Julia 101

Welcome to the first laboratory session for the *Logic and Machine Learning* course!

The goal of this notebook is to get familiar with the **Julia programming language** and its core functionalities.

Specifically, you will learn about:

- how to use `Pkg.jl` to manage your project's working environment
- variables and types in Julia
- Julia's built-in data structures, like arrays, tuples, dictionaries and sets
- functions in Julia (and how to use `BenchmarkTools.jl` to track performance)
- how to define your own data structures in Julia

If you missed the setup instructions, please refer to the `README.md` file in the root folder.

Pkg.jl

`Pkg.jl` is Julia's built-in package manager, that we can leverage to easily manage the project we are working in.

Instead of manually writing esoteric configuration files, we can do everything by simply executing `Pkg.mycommand(...)`!

For example, if we want to add a new dependency `foo.jl` to our project, we need to execute `Pkg.add("foo")`, and all the necessary data will be downloaded from Julia's general package registry. Then, the changings will be tracked in a `Project.toml` file.

You can guess what `Pkg.remove("foo")` does.

To load an already existing `Project.toml`, or to create a new one, you can use `Pkg.activate("filepath")` specifying a relative or absolute filepath.

`Pkg.instantiate()` reads the loaded configuration and resolves it, that is, it tries to precompile all the specified packages, taking care of versioning and populating a `Manifest.toml` file with metadata that we never want to manually change.

`Pkg.update()` forces `Pkg.jl` to visit the general registry and install the newest updates that respect all the versioning constraints of the project.

Finally, `Pkg.status()` consists of a summary of all the dependencies we are dealing with. They will be useful throughout all the lessons.

The next few lines will install all the packages we will need in the following lectures!

```
In [1]: using Pkg              # import the Pkg.jl package
        Pkg.activate("..")  # the Project.toml file is in the parent directory!
        Pkg.instantiate()
        Pkg.update()
        Pkg.status()
```

```
  Activating project at `~/logic-and-machine-learning`
    Updating registry at `~/.julia/registries/General.toml`
    Updating git-repo `https://github.com/aclai-lab/ManyExpertDecisionTree
s.jl`
    Updating git-repo `https://github.com/aclai-lab/SoleReasoners.jl#embed
ding`
  No Changes to `~/logic-and-machine-learning/Project.toml`
  No Changes to `~/logic-and-machine-learning/Manifest.toml`
Status `~/logic-and-machine-learning/Project.toml`
^ [da404889] ARFFFiles v1.5.0
  [6e4b80f9] BenchmarkTools v1.6.3
  [336ed68f] CSV v0.10.15
  [159f3aea] Cairo v1.1.1
  [861a8166] Combinatorics v1.1.0
  [a93c6f00] DataFrames v1.8.1
⌅ [864edb3b] DataStructures v0.18.22
  [7806a523] DecisionTree v0.12.4
  [186bb1d3] Fontconfig v0.4.1
  [271df9f8] FuzzyLogic v0.1.3
  [a2cc645c] GraphPlot v0.6.2
^ [86223c79] Graphs v1.13.1
  [6a3955dd] ImageFiltering v0.7.12
  [f7bf1975] Impute v0.6.13
  [23992714] MAT v0.11.4
⌅ [add582a8] MLJ v0.20.9
^ [a7f614a8] MLJBase v1.9.2
^ [c6f25543] MLJDecisionTreeInterface v0.4.2
  [b59e7f69] ManyExpertDecisionTrees v1.0.0 `https://github.com/aclai-lab/
ManyExpertDecisionTrees.jl#`
  [24e37439] MatrixProfile v1.1.1
  [fb95e5f7] ModalAssociationRules v0.2.1
  [e54bda2e] ModalDecisionTrees v0.5.2
  [8cc5100c] MultiData v0.1.4
  [91a5bcdd] Plots v1.41.4
  [ce6b1742] RDatasets v0.8.1
  [4475fa32] SoleBase v0.13.4
  [123f1ae1] SoleData v0.16.7
  [b002da8f] SoleLogics v0.13.7
  [4249d9c7] SoleModels v0.10.6
  [eb5c4719] SoleReasoners v0.1.0 `https://github.com/aclai-lab/SoleReason
ers.jl#embedding#main`
  [2913bbd2] StatsBase v0.34.10
  [9a3f8284] Random v1.11.0
  [9e88b42a] Serialization v1.11.0
Info Packages marked with ^ and ⌅ have new versions available. Those with
^ may be upgradable, but those with ⌅ are restricted by compatibility cons
traints from upgrading. To see why use `status --outdated`
```

A Julia Cheatsheet

The cells below contain everything you need to start programming in Julia.

You can execute them one after the other by simply selecting the first cell and then pressing `Shift + Enter`.

Note that only the last line of each cell will be printed automatically!

Let us start with the very fundamentals.

In [2]:

```julia
print("Hello, world!")
```

```
Hello, world!
```

Time for some basic math!

In [3]:

```julia
1 + 4
(1 - 5) + (9 * 2)
6 / 5;
```

In [4]:

```julia
35 % 8          # modulo
div(9, 7)       # integer division
9 ^ 3;          # exponentiation
```

In [5]:

```julia
big(2) ^ 38461  # arbitrary precision arithmetic to prevent overflows
```

```
Out[5]: 821605300457270717901267492297310694474211570707136439489464755988617125
        124813828754207520439797950325413302987683645219382296906296433355649926
        394821876423356716933883691439145395940518012011947828367789770651542020
        278834483577902033463410514652835782371123072359595582645983912307785653
        881886102350760504251159886056887539227181981230344275601997119297450582
        201828031499354450725947269170252645682559959111354269510416839315002374
        907219242218641444460430523564932277503520958721371195413421467539254145
        199514704286975744984168611785027177108670215488026989796811850754412677
        919524687901205835240610601052665080396323866989037939922390456984072047
        802419976741186456324086224222520185341388856014286545866254375943490441
        920042909001036897585433145412362218880194989885396050994631080620479046
        398504629395705405287462052770988421061224045790088206086312201993863131
        015648793297099943747468029043663404708907718068931209894912156022566318
        223070869414543436031102930641488247267445803033754436035159808003324644
        362563705965669775568266087346484857756120063971941480502063961153004942
        422915867258447092840555142712976522647266968185029304751464595595497595
        185446397865019001888624922843026579648396273672004724075006400859920523
        718825454663695973533741409639586395292872940907598260831766575555234588
        658720136958604654594132433029629038961742672113375213138992258592403981
        730812846684296127305702749069192936876877836086442168863676537166191528
        022611690028840286366939650447077955002305303120945529822028630489642609
        837681083402882651141952996394106627118732149606535919539866580015831842
        238934060657154438798535950141935999838284875152655225941179799731277191
        302467466044472409138255234732120290555768832468560601562464474126948318
        821232315616757690542491419348353497587114016085067496679537536279173037
        461862128558409802916434992641375812633325976900881121153480257682112816
        674173040683992767625495616299345979922434914172314031467002923208517627
        044055272847412320812697403338225061091955834812673202036912034269232072
        279979580716278447658135406169262704167503129669795881225185505133817681
        655395724688972844794512960673519011190512115922572202820690305776662336
        797110088468633807120184511183326245655995113990950246209711484278856403
        538495870286000671119948530673290540277807624844722816714028249328587599
        675059452059852390953403435296999621986294733743366665480967874350081600
        325249792143885940473798322167171663641096138258614979144124464590424903
        342568722229901414997258091061289711372043678259336260687245139696099662
        214920489758324914230873852704728314643701252251240510446677086843366196
        787270473722063872046693425272673497680492114584707113787753047625289958
        754217630261788283115678042784380552758481403037399275610700077406415362
        845612168392083605296995956874095293953866364732736784613986933900429799
        111746183081370867554968766144315135234947539875400874719510445445753168
        120023745044089941829598967129569597311682237274458939043037367675511847
        534380383516740423221351995749465096576249951305715037369129408410153066
        687828299309425021060601093183982911131256050595691409588221651286301363
        063449227626561346152424365067852491878741919357271585141647233737656930
        294069535417127599958333196250725863147167398641816603446160227439299931
        190875847328861467926134594470511702674581754023410527412445138581534495
        948413295985725651176561741659715987419884958247088705711171636614567789
        7455525203791521211815909615476177945428323363362795144705044915099010801
        808161033379888390839591798658554964368798053047409571221892653913866404
        209505503469766645079014165688396585421194031745122521679506320275161692
        146238828434848791188639798618972039911871879795335725157504537987006186
        752136403089811055892905247963596060147618348546060101651284236389790098
        357659713581362371648740260847849983892245502386138468656066660349815844
        155307561862266352255866077318206121155515478993114416038534630691791254
        317665926821668144603497571010250222675332695350442131563843611967661664
        554414726807259315606955940451751940896317402441947093239811477824115149
        188044471341956900627176234919773891391812363742272543584079346997670269
        316819248422595708849211841837876535552461135932873136194833326447319752
        751712214502159703535043399077567565542209670878008083253162984881827805
        266738423920189632140394509847739844426082759930840969568565113203992161
```

```
410945325850668839801396852508655723850488101109220640624502796829838769
483492394518704395308890548119496349217154879588775738099094406319640688
622709203960978479287141621668679038363847299695430382008611128467307302
076403794477441426619534110790740997578627017322518401744897508518749238
689004906816031046208093578745155668180848212896567408133386820379407410
557823821569025444465598058174822546511877756191209100853788517435775138
242506163768040854042470384012604684261590534845021765792561675783513933
304870308234534886481897160289235053693023031652288678494413478569380844
284188512589700792332410845655950873591873596503335752035618532308093009
503621016093171931639045965089241622457808614487869341536609088933772720
362329147979186322383747648458896791148122423280514731768259220572322794
102511313474090920681182062170796177212095937415605368964212146768048761
985882643910126154965639239938665102846448184028006799037330812807697261
393501187482484059856864037085079400040238454889655641094229338329048838
076276600268532030459679831734571932848763866438861178996893980779083615
418550847373236264849703350791103465768161643426265715794436345275629267
922131003986670857595141189654638590543860153112557157155926790454976346
909515895743283186065523117654250159637970182220954495518313778161635754
903259751407795959436965359505972472181475095968526048823316153693867743
638720709177522145689345673844219719364517840994021311958888848986130923
728246967556070079261402981184644840693100853446390984256147261058252610
293779929496064238616222258428996467475274200098660842895430942742160049
719545879688940310886973546985242800717423186131448746548368069045691589
295347120633201441643531801869608069129354363310033964873367368455807207
529228079596108918928260954519069349641956671253390636332543442821802794
195432218688607784159540074324331352866595319777355654710532564333248222
670907643142688918288769079009866772958566834732305926651976966134549849
728007344366581005020124545224125099696637459490178942245135904198455288
181200948027837760413276511746617206935847003103963298027420382293086288
236029446888719758581236950297173901703898711028777438260018278896259853
746640257550317782522651301328949889353731019605247326439412007720869575
063255623866977676651618529433745225187884825404917222724558857929425328
087719325700274488145974935567882646656720101278894971987590320688661332
233080982933694814991919717109558381037599832953085377696575607196994569
379111086792626918435883333159355288059095756997229002672032633773935043
785923721394871120367257524830788010929111813371838192321897742414859204
500405902263930511394487687068971278083010945060083168940467593208850695
576793170187378841416050672661160361047259038293851922142772026405908080
866363224472005206895349668094031311917665041105614632281866582404368611
805873543800047690662700325762399489943193061965319847830960276693960269
078510264836064324622406918327409914213412022909470811499230212242307508
602807321417437870894590014383698131135392581602134624534676612226626411
491988579332771041640562582771993517241702651592180814355748622095133977
673434538521357754736486743721962247709348226023321655121978125657350173
212641435641817279101646033153054273467234992889587759481337755048826437
884539608761983827117361432409569904786886572378121113294259371762177699
351222401143928464045148484767391570497687980471590290619529482503815834
385362156690847964051828440721080466590471517381667511846368811043225633
014008265600406358879472414853557541485164824774586062129166167718503938
186889950361860512325743894421648303562758644122405359997762672516684847
924104379281745802541119921362333212867204647976032354119211688695162363
554195912457955704706293014291898128373234189094466828686640718812287936
814825697605553503154486880043351628226449035296782419735870687388200835
385271661513511169505367801937536391388743745751081228961511450058828091
602421924150931291387599933585405603143747533956248011997195157000003687
508645801894043696460573800520702077611980240028512167286028327072124035
582321807519454357149694175854463442941810445981571515722404871113581617
989089656277924449658182770657014090274646638963478110604770862949513774
906209988704953524225998629336410324626042781587664198820875267976524705
366087858351277272147351703659445204881189155797928075352595213496987502
```

```
901795870945161219714376370843550050497763714206981567412316070390618171
757646706658800977507016345645176125176033305131499048478061514205374940
837637906595257544551924392324522595742057291492530926347122871708126751
488003105521605474870516994063799356549472334712098187685570390574221811
778021945542896937154447934433370841314715794591199144105472819441326033
170138328433786122205385954309614559126460771599401489711394365295670872
591187997553084438833427895286178761331676261238556721448243890446627208
854152457824694656921051981366126701416345106040314616883429840745059495
541712039040333539243186233793015564154782661154163740593458209681242941
459420299660291402535286780500842897911621903509306615065908656858017624
222788527825897284905022143224109598240495673814468445200006733031901351
304579748889365806065780682334147253130073468674270981104496771140614062
310176948903588172187621170367740304084550309782427789183653437590212409
482423442658344032081864261625424723225193353392729355591964498010395200
847854193440010910178036386895560395867062045935361431438558236242354238
763636111487309202536925482051147014717338006821104510551856072019291408
129939657981694703345817259847963330165389250657380045223901641153399783
101051259529986137825454403562225114777309457968056174096554257805603547
773243500547487063516225754500701942341312421648434782712217277409107616
528792960375055571901401753712935830822735652458378547665431409515743045
005209782093839171651960802591057488616512115179309839945958716770321602
578067873918627346399887286136101576827791640223139983840692761276218777
245401412874060506199586750600728632954856917716071673204468893681547120
085531136669509461790816215821125693942783014453251627907173782764627027
470924099482653189614214558198513805424573314868141864347598687966638986
178833790963942816044815715267451181171269796822452471508232830892774227
841069220803026988561428499678782517567156595798553324900604007288492688
047934972982603059312579855304315461099348171381575018045132608361728453
281196241491296443134507893769053930818899509122139201531148904690266049
650581669968542236265600356806967324884846957844820722750762137951508948
698069273462846772021064013329214550236469558756503949068682689455729308
650453535880712472976452281460991119304125652849744460961039537893988069
123511289156986747891080458618325440656164411639846721226650867770262395
710732890590906712255635202683920906767115622995536908453065592731909901
517982124175497105720834638425287034709456344694769063277812700301452461
651032902560562519581612881607343714566339236616894395933454957015385435
793012975702559303138826768145830527071441390911167504010317837146196675
341543703089135570262656845285263367336452434674324768019460208050396608
273875328607778984616061372596383915439056570130777396418439037567059632
945343907840460177701568800634665678777843884309847354558874299097146664
7594992190062752764411910289420384113851819079430100221952
```

Big, isn't it?

Now time for some logic!

In [6]:
```
true
false

true && false   # logical and
true || false   # or
!true           # not
true ⊻ true;    # xor
```

In [7]:
```
1 & 0;
```

In [8]:
```
6 & 2;
```

In [9]:

```
6 | 1;
```

In [10]:

```
xor(6, 2);
```

And now, let's do some numeric comparisons...

In [11]:

```
26 < 43
38 > 32
79 <= 50    # less or equal than
28 >= 84
19 == 71    # equal to
69 != 39    # not equal to
```

Out[11]:

```
true
```

What about strings?

In [12]:

```
"lexicographical" > "comparison"
```

Out[12]:

```
true
```

In [13]:

```
foo = "string"  # this is a variable
print("This is a $(foo) interpolation!")
```

```
This is a string interpolation!
```

Variables and Types

Variable names start with a letter or an underscore, and they can't be declared without a value.

We can play with all the variables appearing in the previously executed cells.

Every variable is associated with a *type*, and types are organized in hierarchical structures of *abstract types*.

The very bottom of this hierarchical structure includes the *concrete types*, and variables are particular instantiations of such types.

We can investigate the types of a variable using `typeof` , `supertype` and `subtypes` .

In [14]:

```
println(typeof(foo))
println(supertype(typeof(foo)))
println(supertype(supertype(typeof(foo))))
```

```
String
AbstractString
Any
```

In [15]:

```
bar = 93
bar |> typeof |> println # a more readable rewriting of println(typeof(ba
bar |> typeof |> supertype |> println
bar |> typeof |> supertype |> supertype |> println
bar |> typeof |> supertype |> supertype |> supertype |> println
```

```
Int64
Signed
Integer
Real
```

In [16]:
```
subtypes(Real)
```

Out[16]:
```
4-element Vector{Any}:
 AbstractFloat
 AbstractIrrational
 Integer
 Rational
```

In [17]:
```
baz = 75.10
current_type = typeof(baz)

while current_type != Any
    println(current_type)
    current_type = supertype(current_type)
end
```

```
Float64
AbstractFloat
Real
Number
```

In [18]:
```
subtypes(Integer)
```

Out[18]:
```
3-element Vector{Any}:
 Bool
 Signed
 Unsigned
```

In [19]:
```
# "<:" is the subtype operator
Integer <: Number
```

Out[19]:
```
true
```

In [20]:
```
Int <: Integer <: Number
```

Out[20]:
```
true
```

Special Variables and Special Types

Variables can hold special values to gracefully handle errors, such as the Float64 `NaN` value.

Similarly, particular semantics are conveyed by unique types, such as `Nothing`, `Symbol` and `Union`.

The former type can only be instantiated with the value `nothing`.

Symbols are used to encode uninterpreted names instead of values, and helps when dealing with metaprogramming.

Unions are special types, used to represent more than one type at once.

In [21]:

```
println(0/0)
println(isnan(0/0))
```

```
NaN
true
```

In [22]:

```
# this is a special variable representing the absence of any value
placeholder = nothing
isnothing(placeholder)
```

Out[22]:

```
true
```

In [23]:

```
print_return_value = println("Println does not return anything")
println(print_return_value)
```

```
Println does not return anything
nothing
```

In [24]:

```
# symbols are special values intended to represent names rather than valu
plussymbol = Symbol("+")
xsymbol = Symbol("x")
twosymbol = Symbol("2")
```

Out[24]:

```
Symbol("2")
```

In [25]:

```
x = 10
expr = :(x + 2)
dump(expr)
```

```
Expr
  head: Symbol call
  args: Array{Any}((3,))
    1: Symbol +
    2: Symbol x
    3: Int64 2
```

In [26]:

```
const MyCustomType = Union{Float64,String}
```

Out[26]:

```
Union{Float64, String}
```

In [27]:

```
# this is a synonym of: typeof("papadimitriou") <: MyCustomType
"papadimitriou" isa MyCustomType
```

Out[27]:

```
true
```

In [28]:

```
:papadimitriou isa MyCustomType
```

Out[28]:

```
false
```

Control Flow Structures

Let's see how to implement branching decisions and repeating blocks of code.

In [29]:

```
n = 3

if n % 2 == 0
    println("$n is even")
```

```
else
    println("$n is odd")
end
```

```
3 is odd
```

In [30]:

```
threshold1 = 1.0
threshold2 = 2.0
signal = 1.5

if signal < threshold1
    println("The signal is low.")
elseif signal > threshold2
    println("The signal is high.")
else
    println("The signal is neither too low nor too high.")
end
```

```
The signal is neither too low nor too high.
```

In [31]:

```
for i in 0:10
    if i % 2 == 0
        println(i)
    end
end
```

```
0
2
4
6
8
10
```

In [32]:

```
for i in 0:2:10
    println(i)
end
```

```
0
2
4
6
8
10
```

In [33]:

```
for i in 10:-1:0
    println(i)
end
```

```
10
9
8
7
6
5
4
3
2
1
0
```

In [34]:

```
for c in "logic\nis\nfun"
    print(c)
```

```
end
```

```
logic
is
fun
```

In [35]:

```
(c for c in "logicisfun")
```

Out[35]:

```
Base.Generator{String, typeof(identity)}(identity, "logicisfun")
```

In [36]:

```
# Looks good, right?

#=
i = -23
while true
    if i < 0
        continue
    elseif i % 2 == 0
        println(i)
    elseif i > 30
        break
    end

    i += 2
end
=#
```

Data Structures

Arrays

`Array{T,N}` are dynamic ordered collections of dimensionality `N`, embodying elements of type `T`.

For instance, the type `Array{Float64,1}` encodes *vectors* of floats, while `Array{Float64,2}` encodes *matrices* of floats.

Note that the dimension number is not a type by itself, as it is an integer, but it is treated like a type in this context for optimization purposes.

In [37]:

```
baz = [74, 94]
typeof(baz)
```

Out[37]:

```
Vector{Int64} (alias for Array{Int64, 1})
```

In [38]:

```
push!(baz, 4)
println(baz)
```

```
[74, 94, 4]
```

In [39]:

```
try
    push!(baz, 5.9)
catch
    println("You can't push a Float64 into a $(typeof(baz))")
end
```

```
You can't push a Float64 into a Vector{Int64}
```

In [40]:

```
baz = convert(Vector{Float64}, baz)
```

Out[40]:

```
3-element Vector{Float64}:
 74.0
 94.0
  4.0
```

In [41]:

```
baz = [23, 0.7, 81] # the automatic conversion to Vector{Float64} is due
promote_rule(Float64, Int)
```

Out[41]:

```
Float64
```

In [42]:

```
println("The content of baz is: $(baz)")
println("The length of baz is: $(length(baz))")
println("The size of baz is: $(size(baz))")
println("The first element of baz is: $(baz[1])")
println("The first two elements of baz are: $(baz[1:2])")
println("The last element is: $(baz[end])")
```

```
The content of baz is: [23.0, 0.7, 81.0]
The length of baz is: 3
The size of baz is: (3,)
The first element of baz is: 23.0
The first two elements of baz are: [23.0, 0.7]
The last element is: 81.0
```

In [43]:

```
println("The minimum of baz is: $(minimum(baz))")
println("The maximum of baz is: $(maximum(baz))")
println("The sum of baz is: $(sum(baz))")
```

```
The minimum of baz is: 0.7
The maximum of baz is: 81.0
The sum of baz is: 104.7
```

In [44]:

```
mysum = 0

for n in baz
    mysum += n
end

println("The 'manually computed' sum of baz is: $(mysum)")
```

```
The 'manually computed' sum of baz is: 104.7
```

In [45]:

```
for (i,n) in enumerate(baz)
    println("The element $(i) of baz is: $(baz[i])")
end
```

```
The element 1 of baz is: 23.0
The element 2 of baz is: 0.7
The element 3 of baz is: 81.0
```

In [46]:

```
for (n, next_n) in zip(baz, baz[2:end])
```

```
    println("$(n)\t$(next_n)") # \t is the tabulation character
end
```

```
23.0    0.7
0.7     81.0
```

In [47]:

```
# an Int vector is not a subtype of a vector containing elements of arbit
# Real types (even floats!)
Vector{Int} <: Vector{Real}
```

Out[47]:

```
false
```

In [48]:

```
# same reasoning if we consider the whole family of Int8, Int32, Int64...
Vector{Integer} <: Vector{Real}
```

Out[48]:

```
false
```

In [49]:

```
# this is fine
Vector{Int} <: Vector{<:Real}
```

Out[49]:

```
true
```

Exercise

Implement your binary search.

Try to search the index of the number `1427` in the following array.

Solution (Base64):

bGVmdCA9IDEKcmlnaHQgPSBsZW5ndGgoYXJyKQp0YXJnZXQgPSAxNDI3Cgp3aGlsZSBsZ

In [50]:

```
using Random
Random.seed!(1605)
arr = sort(rand(1:3200, 500));
```

In [51]:

```
# implement your binary search here
```

Tuples

Tuples are *immutable* fixed-length ordered collections: we can think about them as an immutable version of Arrays.

Hence, if we want to modify a tuple, we have to recreate it completely.

We can explicitly state the type that each element within a tuple must have by enclosing such types ordered in curly brackets, or we can let Julia infer them.

In [52]:

```
qux = (58, 20.9)    # same as Tuple{Int64, Float64}((58, 20.9))
```

Out[52]:

```
(58, 20.9)
```

In [53]:

```
typeof(qux)
```

Out[53]:

```
Tuple{Int64, Float64}
```

In [54]:

```julia
try
    qux[1] = qux[1] + 2
catch
    println("Remember that tuples are are immutable!")
end
```

```
Remember that tuples are are immutable!
```

Dictionaries

Dictionaries are hash tables `Dict{K,V}` with keys of type `K` and values of type `V`.

Under the hood, keys are hashed using the `hash` function of the Julia standard library.

In [55]:

```julia
mydict = Dict{Int, Float64}(74 => 9.4, 45 => 9.2)
```

Out[55]:

```
Dict{Int64, Float64} with 2 entries:
  45 => 9.2
  74 => 9.4
```

In [56]:

```julia
mydict[74]
```

Out[56]:

```
9.4
```

In [57]:

```julia
9.2 in values(mydict)   # check if 9.2 is in the values of mydict
```

Out[57]:

```
true
```

In [58]:

```julia
try
    mydict[30]
catch
    println("The dictionary does not contain a key with value 30.")
end
```

```
The dictionary does not contain a key with value 30.
```

In [59]:

```julia
# alternatively, we can provide a default value for non-existing entries
get(mydict, 30, -1)
```

Out[59]:

```
-1
```

In [60]:

```julia
metadict = Dict{String, Dict{Int, Float64}}(
    "logic" => mydict,
    "machine learning" => mydict
)

metadict["logic"] == mydict
```

Out[60]:

```
true
```

Watch out! The two dictionaries within `metadict` are not copied by value, but by reference.

In [61]:

```julia
println("The values associated with key 74 in the two dictionaries are:")
for (key, innerdict) in metadict
```

```
    println(innerdict[74])
end

println()

for (key, innerdict) in metadict
    println("Adding one in the $(key == "logic" ? "1st" : "2nd") dictiona
    innerdict[74] += 1
end

println("\nThe values associated with key 74 in the two dictionaries are:
for (key, innerdict) in metadict
    println(innerdict[74])
end
```

```
The values associated with key 74 in the two dictionaries are:
9.4
9.4

Adding one in the 1st dictionary
Adding one in the 2nd dictionary

The values associated with key 74 in the two dictionaries are:
11.4
11.4
```

In [62]:

```
o1, o2 = objectid(metadict["logic"]), objectid(metadict["machine learning

println(objectid(metadict["logic"]))
println(objectid(metadict["machine learning"]))

# === is the "identical" operator: it queries the id associated with each
# variable under the hood, rather than just their values
println(o1 === o2)
```

```
13868113764429803215
13868113764429803215
true
```

In [63]:

```
mydict[78] = 16.40  # adding a new key => value pair to the dictionary
mydict
```

Out[63]:

```
Dict{Int64, Float64} with 3 entries:
  78 => 16.4
  45 => 9.2
  74 => 11.4
```

In [64]:

```
# when a function ends with a bang (!), it usually modifies its first arg
delete!(mydict, 78) # use pop! if you also want to retrieve the deleted p
```

Out[64]:

```
Dict{Int64, Float64} with 2 entries:
  45 => 9.2
  74 => 11.4
```

Sets

Sets are unordered collections of unique elements.

They allow for efficient union, intersection and difference set operations.

We can leverage the `in` operator or `issubset` for checking the membership to a set.

```
In [65]: myset1 = Set{String}(["this", "is", "my", "beautiful", "set"]);
         myset2 = Set{String}(["look", "at", "this", "beautiful", "set"]);
```

```
In [66]: union(myset1, myset2)
```

```
Out[66]: Set{String} with 7 elements:
           "this"
           "is"
           "set"
           "beautiful"
           "at"
           "my"
           "look"
```

```
In [67]: intersect(myset1, myset2)
```

```
Out[67]: Set{String} with 3 elements:
           "this"
           "set"
           "beautiful"
```

```
In [68]: setdiff(myset1, myset2)
```

```
Out[68]: Set{String} with 2 elements:
           "is"
           "my"
```

```
In [69]: setdiff(myset2, myset1)
```

```
Out[69]: Set{String} with 2 elements:
           "at"
           "look"
```

```
In [70]: if "my" in myset1
             println("The string 'my' ∈ myset1.")
         end

         myset3 = Set(["this", "is", "set"])
         if issubset(myset3, myset1)
             println("Also, 'this', 'is', and 'set' strings all belong to myset1")
         end
```

```
The string 'my' ∈ myset1.
Also, 'this', 'is', and 'set' strings all belong to myset1
```

Functions

Functions are mappings between a tuple of arguments and a return value.

Julia functions are first-class citizens, meaning that they can be passed as arguments to other functions, they can be returned from functions and can be stored in data structures.

Functions in Julia can have multiple implementations, each specialized to a specific combination of arguments.

This idea of *multiple dispatching* is at the core of the design of Julia, and is the key to its preformance.

In [71]:

```
function add(x, y)
    return x + y    # return keyword is omitted: the last operation is re
end

add(1, 2)
```

Out[71]:

```
3
```

In [72]:

```
subtract(x, y) = x - y

subtract(1, 2)
```

Out[72]:

```
-1
```

In [73]:

```
# the next line returns an anonymous (i.e., nameless) function
add_five = x -> x + 5

add_five(1)
```

Out[73]:

```
6
```

In [74]:

```
# a function can even return a function
divide_by(y) = return x -> x / y

divide_by(5)(10)
```

Out[74]:

```
2.0
```

In [75]:

```
# functions can return multiple values
function powers(x)
    return x, x^2, x^3
end

a, b, c = powers(3)
```

Out[75]:

```
(3, 9, 27)
```

In [76]:

```
typeofpowers = typeof(powers)

println(typeofpowers)
println(supertype(typeofpowers))
```

```
typeof(powers)
Function
```

In [77]:

```
# function names may contain UTF characters and a variable number of argu
function ∑(args...)
    c = 0

    for arg in args
        c += arg
    end
```

```julia
    return c
end

∑(5, 6, 3, 4, 12)
```

Out[77]: 30

In [78]:
```julia
# functions may provide default values for their arguments
function power(x, y=2)
    return x ^ y
end

power(5)
```

Out[78]: 25

In [79]:
```julia
# the broadcast (.) operator applies the function to each member of a col
power.(collect(0:10))   # 0:10, synonym of 0:1:10, goes from 0 to 10 with
```

Out[79]:
```
11-element Vector{Int64}:
   0
   1
   4
   9
  16
  25
  36
  49
  64
  81
 100
```

In [80]:
```julia
function myprint(x::Int64)
    println("This is an awesome print for the number $(x)")
end

function myprint(x::Float64)
    println("This is a beautiful print for the number $(x)")
end

myprint(1)
myprint(1.0)
```

```
This is an awesome print for the number 1
This is a beautiful print for the number 1.0
```

In [81]:
```julia
# note the difference between positional and keyword arguments;
# the formers are identified by their position in the function signature,
# while the latters are recognized by their name when providing a value.
function myprint(x::String; mode::Symbol=:plain)

    if mode == :plain
        punctuation = ["", ""]
    elseif mode == :punctuation
        punctuation=[",", "."]
    else
        throw(ArgumentError("The specified mode $(mode) is not available.
    end

    println(
        "This is an awesome print$(punctuation[1]) " *
        "wrapping the string '$(x)'$(punctuation[2])"
    )
end

myprint("Hello, World!")
myprint("Hello, World!"; mode=:punctuation)
```

```
This is an awesome print wrapping the string 'Hello, World!'
This is an awesome print, wrapping the string 'Hello, World!'.
```

It is important to track the performance of the functions we write.

Below, we leverage the BenchmarkTools.jl package for comparing the execution time of a naive implementation of the sum function, `naive_sum`, with a smarter one, `efficient_sum`.

The generic type `T` we associate with the given collection, `xs`, is a placeholder possibly indicating any subtype of `Real`. When `efficient_sum` is called with an argument of type `Vector{Int64}`, it has the chance to compile *specialized code*: this is exactly the purpose of multiple dispatch!

Note how we use `@inbounds` and `@simd` macros to speedup the code (remove them if you don't believe us, and run the benchmark again!):

- `@inbounds` disables the default bounds checking that must be performed everytime `xs` is accessed
- `@simd` indicates that the loop can be evaluated out-of-order

In [82]:

```
using BenchmarkTools
```

In [83]:

```
function naive_sum(xs)
    result = 0

    for x in xs
        result += x
    end

    return result
end
```

Out[83]:

```
naive_sum (generic function with 1 method)
```

In [84]:

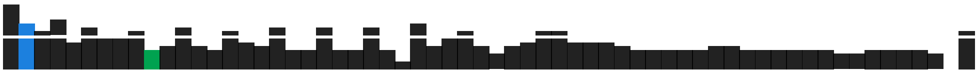
```
# this is nearly the Julia's implementation of the sum function!
function efficient_sum(xs::Vector{T}) where {T<:Real}
    # beware of type stability:
    # this cannot be an Int8(0), or a Float64(0): it has to match T!
    result = zero(T)

    @inbounds @simd for x in xs
        result += x
    end

    return result
end
```

```
Out[84]: efficient_sum (generic function with 1 method)
```

```
In [85]: xs = rand(100000);
```

```
In [86]: @benchmark naive_sum(xs)
```

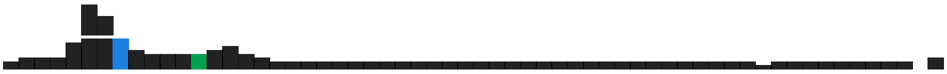
```
Out[86]: BenchmarkTools.Trial: 10000 samples with 1 evaluation per sample.
          Range (min … max):  59.552 μs … 595.685 μs  ┊ GC (min … max): 0.00% …
         0.00%
          Time  (median):     59.613 μs               ┊ GC (median):    0.00%
          Time  (mean ± σ):   60.517 μs ±   9.065 μs  ┊ GC (mean ± σ):  0.00% ±
         0.00%
```



```
          59.6 μs       Histogram: log(frequency) by time       66 μs <

          Memory estimate: 16 bytes, allocs estimate: 1.
```

```
In [87]: @benchmark efficient_sum(xs)
```

```
Out[87]: BenchmarkTools.Trial: 10000 samples with 6 evaluations per sample.
          Range (min … max):  5.611 μs … 73.454 μs  ┊ GC (min … max): 0.00% … 0.0
         0%
          Time  (median):     5.796 μs              ┊ GC (median):    0.00%
          Time  (mean ± σ):   5.941 μs ±  1.712 μs  ┊ GC (mean ± σ):  0.00% ± 0.0
         0%
```



```
          5.61 μs        Histogram: frequency by time        7.2 μs <

          Memory estimate: 16 bytes, allocs estimate: 1.
```

Structures

Julia's user defined composite types are called *structures*.

They are collections of named fields, and can be instantiated via specific functions called `constructors` .

Structures are *concrete types*, meaning that their instances are subtypes of some abstract type (the default is `Any`), and are immutable by default.

Below, we play with structures to model a little scenario involving animals.

```
In [88]: abstract type Animal end     # Let's first define a new abstract type
```

```
In [89]: struct Dog <: Animal     # Dog is a subtype of animal
             name::String
             age::Int

             function Dog(name, age)
                 if age < 0
                     throw(ArgumentError("Age cannot be negative ($(age) is provid
                 end
```

```
        new(name, age)
    end
end

name(d::Dog) = d.name
age(d::Dog) = d.age

speak(d::Dog) = println("Woof, I am $(name(d)) and I am $(age(d))... woof
```

Out[89]:
```
speak (generic function with 1 method)
```

In [90]:
```
buddy = Dog("Marathon", 7)
speak(buddy)
```

```
Woof, I am Marathon and I am 7... woof!
```

In [91]:
```
struct Cat <: Animal
    name::String
    age::Int
    lives::Int

    function Cat(name, age; lives=7)
        if age < 0 || lives < 0
            throw(ArgumentError(
                "Age and lives cannot be negative " *
                "($(age) and $(lives) are provided)"
            ))
        end
        new(name, age, lives)
    end
end

name(c::Cat) = c.name
age(c::Cat) = c.age
lives(c::Cat) = c.lives

function speak(c::Cat)
    println(
        "My name is $(name(c)), I am $(age(c)) and I have $(lives(c)) liv
        "Meaow."
    )
end
```

Out[91]:
```
speak (generic function with 2 methods)
```

In [92]:
```
pal = Cat("Booted Cat", 3)
speak(pal)
```

```
My name is Booted Cat, I am 3 and I have 7 lives. Meaow.
```

In [93]:
```
struct Axolotl <: Animal
end

try
    speak(Axolotl())
catch
    println("This triggers a method error!")
end
```

```
This triggers a method error!
```

In [94]:

```julia
# we can gracefully handle non-existing dispatches thanks to general inte
function speak(a::Animal)
    throw(
        ErrorException(
            "Please provide an implementation of speak(a::$(typeof(a)))"
        )
    )
end
```

Out[94]:

```
speak (generic function with 3 methods)
```

In [95]:

```julia
ozzy = Axolotl()
speak(ozzy)
```

```
Please provide an implementation of speak(a::Axolotl)

Stacktrace:
 [1] speak(a::Axolotl)
   @ Main ./In[94]:3
 [2] top-level scope
   @ In[95]:2
```