Aide-mémoire Julia (v1.5)

inspiré de juliadocs.github.io/Julia-Cheat-Sheet

triscale-innov.com

Fondamentaux

Affectation	`answer = 42`
	`(x, y, z) = (1, π, "A string")`
	`x, y = y, x  # échanger x et y`

Déclaration constante	`const DATE_OF_BIRTH = 2012`
Commentaire en fin de ligne	`i = 1  # Ceci est un commentaire`
Commentaire délimité	`#= Ceci est un autre commentaire =#`

Chainage	`x = y = z = 1  # de droite à gauche`
	`0 < x < 3  # true`
	`5 < x != y < 5 # false`

Définition de fonction	`function add_one(i)`
	`  return i + 1`
	`end`

Insertion de symboles LaTeX	`\delta` puis `<Tab>`

Opérateurs

Opérations fondamentales	`+, -, *, /`
Exponentiation	`2^3 == 8`
Division	`3/12 == 0.25 # Quotient d'entiers -> résultat flottant`
Division inverse	`7\3 == 3/7`
Reste (modulo)	`x % y` ou `rem(x,y)`
ET et OU logiques (court-circuités)	`a && b` et `a || b`
Négation	`!true == false`
Opérateur ternaire	`a == b ? "Equal" : "Not equal"`
Égalité	`a == b`
Equivalence d'objets	`a === b`
Non-égalité	`a != b` ou `a ≠ b`
Inégalités strictes	`<` et `>`
Inégalités larges	`<=` ou `≤` et `>=` ou `≥`

Opérations élément-par-élément	`[1, 2, 3] .+ [1, 2, 3] == [2, 4, 6]`
	`[1, 2, 3] .* [1, 2, 3] == [1, 4, 9]`
	`cos.([0, pi, 2pi])  == [1, -1, 1]`

REPL

Rappeler le dernier résultat	`ans`
Interrompre l'exécution	`<Ctrl> + C`
Effacer l'écran	`<Ctrl> + L`
Rappeler commande précédente	`<Ctrl> + R`
Lancer un programme Julia (script)	`include("filename.jl")`
Aide sur la fonction `func`	`?func`
… ou sur un sujet plus large	`?"topic"` ou `apropos("topic")`
Mode de commandes shell	`;` (en début de ligne)
Mode du gestionnaire de paquets	`]` (en début de ligne)
Mode d'aide	`?` (en début de ligne)
Sortir d'un mode spécifique	`<BackSpace>` (en début de ligne)
Quitter le REPL	`exit()` ou `<Ctrl> + D`

Gestionnaire de paquets (REPL en mode `pkg>`)

Activer un environnement	`activate path/to/project`
Instancier l'environnement courant	`instantiate`
Installer `PackageName`	`add PackageName`
Supprimer `PackageName`	`rm PackageName`
Mettre à jour tous les paquets	`update`
… ou seulement `PackageName`	`update PackageName`
Utiliser une version de dévelop-pement de `PackageName`	`dev PackageName` ou `dev GitRepoUrl` ou `dev path/to/PackageName`
Après `dev`, revenir à une version publiée de `PackageName`	`free PackageName`

Structures de contrôle et boucles

Tests conditionnels	`if ... elseif ... else ... end`

Boucle simple	`for i in 1:10`
	`  println(i)`
	`end`

Boucle à indices multiples	`for i = 1:10, j = 1:5`
	`  println(i*j)`
	`end`

Énumeration	`for (idx, val) in enumerate(arr)`
	`  println("le $idx-ème élément est $val")`
	`end`

Boucle "tant que"	`while bool_expr`
	`  # do stuff`
	`end`

Sortie de boucle prématurée	`break`
Saut à l'itération suivante	`continue`

Fonctions & méthodes

Déclaration d'une fonction "nue"	`function fun end`
Définition d'une méthode (courte)	`fun(x) = 2x  # return optionnel`

Définition d'une méthode (longue)	`function fun(req1, req2, opt1="def"; key1="def", key2)`
	`  # avant ';' : arguments positionnels`
	`  # après ';' : arguments par mot-clé`
	`  # name=value : valeur par défaut => arg. optionnel`
	`end`

Appel d'une méthode	`fun(1, 2, key2=3) # opt1 et key1 non précisés`

Typage des arguments et de la valeur de retour	`function fun(x::Number, y::AbstractArray) :: typeof(x)`
	`  return x * length(y)`
	`end`

Appel élément-par-élément (*broadcasting*)	`f(x, y) = 2x + y` `f.([1, 2, 3], [10, 20, 30]) == [12, 24, 36]`
Liste des méthodes existantes	`methods(fun)`

Aide-mémoire Julia (v1.5)

inspiré de juliadocs.github.io/Julia-Cheat-Sheet

triscale-innov.com

Types

Relations entre types	`SubType <: SuperType`
Parent et descendants	`supertype(TypeName)` et `subtypes(TypeName)`
Supertype universel	`Any`
Type d'une valeur	`typeof(value)`
Assertion de type, annotation	`var::TypeName`
Relation valeur ∈ supertype	`value isa TypeName` ⇔ `typeof(value) <: TypeName`

Déclaration de type	`struct Person  # ou mutable struct`
	`  name :: String`
	`  age  :: Int`
	`end`

Construction d'une instance	`john = Person("John", 42)`

Type abstrait et sous-type	`abstract type Bird end`
	`struct Duck <: Bird`
	`  name :: String`
	`end`
	`Duck("Donald") isa Bird`

Type paramétrique	`struct Point{T <: Real}`
	`  x::T`
	`  y::T`
	`end`
	`p = Point{Float64}(1, 2)`

Type union	`1 isa Union{Int, String}`

Modules

Définition de module	`module ModuleName`
	`# déclarations de fonctions, types...`
	`# 'export' rend les définitions accessibles`
	`end`

Utilisation des noms d'un module	`using ModuleName  # tous les noms exportés`
	`using ModuleName: x, y  # uniquement x, y`
	`import ModuleName  # uniquement ModuleName`

Utilisation des noms d'un module et extension des fonctions	`import ModuleName: x, y # uniquement x, y` `import ModuleName.x  # uniquement x`
Liste des définitions	`names(ModuleName)`

Gestion des erreurs

Signaler une erreur	`error("message")`
Lancer une exception	`throw(SomeException(...))`

Gérer les erreurs	`try`
	`  # ...`
	`catch e`
	`  # exécuté en cas de problème`
	`finally`
	`  # exécuté dans tous les cas`
	`end`

Collections et fonctions d'ordre supérieur

Application d'une fonction à chaque élément de `coll`	`map(fun, [1, 2, 3]) == [fun(1), fun(2), fun(3)]` ou `fun.([1, 2, 3]) # Fusion des boucles si possible`
Réduction d'une collection par un opérateur	`reduce(+, [1,2,3]) == 1 + 2 + 3` `foldr(f, [1,2,3]) == f(1, f(2, 3))`
Rapprochement de plusieurs collections	`zip([1, 2, 3], ['a', 'b', 'c'])` `== [(1, 'a'), (2, 'b'), (3, 'c')]`

Fonctions anonymes	`reduce(+, coll)`
	⇔ `reduce((x,y)->x+y, coll)`
	⇔ `reduce(coll) do x, y`
	`  x + y`
	`end`

Compréhension (instanciée)	`arr = [f(elem) for elem in coll]`
Générateur (non instancié)	`gen = (f(elem) for elem in coll)`

Tableaux

Tableau vide (mais typé)	`Float64[]`
Tableau non initialisé	`Array{Float64, 1}(undef, 10)  # 10 élems, dimension 1`
Rempli de zéros	`zeros(Int64, 10)  # 10 élems entiers`
Rempli de uns	`ones(Int64, 10)  # 10 élems entiers`
Rempli de `val`	`fill(42, 10)  # 10 élems valant 42`
Accès (indexation à partir de 1)	`arr[1] == first(arr) ; arr[end] == last(arr)`
Sous-tableau (copié)	`arr[2:5]`
Mutation	`arr[begin] = 42  # begin -> indice du premier élément`
...d'un sous-tableau	`arr[3:end] = 0  # end  -> indice du dernier élément`
Vue (liée au tableau d'origine)	`@view(arr[2:5])`
Tableaux vus comme piles/files ajout/suppression en début/fin	`push!(arr, 2) ; push_first!(arr, 1)` `pop!(arr); popfirst!(arr)`
Insertion / suppression d'éléments	`insert!(arr, idx, val) ; deleteat!(arr, idx)`

Bibliothèques standard

`DelimitedFiles`	(fichiers données)	`readdlm`
`Random`	(nombres aléatoires)	`rand`, `randn`, `randsubseq`
`Statistics`	(statistiques)	`mean`, `std`, `cor`, `median`, `quantile`
`LinearAlgebra`	(algèbre linéaire)	`I`, `eigvals`, `eigvecs`, `det`, `cholesky`
`SparseArrays`	(matrices creuses)	`sparse`, `SparseVector`, `SparseMatrixCSC`
`Distributed`	(calcul distribué)	`@distributed`, `pmap`, `addprocs`
`Test`	(tests unitaires)	`@testset`, `@test`, `@test_throws`, `@test_broken`
`Logging`	(messages d'info.)	`@debug`, `@info`, `@warn`, `@error`
`Printf`	(chaînes formattées)	`@printf`, `@sprintf`
`Dates`	(dates et heures)	`DateTime`, `Date`

Inspection du processus de compilation

Lowering	`@code_lowered fun(args)`
Inférence de type	`@code_typed fun(args)` ou `@code_warntype fun(args)`
Spécialisation & optimisation	`@code_llvm fun(args)  # Repr. intermédiaire (IR) LLVM` `@code_native fun(args) # Assembleur`