CheatSheet - R - Python - Julia - version 1

Frederico Mestre

Packages and Modules – A note on nomenclature.

	R	Python	Julia Programming
Package	In R, packages are collections of functions, data, and other resources bundled together. Packages extend the functionality of the R language by providing additional functions and tools for specific tasks or domains.	In Python, packages are directories that contain multiple modules and a special __init__.py file. Packages provide a way to organize related modules into a hierarchical structure. They enable you to create reusable code libraries and distribute them for others to use.	Packages in Julia are collections of modules and other resources that are distributed and installed separately. They are commonly used to share and reuse code across different projects. The Julia package manager (Pkg) is used to manage packages and their dependencies.
Module		A module is a file containing Python code that defines functions, classes, and variables. Modules are used for code organization and reusability. They allow you to logically group related code and make it accessible from other parts of your program using the import statement.	a module is a container that organizes related code, variables, and types. It allows you to encapsulate functionality and control the visibility and scope of objects. Modules are used for code organization, avoiding naming conflicts and providing namespaces.
Main repository	The Comprehensive R Archive Network (**CRAN**) is the primary repository for R packages.	The Python Package Index (**PyPI**) is the primary repository for Python packages, and the pip package manager is used to install and manage packages.	The main package repository is the **General Registry**, which is managed by the Julia community. It serves as the central repository for Julia packages and is the primary source for discovering, installing, and managing packages in Julia. The General registry can be accessed through the Julia package manager (Pkg) using the add, update, and dev commands.

IMPORTANT! - A note on indexing

R	Python	Julia Programming
Indexing in Python starts with 0. In R and Julia starts with 1. As such selecting the first element:		

`my_vector <- c(10, 20, 30, 40, 50)`
`# Accessing the first element`
`first_element <- my_vector[1]`
	`my_list = [10, 20, 30, 40, 50]`
	`# Accessing the first element`
	`first_element = my_list[0]`
		`my_vector = [10, 20, 30, 40, 50]`
		`# Accessing the first element`
		`first_element = my_vector[1]`

Basic actions

Action	R	Python	Julia Programming
Get working directory	getwd()	import os os.getcwd()	pwd()
Change working directory	setwd()	import os new_dir = "/path/to/new_dir" os.chdir(new_dir)	new_dir = "/path/to/new_dir" cd(new_dir)
Install packages	install.packages("package")	**<u>Installing packages with conda</u>** conda install package_name **<u>Installing packages with pip</u>** pip install package_name **Note:** When using Anaconda, it is recommended the use conda for package installation, as it can manage packages specifically built and	Pkg.add("package")

		optimized for Anaconda environments. However, pip can still be used if a package is not available in the Anaconda repositories.	
Load packages	library("package_name")	import package_name	using package_name

Delete object	rm()	del()	a=1
			
			#delete a
			a = nothing

List all objects in the environment.	ls()	dir()	#Not exactly the same thing as the R function varinfo()
Free memory	gc()	--	GC.gc()

The division of the following two tables stems from my R-centred mind…

In R, there are two main concepts related to object-oriented programming: types and classes. **Types:** In R, every object has a type. The type of an object determines its behaviour and the operations that can be performed on it. Common types in R include numeric, character, logical, integer, and complex. **Classes:** In addition to types, R also supports classes, which provide a way to define custom object types with specific properties and behaviours. Classes are defined using the class() function, and objects of a specific class are referred to as instances of that class.

Object types/classes

	R	Python	Julia Programming
Evaluate type	**typeof()** **R has 5 basic classes:** **numeric:** Floating-point numbers (e.g., 3.14, -2.5). **integer:** Integer numbers (e.g., 1, 2, -3). **character:** Represents strings of text (e.g., "Hello"). **logical:** Represents boolean values, either TRUE or FALSE. **complex:** Represents complex numbers with real and imaginary parts (e.g., 2 + 3i).	**type()** **int:** Integers (e.g., 1, 2, -3). **float:** Floating-point numbers (e.g., 3.14, -2.5). **complex:** Complex numbers with real and imaginary parts (e.g., 2 + 3j). **bool:** Represents the truth values True or False.	**typeof(x)** **Int:** Integer type. **Float64:** 64-bit floating-point number type. **Bool:** Boolean type (true or false). **Char:** Character type. **String:** String type. **Symbol:** Symbol type (immutable, used for identifiers).
logical/bool	TRUE FALSE	True False	true false
integer/int/Int64	2L, as.integer(2)	2	2
double/float/ Float64	2.5	2.5	2.5

Complex	2 + 3i	2 + 3j	z1 = complex(3, 4) #Create a complex number with real part 3 and imaginary part 4 z2 = 2 + 5im #Create a complex number using the shorthand syntax
character/str/Char and string	"a"	"a"	"a" #Char "abc" #String
Numeric	2, 2.0, pi	--	
Getting help on a function	?function_name ??function_name	help(function_name)	Press ? than the name of the function.

Object types/classes

Action	R	Python	Julia Programming
Evaluate class	**class()** **Date:** Represents dates without time. **POSIXct:** Represents date and time using the POSIX standard. **POSIXlt:** Represents date and time using a list structure. **vector:** Represents a sequence of elements of the same type. **factor:** Represents categorical data with predefined levels. **matrix:** Represents a 2-dimensional array-like structure.	**type()** **list:** Ordered, mutable sequences (e.g., [1, 2, 3]). **tuple:** Ordered, immutable sequences (e.g., (1, 2, 3)). **str:** Strings, immutable sequences of characters (e.g., "Hello"). **dict:** Mutable mappings of keys to values (e.g., {"key": "value"}). **set:** Mutable, unordered collections of unique elements (e.g., {1, 2, 3}).	**typeof(x)** **Primitive Types** **Tuple:** Ordered, immutable collection of values. **Array:** Ordered, mutable collection of values. **Dict:** Associative collection of key-value pairs. **Set:** Collection of unique elements. **These are the primitive types. Then there are other types, such as abstract types.**

	array: Represents a multi-dimensional array. **data.frame**: Represents a tabular data structure with rows and columns. **list**: Represents a collection of objects of different types.	**frozenset**: Immutable sets (e.g., frozenset({1, 2, 3})).	
Data frames			
Definition	Data Frames are data displayed in a format as a table. Items can have different types.	Not defined in Python. Alternatives are lists of lists, lists of dictionaries or packages, such as [pandas]().	Not defined in Julia. Alternatives are the packages [DataFrames.jl]() and [Tables.jl]().

Create	data.frame(vect1, vect2)
	
	as.data.frame(matrix(ncol = 5, nrow = 10))
		#Using the pandas library
		import pandas as pd
		
		# Create a data frame from a dictionary
		data = {'Name': ['Alice', 'Bob', 'Charlie'],
		        'Age': [25, 30, 35],
		        'City': ['New York', 'London',
		'Paris']}
		
		df = pd.DataFrame(data)
			--

Select elements	Df1[line_number,column_number]	# Select the element at row label 0 and column label 'Name' df.loc[0, 'Name']	--
Number of columns	ncol()	# Get the number of columns using the shape attribute df.shape[1] # Alternatively, get the number of columns using the len() function on the columns attribute len(df.columns)	--

Number of rows	nrow()	# Get the number of rows using the shape attribute df.shape[0] # Alternatively, get the number of rows using the len() function on the index attribute len(df.index)	--
Dimensions	dim()	# Evaluate the dimensions of the data frame using the shape attribute num_rows, num_columns = df.shape print("Number of rows:", num_rows) print("Number of columns:", num_columns)	--

Add column	cbind()	import pandas as pd
		
		# Create a sample data frame
		data = {'Name': ['Alice', 'Bob', 'Charlie'],
		        'Age': [25, 30, 35],
		        'City': ['New York', 'London',
		'Paris']}
		
		df = pd.DataFrame(data)
		
		# Add a new column to the data frame
		df['Gender'] = ['Female', 'Male', 'Male']
			--

Remove column	Df1[,-2] remove second column	import pandas as pd
		
		# Create a sample data frame
		data = {'Name': ['Alice', 'Bob', 'Charlie'],
		        'Age': [25, 30, 35],
		        'City': ['New York', 'London',
		'Paris']}
		
		df = pd.DataFrame(data)
		
		# Remove the 'City' column from the data frame
		df = **df.drop**('City', axis=1)
			--

Add row	rbind()	import pandas as pd # Create a sample data frame	--

		data = {'Name': ['Alice', 'Bob'],
		        'Age': [25, 30],
		        'City': ['New York', 'London']}
		
		df = pd.DataFrame(data)
		
		# Create a new row as a dictionary
		new_row = {'Name': 'Charlie', 'Age': 35,
		'City': 'Paris'}
		
		# Append the new row to the data frame
		df = **df.append**(new_row, ignore_index=True)	

Remove row	Df1[-1,] remove first line	import pandas as pd
		
		# Create a sample data frame
		data = {'Name': ['Alice', 'Bob', 'Charlie'],
		        'Age': [25, 30, 35],
		        'City': ['New York', 'London',
		'Paris']}
		
		df = pd.DataFrame(data)
		
		# Remove the row at index 1
		df = **df.drop**(1)
			--

Matrices			
Definition	A matrix is a two-dimensional data set with columns and rows and items of the same type.	In Python, you can work with matrices using various libraries, with the most used one being NumPy. NumPy provides a powerful N-dimensional array object that can be used to represent matrices efficiently.	A two-dimensional array storing elements of the same type.

Create	M1 <- matrix(ncol = 5, nrow = 10)	import numpy as np
		
		# Create a matrix using a nested list
		
		M1 = np.array([[1, 2, 3],
		                [4, 5, 6],
		                [7, 8, 9]])
			M1 = [1 0 2; 0 1 1]

Select elements	Accessing the element at row 2, column 3
	M1[2,3]
		import numpy as np
		
		matrix = np.array([[1, 2, 3],
		                  [4, 5, 6],
		                  [7, 8, 9]])
		
		# Select the element at row index 1, column index 2
		element = matrix[1, 2]
			Accessing the element at row 2, column 3
			M1[2,3]

Number of columns	ncol()	matrix.shape[1]	The number of columns is the number of elements in the second dimension. size(M1, 2)
Number of rows	nrow()	matrix.shape[0]	The number of rows is the number of elements in the first dimension. size(M1, 1)
Dimension	dim()	matrix1.shape	size()

Add column		import numpy as np
		
		m1 = np.array([[1, 2, 3],
		              [4, 5, 6],
		              [7, 8, 9]])
		
		column = np.array([10, 11, 12])
		
		# Add the column to the matrix
		M2 = np.hstack((m1, column.reshape(-1, 1)))
		
		#The reshaping of the column using column.reshape(-1, 1) is necessary to ensure the dimensions match when concatenating the matrices.
			# Creating a matrix
			A = [1 2 3; 4 5 6; 7 8 9]
			
			# Creating a column vector
			col = [10, 11, 12]
			
			# Adding the column to the matrix
			B = **hcat**(A, col)

Remove column	# Removing the first column
	
	M1[,-1]
		np.delete(matrix, 1, axis=1)
		
		#The 1 in np.delete(matrix, 1, axis=1) specifies the index of the column to be removed, and axis=1 indicates that the operation is performed along the columns.
			# Creating a matrix
			A = [1 2 3; 4 5 6; 7 8 9]
			
			# Removing the second column
			B = A[:, [1, 3]]

Add row		import numpy as np
		
		matrix = np.array([[1, 2, 3],
		[4, 5, 6],
		[7, 8, 9]])
		
		row = np.array([10, 11, 12])
		
		# Add the row to the matrix
		new_matrix = np.vstack((matrix, row))
			# Creating a matrix
			A = [1 2 3; 4 5 6; 7 8 9]
			
			# Creating a row vector
			row = [10 11 12]
			
			# Adding the row to the matrix
			B = **vcat**(A, row)

Remove row	# Removing the first row
	
	M1[-1,]
		import numpy as np
		
		matrix = np.array([[1, 2, 3],
		[4, 5, 6],
		[7, 8, 9]])
		
		# Delete the second row
		new_matrix = np.delete(matrix, 1, axis=0)
			# Creating a matrix
			A = [1 2 3; 4 5 6; 7 8 9]
			
			# Removing the second row
			B = A[[1, 3], :]

Vectors			
Definition	A vector is simply a list of items that are of the same type.	Not defined in Python. Alternatives are lists or the NumPy library, which provides a powerful data structure for representing arrays and vectors in Python.	In Julia, vectors are a particular type of array, a 1-dimensional array. If the elements are of the same type, vectors are homogeneous. However, Julia also allows the creation of heterogeneous vectors.
Create	V1 <- c(1,2,3)	# Creating a vector using a list	#Homogeneous vector

		V1 = [1, 2, 3, 4, 5]
		
		# Using NumPy
		import numpy as np
		
		# Creating a vector using a NumPy array
		V2 = np.array([1, 2, 3, 4, 5])
			V1 = [1, 2, 3, 4, 5]
			
			#Heterogeneous vector
			V2 = [1, 2.0, "three"]

Length	length(V1)	--	length(V1)
Select elements	V1 <- c(1,2,3) #Selecting the second element V1[2]	--	V1 = [1, 2, 3, 4, 5] #Selecting the second element V1[2]

Concatenate	c(vect1, vect2)	--	**vcat()**
			Example
			# Creating vectors
			v1 = [1, 2, 3]
			v2 = [4, 5, 6]
			v3 = [7, 8, 9]
			
			v4 = vcat(v1, v2, v3)
			
			cat()
			Example
			# Creating vectors
			v1 = [1, 2, 3]
			v2 = [4, 5, 6]
			v3 = [7, 8, 9]
			
			v5 = cat(v1, v2, v3, dims=1)
			By using dims=1, we concatenate them horizontally along dimension 1 to create a single vector.

Remove elements	#Remove the third element V1[-3]	--	#Create a sample 1-dimensional array arr1 = ["John", "Paul", "Ringo", "George"] #Delete one element deleteat!(arr1, 2)
Lists			
Definition	A list can contain many different data types inside it. A list is a collection of data which is ordered and changeable.	Lists are used to store multiple items in a single variable.	In Julia, lists are known as arrays, and they provide a flexible and powerful way to store and manipulate collections of elements. Arrays in Julia can hold elements of any type, including numbers, strings, and even other arrays.
Create	lst1 <- list(TRUE, "hello", c(3,5,4)) lst2 <- vector(mode='list', length=10)	lst1 = [1, 2, 3, 4, 5]	--
Length	length(lst1)	len(my_list1)	--
Select elements	lst1[[element_position]]	lst2 = [1, 2, 3, 4, 5] second_element = lst2[1]	--
Add elements	lst2 <- list(1, 2, 3) lst3 <- c(lst2, 4)	lst3 = [1, 2, 3, 4, 5] lst3.append(6)	--
Remove elements	lst4 <- list(1, 2, 3, 4, 5) lst 5 <- lst4[-3]	lst3 = [1, 2, 3, 4, 5] Lst3.remove(3)	--
Join two	lst1 <- list(value1, value2) lst2 <- list(value3, value4) c(lst1, lst2)	lst4 = [1, 2, 3] lst5 = [4, 5, 6] lst6 = lst4 + lst5	--
Arrays			

Definition	Compared to matrices, arrays can have more than two dimensions.	Not defined in Python. Alternatives are lists or the NumPy library, which provides a powerful data structure for representing arrays and vectors in Python. Additionally, Python has built-in support for arrays through the array module. It provides an array object that can be used to store homogeneous data efficiently.	Vectors are 1-dimensional arrays, matrices are 2-dimensional arrays. Julia has arrays rather than vectors and matrices/data frames.

Creation	#Creating a 3-dimensional array
	arr2 <- array(data = 1:24, dim = c(3, 4, 2))
	
	# The dimensions refer to 3 rows, 4 columns and 2 matrices.
		#Using numpy
		import numpy as np
		
		# Creating a 3-dimensional array
		arr2 = np.array([[[1, 2], [3, 4], [5, 6]],
		[[7, 8], [9, 10], [11, 12]],
		[[13, 14], [15, 16], [17, 18]]])
			arr1 = [1, 2, 3, 4, 5]

Dimensions	dim()	import numpy as np
		
		# Creating a 3-dimensional array
		arr2 = np.array([[[1, 2], [3, 4], [5, 6]],
		[[7, 8], [9, 10], [11, 12]],
		[[13, 14], [15, 16], [17, 18]]])
		
		arr2.ndim
			size()

Element selection	# Creating a 3-dimensional array
	arr <- array(1:27, dim = c(3, 3, 3))
	
	# Selecting an element
	element <- arr[2, 3, 1]
		import numpy as np
		
		# Creating a 2-dimensional array
		arr = np.array([[1, 2, 3],
		[4, 5, 6],
		[7, 8, 9]])
		
		# Selecting an element
		element = arr[1, 2]
			# Creating a 3-dimensional array
			arr = reshape(1:27, (3, 3, 3))
			
			# Selecting an element
			element = arr[2, 3, 1]

Dictionaries			

Definition	Not defined in R. Alternatives are named lists.	Dictionaries store data values in key:value pairs. A dictionary is ordered, changeable and does not allow duplicates.	In Julia, dictionaries are a built-in data structure that allows you to store and retrieve key-value pairs. Dictionaries in Julia are called "Dict" and provide an efficient way to associate values with unique keys.

Creation	`my_list <- list(name1 = value1, name2 = value2, name3 = value3)`	`a = dict(one=1, two=2, three=3)`
		
		`b = {'one': 1, 'two': 2, 'three': 3}`
			`dt1 = Dict("key1" => 1, "key2" => 2, "key3" => 3)`

Select elements	--	`D1 = {`
		`  "name": "Michael",`
		`  "species": "dog",`
		`  "age": 5`
		`}`
		`D1["name"]`
			`dt1["key2"]`

Add elements	--	`D1 = {`
		`  "name": "Michael",`
		`  "species": "dog",`
		`  "age": 5`
		`}`
		`D1["colour"] = “brown”`
			`dt1["key4"] = 4`

Remove elements	--	`D1 = {`
		`  "name": "Michael",`
		`  "species": "dog",`
		`  "age": 5`
		`}`
		`thisdict.pop("species")`
			`pop!(dt1, "key2")`

Tuples			
Definition	Not defined in R.	Tuples are used to store multiple items in a single variable. A tuple	In Julia, tuples are a built-in data structure that allows you to store an ordered collection of

		is a collection which is ordered and unchangeable.	elements. Tuples are defined using parentheses () and are immutable, meaning their elements cannot be modified once created.
Creation	--	tp1 = ("a", "b", "c")	tp1 = (value1, value2, value3)
Select elements	--	tp1 = (value1, value2, value3) #Select second element tp1[1]	tp2 = (1, 3, 2) #Selecting the second element tp2[2]
Add elements	--	Not possible.	Not possible.
Remove elements	--	Not possible.	Not possible.
Join two	--	tl1 = ("a", "b", "c") tp2 = (1, 2, 3) tp3 = tp1 + tp2	tp1 = (1, 2) tp2 = (3, 4) concatenated_tuple = tuple(tp1..., tp2...) #or (tp1..., tp2...)
Methods	--	**count()** Returns the number of times a specified value occurs in a tuple. **index()** Searches the tuple for a specified value, returning its position.	--

Comparison/Relational operators

R	Python	Julia Programming
== equal to	**==** equal to	**==** equality
!= different from	**!=** different from	**!=** inequality
> greater than	**>** greater than	**<** less than
< smaller than	**<** smaller than	**<=** less than or equal to
>= greater or equal to	**>=** greater than or equal to	**>** greater than
<= smaller or equal to	**<=** smaller than or equal to	**>=** greater than or equal to

Assignment Operators

R	Python			Julia Programming
a **<-** 3			Same as…	a = 5
	=	x = 5	x = 5	
a **<<-** 3	**+=**	x += 3	x = x + 3	
	-=	x -= 3	x = x - 3	
a **=** 3	***=**	x *= 3	x = x * 3	
	/=	x /= 3	x = x / 3	
	%=	x %= 3	x = x % 3	
	//=	x //= 3	x = x // 3	
	****=**	x **= 3	x = x ** 3	
	&=	x &= 3	x = x & 3	
	|=	x |= 3	x = x | 3	
	^=	x ^= 3	x = x ^ 3	
	>>=	x >>= 3	x = x >> 3	
	<<=	x <<= 3	x = x << 3	

Identity Operators

R	Python	Julia Programming
== (Equality): Tests if two objects have the same values. **!=** (Inequality): Tests if two objects have different values.	**is** Returns True if both variables are the same object. **is not** Returns True if both variables are not the same object.	**===** (Identity Equality): Tests if two objects or values have the same identity. **!==** (Identity Inequality): Tests if two objects or values have different identities.

Membership Operators

R	Python	Julia Programming
%in% Find out if an element belongs to a vector.	**in** Returns True if a sequence with the specified value is present in the object. **not in** Returns True if a sequence with the specified value is not present in the object.	**in** Tests if an element is present in a collection. **!in** Tests if an element is not present in a collection.

Bitwise operators

<table>
<tr><th>R</th><th>Python</th><th>Julia Programming</th></tr>
<tr>
<td>--</td>
<td>& AND Sets each bit to 1 if both bits are 1 x & y
| OR Sets each bit to 1 if one of two bits is 1 x | y
^ XOR Sets each bit to 1 if only one of two bits is 1 x ^ y
~ NOT Inverts all the bits ~x
<< Zero fill left shift Shift left by pushing zeros in from the right and let the leftmost bits fall off x << 2
>> Signed right shift Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off x >> 2</td>
<td>~x bitwise not
x & y bitwise and
x | y bitwise or
x ⊻ y bitwise xor (exclusive or)
x ⊼ y bitwise nand (not and)
x ⊽ y bitwise nor (not or)
x >>> y logical shift right
x >> y arithmetic shift right
x << y logical/arithmetic shift left</td>
</tr>
</table>

Logical operators

<table>
<tr><th>R</th><th>Python</th><th>Julia Programming</th></tr>
<tr>
<td>& AND – Vectorized version. Compares two vectors returning a vector of TRUE and FALSE.

&& AND – Non-vectorized version. Compares the first value of each vector returning one logical value.
| OR – Vectorized version. Compares two vectors returning a vector of TRUE and FALSE.</td>
<td>and Returns True if both statements are true.

or Returns True if one of the statements is true.

not Reverse the result, returns False if the result is true.</td>
<td>!x negation
x && y short-circuiting and
x || y short-circuiting or</td>
</tr>
</table>

|| OR – Non-vectorized version. Compares the first value of each vector returning one logical value. **!** NOT - Returns a unique logical value or a vector of TRUE/FALSE. **xor** XOR - Returns the value TRUE if both entry values are different and returns FALSE if the values are equal.		

Arithmetic operators

R	Python	Julia Programming
+ Addition x + y **-** Subtraction x - y ***** Multiplication x * y **/** Division x / y **^** Exponent x ^ y **%%** Modulus (Remainder from division) x %% y **%/%** Integer Division x%/%y	**+** Addition x + y **-** Subtraction x - y ***** Multiplication x * y **/** Division x / y **%** Modulus x % y ****** Exponentiation x ** y **//** Floor division x // y	**+x** the identity operation **-x** maps values to their additive inverses **x + y** performs addition **x - y** performs subtraction **x * y** performs multiplication **x / y** performs division **x ÷ y** divide x / y, truncated to an integer **x \ y** equivalent to y / x **x ^ y** raises x to the y^{th} power x % y equivalent to rem(x,y)

Create functions

R	Python	Julia Programming

f <- function(x, y) {
  x + y
}
	def f(x, y):
	  print(x + y)
		function f(x,y)
		          x + y
		      end
		f (generic function with 1 method)
		
		or…

		f(x,y) = x + y f (generic function with 1 method)

Control Structures and Loops

R		Python	Julia Programming

for	for (i in 1:10)
	{
	  statement
	}
		for iterator_var in sequence:
		    statements(s)
		
		Example:
		n = 4
		for i in range(0, n):
		    print(i)
			for iterator in range
			    statements(s)
			end
			
			Example:
			for i in 1:10
			    println(i)
			end

while	while (condition)
	{
	  statement
	}
		while expression:
		    statement(s)
		
		Example:
		i = 1
		while i < 6:
		  print(i)
		  i += 1
			while condition
			    # Code block to be executed
			End
			
			Example:
			while i <= 3
			        println(i)
			        global i += 1
			    end

Repeat	repeat
	{
	   statement
	
	   if(condition)
	   {
	      break
	   }
	}
		--	--

If else	if (condition) {
	  # Code block executed when condition is true
	} else {
		if condition:
		    # Code block executed when condition is true
		else:
			if condition
			    # Code block executed when condition is true
			else

	# Code block executed when condition is false
	}
	
	Example:
	x <- 10
	
	if (x > 0) {
	  print("x is positive")
	} else if (x < 0) {
	  print("x is negative")
	} else {
	  print("x is zero")
	}
		# Code block executed when condition is false
		
		Example:
		x = 10
		
		if x > 0:
		    print("x is positive")
		else:
		    print("x is non-positive")
			# Code block executed when condition is false
			end
			
			Example:
			if x < y
			    println("x is less than y")
			elseif x > y
			    println("x is greater than y")
			else
			    println("x is equal to y")
			end

Useful links

R

https://www.w3schools.com/r/default.asp

Julia

https://docs.julialang.org/en/v1/

https://www.datacamp.com/cheat-sheet/julia-basics-cheat-sheet

https://cheatsheet.juliadocs.org/

https://julia.school/julia/

Python

https://www.w3schools.com/python/default.asp

https://docs.python.org/3/contents.html