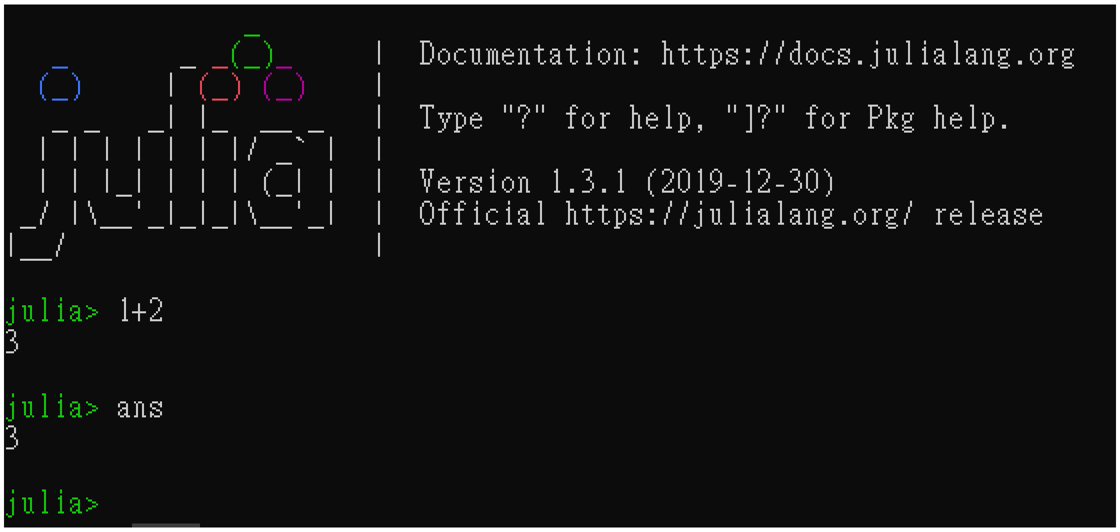
Get started on Julia Programming

The First interactive session



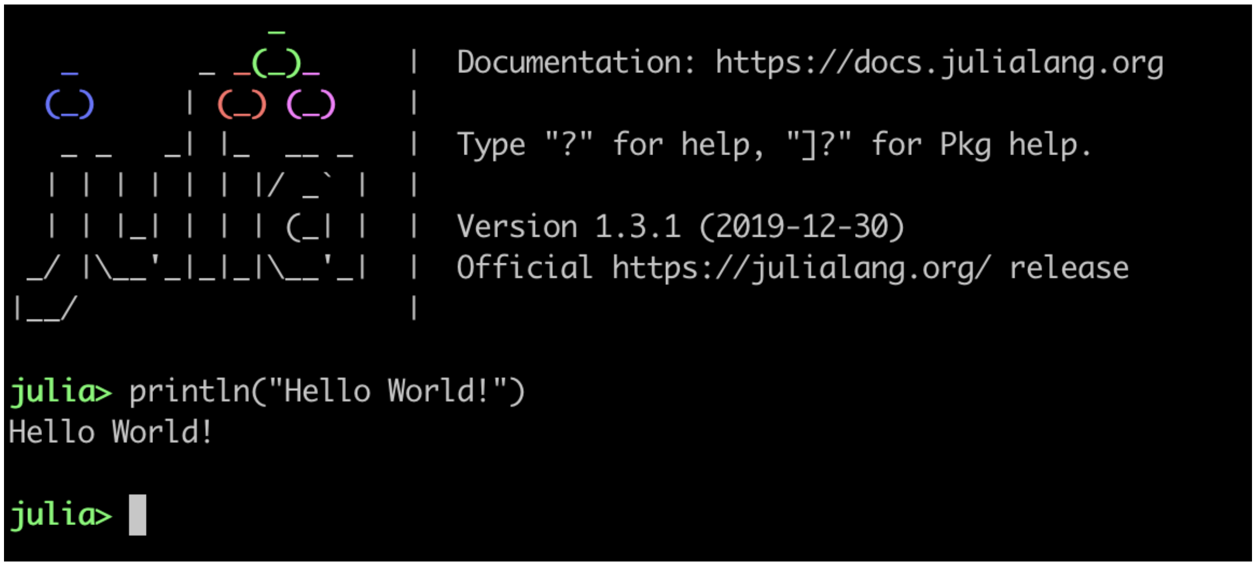
```
               _
   _       _ _(_)_     |  Documentation: https://docs.julialang.org
  (_)     | (_) (_)    |
   _ _   _| |_  __ _   |  Type "?" for help, "]?" for Pkg help.
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 1.3.1 (2019-12-30)
 _/ |\__'_|_|_|\__'_|  |  Official https://julialang.org/ release
|__/                   |

julia> 1+2
3

julia> ans
3

julia>
```

Steps:

1. Running julia from the command line (or double-clicking the Julia executable)
2. Enter an expression and press enter. In this example, the expression is "1+2"
3. Type “ans” and press enter

The interactive session will evaluate the expression and show its value. The variable “ans” is bound to the value of the last evaluated expression. Therefore, when you type the expression, it will display 3 (for this example).

“Hello world” program



```
               _
   _       _ _(_)_     |  Documentation: https://docs.julialang.org
  (_)     | (_) (_)    |
   _ _   _| |_  __ _   |  Type "?" for help, "]?" for Pkg help.
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 1.3.1 (2019-12-30)
 _/ |\__'_|_|_|\__'_|  |  Official https://julialang.org/ release
|__/                   |

julia> println("Hello World!")
Hello World!

julia>
```

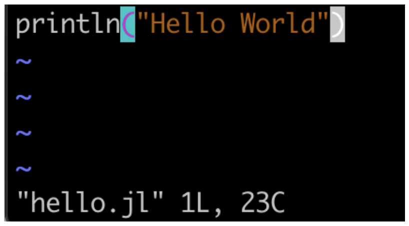
Steps:

1. Running julia from the command line (or double-clicking the Julia executable)
2. Type “println(“Hello World”)” and press enter

This task is an example of how to “print”(output) a string to your screen. println() is a built-in function of Julia, which is printing the input parameters to the screen. We will discuss more about function in Julia later.

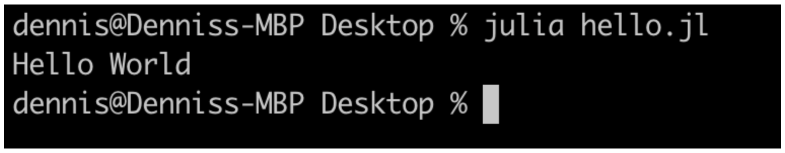
Run Julia from file

Besides running in Julia interactive mode, you can also create a ".jl” file and put all your code in this file. To run the Julia program, you can execute the program by entering the command:“julia <filename>.jl”. It is the way that we mostly used.



```
println("Hello World")
~
~
~
~
"hello.jl" 1L, 23C
```

hello.jl



```
dennis@Denniss-MBP Desktop % julia hello.jl
Hello World
dennis@Denniss-MBP Desktop %
```

Run hello.jl

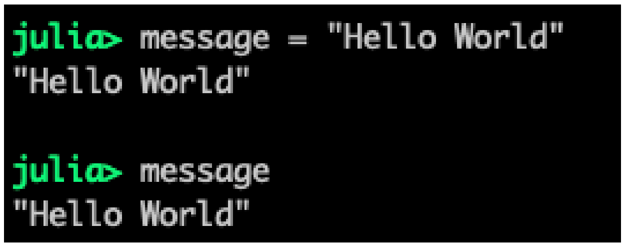
Steps:

1. Create a “hello.jl” file with your notepad (or any editor)
2. Open command line
3. Change to the directory that storing the “hello.jl” file
4. Execute the file by entering “julia hello.jl”

Basic Concept of Julia

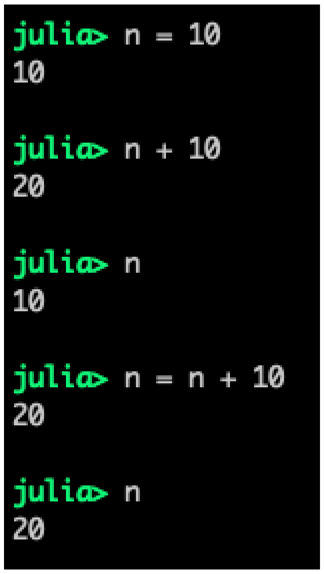
Variables, Expressions and Statements

In Julia, the definition of “Variables”, “Expressions” and “Statements” are:
Variable: Used to store a value for later use. It maps the identifier(name) to a value.
Expression: A combination of values, variables and operators.
Statement: An instruction forJulia to perform an action
For example:
Example 1



```
julia> message = "Hello World"
"Hello World"

julia> message
"Hello World"
```

Line 1 is an Assignment statement, which assign the value “Hello World” to variable “message”.
Line 2 is displaying the value of variable “message”.



```
julia> n = 10
10

julia> n + 10
20

julia> n
10

julia> n = n + 10
20

julia> n
20
```

Line 1 is assigning 10 to variable “n”
Line 2 is displaying the result of the expression “n + 10”. It does not change the value of n since the statement does not include reassignment of variable “n”. n still is 10.
Line 3 is displaying the value of n at that moment
Line 4 is evaluating the expression “n + 10”, and assign the result to the variable “n”
Line 5 is displaying the value of n. Since the value of n changed in Line 4, the current value of variable n is 20

Names of Variables

Julia supports using non-english characters(in UTF-8 encoding) as the variblear names(but it is not recommended).

The names must begin with a letter (A-Z or a-z), underscore, or a subset of Unicode code points greater than 00A0(other non-english characters). The following characters may also include ! and digits.

Some of the names are reserved words and not allowed to be used as the identifier(name) of a variable. Also, it is not suggested to redefine the value of built-in constants or functions. There are some suggestion for naming the variable:

- Names of variables are in lower case.
- Word separation can be indicated by underscores ('_'), but use of underscores is discouraged unless the name would be hard to read otherwise.
- Names of Types and Modules begin with a capital letter and word separation is shown with upper camel case instead of underscores.
- Names of functions and macros are in lower case, without underscores.

Examples of variable names

Valid	Invalid
abc	!abc
_abc	123abc
abc!123	try
δ	
안녕하세요 abc123!δ	

Operators in Julia

Arithmetic Operators

The following arithmetic operators are supported by Julia on all primitive numeric types:

Operator	Usage	Example	Output
+	Addition	1 + 1	2
-	Subtraction	10 - 1	9
*	Multiplication	10 * 2	20
/	Division	10 / 4	2.5
÷	Integer Division	10 ÷ 4	2
^	Exponentiation	2^3	8
\	Inverse Division (equivalent to y / x)	4 \ 10	2.5
%	Remainder	10 % 3	1

Bitwise Operators

The following bitwise operators are supported on all primitive integer types, which is considering the integer stores as a binary number and operating in bitwise level(bit by bit). It is rarely used in our course but you may encounter it when you are reading other libraries or other's work.

You may refer to the logic truth table of each operator if you are not familiar with these operators.

Operator	Usage	Example(In 8-bits integer)	Output
~	Bitwise not	~123 (In binary level, 123 = 01111011)	-124 (10000100, which is the two's complement representation of -124.)
&	Bitwise and	123 & 7 (In binary level, 123 = 01111011 7 = 00000111)	3 (Binary: 00000011)
|	Bitwise or	123 | 7	127 (Binary: 01111111)

⊻	Bitwise xor (Exclusive or)	123 ⊻ 7	124 (Binary: 01111100)
>>>	Logical shift right	-124 >>> 1 (Logical shift right 1 bit)	66 (Binary: 01000010)
>>	Arithmetic shift right	-124 >> 1 (Arithmetic shift right 1 bit)	-62 (Binary: 11000010)
<<	(Logical/Arithmetic) Shift Left	-124 << 1 (Shift left 1 bit)	8 (Binary: 00001000)

Updating operators

These operators combine the functions of calculation and assignment on a variable.

Operator(example)	Equivalent statement	Operator(example)	Equivalent statement
+= (x += 2)	x = x + 2	^= (x ^= 2)	x = x ^ 2
-= (x -= 2)	x = x - 2	&= (x &= 2)	x = x & 2
*= (x *= 2)	x = x * 2	|= (x |= 2)	x = x | 2
/= (x /= 2)	x = x / 2	⊻= (x ⊻= 2)	x = x ⊻ 2
\= (x \= 2)	x = x \ 2	>>>= (x >>>= 2)	x = x >>> 2
÷= (x ÷= 2)	x = x ÷ 2	>>= (x >>= 2)	x = x >> 2
%= (x %= 2)	x = x % 2	<<= (x <<= 2)	x = x << 2

Data Types in Julia

Data Types

The followings are the common data type in Julia:

Type(Example)	Example
Unsigned Integer (UInt8, UInt16...)	0,1,100
Signed Integer (Int8, Int16...)	-100.0,1,100
Float(Float16, Float32...)	1.0, 2.345
String	“Hello World”, “1”,”2.345”
Character(Char)	‘A’, ‘我’
Bool	true, false

If you are entering a very large number, DO NOT add commas. Otherwise Julia will parse it as a tuple. For example:

✓ 1000000000

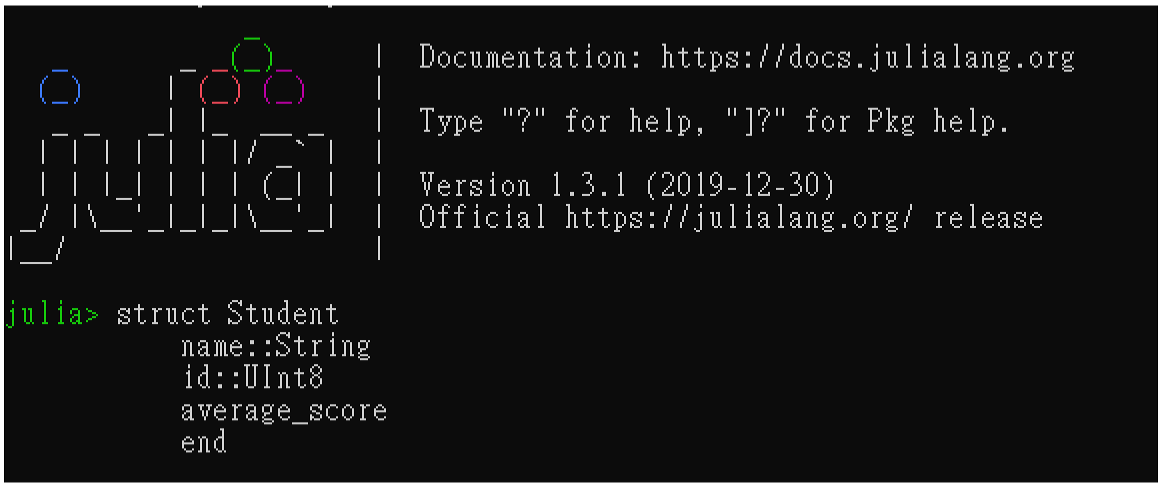
✗ 1,000,000,000

Since Julia is a dynamically typed language, the programmer does not need to oftenly handle the problem of data types. However, it is necessary to convey the data type to “fit in” to some function. You may use the convert(<Type>,<Variable>) function to convert data types.

Immutable and Mutable Composite Types

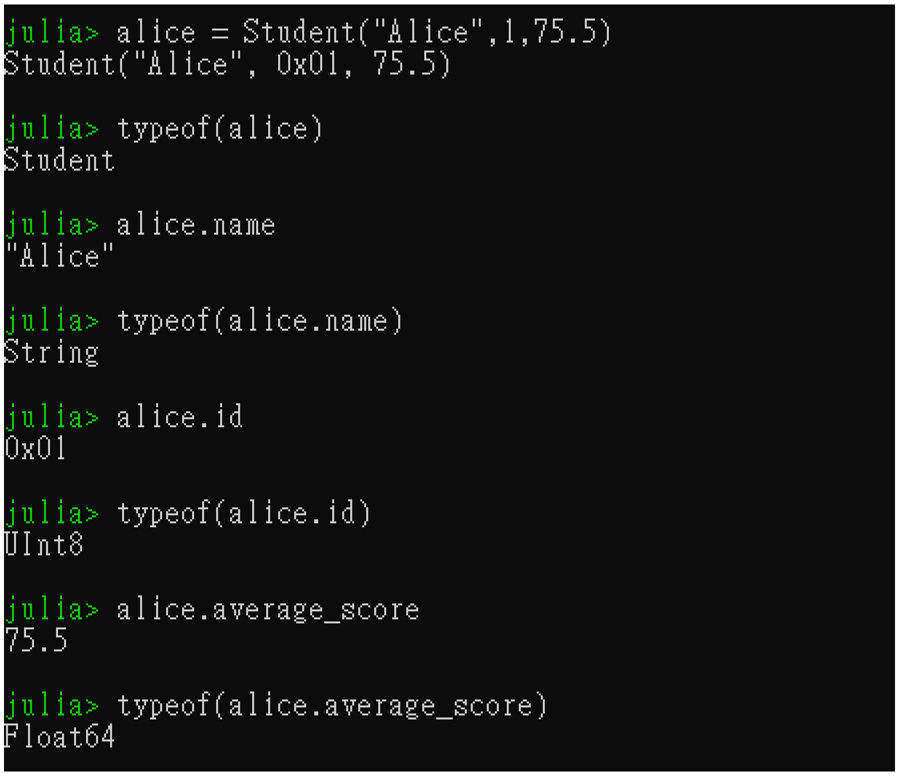
Composite type is a collection of variables, which means that it groups items of variant types into a single type. It is called records, structs, or objects in various languages. Composite type is one of the user-defined types in Julia.

A new composite type is defined by the keyword “struct”. If necessary, we may use :: operator to assign data type. The following is the example of the composite type:



```
               _
   _       _ _(_)_     |  Documentation: https://docs.julialang.org
  (_)     | (_) (_)    |
   _ _   _| |_  __ _   |  Type "?" for help, "]?" for Pkg help.
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 1.3.1 (2019-12-30)
 _/ |\__'_|_|_|\__'_|  |  Official https://julialang.org/ release
|__/                   |

julia> struct Student
           name::String
           id::UInt8
           average_score
           end
```

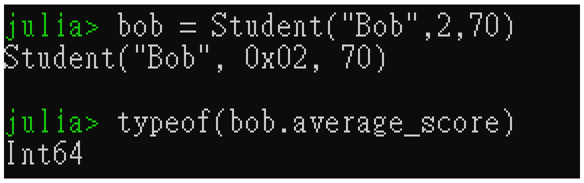
In this example, we defined a new composite type called Student. The Student type contains three variables: name(String), id(Unsigned 8-bits integer) and average_score. The type of average_score will be handled by Julia.



```
julia> alice = Student("Alice",1,75.5)
Student("Alice", 0x01, 75.5)

julia> typeof(alice)
Student

julia> alice.name
"Alice"

julia> typeof(alice.name)
String

julia> alice.id
0x01

julia> typeof(alice.id)
UInt8

julia> alice.average_score
75.5

julia> typeof(alice.average_score)
Float64
```

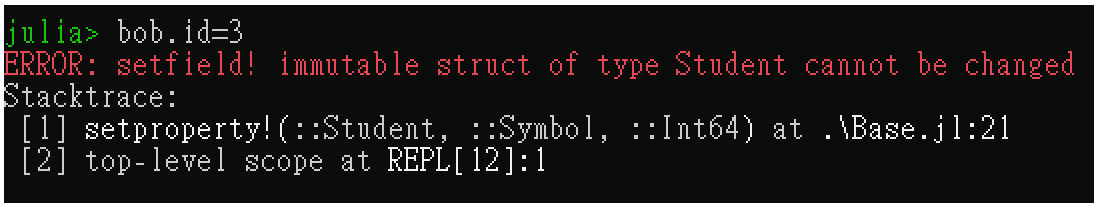
We also defined a new variable called alice and the type of alice is Student. The name of alice is “Alice”, her id is 1 and her average score is 75.5. Since we have not defined the type of average_score, Julia decided the type of average_score in alice is Float64.

To access the variable in the alice, we use the following notation: alice.[variable name].



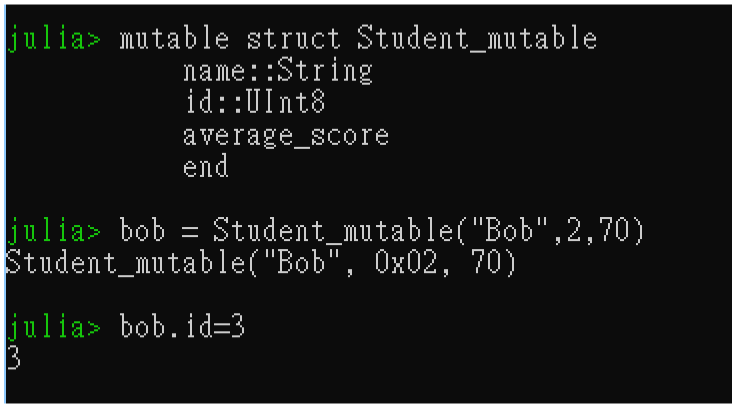
```
julia> bob = Student("Bob",2,70)
Student("Bob", 0x02, 70)

julia> typeof(bob.average_score)
Int64
```

P.s. The type of average_score is different for different students. For example, we have another Student called bob. The average_score is 70 and Julia decided the type of average_score in bob is Int64. Therefore it is necessary for us to handle the types.



```
julia> bob.id=3
ERROR: setfield! immutable struct of type Student cannot be changed
Stacktrace:
 [1] setproperty!(::Student, ::Symbol, ::Int64) at .\Base.jl:21
 [2] top-level scope at REPL[12]:1
```

It is not permitted to modify the value in the composite type defined by the keyword “struct” only. Instead, we need to add another keyword “mutable”. Here is the example

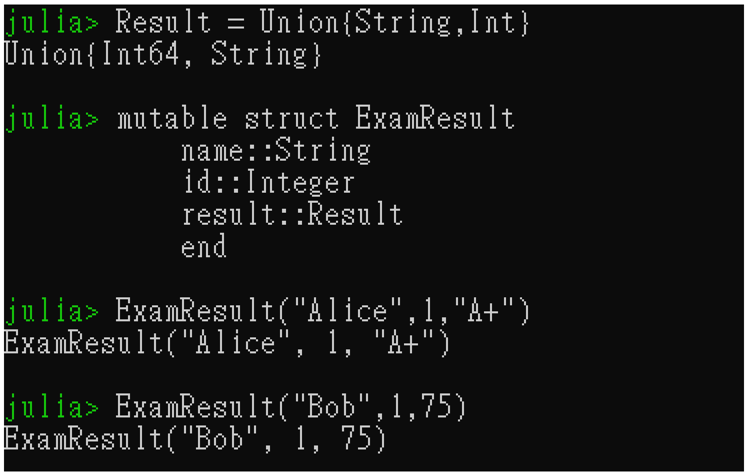


```
julia> mutable struct Student_mutable
           name::String
           id::UInt8
           average_score
           end

julia> bob = Student_mutable("Bob",2,70)
Student_mutable("Bob", 0x02, 70)

julia> bob.id=3
3
```

Type Unions

A type union is a special abstract type defined by the user. The user defines a new type union containing several types. The data contained in the variable defined as the new type must be either one of the data types in the union. To define a new union, we use the keyword Union.

For example:

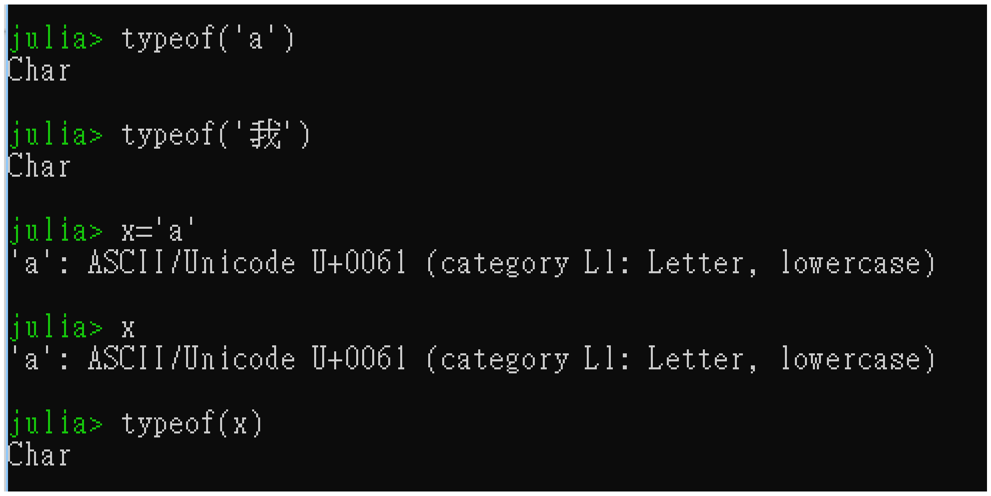


```
julia> Result = Union{String,Int}
Union{Int64, String}

julia> mutable struct ExamResult
           name::String
           id::Integer
           result::Result
           end

julia> ExamResult("Alice",1,"A+")
ExamResult("Alice", 1, "A+")

julia> ExamResult("Bob",1,75)
ExamResult("Bob", 1, 75)
```

In this example, we defined a new type union called Result and the union contains String and Int. Then, we create a mutable struct with 3 variables. One of the variables is the result with the type Result. As you can see, the variable could contain either string or integer.

Character & String in Julia

Character



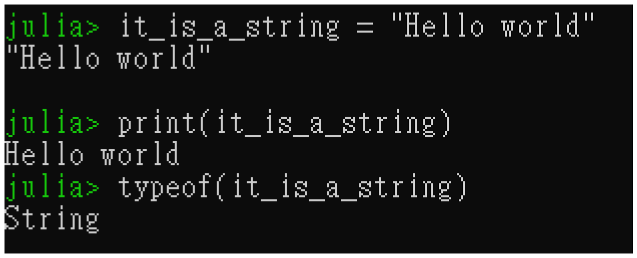
```
julia> typeof('a')
Char

julia> typeof('我')
Char

julia> x='a'
'a': ASCII/Unicode U+0061 (category Ll: Letter, lowercase)

julia> x
'a': ASCII/Unicode U+0061 (category Ll: Letter, lowercase)

julia> typeof(x)
Char
```

In Julia, the type Char is defined for storing a single character. Char is a 32-bit primitive type in Julia and Unicode characters are supported. In other languages, the Char type usually is 8 bits and usually only supports ASCII characters.

We will use a single quote to wrap the character such that Julia will know that we are talking about character. In the example, we defined a new variable x with initial value ‘a’. Since we are using a single quote, Julia will recognize it and decide the type of x is Char.

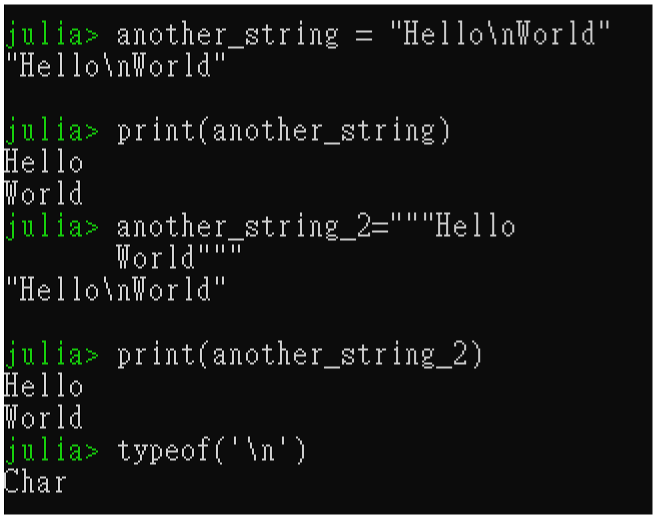
String

String is a sequence of characters. In Julia, string is wrapped by double quotes or triple double quotes.



```
julia> it_is_a_string = "Hello world"
"Hello world"

julia> print(it_is_a_string)
Hello world
julia> typeof(it_is_a_string)
String
```

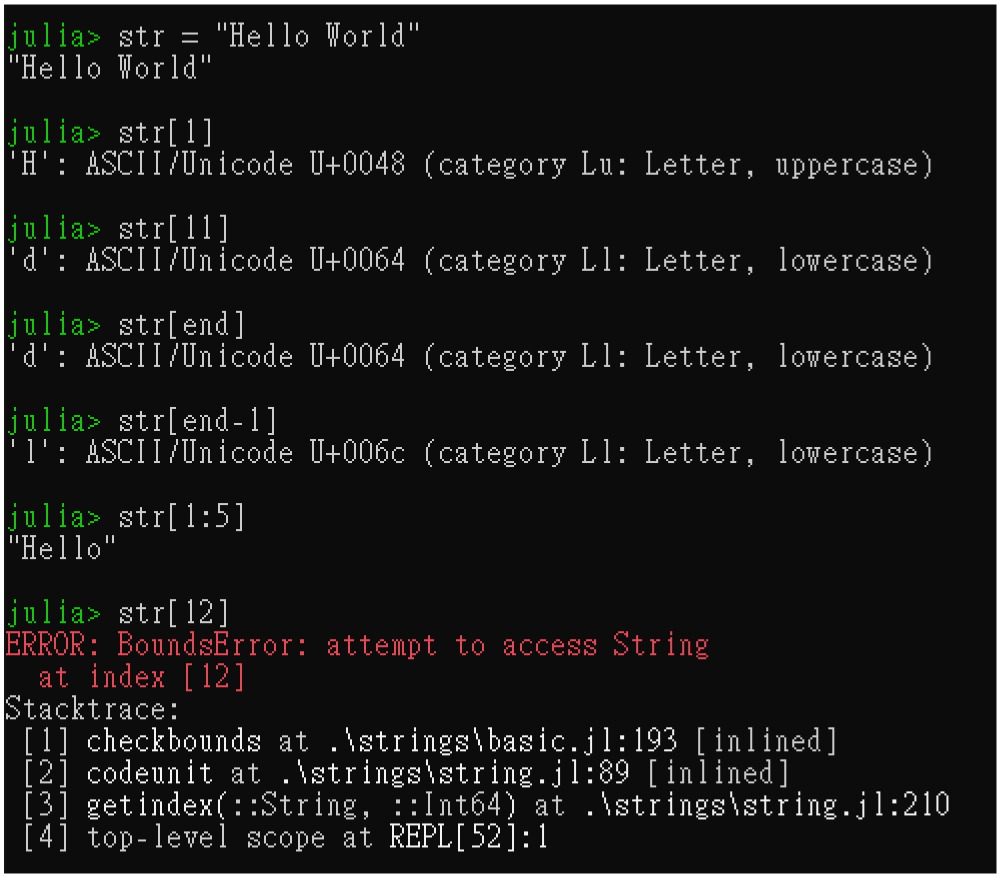
In this example, we defined a new variable it_is_a_string with initial value “Hello world”. When we use the print function to print out the string, it will be printed in one line.



```
julia> another_string = "Hello\nWorld"
"Hello\nWorld"

julia> print(another_string)
Hello
World
julia> another_string_2="""Hello
       World"""
"Hello\nWorld"

julia> print(another_string_2)
Hello
World
julia> typeof('\n')
Char
```

If we need to create a new line in the string, we need to use the '\n' character (newline character). Although it is the combination of backslash and character n, it is considered as one character. When we add the newline character in the string, it will splitted into two lines when we print the variable.

The alternative method to split the string to multiple lines is use triple quotation to define the string. Julia will automatically convert the formatted string into a normal string.



```
julia> str = "Hello World"
"Hello World"

julia> str[1]
'H': ASCII/Unicode U+0048 (category Lu: Letter, uppercase)

julia> str[11]
'd': ASCII/Unicode U+0064 (category Ll: Letter, lowercase)

julia> str[end]
'd': ASCII/Unicode U+0064 (category Ll: Letter, lowercase)

julia> str[end-1]
'l': ASCII/Unicode U+006c (category Ll: Letter, lowercase)

julia> str[1:5]
"Hello"

julia> str[12]
ERROR: BoundsError: attempt to access String
  at index [12]
Stacktrace:
 [1] checkbounds at .\strings\basic.jl:193 [inlined]
 [2] codeunit at .\strings\string.jl:89 [inlined]
 [3] getindex(::String, ::Int64) at .\strings\string.jl:210
 [4] top-level scope at REPL[52]:1
```

In this example, we demonstrated how to access the character(element) in a string. In Julia, the statement for access to a character is (name)[(position)]. The position is starting from 1.

For example, str[1] is accessing the first character in String str, which is ‘H’ in the example. We can also use str[end] to access the last character in str.

Expression in square brackets is allowed. For example, we can use str[end-1] to access the second last character in a string. If we are trying to access the character that does not exist(e.g. The 12th character in str), Julia will throw an error.

If we need to access the substring from a string, we may use the slice operator: statement (name)[(start positive):(end positive)]. For example, str[1:5] means that we are accessing the substring from character 1 to 5 in String str. In alternative, we may use the function SubString(str, 1, 5).

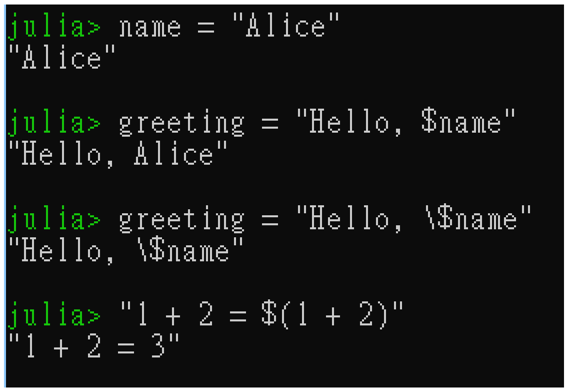
String Operators

The following operators are supported by Julia on string. The “^” operator has higher priority than “*” operator.

Operator	Usage	Example	Output
*	Concatenation	“Hello” * “World”	“HelloWorld”
^	Subtraction	“Hello”^3	“HelloHelloHello”

Note that ^ has higher priority than *. For example, evaluating “Hello” * ”World” ^ 3 will become “HelloWorldWorldWorld”. In alternative, you may use the function string() to concatenate multiple strings.

String Interpolation



```
julia> name = "Alice"
"Alice"

julia> greeting = "Hello, $name"
"Hello, Alice"

julia> greeting = "Hello, \$name"
"Hello, \$name"

julia> "1 + 2 = $(1 + 2)"
"1 + 2 = 3"
```

In Julia, it is allowed to create a new string based on the previous defined string or a expression. It is called String Interpolation.

For example, we created the variable(name) with the value `Alice”. If we would like to create a greeting string based on the current value of the name, we can just simply use $[the name of variable] in the string instead of using string operator to concatenate strings. Julia will automatically replace the “$name” in string to the current value of the variable.

Also, it is allowed to put an expression in a string. The expression must be wrapped by a pair of parentheses.

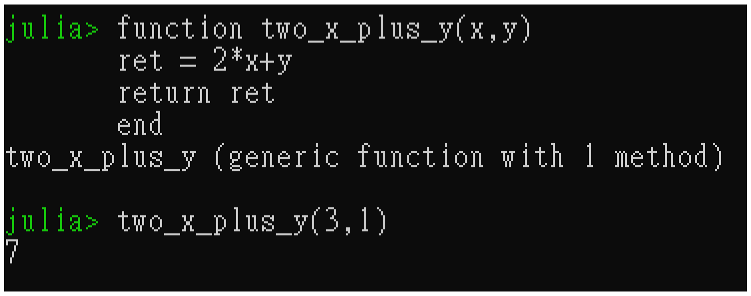
If we really need to use $ character in string, it is considered a special character and represented as ‘\$’. Similar to ‘\n’, it is considered as one character even though it is combined with two characters

Functions in Julia

Concept of function

As mentioned in the previous, a computer program can be considered as a IPO model. In most cases, the program is very large and complicated. Therefore, we introduce a new concept - function.

Function is defined as “named sequence of statements that performs a computation” which takes several inputs, do some calculations and returns(output) the value. In other words, it could be considered as the “sub-program” inside the program. In Julia, this is how we define a function:



```
julia> function two_x_plus_y(x,y)
       ret = 2*x+y
       return ret
       end
two_x_plus_y (generic function with 1 method)

julia> two_x_plus_y(3,1)
7
```

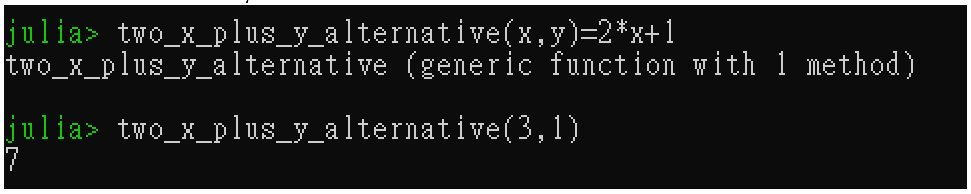
In this example, we defined the function to calculate 2x+y and used it to calculate 2*3+1

First, we use the keyword “function”, and we define the name of the function. There are 2 inputs in this example: x and y. The input(s) of the function have a special name: argument(s). Therefore, from the first line of the example, we defined a function called two_x_plus_y, with two arguments.

The second line defined a new variable (inside the function) called ret, which is calculating 2x+y and storing to the variable ret.

The third line is the return statement. It is the output of the function. In the example, the value of ret is returned(outputted). At the end, we need to add another keyword “end” to mention that we are finished.

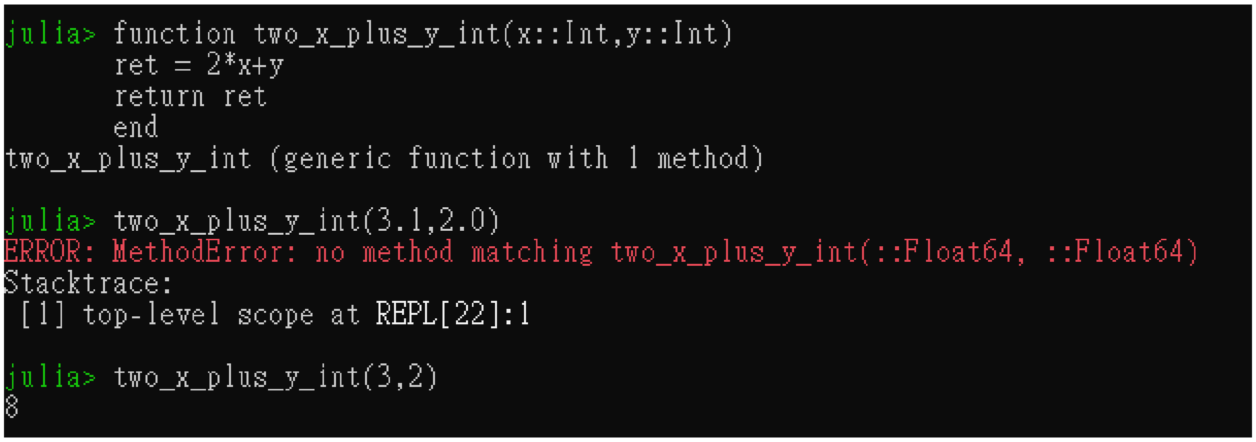
Then, we use the function to perform calculations. It is called “call the function”. We call the function two_x_plus_y with values 3 and 1. 3 and 1 are the “parameters” of the function. As the result, the function returned 7, which is the value of 2*3+1.



```
julia> two_x_plus_y_alternative(x,y)=2*x+1
two_x_plus_y_alternative (generic function with 1 method)

julia> two_x_plus_y_alternative(3,1)
7
```

There is another method to define a simple function. In this method, we just define the function like mathematical functions: [function name]([arguments]) = [expression]. This example is doing exactly the same with the previous example.

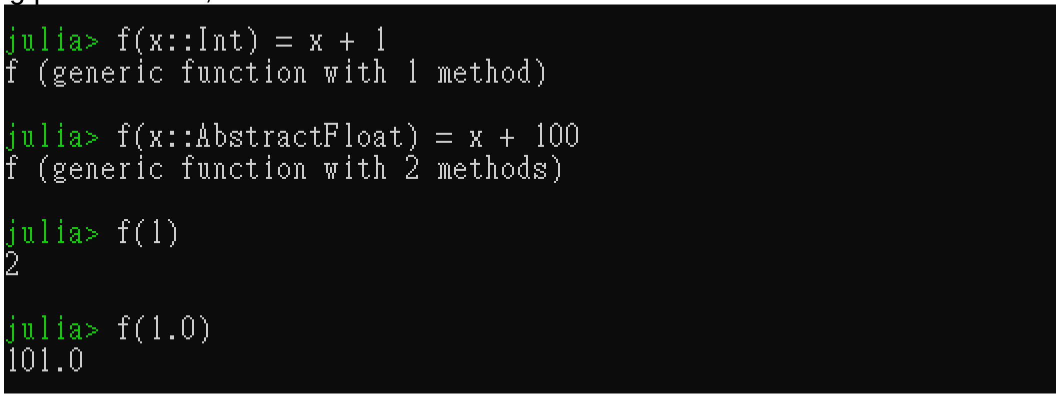
Function on specific type and function overloading



```
julia> function two_x_plus_y_int(x::Int,y::Int)
       ret = 2*x+y
       return ret
       end
two_x_plus_y_int (generic function with 1 method)

julia> two_x_plus_y_int(3.1,2.0)
ERROR: MethodError: no method matching two_x_plus_y_int(::Float64, ::Float64)
Stacktrace:
 [1] top-level scope at REPL[22]:1

julia> two_x_plus_y_int(3,2)
8
```

In previous examples, the type of input arguments is not restricted. In some cases, we would like the function only to work on specific types. We can use :: operator to restrict the type of argument.

In this example, we restricted the input type to Integer. Therefore, when we call the function with a floating point number, it will cause error.



```
julia> f(x::Int) = x + 1
f (generic function with 1 method)

julia> f(x::AbstractFloat) = x + 100
f (generic function with 2 methods)

julia> f(1)
2

julia> f(1.0)
101.0
```

Julia allows function overloading. The meaning of function overloading is: we can define two functions with the same name, but doing different things. It is based on the types of input arguments. In this example, we defined two functions with the same name f. If the input is integer, it will add 1. Else if the input is a floating point number, it will add 100.

Why we need function

There are many benefits for us to split a program into several functions.

One of the benefits is that we could reuse the code. For example, when you are doing the same complicated calculation on a number of variables, you just need to define a function and do not need to repeat the expression multiple times.
It is possible for a language to do some optimization based on the function. Since the number of repeated statements is reduced, the size of the program is reduced as well.

Second, function can improve the readability of a program. Humans takes time to understand an expression. Clear and organized code is helpful on debugging and tracing the logic. If there are many repeated complicated expressions in a single program, the program will become very long and messy. Using function could solve this problem.

Built-in Functions

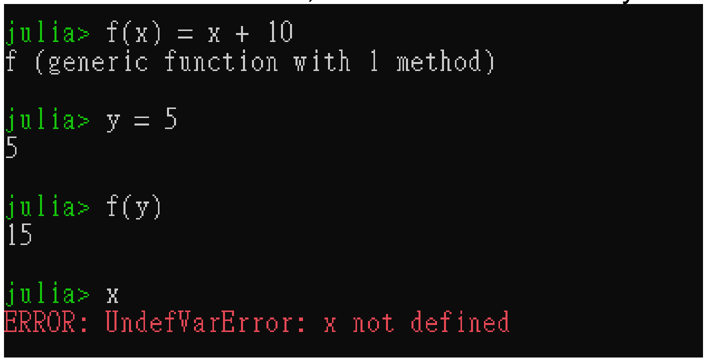
In Julia, there are several build-in functions which are provided by the developers of Julia. Although Julia allows the programmer to re-define the function, it is not suggested to do so.

The following is the list of common built-in functions in Julia. There are much more built-in functions in Julia.

Function(arguments)	Description
print(content)	Output the content to the screen
println(content)	Output the content to the screen, and add a new line character and the end of screen(\n)
typeof(content)	Check the type of the input parameter.
parse(type , string)	Convert the string to the specific number type, if possible
trunc(type , float)	Convert the floating point number to the specific integer. It will round down to the nearest integer, if possible.
float(integer)	Converts its argument to a floating point number
string(Number)	Converts its argument to a string

Scope: global vs local

Variable inside a function are “local variable”, which means that it only exists inside the function.

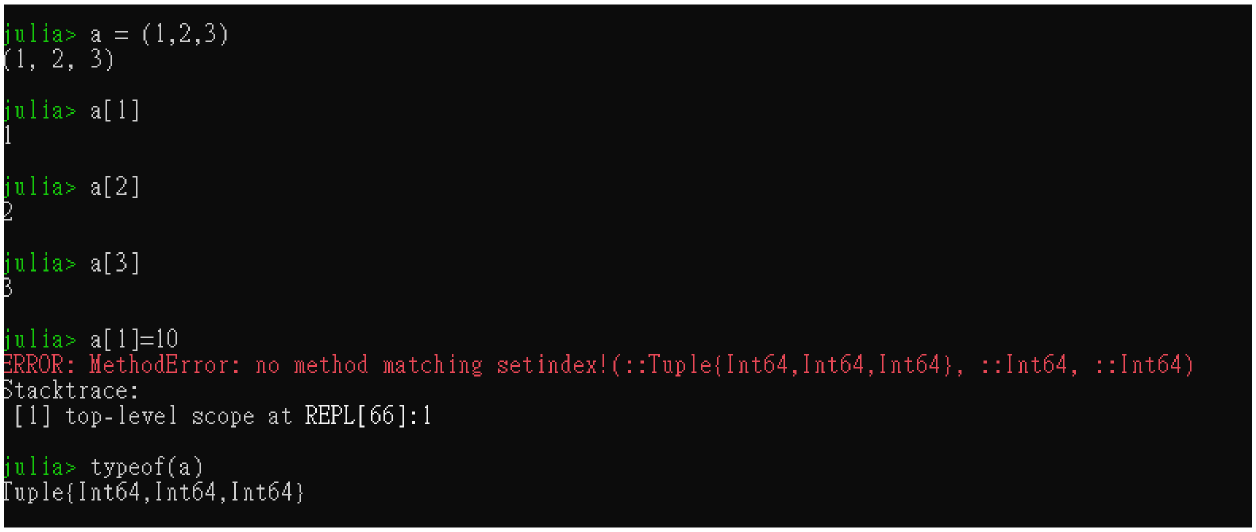


```
julia> f(x) = x + 10
f (generic function with 1 method)

julia> y = 5
5

julia> f(y)
15

julia> x
ERROR: UndefVarError: x not defined
```

In this example, we defined a function f and we used it to calculate y+10. Since x is in the function, we cannot access x. There is another alternative concept called global variable. However, it is not recommended to use.

Tuples, Arrays and Dictionaries in Julia

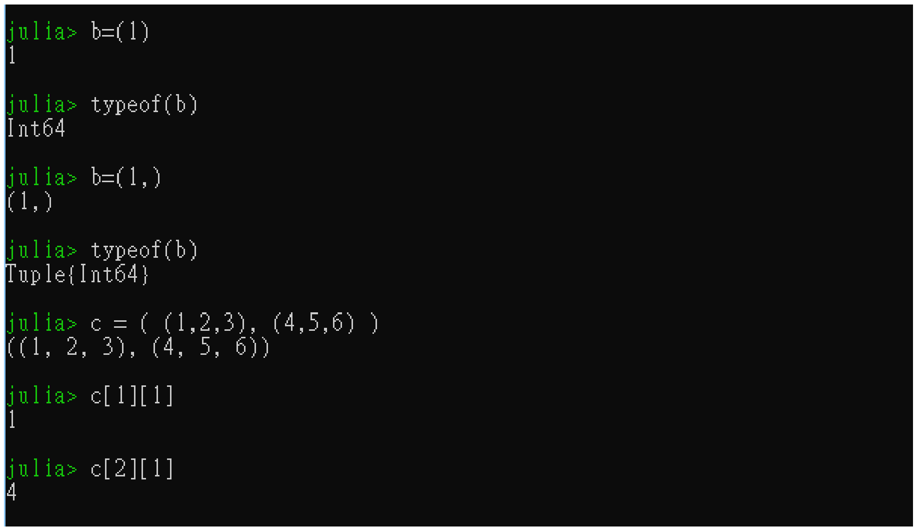
Tuples



```
julia> a = (1,2,3)
(1, 2, 3)

julia> a[1]
1

julia> a[2]
2

julia> a[3]
3

julia> a[1]=10
ERROR: MethodError: no method matching setindex!(::Tuple{Int64,Int64,Int64}, ::Int64, ::Int64)
Stacktrace:
 [1] top-level scope at REPL[66]:1

julia> typeof(a)
Tuple{Int64,Int64,Int64}
```

Tuples is a collection of values. The values(elements) can be of any type, and they are indexed by integers. Tuples are immutable and we cannot append /remove values to / from a tuple, nor change the value inside the tuple. To define a new tuple, we need to use a pair of parentheses. All the values in a tuple is splitted by comma.

In the example, a tuple with three elements (1, 2, 3) is created and assigned to the variable a. The method of accessing elements in a tuple is similar to string: using the name of variable, square brackets and position. If you are trying to change the value inside the turple, Julia will throw an error.

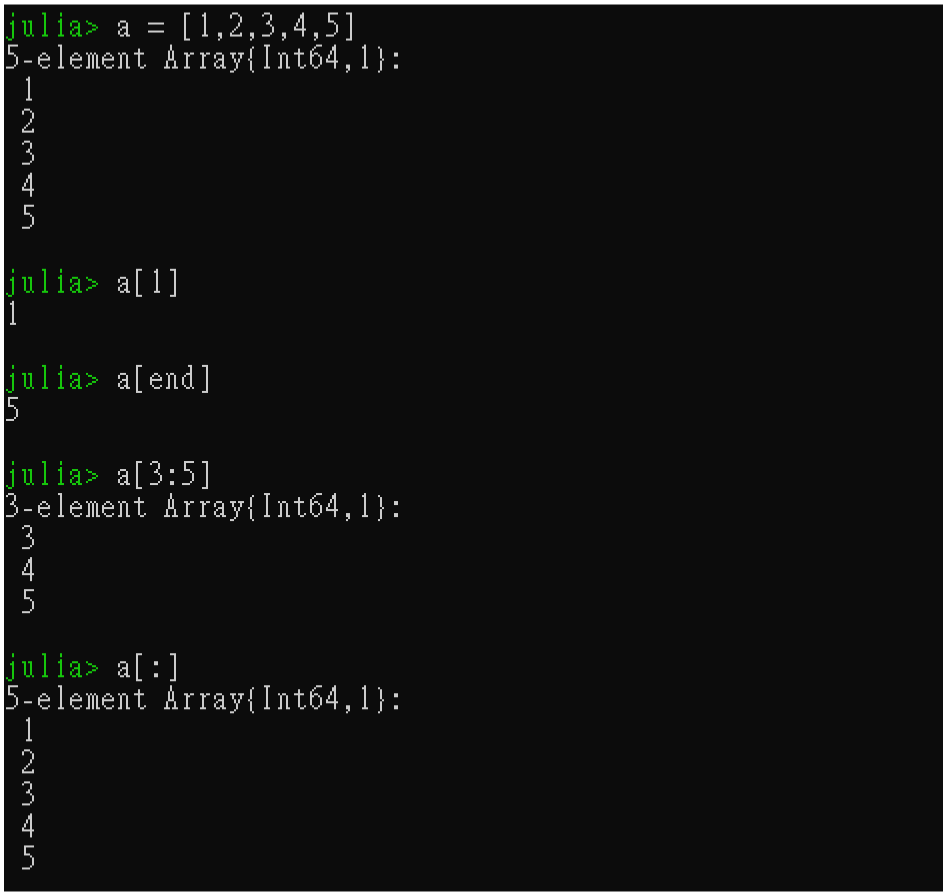


```
julia> b=(1)
1

julia> typeof(b)
Int64

julia> b=(1,)
(1,)

julia> typeof(b)
Tuple{Int64}

julia> c = ( (1,2,3), (4,5,6) )
((1, 2, 3), (4, 5, 6))

julia> c[1][1]
1

julia> c[2][1]
4
```

There is a special case for creating a tuple with only one element. We need to add a comma inside the parentheses to let Julia know that you are creating a tuple. If it is necessary, you may create a tuple, which contains other tuples.

Array

Array is the mutable version of Tuple. We can append/remove values to/from an array. To define a new tuple, we need to use a pair of square brackets.

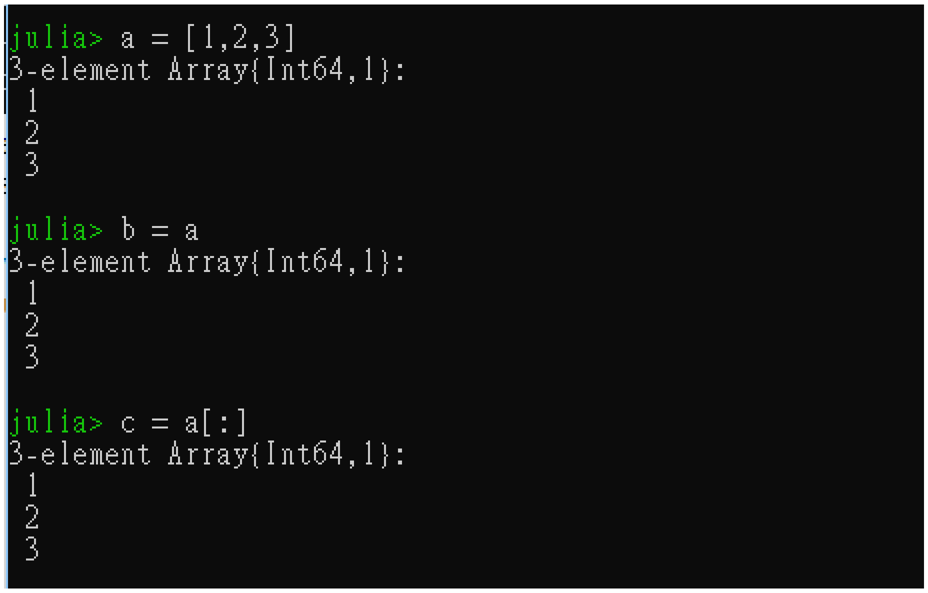


```
julia> a = [1,2,3,4,5]
5-element Array{Int64,1}:
 1
 2
 3
 4
 5

julia> a[1]
1

julia> a[end]
5

julia> a[3:5]
3-element Array{Int64,1}:
 3
 4
 5

julia> a[:]
5-element Array{Int64,1}:
 1
 2
 3
 4
 5
```

In the example, an array with three elements [1, 2, 3, 4, 5] is created and assigned to the variable a. The method of accessing elements is the same as accessing elements in tuple.

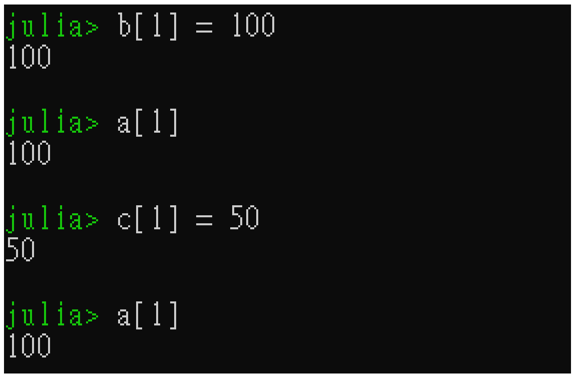
If we need to access the sub-array, it is also similar with how we access the sub-string. By using a slice operator, we can create a sub-array. It is the copy of the array.

Aliasing and Copying an array



```
julia> a = [1,2,3]
3-element Array{Int64,1}:
 1
 2
 3

julia> b = a
3-element Array{Int64,1}:
 1
 2
 3

julia> c = a[:]
3-element Array{Int64,1}:
 1
 2
 3
```

In this example, we initiated an array. Then we used two different methods to create two new arrays based on array a. What is the difference between them?

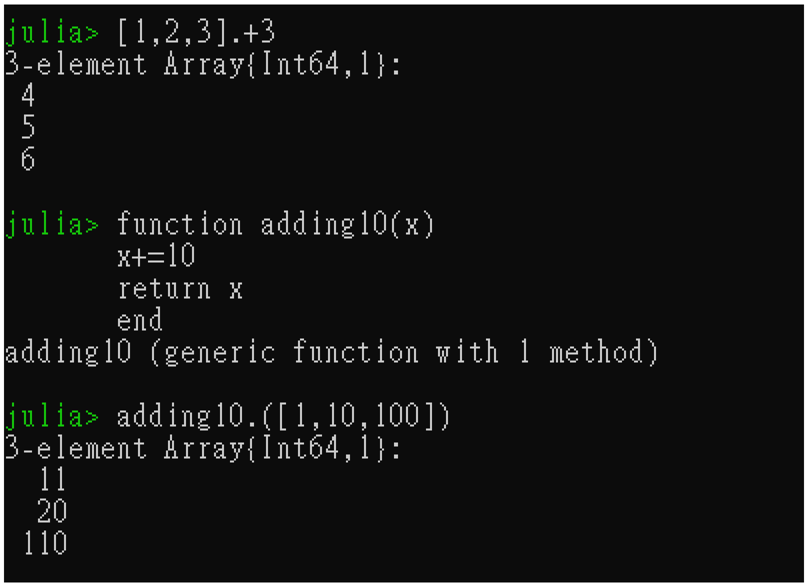


```
julia> b[1] = 100
100

julia> a[1]
100

julia> c[1] = 50
50

julia> a[1]
100
```

The first one is “aliasing” the array a, which means that a and b are pointing to the same array. If we changed the value of elements in array b, the value in array a will be changed as well. The second method is “copying” the array, which means that array a and c are independent.

Dot Syntax

When we are doing mathematical calculation on array, there is a simply shortcut to apply the calculation on all elements: Dot Syntax.For example, when we have the expression: [1, 2, 3] .^ 3, it is equivalent to [1^3, 2^3, 3^3]. The Dot Syntax could apply on function also.



```
julia> [1,2,3].+3
3-element Array{Int64,1}:
 4
 5
 6

julia> function adding10(x)
       x+=10
       return x
       end
adding10 (generic function with 1 method)

julia> adding10.([1,10,100])
3-element Array{Int64,1}:
  11
  20
 110
```

In this example, we used the dot syntax for adding 3 to the array [1,2,3]. Also, we applied the dot syntax on a self-defined function which is used to add 10 to an integer. The function originally only works on integer.

Deleting and Inserting Elements

In array, we can remove, add or change the elements. The following are the method of modifying the array:

Function	Description
splice!(array,position)	Delete the specific elements and return it from an array
pop!(array)	Deletes and returns the last element from an array
popfirst!(array)	Deletes and returns the first element from an array
push!(array, value)	Insert a element at the end of array
pushfirst!(array, value)	Insert a element at the beginning of array
deleteat(array,position)	Delete the specific elements from an array
insert!(array,position , value)	Insert a element at the specific position of the array

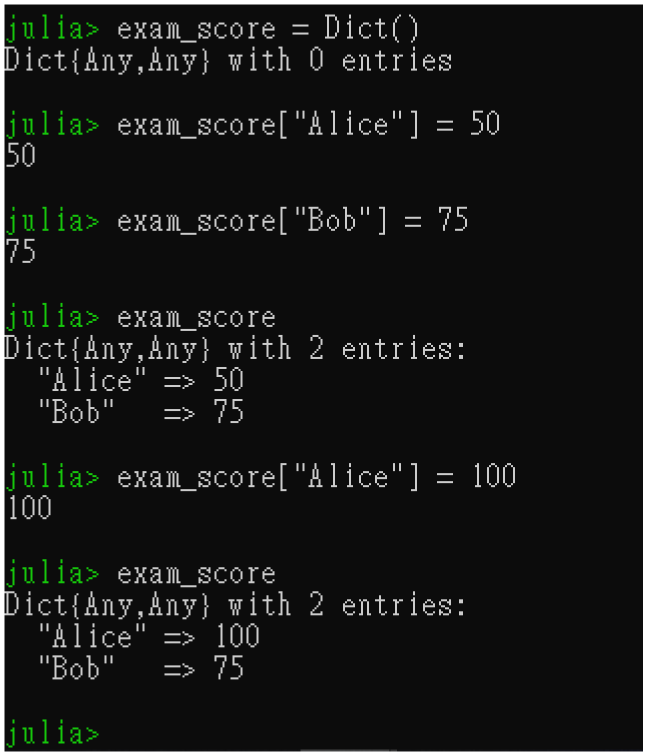
Dictionaries

There is another built-in type called a dictionary in Julia. It is similar to array: both of them are storing several values and mutable. In array, we need the position to access the element, which must be an integer and continuous (i.e. start from 1 and would not skip any number).

Instead of using the continuous position(integer) to access the elements in array, we use the key to access the elements in the dictionary. Each key is associated with a single value. The key could be either number(both integer and floating point), character or string.

We can consider a dictionary to be a key-value mapping. It maps a value to a key.

The dictionary is initialized by the built-in function Dict(). To add a new element in the dictionary, we can simply assign the value, just simple like how we access the value in the dictionary.

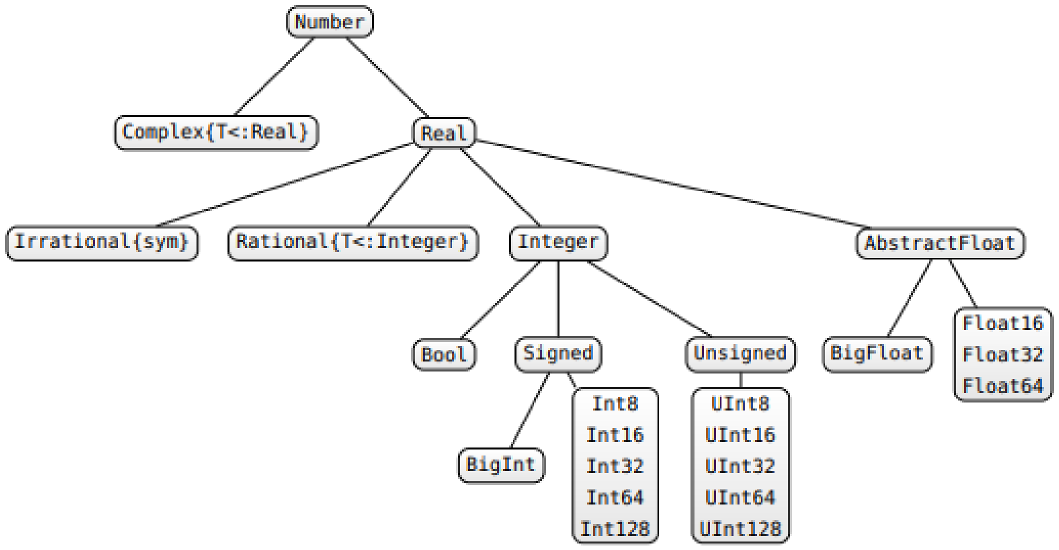


```
julia> exam_score = Dict()
Dict{Any,Any} with 0 entries

julia> exam_score["Alice"] = 50
50

julia> exam_score["Bob"] = 75
75

julia> exam_score
Dict{Any,Any} with 2 entries:
  "Alice" => 50
  "Bob"   => 75

julia> exam_score["Alice"] = 100
100

julia> exam_score
Dict{Any,Any} with 2 entries:
  "Alice" => 100
  "Bob"   => 75

julia>
```

In this example, we initialized a new dictionary called exam_score. We added the score of two students Alice and Bob by exam_score[(name of student)] = value, and the method of modifying the element is the same.

Generic programming

Abstract types

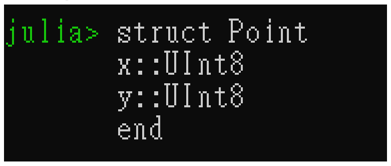


Hierarchy of primitive numeric types in Julia, Source: http://bogumilkaminski.pl/files/julia_express.pdf

Abstract types are the types that cannot be declared as a new variable. It is used to shows the conceptual relationship between types. The figure shows the hierarchy of primitive numeric types in Julia. For example, you cannot declare a new variable with type “Integer”. Instead, you can declare a new variable with type “Int8” or “UInt8”.

Generic Programming

Generic programming is a style of computer programming. In the previous examples, we define a struct with primitive types. For example:



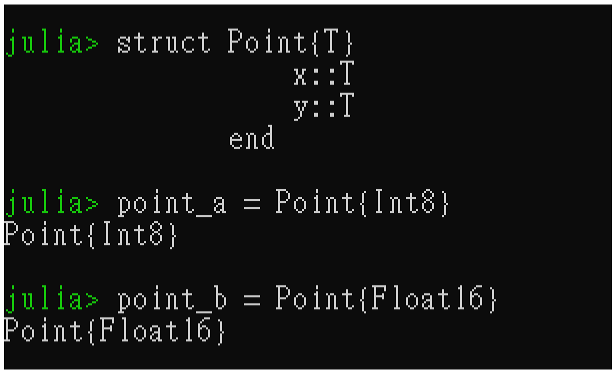
```
julia> struct Point
       x::UInt8
       y::UInt8
       end
```

Here is a typical example of the struct. In this example, a Point struct is defined with two variables x and y with UInt8 type. If we need to define point struct with different types of variables, we need to define many struct with unique names.

If we have generic programming, we can define that the x and y are the same type. Then, we define the type of x and y when we are creating the struct variable. Noted that generic programming does not only apply on struct only. It is a concept that could be adopted and applied to other concepts such as objects and functions.

Parametric Composite Types

In Julia, Parametric Composite Types is one of the types that uses the generic programming concept. Rather that defining the type of variables in composite types, we use capital letters to declare the composite types, and decide the type of variables when we create the variable



```
julia> struct Point{T}
                  x::T
                  y::T
              end

julia> point_a = Point{Int8}
Point{Int8}

julia> point_b = Point{Float16}
Point{Float16}
```

It is an example of parametric composite types. When we defined the struct, we did not assign the type of x and y. When we create the point_a and point_b, we declare they are the type of x and y is Int8 and Float16.