A Quick Introduction to `Groovy` for `Nextflow` workflows

prosaddas888@gmail.com
Systems and Chemical Biology
CSIR-Institute of Genomics and Integrative Biology (IGIB), New Delhi, India

July 14, 2024

- Printing in groovy can be done with or without the parentheses.

```
println("Hello, World!"); print "Hello, World!" // These two are equivalent
```

- Comments are C/C++-styled:

```
// comment a single line
/* a comment spanning multiple lines */
```

- Data types. Groovy, being a dynamically-typed language, allows you to work with various data types without having to explicitly declare them. However, it still supports static typing where you can explicitly declare the types. Data types are of five types: **primitive**, **object**, **collections**, **special types**, and **groovy-specific types**. Local variables are always declared with `def` keyword.
  Primitive types:
  - **byte**: 8-bit signed integer `byte b = 1`
  - **short**: 16-bit signed integer `short s = 10`
  - **int**: 32-bit signed integer `int i = 100`
  - **long**: 64-bit signed integer `long l = 1000L`
  - **float**: 32-bit signed floating-point number `float f = 1.23f`
  - **double**: 64-bit signed floating-point number `double d = 1.23456789`
  - **char**: 16-bit Unicode character `char c = "A"`
  - **boolean**: true or false `boolean b = true`

  Object types:
  - **String**: sequence of characters `String str = "Hello, Groovy"`
  - **GString**: Groovy-specfic string with embedded expressions
    ```
    def name = "World"
    def greeting = "Hello, ${name}$" // Gstring
    ```
  - **Wrapper Classes**: `Integer intWrapper = 100`

  Collections:
  - **Lists**: `def list = [1, 2, 3, 4]`
  - **Maps**: `def map [name: "John", age: 30]`
  - **Sets**: `def set = [1, 2, 3, 4] as set`

  Special Types:
  - **Ranges**: `def range = 1..5 // [1, 2, 3, 4, 5]`
  - **Closures**: Anonymous blocks of code that can be assigned to variables, passed as parameters or executed. `def closure = {println "Hello, Closure"} // Output: Hello, Closure`

- Type casting in Groovy can be done in several ways, similar to Java but with additional syntactic sugar and flexibility. Here are some of the methods to perform type casting in Groovy.

```
def num = "123"
def intNum = num as Integer
println(intNum) // Output: 123
```

Using explicit type casting similar to Java:

```
def num = "123"
def intNum = (Integer) num
println intNum // Output: 123
```

Using methods for type conversion:

```
def num = "123"
def intNum = num.toInteger()
println intNum // Output: 123
def num_ = 123
def strNum = num_.toString()
println strNum // Output: "123"
```

String to Integer:

```
def str = "456"; def intNum = str as Integer; println intNum // Output: 456
```

String to Double:

```
def str = "456.78"; def doubleNum = str as Double; println doubleNum // Output: 456.78
```

Integer to String

```
def num = 789; def str = num as String; println str // Output: "789"
```

List to Array

```
def list = [1, 2, 3]
def array = list as int[]
println array // Output: [I@6d6f6e28 (array's memory address)]
println array.toList() // Output; [1, 2, 3]
```

- A list object can be defined by placing the list items in square brackets:

```
list = [10, 20, 30, 40]

// Get values by index
println list[0]

// Get values by get method
println list.get(0)

// length of list
println list.size()

/* Check values by assert method, prints nothing if
a condition is true else raise an AssertionError message */

list = [10, 20, 30, 40]
```

```
assert list[0] == 10

// Neagtive indexing can also be used
list = [0,1,2]
assert list[-1] == 2
assert list[-1..0] == list.reverse()

// Appending an item to a list
list = [1,2,3]
list1 = list << 4 //or,
list2 = list + 4
assert list1 == list2

// to be checked
assert [1, 2, 3] << 1 == [1, 2, 3, 1]
assert [1, 2, 3] + [1] == [1, 2, 3, 1]
assert [1, 2, 3, 1] - [1] == [2, 3]
assert [1, 2, 3] * 2 == [1, 2, 3, 1, 2, 3]
assert [1, [2, 3]].flatten() == [1, 2, 3]
assert [1, 2, 3].reverse() == [3, 2, 1]
assert [1, 2, 3].collect { it + 3 } == [4, 5, 6]
assert [1, 2, 3, 1].unique().size() == 3
assert [1, 2, 3, 1].count(1) == 2
assert [1, 2, 3, 4].min() == 1
assert [1, 2, 3, 4].max() == 4
assert [1, 2, 3, 4].sum() == 10
assert [4, 2, 1, 3].sort() == [1, 2, 3, 4]
assert [4, 2, 1, 3].find { it % 2 == 0 } == 4
assert [4, 2, 1, 3].findAll { it % 2 == 0 } == [4, 2]
```

- Maps. Maps are like lists that have an arbitrary key instead of an integer. Therefore, the syntax is very much aligned.

```
map = [a: 0, b: 1, c: 2]
assert map["a"] == 0
assert map.b == 1
assert map.get("c") == 2
// Modify items
map["a"] = "x"
map.b = "y"
map.put("c", "z")
assert map == [a: "x", b: "y", c: "z"]
```

- String interpolation

```
foxtype = "quick"
foxcolor = ["b", "r", "o", "w", "n"]
println "The $foxtype ${foxcolor.join()} fox"

x = "Hello"
y = "World"
println "$x $y"
```

String literals can also be defined using the / character as a delimeter. They are known as **slashy** string

and are useful for defining regular expressions and patterns, as there is no need to escape backslashes. As with double-quote strings they allow to interpolate variables prefixed with a $ character.

```
x = /tic\tac\toe/
y = 'tic\tac\toe'
z = "tic\tac\toe"
```

Multiline strings can be defined by delimiting it with triple or single or double quotes

```
text = """
    Hello there!
    How are you today?
    """
```

The same can also acheived by using / as delimeter:

```
text = /
    Hello there!
    Isn't is amazing?
    /
```

- If statement:

```
if (< boolean expression >) {
    // truth branch
}
else {
    //false branch
}
```

The `else` branch is optional. Also, the curly brackets are optional when the branch defines a single statement.

```
x = 1
if (x > 10)
    println "Hello"
```

`null`, empty strings, and empty collections are evaluated to `false`. Therefore statements like:

```
list = [1, 2, 3]
if (list != null && list.size() > 0) {
    println list
}
else {
    println "The list is empty"
}
```

Can be written as:

```
list = [1, 2, 3]
if (list)
    println list
else
    println "The list is empty"
```

Tip In some cases it can be useful to replace to the `if` statement with a ternary expression (aka a conditional expression):

```
println list ? list: "The list is empty"
```

The previous statement can be further simplified using the **Elvis operator**

```
println list ?: "The list is empty"
```

- For statement

```
for (int i = 0; i < 3; i++) {
    println("Hello World $i")
}
```

Iteration over list objects is also possible using the syntax below:

```
list = ["a", "b", "c"]
for (String elem:list) {
    println elm
}
```

- Functions

```
def fib(int n) {
    return n < 2 ? 1 : fib(n - 1) + fib(n - 2)
}
// Call
assert fib(10) == 89
```

A function can take multiple arguments separating them with a comma. The `return` keyword can be omitted and function implicitly returns the value of the last evaluated expression. Also, explicit types can be omitted, though not recommended:

```
def fact(n) {
    n > 1 ? n * fact(n - 1) : 1
}
// Call
assert fact(5) == 120
```

- Closures, they are the Swiss army knife of Nextflow/Groovy programming. In a nutshell, a closure is a block of code that can be passed as an argument to a function. A closure can also be used to define an anonymous function. More formally, a closure allows the definition of functions as first-class objects.

```
square = { it * it }
```

The curly braces around the expression `it * it` tells the script interpreter to treat this expression as code. The `it` identifier is an implicit variable that represents the value that is passed to the function when it is invoked. Once compiled, the function object is assigned to the variable `square` as any other variable assignment shown previously. To invoke the closure execution use the special method `call` or just the parentheses to specify the closure parameter(s).

```
assert square.call(5) == 25
assert square(9) == 81
```

The `square` function can be passed to other functions or methods. Some built-in functions take a function like this as an argument. One example is the `collect` method on lists:

```
x = [1, 2, 3, 4].collect(square)
println x // Output: [1, 4, 9, 16]
```

By default, closures take a single parameter called `it`. To give it a different name use the `->` syntax.

```
square = { num -> num * num}
```

It is also possible to define closures with multiple, custom-named parameters. For example, when the method `each()` is applied to a map it can take a closure with two arguments, to which it passes the *key-value* pair for each entry in the `map` object. For example:

```
printMap = {a, b -> println "$a with value $b"}
values = ["Yue": "Wu", "Mark": "Williams", "John": "Doe"]
values.each(printMap)

/* Output
Yue with value Wu
Mark with value Williams
John with value Doe
*/
```

A closure has two other important features. First, it can access and modify variables in the scope where it is defined. Second, a closure can be defined in an *anonymous* manner, meaning that it is not given a name, and is only defined in the place where it needs to be used. For example:

```
result = 0
values = ["China": 1, "India": 2, "USA": 3]
values.keySet().each { result += values[it] } // Output: 6
```