Compilation and More

Compiling a program can be done with `nim c`. Compiling and then running a program is possible using the `-r` flag, i. e. `nim c -r filename.nim`.

Variables

Nim is statically typed, hence the type of a declaration must be provided when defining a variable (much like Java and unlike Ruby). A variable can be defined as follows: —

```
# If value is unknown
var <name>: <type>

# If value is known
var <name>: <type> = <value>
```

Nim can, however, infer the type of a declaration, so that the following line of code is also valid: —

```
var <name> = <value>
```

It is, however, strongly recommended to always define the type of a declaration. The following are a handful of examples of variable declaration: —

```
var a: int
var b = 7
var a = 10
```

Nim developers typically use camel case for variable names; it is, however, rather important to note that Nim is both underscore- and case-insensitive, meaning that the following variables would all be viewed as having the same name in Nim: —

```
hello_world: int
helloWorld: int
hello_World: int
# etc.
```

Nim variables can, however, contain virtually any UTF-8 characters, including emojis; thus, the following examples would compile and run without any issues: —

```
var Κλεόφιλος: int = 100
echo Κλεόφιλος # Output: 100

var 💖: string = "<3"
echo 💖 # Output: <3
```

Several variables (which can be of different types) can be defined inside one `var` block as follows: —

```
var
    c = -11
    d = "Hello"
    e = '!'
```

The indentation here is vital, as Nim uses spaces to delimit the scope of a declaration, much like Python (but quite unlike Ruby); indentation can, thus, be seen as a replacement for curly brackets in programming languages such as PHP or Rust or the `end` placed after a declaration (such as a `def`). The only type of indentation that can, however, be used is **spaces**. The editor must, thus, convert tabs into any number of spaces (like two or four).

There are two other ways of defining values, `let` and `const`. These are under Immutable Values and Declarations. A list of available data types can be found in Data Types.

Immutable Values and Declarations

Variables, as the name would already suggest, are *variable*; their values can change over the course of the program. At times, however, this is not intended and a static value is, instead, required. There are two ways of declaring these immutable, static values: `const` and `let`.

Const

Constants, which are defined by `const`, must be declared and known at compile time. Thus, a constant cannot be dependent on the output of, for example, a function. Therefore, the following declaration would result in a compilation error: —

```
var five: int = -5
const hyper: int = five + 5
```

Here, the value of `hyper` will be known only after the program has been run, but it will not be known during compilation — hence the error. The following declarations will, however, work without any issues because their values will be known at the time of compilation: —

```
const developerName: string = "Marvin"
const pi: float = 3.141592653589793
```

Let

The other method of defining an immutable value is by using the `let` keyword. The main difference between `const` and `let` is the fact that the value of a constant declared by `let` does *not* need to be known at compile time. Thus, the above-example can be rewritten using `let` to make it work: —

```
var five: int = -5
let hyper: int = five + 5
```

Data Types

As with most other programming languages, there are a number of data types that can be defined when declaring a variable or constant. The following is a list of these data types.

Numbers

The first type of data we will be looking at is numbers. There are a couple of different data types concerning numbers.

Integers

Integers can be declared using the previously-discussed `int` declaration. An underscore can be used as a thousands separator to make larger numbers more easily readable as follows: —

```
let monthlyIncome: int = 10_000
```

Floats

Floats work as one would expect and can be declared using `float`. Scientific notation for larger floats is also possible, such as `4e7`.

Conversion Between Floats and Integers

If one tries to add a float to an integer as in the example below, an error would occur: —

```
let
    monthlyIncome = 10_000
    monthlyTax    = 2_300.5142

echo monthlyIncome - monthlyTax
```

The following error would be thrown: —

```
proc `-`(x, y: int): int
first type mismatch at position: 2
required type for y: int
but expression 'monthlyTax' is of type: float64
```

Therefore, one must either the float to an integer or — more logically in this instance — the integer to a float. This can be achieved by using the aptly named `integer` and `float` functions within Nim as follows: —

```
let
    monthlyIncome = 10_000
    monthlyTax    = 2_300.5142

echo float(monthlyIncome) - monthlyTax # Output: 7699.4858
```

Characters and Strings

A different type of frequently used data types are characters and strings. They are used for displaying either single characters (or a small amount thereof) or larger pieces of text.

Characters

The `char` type can only be used for single ASCII characters (no Unicode here, οὖν πάνυ ἀθυμῶ εἰμὶ ἔγωγε). They always have to be placed within single quotation marks as follows: —

```
let hyperChar: char = 'c'
```

Strings

Strings are usually the much more frequently used and can be declared using `string` (as we have previously seen). Their content is placed in-between double quotes as follows: —

```
let καλώδιον: string = "Οὗτός ἐστιν καλώδιόν τί."
```

Special Characters

There are also a handful of *special characters* that can be used, such as `\n` for a new line, `\t` for a tab and, because `\` is used as the escape character, `\\` must be used when one wishes to write a backslash. The latter problem can be mitigated by using so-called *raw strings*: —

```
echo "Hello \\ there" # Output: Hello \ there
echo r"Hello \ there" # Output: Hello \ there
```

```
echo "Hello \ there"  # Error: invalid character constant
```

Concatenation

Strings in Nim are mutable and can, therefore, be modified — as long as they are defined as a variable and not `const` or `let`. Appending a character or a different string to a string can be done using the `add` function, and simply concatenating different strings together — without altering their values — can be done with `&`. The following examples illustrate this: —

```
var
    hyperString = "I am testing."
    καλώδιον    = "Hello there, my friend."
    hallå       = "Låt oss programmera."

hyperString.add(" " & καλώδιον)
echo hyperString # Output: I am testing. Hello there, my friend.
```

Booleans

Booleans are declared using `bool`. The relational operators are basically the same as in Ruby, i. e. `==` to compare to values, `=` to define a value etc. Strings can be compared using the same operators.

Logical Operators

The logical operators are basically the same as in Ruby, though there do not appear to be “shortcuts” as in Ruby, such as `||` for `or`.

Containers

Containers are iterable data types such as arrays or tuples. As they are iterable, loops can be used to iterate through their values. The simplest of these containers is the array.

Array

An important aspect to know about arrays in Nim is that an array's elements must *all* be of the same data type; mixing different data types within the same array will result in an error. Additionally, an array's size must be known at compile time; their size is, therefore, immutable (unlike in Ruby).

Arrays are declared using `array[<length>, <type>]`. If the length and the type can be inferred from the elements passed to the array, the declaration can be omitted. An array's elements are enclosed inside square brackets, as in Ruby. Here are a few examples: —

```
var
    years: array[5, int]     = [2021, 2020, 2019, 2005, 1999]
    names: array[3, string]  = ["Ἡρόδοτος", "Marvin", "Πλάτων"]
    💕: array = ["💜", "💕", "💗", "💘", "💝"]
```

Sequences

Sequences, unlike arrays, have neither an immutable length nor does their length need to be known at compile-time. Sequences are defined by using `seq[<data type>]` and its elements are placed inside square brackets prepended with an @ sign as follows: `@[<values>]`. The following is an example of such a sequence: —

```
var
    hyperSequence: seq[string] = @["Jag", "dricker", "kaffe"]
    inferredSequence = @[1, 2, 3]
```

Adding to a Sequence

Elements can be added (appended) onto a sequence using the same `add` function that was used to append a string to another string. The sequence must be mutable (i. e. declared using `var`) and and data type of the element being added must be the same as the data type of the sequence: —

```
name: seq[string] = @["M", "a", "r", "v", "i"]
name.add("n")
echo name # Output: @["M", "a", "r", "v", "i", "n"]
```

Finding out a Sequence's Length

Finding out a sequence's length can be achieved by using the rather aptly named `len` function as follows: —

```
name: seq[string] = @["M", "a", "r", "v", "i", "n"]
echo name.len # Output: 6
```

Slicing and Indexing

Slicing and indexing can be used to get a specific values — or series of values — from a container. Indexing, as with virtually all other programming languages (except for Lua, as far as I know), starts at 0.

Indexing

The syntax here is identical to Ruby, i. e. `<container>[<index>]`. A caret can be prepended to the index to index "from the back" (i. e. the last element becomes the first). The last element has index `^1`. Below a short example: —

```
var john: array[5, string] = ["ἐν", "ἀρχῇ", "ἦν", "ὁ", "λόγος"]

echo john[0] # Output: ἐν
echo john[^1] # Output: λόγος
```

Slicing

Slicing is basically the same as indexing, just that it allows one to get a series of elements using one statement. The `start ... stop` syntax can be used here as well: —

```
var
    john: array[5, string] = ["ἐν", "ἀρχῇ", "ἦν", "ὁ", "λόγος"]
    name: seq[string] = @["M", "a", "r", "v", "i", "n"]

echo john[1..2] # Output: @["ἀρχῇ", "ἦν"]
echo name[0..5] # Output: @["M", "a", "r", "v", "i", "n"]
```

Assigning New Values

Both slicing and indexing can be used to assign a new value to a sequence or array — or to change an element: —

```
var
    name: seq[string] = @["G", "u", "ð", "m", "u", "n", "d"]
    years: array[3, int] = [2019, 2020, 2021]
    hyperYears: array[3, int]

for i in 0 .. 2:
    hyperYears[i] = years[i] * 2

name[0] = "K"

echo hyperYears # Output: [4038, 4040, 4042]
echo name # Output: @["K", "u", "ð", "m", "u", "n", "d"]
```

Tuples

Tuples contain heterogeneous data, i. e. elements of a tuple can be of different data types. Like arrays, they have a fixed size. A tuple's elements are enclosed within parentheses: —

```
let hyperTuple = ("Marvin", 1999, 'M')
```

It is also possible to assign a name to each of the fields of a tuple to distinguish them more easily: —

```
let hyperTuple = (name: "Marvin", year: 1999, gender: 'M')

echo hyperTuple.name # Output: Marvin
```

If Statements

If statements are part of *control flow*, i. e. only executing a certain piece of code when a condition is fulfilled. If statements in Nim are written as follows: —

```
if <condition>:
    <indented block of code>
```

It is, once again, important to intend the code not using tabs, but using spaces. If statements can actually be nested, so that the following would be possible: —

```
var Κλεόφιλος: string = "Marv"
if Κλεόφιλος ≠ "Marvin":
    echo "Οὐκ ἐστιν ὁ Κλεόφιλος ἀληθῶς ὁ Κλεόφιλος."
    if Κλεόφιλος & "in" == "Marvin":
      echo "Marvin has returned."
```

Else and Else If

There are, of course, also else and else if statements. Else statements are done as follows: —

```
var hyperName: string = "Marvin"
if hyperName == "Marvin":
    echo "Truly, it is him!"
else:
    echo "Hinfort!"
```

And else if statements are done using `elif` in the following manner: —

```
var 🎄: string = "Christmas"
if 🎄 == "Christmas":
```

```
    echo "Merry Christmas!"
elif 🎄 == "Jul":
    echo "God jul till dig!"
elif 🎄 == "Weihnachten":
    echo "Fröhliche Weihnachten!"
else:
    echo "Οὐ καταλαμβάνω τὴν γλῶτταν ταύτην."
```

Case Statements are quite similar to if statements.

Case Statements

Case statements work quite similarly to those found in Ruby. Their syntax differs somewhat, however: —

```
var 🎄: string = "Christmas"
case 🎄
of "Christmas":
    echo "Merry Christmas!"
of "Weihnachten":
    echo "Fröhliche Weihnachten!"
else: echo "Det här språket kan jag inte förstå."
```

Unlike if statements (and unlike Ruby's case statements), case statements *must cover all possible cases*; therefore, when nothing is supposed to happen in the `else` portion of the case statement, the following must be written: —

```
var 🎄: string = "Christmas"
case 🎄
of "Christmas":
    echo "Merry Christmas!"
of "Weihnachten":
    echo "Fröhliche Weihnachten!"
else: discard
```

Several values can be added to one branch of a case statement by separating each value with a comma: —

```
var 🎄: string = "Christmas"
case 🎄
of "Christmas", "Yule":
    echo "Merry Christmas!"
```

Loops

Nim has a number of different loops, just like Ruby does too. The most commonly used ones are the for- and while-loops.

For-loop

The for-loop is pretty straight-forward and looks as follows, wherein the `iterable` is any object which one can iterate through: —

```
for <loopVariable> in <iterable>:
    <loop body>
```

Iterating through a range of inter numbers (or even chars) can be done quite easily — and in a similar manner to Ruby — using the `<start> .. <finish>` syntax as follows: —

```
for i in 'a' .. 'z':
    echo i # outputs the letters 'a' through 'z'

for i in 1 .. 10:
    echo i # outputs the numbers 1 to 10
```

If you do not wish to include the last number or char, the `..<` syntax can be used.

```
for i in 1 ..< 10:
    echo i # outputs the numbers 1 to 9
```

Different step sizes can be defined using the `countup` or `countdown` functions as follows: —

```
for i in countup(0, 100, 10):
    echo i # outputs 0, 10, 20, 30 …

for i in countdown(100, 0, 10):
    echo i #outputs 100, 90, 80, 80 …
```

As strings are iterables, they can be iterated through using a for-loop as well: —

```
let programming = "Nim Programming"
for l in programming:
    echo l #outputs each letter of the string individually
```

One can also use *two* iterables within one for-loop in the following way: —

```
let programming = "Nim Programming"
for i, l in programming:
    echo "Letter number ", i, " is: ", l
```

While loop

While loops are also quite similar to those found in Ruby. The `inc` value can be used in a similar manner as the `++` found in a variety of other programming language (i. e. to increment a variable value by one). An example of a simple while look is the following: —

```
var a = 1
while a * a < 10:
    echo "a is: ", a
    inc a
```

Break and Continue

There are two special commands that can be used with loops (mostly while loops), namely `break` and `continue`.

Break

As with other programming languages, the `break` command can be used to prematurely exit a loop. The following is an example of that: —

```
var year: int = 2021
while year < 2050:
    echo year
    inc year
    if year == 2030:
        break # Output: 2021, 2022 … 2029
```

Continue

The `continue` statement immediately executes the next iteration of a loop; in this way, certain values can be “skipped”, so to speak. An example is the following: —

```
var name = "Marvin Johanning"

for l in name:
    if l == "n":
        continue
    echo l # Will remove all "n"s from my name
```

Procedures (Functions)

Exercises

Collatz conjecture

> *First pick a number. If it is odd, multiply it by three and add one; if it is even, divide it by two. Repeat this procedure until you arrive at one. E.g. 5 → odd → 3*5 + 1 = 16 → even → 16 / 2 = 8 → even → 4 → 2 → 1 → end*

```
# Collatz Conjecture

var number: int = 123
```

```
while number ≠ 1:
    echo number
    if number mod 2 == 0:
        number = int(number / 2)
    else:
        number = number * 3 + 1
```

Fizz Buzz

> *We count numbers from one upwards. If a number is divisible by 3 replace it with fizz, if it is divisible by 5 replace it with buzz, and if a number is divisible by 15 (both 3 and 5) replace it with fizzbuzz. First few rounds would look like this: 1, 2, fizz, 4, buzz, fizz, 7, ... Create a program which will print first 30 rounds of Fizz buzz.*

```
# Fizz Buzz

for i in 1..30:
    if i mod 15 == 0:
        echo "fizzbuzz"
    elif i mod 3 == 0:
        echo "fizz"
    elif i mod 5 == 0:
        echo "buzz"
    else:
        echo i
```

Output

```
1 2 fizz 4 buzz fizz 7 8 fizz buzz 11 fizz 13 14 fizzbuzz 16 17 fizz 19
buzz fizz 22 23 fizz buzz 26 fizz 28 29 fizzbuzz
```

Vowels Only

> *Create an immutable variable containing your full name. Write a for-loop which will iterate through that string and print only the vowels.*

```
let name: string = "Marvin Johanning"

for l in name:
    case l
    of 'a', 'e', 'i', 'o', 'u':
        echo l
    else:
        discard
```

Output

```
aioai
```

Collatz Conjecture — Revisted

> *Re-do the Collatz conjecture exercise, but this time instead of printing each step, add it to a sequence: Create a sequence whose only member is that starting number; using the same logic as before, keep adding elements to the sequence until you reach 1; print the length of the sequence, and the sequence itself*

```
# Collatz Conjecture Revisted
var numbers: seq[int] = @[9]

while numbers[^1] ≠ 1:
    if numbers[^1] mod 2 == 0:
        numbers.add(int(numbers[^1] / 2))
    else:
        numbers.add(numbers[^1] * 3 + 1)

echo "The following numbers were found: ", numbers
echo "In total, there were ", numbers.len, " numbers found"
```

Output

```
The following numbers were found: @[9, 28, 14, 7, 22, 11, 34, 17, 52, 26,
13, 40, 20, 10, 5, 16, 8, 4, 2, 1]
In total, there were 20 numbers found
```

Hyper Collatz

> *Find the number in a range from 2 to 100 which will produce the longest Collatz sequence: For each number in the given range calculate its Collatz sequence; if the length of current sequence is longer than the previous record, save the current length and the starting number as a new record; print the starting number which gives the longest sequence, and its length*

```
# Hyper Collatz
var bigCollatz = (longestLength: 0, startingNumber: 0)
var bigCandidate: seq[int]

for numberCandidate in 2 .. 100:
    bigCandidate = @[numberCandidate]

    while bigCandidate[^1] ≠ 1:
        if bigCandidate[^1] mod 2 == 0:
            bigCandidate.add(int(bigCandidate[^1] / 2))
        else:
            bigCandidate.add(bigCandidate[^1] * 3 + 1)
        if bigCandidate.len > bigCollatz.longestLength:
            bigCollatz.longestLength = bigCandidate.len
            bigCollatz.startingNumber = numberCandidate


echo "The longest Collatz sequence is ", bigCollatz.longestLength , "
numbers long."
echo "The number starting number for this sequence was: ",
bigCollatz.startingNumber
```

Output

```
The longest Collatz sequence is 119 numbers long.
The number starting number for this sequence was: 97
```

Array stuff

Create an empty array which can contain ten integers: Fill that array with numbers 10, 20, ..., 100; - Print only the elements of that array that are on odd indices (values 20, 40, ...); multiply elements on even indices by 5. Print the modified array.

```
# Array Stuff
var hyperSeq: seq[int]

for num in countup(10, 100, 10):
    hyperSeq.add(num)

for i in 0 .. (hyperSeq.len - 1):
    if i mod 2 == 1:
        echo hyperSeq[i]
    else:
        hyperSeq[i] = hyperSeq[i] * 5

echo hyperSeq
```

Output

```
20
40
60
80
100
@[50, 20, 150, 40, 250, 60, 350, 80, 450, 100]
```