Nim basics

Nim basics

Table of Contents

Nim is a relatively new programming language which allows users to write easy-to-read high-performance code. But if you are reading this Nim tutorial, the chances are that you already know about Nim.

The tutorial is available both online and as a PDF.

This is a work-in-progress: if you spot any errors and/or you have an idea how to make this tutorial better, please report it to the issue tracker.

Who is this for?

- People with no or minimal previous programming experience
- People with some programming experience in other programming languages
- People who want to explore Nim for the first time, starting from scratch

Who is this *not* for?

- People with lots of programming experience: other, more advanced, tutorials might suit you better. See Official Tutorial or Nim by Example.
- People experienced in Nim (feel free to help make this tutorial better)

How to use this tutorial?

The aim of this tutorial is to give you the basics of programming and the Nim syntax so you can have an easier time following other tutorials and/or explore further by yourself.

Instead of just reading what is written, it would be the best if you try the stuff by yourself, modify the examples, think of some examples of your own, and be curious in general. The exercises at the end of some chapters should be non-negotiable — don't skip them.

If you need additional help understanding some parts of the tutorial or with the exercises, you can always ask for help on the Nim forum, the Nim Gitter channel, their Discord server, or Nim's IRC channel on freenode, #nim.

Installation

Installing Nim

Nim has ready made distributions for all three major operating systems and there are several options when it comes to installing Nim.

You can follow the official installation procedure to install the latest stable version, or you can use a tool called choosenim which enables you to easily switch between the stable and the latest development version if you're interested in the latest features and bugfixes.

Whichever way you choose, just follow the installation procedure explained at each link and Nim should be installed. We will check that the installation went well in a coming chapter.

If you're using Linux, there is a high probability that your distribution has Nim in the package manager. If you are installing it that way, make sure it's the most recent version (see the website for what is the latest version), otherwise install via one of two methods mentioned above.

In this tutorial we will use the stable version. Originally, this tutorial was written for Nim 0.19 (released in September 2018), and it should work for any newer version, including Nim 1.0.

Installing additional tools

You can write Nim code in any text editor, and then compile and run it from the terminal. If you want syntax highlighting and code completion there are plugins for popular code editors which provide these features.

Most of Nim users prefer the VS Code editor, with the Nim extension which provides syntax highlighting and code completion, and the Code Runner extension for quick compiling and running.

The author personally uses NeoVim editor, with this plugin which provides additional features like syntax highlighting and code completion.

If you're using other code editors, see the wiki for available editor support.

Testing the installation

To check if the installation was successful, we will write a program which is traditionally used as an introductory example: Hello World.

Printing (as in: displaying on the screen; not on a paper with a printer) the phrase `Hello World!` in Nim is straightforward and it doesn't require any boilerplate code.

In a new text file called e.g. `helloworld.nim` we need to write just one line of code:

helloworld.nim

```
echo "Hello World!"
```

> ℹ️ The phrase you want to print must follow the `echo` command and must be enclosed in double-quotes (`"`).

First we need to compile our program, and then run it to see if it works as expected.

Open your terminal in the same directory where your file is (on Linux you can get "Open Terminal here" if you right-click the directory in your file manager, on Windows you should use Shift + right-click to get the menu option for opening the command line).

We compile our program by typing in the terminal:

```
nim c helloworld.nim
```

After a successful compilation, we can run our program. On Linux we can run our program by typing `./helloworld` in the terminal, and on Windows we do it by typing `helloworld.exe`.

There is also a possibility to both compile and run the program with just one command. We need to type:

```
nim c -r helloworld.nim
```

`c` is telling Nim to compile the file, and `-r` is telling it to run it immediately.

To see all compiler options, type `nim --help` in your terminal.

If you're using VSCode with the Code Runner extension mentioned before, you'll just have to press `Ctrl+Alt+N` and your file will be compiled and run.

Whichever way you chose to run your program, after a brief moment in the output window (or in your terminal) you should see:

```
Hello World!
```

Congratulations, you have successfully run your first Nim program!

Now you know how to print some stuff on the screen (using the `echo` command), compile your program (typing `nim c programName.nim` in your terminal), and run it (various possibilities).

We can now start to explore the basic elements which will help us to write simple Nim programs.

Naming values

It is often helpful to give the values in our programs names to help us keep track of things. If we ask a user for his/her name, we want to store it for the later usage, without asking for it again and again every time we need to do some computation with it.

In the example `pi = 3.14`, the name `pi` is connected to the value `3.14`. From our experience, we can tell that the type of a variable `pi` is a (decimal) number.

Another example would be `firstName = Alice`, where `firstName` is the name of a variable with the value `Alice`. We would say that the type of this variable is a word.

In programming languages this works similarly. These name assignments have their *name*, the *value*, and a *type*.

Variable declaration

Nim is a statically typed programming language, meaning that the type of an assignment needs to be declared before using the value.

In Nim we also distinguish values that can change, or mutate, from those that can't, but more on this later. We can declare a variable (a mutable assignment) using the `var` keyword, just by stating its name and type (the value can be added later) by using this syntax:

```
var <name>: <type>
```

If we already know its value, we can declare a variable and give it a value immediately:

```
var <name>: <type> = <value>
```

> ℹ️ Angular brackets(`<>`) are used to show something you can change.
> So `<name>` is not literally the word `name` in angular brackets but rather any name.

Nim also has type inference ability: the compiler can automatically detect the type of a name assignment from its value, without explicitly stating the type.

We'll look more into the various types in the next chapter.

So we can assign a variable without an explicit type like this:

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

An example of this in Nim looks like this:

```
var a: int    ①
var b = 7     ②
```

① Variable `a` is of type `int` (integer) with no value explicitly set.

② Variable `b` has a value of `7`. Its type is automatically detected as an integer.

When assigning names it is important to choose names that mean something for your program. Simply naming them `a`, `b`, `c`, and so forth will quickly become confusing. It is not possible to use spaces in a name, as that would split it into two. So if the name you choose consists of more than one word the usual way is to write it in `camelCase` style (notice that the first letter in a name should be lowercase).

Note however that Nim is both case- and underscore-insensitive meaning that `helloWorld` and `hello_world` would be the same name. The exception to this is the first character, which *is* case-sensitive. Names can also include both numbers and other UTF-8 characters, even emojis should you wish that, but keep in mind you and possibly others will have to type them.

Instead of typing `var` for each variable, multiple variables (not necessarily of the same type) can be declared in the same `var` block. In Nim, blocks are parts of code with the same indentation (same number of spaces before the first character), and the default indentation level is two spaces. You will see such blocks everywhere in a Nim program, not only for assigning names.

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

> ℹ️ In Nim tabs are not allowed as indentation.
> You can set up your code editor to convert pressing `Tab` to any number of spaces.
> In VS Code, the default setting is to convert `Tab` to four spaces. This is easily overridden in settings (`Ctrl+,`) by setting `"editor.tabSize": 2`.

As previously mentioned variables are mutable, i.e. their value can change (multiple times), but their type must stay the same as declared.

```nim
var f = 7              ①

f = -3                ②
f = 19
f = "Hello" # error   ③ ④
```

① Variable `f` has an initial value of `7` and its type is inferred as `int`.

② The value of `f` is first changed to `-3`, and then to `19`. Both of these are integers, the same as the original value.

③ Trying to change the value of `f` to `"Hello"` produces an error because `Hello` is not a number, and this would change the type of `f` from an integer to a string.

④ `# error` is a comment. Comments in Nim code are written after a `#` character. Everything after it on the same line will be ignored.

Immutable assignment

Unlike variables declared with `var` keyword, two more types of assignment exist in Nim, whose value cannot change, one declared with the `const` keyword, and the other declared with the `let` keyword.

Const

The value of an immutable assignment declared with `const` keyword must be known at compile time (before the program is run).

For example, we can declare the acceleration of gravity as `const g = 9.81` or pi as `const pi = 3.14`, as we know their values in advance and these values will not change during the execution of our program.

```
const g = 35
g = -27          # error ①

var h = -5
const i = h + 7 # error ②
```

① The value of a constant cannot be changed.

② Variable `h` is not evaluated at compile time (it is a variable and its value can change during the execution of a program), consequently the value of constant `i` can't be known at compile time, and this will raise an error.

In some programming languages it is a common practice to have the names of constants written in `ALL_CAPS`. Constants in Nim are written just like any other variable.

Let

Immutable assignments declared with `let` don't need to be known at compile time, their value can be set at any time during the execution of a program, but once it is set, their value cannot change.

```
let j = 35
j = -27 # error ①

var k = -5
let l = k + 7    ②
```

① The value of an immutable cannot be changed.

② In contrast to `const` example above, this works.

In practice, you will see/use `let` more frequently than `const`.

While you could use `var` for everything, your default choice should be `let`. Use `var` only for the variables which will be modified.

Basic data types

Integers

As seen in the previous chapter, integers are numbers which are written without a fractional component and without a decimal point.

For example: `32`, `-174`, `0`, `10_000_000` are all integers. Notice that we can use `_` as a thousands separator, to make larger numbers more readable (it is easier to see that we're talking about 10 million when it's written as 10_000_000 rather than as 10000000).

The usual mathematical operators — addition (`+`), subtraction (`-`), multiplication (`*`), and division (`/`) — work as one would expect. The first three operations always produce integers, while dividing two integers always gives a floating point number (a number with a decimal point) as a result, even if two numbers can be divided without a remainder.

Integer division (division where the fractional part is discarded) can be achieved with the `div` operator. An operator `mod` is used if one is interested in the remainder (modulus) of an integer division. The result of these two operations is always an integer.

integers.nim

```
let
  a = 11
  b = 4

echo "a + b = ", a + b ①
echo "a - b = ", a - b
echo "a * b = ", a * b
echo "a / b = ", a / b
echo "a div b = ", a div b
echo "a mod b = ", a mod b
```

① The `echo` command will print to the screen everything that follows it separated by commas. In this case, it first prints the string `a + b = `, and then after it, in the same row, it prints the result of the expression `a + b`.

We can compile and run the above code, and the output should be:

```
a + b = 15
a - b = 7
a * b = 44
a / b = 2.75
a div b = 2
a mod b = 3
```

Floats

Floating-point numbers, or floats for short, are an approximate representation of real numbers.

For example: `2.73`, `-3.14`, `5.0`, `4e7` are floats. Notice that we can use scientific notation for large floats, where the number after the `e` is the exponent. In this example, `4e7` is a notation representing `4 * 10^7`.

We can also use the four basic mathematical operations between two floats. Operators `div` and `mod` are not defined for floats.

floats.nim

```
let
  c = 6.75
  d = 2.25

echo "c + d = ", c + d
echo "c - d = ", c - d
echo "c * d = ", c * d
echo "c / d = ", c / d
```

```
c + d = 9.0    ①
c - d = 4.5
c * d = 15.1875
c / d = 3.0    ①
```

① Notice that in the addition and division examples, even though we get a number without a decimal part, the result is still of the floating type.

The precedence of mathematical operations is as one would expect: multiplication and division have higher priority than addition and subtraction.

```
echo 2 + 3 * 4
echo 24 - 8 / 4
```

```
14
22.0
```

Converting floats and integers

Mathematical operations between variables of different numerical types are not possible in Nim, and they will produce an error:

```
let
  e = 5
  f = 23.456

echo e + f   # error
```

The values of variables need to be converted to the same type. Conversion is straight-forward: to convert to an integer, we use the `int` function, and to convert to a float the `float` function is used.

```
let
  e = 5
  f = 23.987

echo float(e)      ①
echo int(f)        ②

echo float(e) + f  ③
echo e + int(f)    ④
```

① Printing a `float` version of an integer `e`. (`e` remains of integer type)

② Printing an `int` version of a float `f`.

③ Both operands are floats and can be added.

④ Both operands are integers and can be added.

```
5.0
23
28.987
28
```

ℹ When using the `int` function to convert a float to an integer no rounding will be performed. The number simply drops any decimals. To perform rounding we must call another function, but for that we must know a bit more about how to use Nim.

Characters

The `char` type is used for representing a single ASCII character.

Chars are written between two single ticks (`'`). Chars can be letters, symbols, or single digits. Multiple digits or multiple letters produce an error.

```
let
  h = 'z'
  i = '+'
  j = '2'
  k = '35' # error
  l = 'xy' # error
```

Strings

Strings can be described as a series of characters. Their content is written between two double quotes (`"`).

We might think of strings as words, but they can contain more than one word, some symbols, or digits.

strings.nim

```
let
  m = "word"
  n = "A sentence with interpunction."
  o = ""        ①
  p = "32"      ②
  q = "!"       ③
```

① An empty string.

② This is not a number (int). It is inside double quotes, making it a string.

③ Even though this is only one character, it is not a char because it is enclosed inside of double quotes.

Special characters

If we try to print the following string:

```
echo "some\nim\tips"
```

the result might surprise us:

```
some
im  ips
```

This is because there are several characters which have a special meaning. They are used by prepending the escape character `\` to them.

- `\n` is a newline character
- `\t` is a tab character
- `\\` is a backslash (since one `\` is used as the escape character)

If we wanted to print the above example as it was written, we have two possibilities:

- Use `\\` instead of `\` to print backslashes, or
- Use raw strings which have syntax `r"▯"` (putting a letter `r` immediately before the first quote), in which there are no escape characters and no special meanings: everything is printed as it is.

```
echo "some\\nim\\tips"
echo r"some\nim\tips"
```

```
some\nim\tips
some\nim\tips
```

There are more special characters than the ones listed above, and they are all found in the Nim manual.

String concatenation

Strings in Nim are mutable, meaning their content can change. With the `add` function we can add (append) either another string or a char to an existing string. If we don't want to change the original string, we can also concatenate (join together) strings with the `&` operator, this returns a new string.

stringConcat.nim

```
var                        ①
  p = "abc"
  q = "xy"
  r = 'z'

p.add("def")               ②
echo "p is now: ", p

q.add(r)                   ③
echo "q is now: ", q

echo "concat: ", p & q     ④

echo "p is still: ", p
echo "q is still: ", q
```

① If we plan to modify strings, they should be declared as `var`.

② Adding another string modifies the existing string `p` in-place, changing its value.

③ We can also add a `char` to a string.

④ Concatenating two strings produces a new string, without modifying the original strings.

```
p is now: abcdef
q is now: xyz
concat: abcdefxyz
p is still: abcdef
q is still: xyz
```

Boolean

A boolean (or just `bool`) data type can only have two values: `true` or `false`. Booleans are usually used for control flow (see next chapter), and they are often a result of relational operators.

The usual naming convention for boolean variables is to write them as a simple yes/no (true/false) questions, e.g. `isEmpty`, `isFinished`, `isMoving`, etc.

Relational operators

Relational operators test the relation between two entities, which must be comparable.

To compare if two values are the same, `==` (two equal signs) is used. Do not confuse this with `=`, which is used for assignment as we saw earlier.

Here are all the relational operators defined for integers:

relationalOperators.nim

```nim
let
  g = 31
  h = 99

echo "g is greater than h: ", g > h
echo "g is smaller than h: ", g < h
echo "g is equal to h: ", g == h
echo "g is not equal to h: ", g != h
echo "g is greater or equal to h: ", g >= h
echo "g is smaller or equal to h: ", g <= h
```

```
g is greater than h: false
g is smaller than h: true
g is equal to h: false
g is not equal to h: true
g is greater or equal to h: false
g is smaller or equal to h: true
```

We can also compare characters and strings:

relationalOperators.nim

```nim
let
  i = 'a'
  j = 'd'
  k = 'Z'

echo i < j
echo i < k   ①

let
  m = "axyb"
  n = "axyz"
  o = "ba"
  p = "ba "

echo m < n   ②
echo n < o   ③
echo o < p   ④
```

① All uppercase letters come before lowercase letters.

② String comparison works char-by-char. First three characters are the same, and character `b` is smaller than character `z`.

③ String length doesn't matter for comparison if their characters are not identical.

④ Shorter string is smaller than the longer one.

```
true
false
true
true
true
```

Logical operators

Logical operators are used to test the truthiness of an expression consisting of one or more boolean values.

- Logical `and` returns `true` only if both members are `true`
- Logical `or` returns `true` if there is at least one member which is `true`
- Logical `xor` returns `true` if one member is true, but the other is not

- Logical `not` negates the truthiness of its member: changing `true` to `false`, and vice versa (it is the only logical operator that takes just one operand)

logicalOperators.nim

```
echo "T and T: ", true and true
echo "T and F: ", true and false
echo "F and F: ", false and false
echo "---"
echo "T or T: ", true or true
echo "T or F: ", true or false
echo "F or F: ", false or false
echo "---"
echo "T xor T: ", true xor true
echo "T xor F: ", true xor false
echo "F xor F: ", false xor false
echo "---"
echo "not T: ", not true
echo "not F: ", not false
```

```
T and T: true
T and F: false
F and F: false
---
T or T: true
T or F: true
F or F: false
---
T xor T: false
T xor F: true
F xor F: false
---
not T: false
not F: true
```

Relational and logical operators can be combined together to form more complex expressions.

For example: `(5 < 7) and (11 + 9 == 32 - 2*6)` will become `true and (20 == 20)`, which becomes `true and true`, and in the end this will give the final result of `true`.

Recap

This was the longest chapter in this tutorial and we covered a lot of ground. Take your time to go trough each data type and experiment with what you can do with each of them.

Types might seem like a restriction at first, but they allow the Nim compiler to both make your code faster, and make sure you're not doing something wrong by accident — this is especially beneficial in large code bases.

Now you know the basic data types and several operations on them, which should be enough to do some simple calculations in Nim. Test your knowledge by doing the following exercises.

Exercises

1. Create an immutable variable containing your age (in years). Print your age in days. (1 year = 365 days)

2. Check if your age is divisible by 3. (Hint: use `mod`)

3. Create an immutable variable containing your height in centimeters. Print your height in inches. (1 in = 2.54 cm)

4. A pipe has a 3/8 inch diameter. Express the diameter in centimeters.

5. Create an immutable variable containing your first name, and another one containing your last name. Make a variable `fullName` by concatenating the previous two variables. Don't forget to put a whitespace in-between. Print your full name.

6. Alice earns $400 every 15 days. Bob earns $3.14 per hour and works 8 hours a day, 7 days a week. After 30 days, has Alice earned more than Bob? (Hint: use relational operators)

Control flow

So far in our programs every line of code was executed at some point. Control flow statements allow us to have parts of code which will be executed only if some boolean condition is satisfied.

If we think of our program as a road we can think of control flow as various branches, and we pick our path depending on some condition. For example, we will buy eggs only *if* their price is less than some value. Or, *if* it is raining, we will bring an umbrella, otherwise (*else*) we will bring sunglasses.

Written in pseudocode, these two examples would look like this:

```
if eggPrice < wantedPrice:
  buyEggs

if isRaining:
  bring umbrella
else:
  bring sunglasses
```

Nim syntax is very similar, as you'll see below.

If statement

An if statement as shown above is the simplest way to branch our program.

The Nim syntax for writing if statement is:

```
if <condition>:      ①
  <indented block>   ②
```

① The `condition` must be of boolean type: either a boolean variable or a relational and/or logical expression.

② All lines following the `if` line which are indented two spaces make the same block and will be executed only if the condition is `true`.

If statements can be nested, i.e. inside one if-block there can be another if statement.

if.nim

```nim
let
  a = 11
  b = 22
  c = 999

if a < b:
  echo "a is smaller than b"
  if 10*a < b:   ①
    echo "not only that, a is *much* smaller than b"

if b < c:
  echo "b is smaller than c"
  if 10*b < c:   ②
    echo "not only that, b is *much* smaller than c"

if a+b > c:       ③
  echo "a and b are larger than c"
  if 1 < 100 and 321 > 123:   ④
    echo "did you know that 1 is smaller than 100?"
    echo "and 321 is larger than 123! wow!"
```

① The first condition is true, the second is false — inner `echo` is not executed.

② Both conditions are true and both lines are printed.

③ The first condition is false — all lines inside of its block will be skipped, nothing is printed.

④ Using the logical `and` inside of the `if` statement.

```
a is smaller than b
b is smaller than c
not only that, b is *much* smaller than c
```

Else

Else follows after an if-block and allows us to have a branch of code which will be executed when the condition in the if statement is not true.

else.nim

```
let
  d = 63
  e = 2.718

if d < 10:
  echo "d is a small number"
else:
  echo "d is a large number"

if e < 10:
  echo "e is a small number"
else:
  echo "e is a large number"
```

```
d is a large number
e is a small number
```

> ℹ️ If you only want to execute a block if the statement is `false`, you can simply negate the condition with the `not` operator.

Elif

Elif is short for "else if", and enables us to chain multiple if statements together.

The program tests every statement until it finds one which is true. After that, all further statements are ignored.

elif.nim

```nim
let
  f = 3456
  g = 7

if f < 10:
  echo "f is smaller than 10"
elif f < 100:
  echo "f is between 10 and 100"
elif f < 1000:
  echo "f is between 100 and 1000"
else:
  echo "f is larger than 1000"

if g < 1000:
  echo "g is smaller than 1000"
elif g < 100:
  echo "g is smaller than 100"
elif g < 10:
  echo "g is smaller than 10"
```

```
f is larger than 1000
g is smaller than 1000
```

> In the case of g, even though g satisfies all three conditions, only the first branch is executed, automatically skipping all the other branches.

Case

A case statement is another way to only choose one of multiple possible paths, similar to the if statement with multiple `elif`s. A `case` statement, however, doesn't take multiple boolean conditions, but rather any value with distinct states and a path for each possible value.

Code written with in if-elif block looking like this:

```nim
if x == 5:
  echo "Five!"
elif x == 7:
  echo "Seven!"
elif x == 10:
  echo "Ten!"
else:
  echo "unknown number"
```

can be written with case statement like this:

```nim
case x
of 5:
  echo "Five!"
of 7:
  echo "Seven!"
of 10:
  echo "Ten!"
else:
  echo "unknown number"
```

Unlike the if statement, case statement must cover *all* possible cases. If one is not interested in some of those cases, `else: discard` can be used.

case.nim

```nim
let h = 'y'

case h
of 'x':
  echo "You've chosen x"
of 'y':
  echo "You've chosen y"
of 'z':
  echo "You've chosen z"
else: discard   ①
```

① Even though we are interested in only three values of `h`, we must include this line to cover all other possible cases (all other characters). Without it, the code would not compile.

```
You've chosen y
```

We can also use multiple values for each branch if the same action should happen for more than one value.

multipleCase.nim

```nim
let i = 7

case i
  of 0:
    echo "i is zero"
  of 1, 3, 5, 7, 9:
    echo "i is odd"
  of 2, 4, 6, 8:
    echo "i is even"
  else:
    echo "i is too large"
```

```
i is odd
```

Loops

Loops are another control flow construct which allow us to run some parts of code multiple times.

In this chapter we will meet two kinds of loops:

- for-loop: run a known number of times

- while-loop: run as long some condition is satisfied

For loop

Syntax of a for-loop is:

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

Traditionally, `i` is often used as a `loopVariable` name, but any other name can be used. That variable will be available only inside the loop. Once the loop has finished, the value of the variable is discarded.

The `iterable` is any object we can iterate through. Of the types already mentioned, strings are iterable objects. (More iterable types will be introduced in the next chapter.)

All lines in the `loop body` are executed at every loop, which allows us to efficiently write repeating parts of code.

If we want to iterate through a range of (integer) numbers in Nim, the syntax for the `iterable` is `start .. finish` where `start` and `finish` are numbers. This will iterate through all the numbers between `start` and `finish`, including both `start` and `finish`. For the default range iterable, `start` needs to be smaller than `finish`.

If we want to iterate *until* a number (not including it), we can use `..<`:

for1.nim

```nim
for n in 5 .. 9:    ①
  echo n

echo ""

for n in 5 ..< 9:   ②
  echo n
```

① Iterating through a range of numbers using `..` — both ends are included in the range.

② Iterating through the same range using `..<` — it iterates until the higher end, not including it.

```
5
6
7
8
9

5
6
7
8
```

If we want to iterate through a range of numbers with a step size different than one, `countup` is used. With `countup` we define the starting value, the stopping value (included in the range), and the step size.

for2.nim

```nim
for n in countup(0, 16, 4):   ①
  echo n
```

① Counting up from zero to 16, with a step size of 4. The end (16) is included in the range.

```
0
4
8
12
16
```

To iterate through a range of numbers where the `start` is larger than `finish`, a similar function called `countdown` is used. Even if we're counting down, the step size must be positive.

for2.nim

```
for n in countdown(4, 0):        ①
  echo n

echo ""

for n in countdown(-3, -9, 2):   ②
  echo n
```

① To iterate from a higher to a lower number, we must use `countdown` (The `..` operator can only be used when the starting value is smaller than the end value).

② Even when counting down, the step size must be a positive number.

```
4
3
2
1
0

-3
-5
-7
-9
```

Since string is an iterable, we can use a for-loop to iterate through each character of the string (this kind of iteration is sometimes called a *for-each* loop).

for3.nim

```
let word = "alphabet"

for letter in word:
  echo letter
```

```
a
l
p
h
a
b
e
t
```

If we also need to have an iteration counter (starting from zero), we can achieve that by using `for <counterVariable>, <loopVariable> in <iterator>:` syntax. This is very practical if you want to iterate through one iterable, and simultaneously access another iterable at the same offset.

for3.nim

```
for i, letter in word:
  echo "letter ", i, " is: ", letter
```

```
letter 0 is: a
letter 1 is: l
letter 2 is: p
letter 3 is: h
letter 4 is: a
letter 5 is: b
letter 6 is: e
letter 7 is: t
```

While loop

While loops are similar to if statements, but they keep executing their block of code as long as the condition remains true. They are used when we don't know in advance how many times the loop will run.

We must make sure the loop will terminate at some point and not become an infinite loop.

```nim
var a = 1

while a*a < 10: ①
  echo "a is: ", a
  inc a        ②

echo "final value of a: ", a
```

① This condition will be checked every time before entering the new loop and executing the code inside of it.

② `inc` is used to increment `a` by one. It is the same as writing `a = a + 1` or `a += 1`.

```
a is: 1
a is: 2
a is: 3
final value of a: 4
```

Break and continue

The `break` statement is used to prematurely exit from a loop, usually if some condition is met.

In the next example, if there were no if statement with `break` in it, the loop would continue to run and print until `i` becomes 1000. With the `break` statement, when `i` becomes 3, we immediately exit the loop (before printing the value of `i`).

break.nim

```nim
var i = 1

while i < 1000:
  if i == 3:
    break
  echo i
  inc i
```

```
1
2
```

The `continue` statement starts the next iteration of a loop immediately, without executing the remaining lines of the current iteration. Notice how 3 and 6 are missing from the output of the following code:

continue.nim

```
for i in 1 .. 8:
  if (i == 3) or (i == 6):
    continue
  echo i
```

```
1
2
4
5
7
8
```

Exercises

1. Collatz conjecture is a popular mathematical problem with simple rules. 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!
Pick an integer (as a mutable variable) and create a loop which will print every step of the Collatz conjecture. (Hint: use `div` for division)

2. Create an immutable variable containing your full name. Write a for-loop which will iterate through that string and print only the vowels (a, e, i, o, u). (Hint: use `case` statement with multiple values per branch)

3. Fizz buzz is a kids game sometimes used to test basic programming knowledge. 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. (Hint: beware of the order of divisibility tests)

4. In the previous exercises you have converted inches to centimeters, and vice versa. Create a conversion table with multiple values. For example, the table might look like this:

```
in | cm
----------------
1   | 2.54
4   | 10.16
7   | 17.78
10  | 25.4
13  | 33.02
16  | 40.64
19  | 48.26
```

Containers

Containers are data types which contain a collection of items and allow us to access those elements. Typically a container is also iterable, meaning that we can use them the same way we used strings in the loops chapter.

For example, a grocery list is a container of items we want to buy, and a list of primes is a container of numbers. Written in pseudocode:

```
groceryList = [ham, eggs, bread, apples]
primes = [1, 2, 3, 5, 7]
```

Arrays

An array is the simplest container type. Arrays are homogeneous, i.e. all elements in an array must have the same type. Arrays are also of a constant size, meaning that the amount of elements (or rather: the amount of *possible* elements), must be known at compile-time. This means that we call arrays a "homogeneous container of a constant length".

The array type is declared using `array[<length>, <type>]`, where `length` is the total capacity of the array (number of elements it can fit), and `type` is a type of all its elements. The declaration can be omitted if both length and type can be inferred from the passed elements.

The elements of an array are enclosed inside of square brackets.

```
var
  a: array[3, int] = [5, 7, 9]
  b = [5, 7, 9]          ①
  c = []   # error       ②
  d: array[7, string]    ③
```

① If we provide the values, the length and type of array `b` are known at compile time. Although correct, there is no need to specifically declare it like array `a`.

② Neither the length nor the type of the elements can be inferred from this kind of declaration — this produces an error.

③ The correct way to declare an empty array (which will be filled later) is to

give its length and type, without providing the values of its elements — array `d` can contain seven strings.

Since the length of an array has to be known at compile-time, this will not work:

```
const m = 3
let n = 5

var a: array[m, char]
var b: array[n, char] # error ①
```

① This produces an error because `n` is declared using `let` — its value is not known at compile time. We can only use values declared with `const` as a `length` parameter for an array initialization.

Sequences

Sequences are containers similar to arrays, but their length doesn't have to be known at compile time, and it can change during runtime: we declare only the type of the contained elements with `seq[<type>]`. Sequences are also homogeneous, i.e. every element in a sequence has to be the same type.

The elements of a sequence are enclosed between `@[` and `]`.

```
var
  e1: seq[int] = @[]   ①
  f = @["abc", "def"]   ②
```

① The type of an empty sequence must be declared.

② The type of a non-empty sequence can be inferred. In this case, it is a sequence containing strings.

Another way to initialize an empty sequence is to call the `newSeq` procedure. We'll look more at procedure calls in the next chapter but for now just know that this is also a possibility:

```
var
  e = newSeq[int]() ①
```

① Providing the type parameter inside of square brackets allows the

procedure to know that it shall return a sequence of a certain type.

A frequent error is omission of the final `()`, which must be included.

We can add new elements to a sequence with the `add` function, similar to how we did with strings. For this to work the sequence must be mutable (defined with `var`), and the element we're adding must be of the same type as the elements in the sequence.

seq.nim

```
var
  g = @['x', 'y']
  h = @['1', '2', '3']

g.add('z')   ①
echo g

h.add(g)     ②
echo h
```

① Adding a new element of the same type (char).

② Adding another sequence containing the same type.

```
@['x', 'y', 'z']
@['1', '2', '3', 'x', 'y', 'z']
```

Trying to pass different types to the existing sequences will produce an error:

```
var i = @[9, 8, 7]

i.add(9.81) # error ①
g.add(i)    # error ②
```

① Trying to add a `float` to a sequence of `int`.

② Trying to add a sequence of `int` to a sequence of `char`.

Since sequences can vary in length we need a way to get their length, for this we can use the `len` function.

```
var i = @[9, 8, 7]
echo i.len

i.add(6)
echo i.len
```

```
3
4
```

Indexing and slicing

Indexing allows us to get a specific element from a container by its index. Think of the index as a position inside of the container.

Nim, like many other programming languages, has zero-based indexing, meaning that the first element in a container has the index zero, the second element has the index one, etc.

If we want to index "from the back", it is done by using the ^ prefix. The last element (first from the back) has index ^1.

The syntax for indexing is `<container>[<index>]`.

indexing.nim

```
let j = ['a', 'b', 'c', 'd', 'e']

echo j[1]    ①
echo j[^1]   ②
```

① Zero-based indexing: the element at index 1 is `b`.

② Getting the last element.

```
b
e
```

Slicing allows us to get a series of elements with one call. It uses the same syntax as ranges (introduced in the for loop section).

If we use `start .. stop` syntax, both ends are included in the slice. Using `start ..< stop` syntax, the `stop` index is not included in the slice.

The syntax for slicing is `<container>[<start> .. <stop>]`.

indexing.nim

```
echo j[0 .. 3]
echo j[0 ..< 3]
```

```
@[a, b, c, d]
@[a, b, c]
```

Both indexing and slicing can be used to assign new values to the existing mutable containers and strings.

assign.nim

```
var
  k: array[5, int]
  l = @['p', 'w', 'r']
  m = "Tom and Jerry"

for i in 0 .. 4:    ①
  k[i] = 7 * i
echo k

l[1] = 'q'          ②
echo l

m[8 .. 9] = "Ba"    ③
echo m
```

① Array of length 5 has indexes from zero to four. We will assign a value to each element of the array.

② Assigning (changing) the second element (index 1) of a sequence.

③ Changing characters of a string at indexes 8 and 9.

```
[0, 7, 14, 21, 28]
@['p', 'q', 'r']
Tom and Barry
```

Tuples

Both of the containers we've seen so far have been homogeneous. Tuples, on the other hand, contain heterogeneous data, i.e. elements of a tuple can be of different types. Similarly to arrays, tuples have fixed-size.

The elements of a tuple are enclosed inside of parentheses.

tuples.nim

```nim
let n = ("Banana", 2, 'c')   ①
echo n
```

① Tuples can contain fields of different types. In this case: `string`, `int`, and `char`.

```
(Field0: "Banana", Field1: 2, Field2: 'c')
```

We can also name each field in a tuple to distinguish them. This can be used for accessing the elements of the tuple, instead of indexing.

tuples.nim

```nim
var o = (name: "Banana", weight: 2, rating: 'c')

o[1] = 7          ①
o.name = "Apple"  ②
echo o
```

① Changing the value of a field by using the field's index.

② Changing the value of a field by using the field's name.

```
(name: "Apple", weight: 7, rating: 'c')
```

Exercises

1. Create an empty array which can contain ten integers.

   - Fill that array with numbers 10, 20, ..., 100. (Hint: use loops)

   - 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.

2. Re-do the Collatz conjecture exercise, but this time instead of printing each step, add it to a sequence.

   - Pick a starting number. Interesting choices, among others, are 9, 19, 25 and 27.

   - 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

3. 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 (you can use the tuple `(longestLength, startingNumber)` or two separate variables)

   - Print the starting number which gives the longest sequence, and its length

Procedures

Procedures, or *functions* as they are called in some other programming languages, are parts of code that perform a specific task, packaged as a unit. The benefit of grouping code together like this is that we can *call* these procedures instead of writing all the code over again when we wish to use the procedure's code.

In some of the previous chapters we've looked at the Collatz conjecture in various different scenarios. By wrapping up the Collatz conjecture logic into a procedure we could have called the same code for all the exercises.

So far we have used many built-in procedures, such as `echo` for printing, `add` for adding elements to a sequence, `inc` to increase the value of an integer, `len` to get the length of a container, etc. Now we'll see how to create and use our own procedures.

Some of the advantages of using procedures are:

- Reducing code duplication
- Easier to read code as we can name pieces by what they do
- Decomposing a complex task into simpler steps

As mentioned in the beginning of this section, procedures are often called functions in other languages. This is actually a bit of a misnomer if we consider the mathematical definition of a function. Mathematical functions take a set of arguments (like `f(x, y)`, where `f` is a function, and `x` and `y` are its arguments) and *always* return the same answer for the same input.

Programmatic procedures on the other hand don't always return the same output for a given input. Sometimes they don't return anything at all. This is because our computer programs can store state in the variables we mentioned earlier which procedures can read and change. In Nim, the word `func` is currently reserved to be used as the more mathematically correct kind of function, forcing no side-effects.

Declaring a procedure

Before we can use (call) our procedure, we need to create it and define what it does.

A procedure is declared by using the `proc` keyword and the procedure name, followed by the input parameters and their type inside of parentheses, and the last part is a colon and the type of the value returned from a procedure, like this:

```
proc <name>(<p1>: <type1>, <p2>: <type2>, ...): <returnType>
```

The body of a procedure is written in the indented block following the declaration appended with a `=` sign.

callProcs.nim

```
proc findMax(x: int, y: int): int =   ①
  if x > y:
    return x   ②
  else:
    return y
  # this is inside of the procedure
# this is outside of the procedure
```

① Declaring procedure called `findMax`, which has two parameters, `x` and `y`, and it returns an `int` type.

② To return a value from a procedure, we use the `return` keyword.

```
proc echoLanguageRating(language: string) =   ①
  case language
  of "Nim", "nim", "NIM":
    echo language, " is the best language!"
  else:
    echo language, " might be a second-best language."
```

① The `echoLanguageRating` procedure just echoes the given name, it doesn't return anything, so the return type is not declared.

Normally we're not allowed to change any of the parameters we are given. Doing something like this will throw an error:

```nim
proc changeArgument(argument: int) =
  argument += 5

var ourVariable = 10
changeArgument(ourVariable)
```

In order for this to work we need to allow Nim, and the programmer using our procedure, to change the argument by declaring it as a variable:

```nim
proc changeArgument(argument: var int) = ①
  argument += 5

var ourVariable = 10
changeArgument(ourVariable)
echo ourVariable
changeArgument(ourVariable)
echo ourVariable
```

① Notice how `argument` is now declared as a `var int` and not just as an `int`.

```
15
20
```

This of course means that the name we pass it must be declared as a variable as well, passing in something assigned with `const` or `let` will throw an error.

While it is good practice to pass things as arguments it is also possible to use names declared outside the procedure, both variables and constants:

```nim
var x = 100

proc echoX() =
  echo x    ①
  x += 1    ②

echoX()
echoX()
```

① Here we access the outside variable `x`.

② We can also update its value, since it's declared as a variable.

```
100
101
```

Calling the procedures

After we have declared a procedure, we can call it. The usual way of calling procedures/functions in many programming languages is to state its name and provide the arguments in the parentheses, like this:

```
<procName>(<arg1>, <arg2>, ...)
```

The result from calling a procedure can be stored in a variable.

If we want to call our `findMax` procedure from the above example, and save the return value in a variable we can do that with:

callProcs.nim

```
let
  a = findMax(987, 789)
  b = findMax(123, 321)
  c = findMax(a, b)   ①

echo a
echo b
echo c
```

① The result from the function `findMax` is here named `c`, and is called with the results of our first two calls (`findMax(987, 321)`).

```
987
321
987
```

Nim, unlike many other languages, also supports Uniform Function Call Syntax, which allows many different ways of calling procedures.

This one is a call where the first argument is written before the function name, and the rest of the parameters are stated in parentheses:

```
<arg1>.<procName>(<arg2>, ...)
```

We have used this syntax when we were adding elements to an existing sequence (`<seq>.add(<element>)`), as this makes it more readable and expresses our intent more clearly than writing `add(<seq>, <element>)`. We can also omit the parentheses around the arguments:

```
<procName> <arg1>, <arg2>, ...
```

We've seen this style being used when we call the `echo` procedure, and when calling the `len` procedure without any arguments. These two can also be combined like this, but this syntax however is not seen very often:

```
<arg1>.<procName> <arg2>, <arg3>, ...
```

The uniform call syntax allows for more readable chaining of multiple procedures:

```nim
proc plus(x, y: int): int =    ①
  return x + y

proc multi(x, y: int): int =
  return x * y

let
  a = 2
  b = 3
  c = 4

echo a.plus(b) == plus(a, b)
echo c.multi(a) == multi(c, a)


echo a.plus(b).multi(c)   ②
echo c.multi(b).plus(a)   ③
```

① If multiple parameters are of the same type, we can declare their type in this compact way.

② First we add a and b, then the result of that operation (2 + 3 = 5) is passed as the first parameter to the `multi` procedure, where it is multiplied by c (5 * 4 = 20).

③ First we multiply c and b, then the result of that operation (4 * 3 = 12) is passed as the first parameter to the `plus` procedure, where it is added with a (12 + 2 = 14).

```
true
true
20
14
```

Result variable

In Nim, every procedure that returns a value has an implicitly declared and initialized (with a default value) `result` variable. The procedure will return the value of this `result` variable when it reaches the end of its indented block, even with no `return` statement.

result.nim

```nim
proc findBiggest(a: seq[int]): int =   ①
  for number in a:
    if number > result:
      result = number
  # the end of proc                     ②

let d = @[3, -5, 11, 33, 7, -15]
echo findBiggest(d)
```

① The return type is `int`. The `result` variable is initialized with the default value for `int`: `0`.

② When the end of the procedure is reached, the value of `result` is returned.

```
33
```

> ⚠️ Beware! In older Nim versions (before Nim 0.19.0), the default value of strings and sequences was `nil`, and when we would use them as returning types, the `result` variable would need to be initialized as an empty string (`""`) or as an empty sequence (`@[]`).

result.nim

```nim
proc keepOdds(a: seq[int]): seq[int] =
  # result = @[]          ①
  for number in a:
    if number mod 2 == 1:
      result.add(number)


let f = @[1, 6, 4, 43, 57, 34, 98]
echo keepOdds(f)
```

① In Nim version 0.19.0 and newer, this line is not needed — sequences are automatically initialized as empty sequences.

In older Nim versions, sequences must be initialized, and without this line the compiler would throw an error. (Notice that `var` must *not* be used, as `result` is already implicitly declared.)

```
@[1, 43, 57]
```

Inside of a procedure we can also call other procedures.

filterOdds.nim

```
proc isDivisibleBy3(x: int): bool =
  return x mod 3 == 0

proc filterMultiplesOf3(a: seq[int]): seq[int] =
  # result = @[]              ①
  for i in a:
    if i.isDivisibleBy3():   ②
      result.add(i)


let
  g = @[2, 6, 5, 7, 9, 0, 5, 3]
  h = @[5, 4, 3, 2, 1]
  i = @[626, 45390, 3219, 4210, 4126]

echo filterMultiplesOf3(g)
echo h.filterMultiplesOf3()
echo filterMultiplesOf3 i    ③
```

① Once again, this line is not needed in the newer versions of Nim.

② Calling the previously declared procedure. Its return type is `bool` and can be used in the if-statement.

③ The third way of calling a procedure, as we saw above.

```
@[6, 9, 0, 3]
@[3]
@[45390, 3219]
```

Forward declaration

As mentioned in the very beginning of this section we can declare a procedure without a code block. The reason for this is that we have to declare procedures before we can call them, doing this will not work:

```
echo 5.plus(10) # error        ①

proc plus(x, y: int): int =    ②
  return x + y
```

① This will throw an error as `plus` isn't defined yet.

② Here we define `plus`, but since it's after we use it Nim doesn't know about it yet.

The way to get around this is what's called a forward declaration:

```
proc plus(x, y: int): int      ①

echo 5.plus(10)                ②

proc plus(x, y: int): int =    ③
  return x + y
```

① Here we tell Nim that it should consider the `plus` procedure to exist with this definition.

② Now we are free to use it in our code, this will work.

③ This is were `plus` is actually implemented, this must of course match our previous definition.

Exercises

1. Create a procedure which will greet a person (print "Hello <name>") based on the provided name. Create a sequence of names. Greet each person using the created procedure.

2. Create a procedure `findMax3` which will return the largest of three values.

3. Points in 2D plane can be represented as `tuple[x, y: float]`. Write a procedure which will receive two points and return a new point which is a sum of those two points (add x's and y's separately).

4. Create two procedures `tick` and `tock` which echo out the words "tick" and "tock". Have a global variable to keep track of how many times they have run, and run one from the other until the counter reaches 20. The expected output is to get lines with "tick" and "tock" alternating 20 times. (Hint: use forward declarations.)

> ℹ️ You can press Ctrl+C to stop execution of a program if you enter an infinite loop.

Test all procedures by calling them with different parameters.

Modules

So far we have used the functionality which is available by default every time we start a new Nim file. This can be extended with modules, which give more functionality for some specific topic.

Some of the most used Nim modules are:

- `strutils`: additional functionality when dealing with strings

- `sequtils`: additional functionality for sequences

- `math`: mathematical functions (logarithms, square roots, ...), trigonometry (sin, cos, ...)

- `times`: measure and deal with time

But there are many more, both in what's called the standard library and in the nimble package manager.

Importing a module

If we want to import a module and all of its functionality, all we have to do is put `import <moduleName>` in our file. This is commonly done on the top of the file so we can easily see what our code uses.

stringutils.nim

```
import strutils        ①

let
  a = "My string with whitespace."
  b = '!'

echo a.split()         ②
echo a.toUpperAscii()  ③
echo b.repeat(5)       ④
```

① Importing strutils.

② Using `split` from `strutils` module. It splits the string in a sequence of words.

③ `toUpperAscii` converts all ASCII letters to uppercase.

④ `repeat` is also from `strutils` module, and it repeats either a character or a

whole string the requested amount of times.

```
@["My", "string", "with", "whitespace."]
MY STRING WITH WHITESPACE.
!!!!!
```

> ℹ To the users coming from other programming languages (especially Python), the way that imports work in Nim might seem "wrong". If that's the case, this is the recommended reading.

maths.nim

```
import math                    ①

let
  c = 30.0 # degrees
  cRadians = c.degToRad()    ②

echo cRadians
echo sin(cRadians).round(2)  ③

echo 2^5                      ④
```

① Importing math.

② Converting degrees to radians with degToRad.

③ sin takes radians. We round (also from math module) the result to at most 2 decimal places. (Otherwise the result would be: 0.4999999999999999)

④ Math module also has ^ operator for calculating powers of a number.

```
0.5235987755982988
0.5
32
```

Creating our own

Often times we have so much code in a project that it makes sense to split it into pieces that each does a certain thing. If you create two files side by side in a folder, let's call them `firstFile.nim` and `secondFile.nim`, you can import one from the other as a module:

firstFile.nim

```
proc plus*(a, b: int): int = ①
  return a + b

proc minus(a, b: int): int = ②
  return a - b
```

① Notice how the `plus` procedure now has an asterisk (*) after it's name, this tells Nim that another file importing this one will be able to use this procedure.

② By contrast this will not be visible when importing this file.

secondFile.nim

```
import firstFile          ①

echo plus(5, 10)          ②
echo minus(10, 5) # error ③
```

① Here we import `firstFile.nim`. We don't need to put the `.nim` extension on here.

② This will work fine and output `15` as it's declared in `firstFile` and visible to us.

③ However this will throw an error as the `minus` procedure is not visible since it doesn't have an asterisk behind it's name.

In case you have more than these two files, you might want to organize them in a subdirectory (or more than one subdirectory). With the following directory structure:

```
.
├────── myOtherSubdir
|     ├────── fifthFile.nim
|     └────── fourthFile.nim
├────── mySubdir
|     └────── thirdFile.nim
├────── firstFile.nim
└────── secondFile.nim
```

if you wanted to import all other files in your `secondFile.nim` this is how you would do it:

secondFile.nim

```
import firstFile
import mySubdir/thirdFile
import myOtherSubdir / [fourthFile, fifthFile]
```

Interacting with user input

Using the stuff we've introduced so far (basic data types and containers, control flow, loops) allows us to make quite a few simple programs.

In this chapter we will learn how to make our programs more interactive. For that we need an option to read data from a file, or ask a user for an input.

Reading from a file

Let's say we have a text file called `people.txt` in the same directory as our Nim code. The contents of that file looks like this:

people.txt

```
Alice A.
Bob B.
Carol C.
```

We want to use the contents of that file in our program, as a list (sequence) of names.

readFromFile.nim

```nim
import strutils

let contents = readFile("people.txt")  ①
echo contents

let people = contents.splitLines()    ②
echo people
```

① To read contents of a file, we use the `readFile` procedure, and we provide a path to the file from which to read (if the file is in the same directory as our Nim program, providing a filename is enough). The result is a multiline string.

② To split a multiline string into a sequence of strings (each string contains all the contents of a single line) we use `splitLines` from the `strutils` module.

```
Alice A.
Bob B.
Carol C.
                ①
@["Alice A.", "Bob B.", "Carol C.", ""] ②
```

① There was a final new line (empty last line) in the original file, which is also present here.

② Because of the final new line, our sequence is longer than we expected/wanted.

To solve the problem of a final new line, we can use the `strip` procedure from `strutils` after we have read from a file. All this does is remove any so-called whitespace from the start and end of our string. Whitespace is simply any character that makes some space, new-lines, spaces, tabs, etc.

readFromFile2.nim

```
import strutils

let contents = readFile("people.txt").strip() ①
echo contents

let people = contents.splitLines()
echo people
```

① Using `strip` provides the expected results.

```
Alice A.
Bob B.
Carol C.
@["Alice A.", "Bob B.", "Carol C."]
```

Reading user input

If we want to interact with a user, we must be able to ask them for an input, and then process it and use it. We need to read from standard input (stdin) by passing `stdin` to the `readLine` procedure.

```
echo "Please enter your name:"
let name = readLine(stdin)  ①

echo "Hello ", name, ", nice to meet you!"
```

① The type of name is inferred to be a string.

```
Please enter your name:
        ①
```

① Waiting for user input. After we write our name and press Enter, the program will continue.

```
Please enter your name:
Alice
Hello Alice, nice to meet you!
```

> **ℹ** If you are using an outdated version of VS Code, you cannot run this the usual way (using Ctrl+Alt+N) because output window doesn't allow user inputs — you need to run these examples in the terminal. With the newer versions of VS Code there is no such limitation.

Dealing with numbers

Reading from a file or from a user input always gives a string as a result. If we would like to use numbers, we need to convert strings to numbers: we again use the strutils module and use parseInt to convert to integers or parseFloat to convert into a float.

interaction2.nim

```
import strutils

echo "Please enter your year of birth:"
let yearOfBirth = readLine(stdin).parseInt()  ①

let age = 2018 - yearOfBirth

echo "You are ", age, " years old."
```

① Convert a string to an integer. When written like this, we trust our user to give a valid integer. What would happen if a user inputs `'79` or `ninety-three`? Try it yourself.

```
Please enter your year of birth:
1934
You are 84 years old.
```

If we have file `numbers.txt` in the same directory as our Nim code, with the following content:

numbers.txt

```
27.3
98.24
11.93
33.67
55.01
```

and we want to read that file and find the sum and average of the numbers provided, we can do something like this:

interaction3.nim

```nim
import strutils, sequtils, math          ①

let
  strNums = readFile("numbers.txt").strip().splitLines()  ②
  nums = strNums.map(parseFloat)          ③

let
  sumNums = sum(nums)                      ④
  average = sumNums / float(nums.len)      ⑤

echo sumNums
echo average
```

① We import multiple modules. `strutils` gives us `strip` and `splitLines`, `sequtils` gives `map`, and `math` gives `sum`.

② We strip the final new line, and split lines to create a sequence of strings.

③ `map` works by applying a procedure (in this case `parseFloat`) to each member

of a container. In other words, we convert each string to a float, returning a new sequence of floats.

④ Using `sum` from `math` module to give us the sum of all elements in a sequence.

⑤ We need to convert the length of a sequence to float, because `sumNums` is a float.

```
226.15
45.23
```

Exercises

1. Ask a user for their height and weight. Calculate their BMI. Report them the BMI value and the category.

2. Repeat Collatz conjecture exercise so your program asks a user for a starting number. Print the resulting sequence.

3. Ask a user for a string they want to have reversed. Create a procedure which takes a string and returns a reversed version. For example, if user types `Nim-lang`, the procedure should return `gnal-miN`. (Hint: use indexing and `countdown`)

Conclusion

It is time to conclude this tutorial. Hopefully this has been useful to you, and you managed to make your first steps in programming and/or the Nim programming language.

These have only been the basics and we've only scratched the surface, but this should be enough to enable you to make simple programs and solve some simple tasks or puzzles. Nim has a lot more to offer, and hopefully you will continue to explore its possibilities.

Next steps

If you want to continue learning from Nim tutorials:

- Official Nim tutorial

- Nim by example

If you want to solve some programming puzzles:

- Advent of Code: Series of interesting puzzles released every December. Archive of old puzzles (from 2015 onwards) is available.

- Project Euler: Mostly mathematical tasks.

Happy coding!

The source files are available on Github.