Nim basics

Nim basics

Table of Contents

Nimは比較的新しいプログラミング言語で，読みやすく高性能なコードを書くことが可能です。このNimチュートリアルの読者は，おそらくNimについてご存知でしょう。

このチュートリアルの原文はオンラインとPDFで，和訳はオンラインとPDFで提供されています。

このチュートリアルは作成途中です。もし誤りを見つけたり，改善案があればissue trackerに報告してください。

対象とする読者

- プログラミングの経験がないか，始めて日が浅い方
- 他の言語でのプログラミング経験がある方
- 初めてNimについて探求したいので，最初から始めたい方

読者として対象ではない 方

- 多くのプログラミング経験がある方は他のより進んだチュートリアルをお勧めします。 公式チュートリアルやNim by Example参照。
- Nimの経験がある方（このチュートリアルの改善にご助力いただければ幸いです。）

このチュートリアルの使い方

このチュートリアルの目的はプログラミングとNimの文法の基礎を示すことにより，読者が他のチュートリアルを読み，独力で進むことができるようにすることです。

書かれていることを単に読むだけでなく，自分で試してたり，例を改変したり，独自の例を考えてみたり好奇心を持てれば最善です。いくつかの章の終わりにある練習問題は，解けるように作ってありますで，飛ばさないでください。

チュートリアルの一部や練習問題について理解するために，さらに助けが必要ならNim forumやNim Gitter channel，Discord server，または Nim's IRC channel on freenode, #nim で質問してください。

インストール

Nimのインストール

Nimは3大オペレーティングシステム向けに完成された配布物が用意されており，Nimのインストールについてはいくつかの選択肢があります。

公式インストール手順に従えば最新版をインストールできます。最新の機能やバグ修正が必要なら，choosenimと呼ばれるツールを使えば簡単に安定版と最新開発版を切り替えることが可能です。

どちらを選んでも，個々のリンク先で説明されているインストール手順に従うだけでNimはインストールされるはずです。後で，インストールがうまくいったか確認します。

Linuxを使っている場合は，使用しているディストリビューションのパッケージマネージャにNimが含まれている可能性が高いです。この方法でインストールしようとする場合は，最新版かどうか確認しましょう（何が最新版か確認するにはウェブサイト参照）。最新版でない場合は，上の二つのうちのどちらかの方法でインストールしましょう。

このチュートリアルでは安定版を使います。当初このチュートリアルはNim 0.19（2018年9月リリース）に対して執筆されましたが，Nim 1.0を含む新しい版でも動くはずです。

追加のツールのインストール

Nimコードはどんなエディタでも書くことは可能で，ターミナルからコンパイルと実行ができます。構文強調やコード補完を求めるなら，人気のあるコードエディタにはそのような機能を提供するプラグインがあります。

多くのNimユーザはVS Codeを好み，構文強調とコード補完にNim extension，迅速なコンパイルと実行にCode Runner extensionを使っています。

著者はNeoVimとこのプラグイン（訳者はこちらのプラグイン）を使って構文強調やコード補完のような機能を追加しています。

他のコードエディタで利用できるエディタサポートについては，ウィキを参照してください。

インストールの確認

インストールが成功したか確認するために，入門用の例として昔から使われているHello Worldを書いてみます。

NimでHello Worldという語句を印字（スクリーンへの表示でプリンタを使った紙への印刷ではない）簡単で余計なコードは不要です。

新しいテキストファイル，例えばhelloworld.nimに1行のコードを書くだけです。

helloworld.nim

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

印字したい語句はechoコマンドの後に置いて二重引用符（"）囲む必要があります。

最初にプログラムをコンパイルしてから，走らせて期待通りに動作するか確認します。

ファイルがあるターミナルを開いてください（Linux　ではファイルマネージャで右クリックして "Open Terminal here"，ウィンドウズではシフト+ 右クリックでコマンドラインを開くメニューが出ます）。

ターミナルで次のように打鍵してプログラムをコンパイルします。

```
nim c helloworld.nim
```

コンパイルに成功した後でプログラムを実行ができます。Linuxではプログラムを実行するにはターミナルで./helloworldウィンドウズではhelloworld.exeと打鍵します。

ただ一つのコマンドでコンパイルと実行をすることもできます。次のように打鍵してください。

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

cはNimにファイルをコンパルすることを命じ，-rは直ちに実行することを命令しています。
全てのコンパイラオプションを参照するには，ターミナルでnim　--helpと打鍵してください。

上述の VSCode と Code Runner 拡張機能，Ctrl+Alt+Nを押すだけでファイルはコンパイル・実行されます。

プログラムの実行にどちらの方法を選択した場合でも，すぐに出力ウィンドウ（またはターミナル）に次のように表示されるはずです。

```
Hello World!
```

おめでとうございます。初めてのNimプログラムの実行に成功しました。

画面に何かを印字する方法（echoコマンドを利用）やプログラムのコンパイル（ターミナルでnim c programName.nimのように打鍵）し，それを実行する（複数の方法）を覚えました。

これからNimの基本要素を探索を始めます。これらは簡単なNimプログラムを書くために役立ちます。

値の命名

プログラムの中の値に名前をつけると，分かりやすくなります。ユーザの名前を聞いて後で使うために保存しておけば，再度聞き返さずに利用することができます。

例としてpi = 3.14で名前piは値3.14に結びついています。 明らかにpiの型は（十進法の）数値です。

別の例としてfirstName = Aliceではは値Aliceを持つ変数の名前です。 この変数の型は単語ということができるでしょう。

代入は，多くのプログラム言語で同様に動作します。 代入では 名前, 値, そして 型 があります。

変数の宣言

Nimは静的型付けプログラミング言語で，代入の型は値を使う前に宣言されていなければなりません。

Nimでは変更できる値を変更できないものと区別しなければなりませんが，後ほど詳しく説明します。 変数を宣言（変更可能な代入）するにはvarキーワードを使って，次のような文法により名前と型を書きます（値は後で与えられます）。

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

値がわかっているときは，変数を宣言して直ちに値を与えることができます。

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

山括弧 (<>) は変更してよいものを示しています。
従って<name>は山括弧の中の文字通りの単語nameではなくどんな名前でも構いません。

Nimは型推論機能があり，明示的に型を指定しなくてもコンパイラが自動的に値から名前代入の型を検出できます。

様々な型については次章で述べます。

型推論を使うと，次のように明示的な型なしに変数の代入ができます。

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

Nimにおける代入の例は次のようなものです。

```
var a: int  ①
var b = 7   ②
```

① 変数aはint（整数）で値は明示的に定めていません。

② 変数bは値が7でその型は整数として自動的に検出されています。

代入では，プログラムにおいて意味のある名前を選ぶことが重要です。単にa，b　，cなどとすると，すぐに混乱してきます。一つの単語よりも多くの語からなる名前を選ぶときは，通常camelCaseスタイル（最初の文字は小文字であることに注意）で書かれます。

ただし，Nimは大文字と小文字とを区別せず，アンダスコアを無視することに注意が必要です。つまり，helloWroldとhello_worldは同じです。例外として最初の文字は大文字と小文字とを区別します。変数の名前には数字やUTF-8文字を含めることができます。望むのなら絵文字も使えますが，他の人が打鍵することになる可能性に配慮する必要があります。

varをそれぞれの変数に対して打鍵する代わりに，複数の変数（必ずしも同一の型ではない）は同じvarブロックで宣言することができます。Nimではブロックは，字下げ（最初の文字の前にある同数の空白文字）が同一であるコードの一部で，既定の字下げ幅は空白2文字です。このようなブロックはNimプログラムの随所に見られ，代入に限りません。

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

Nimではタブは字下げとして認められていません。
コードエディタを設定してTabを任意の数の空白文字に置き換えることができます。
VS Codeでは既定はTabを4つの空白文字に置き換えます。変更するには，設定（Ctrl+,）で"editor.tabSize": 2とします。

前に述べたように変数は書き換え可能，つまり変数の値は何度でも変えることができますが，型は宣言したときと同一でなければなりません。

```
var f = 7           ①

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

① 変数fは初期値7でその型は推定により``int`となります。

② fの値はまず-3，次に19に変更されます。これらは両方とも整数で，元の型と同じです。

③ fの値を"Hello"に変更しようとすると，　エラーが発生します。Helloは数値ではなくので，fの型を整数から文字列にしようとしています。　<4>#　エラーはコメントです。Nimコード中のコメントは文字#の後に書きます。これに続く全ての文字は行の終わりまで無視されます。

書き換え不可の代入

キーワードvarを使って宣言された変数とは異なり，Nimにはあと2種類の代入があります。これらの値は変えることができません。一つはキーワードconst，もう一つはキーワードletを使って宣言されます。

Const

キーワードconstで宣言された書き換え不可な代入に用いられる値は，コンパイル時（プログラム実行前）に既知なければなりません。

例えば，重力加速度をconst g = 9.81あるいは円周率をconst pi = 3.14として宣言することができます。これらの値は事前に分かっていてプログラムの実行の間不変です。

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

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

① 定数の値は変えられません。

② 変数hはコンパイル時に評価されません（これは変数でその値はプログラムの実行中にかわりえます。），　従って定数iの値はコンパイル時には知り得ないので，エラーが発生します。

プログラミング言語によっては，定数の名前を``ALL_CAPS`のように全て大文字で書く習慣があります。Nimの定数は他の変数と同様に書きます。

Let

letで宣言された書き換え不可の代入は，コンパイル時に既知である必要はなく，その値はプログラムの実行中いつでも設定できますが，ひとたび設定したら値を変えることはできません。

```
let j = 35
j = -27 # □□□ ①

var k = -5
let l = k + 7    ②
```

① 書き換え不可の値は変更できません。

② constの例とは対照的に，これは動作します。

実際には，letはconstよりも多く使われます。

varを何にでも使うことはできますが，letを基本とすべきです。varは変更することになる変数のみに使いましょう。

基本データ型

整数

前章で述べたように，整数は小数部や小数点がない数値です。

例えば32, -174, 0, 10_000_000は全て整数です。3桁毎の区切りに_を用いて大きな数値を見やすくしていることに注目してください（1千万を10_000_000と書いた方が10000000と書くより分かりやすくなります)。

通常の数学演算子，加算（+），減算（-），乗算（*），そして除算（/）は期待した通りに動作します。最初の三つの演算子は整数を生成しますが，二つの整数の除算は，余りがない場合でも常に浮動小数点数（小数点のある数値）が結果として与えられます。

整数の除算（小数部が無視される除算）にはdiv演算子を使います。演算子mod（modulus，法）は整数の除算の余りを得たいときに用います。これらの演算の結果は常に整数です。

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 echoコマンドはそれに続くコンマで区切られたもの全てを画面に印字します。この場合，まず文字a + b = を印字し，続いて同じ行に式a + bの結果を印字します。

コードをコンパイルして実行すると出力は次のようになるはずです。

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

浮動小数点数

浮動小数点数（float）実数の近似表現です。

例えば2.73, -3.14, 5.0, 4e7は浮動小数点数です。大きな浮動小数点数には科学的表記を用いることに注目してください。eの後の数値は指数です。この例では，4e7は4 * 10^7を表します。

二つの浮動小数点数に対して，四つの基本数学演算子を用いることができます。演算子divとmodは浮動小数点数に対しては定義されていません。

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 加算と除算の例では，小数部がない数値が得られますが，結果は浮動小数点型であることに注意してください。

数学演算の優先順位は期待通りです。乗算と除算は加算や減算よりも優先されます。

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

```
14
22.0
```

浮動小数点数と整数の変換

Nimでは異なる数値型の変数間の数学演算が許されておらず，エラーが発生します。

```
let
  e = 5
  f = 23.456

echo e + f    # □□□
```

変数の値は同じ型に変換する必要があります。変換は簡単で整数への変換にはint関数，浮動小数点数への変換にはfloat関数を使います。

```
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 floatに変換した整数eを印字する（eは整数のまま）。

② intに変換した浮動小数点数fを印字する。

③ 被演算子は共に浮動小数点数で加算可能。

④ 被演算子は共に整数で加算可能。

```
5.0
23
28.987
28
```

int関数を使って浮動小数点数を整数に変換するときに四捨五入は行われません。単に数値から小数部が落ちるだけです。
四捨五入をするには別の関数を持ちなければなりませんが，それにはNimの使い方をもっと勉強する必要があります。

文字

char型は単一のASCII文字を表すために用いられます。

文字は二つの一重引用符（'）の間に書きます。文字はアルファベット，記号または数字です。複数の数字や文字はエラーになります。

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

文字列

文字列は文字が並んだものです。文字列の中身は二つの二重引用符　（"）の間に書きます。

文字列というと単語を思い浮かべますが，複数の単語や記号，数字を含むことができます。

strings.nim

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

① 空の文字列。

② これは数値（int）ではありません。二重引用符で囲まれているので，文字列になります。

③ これは一つの文字ですが，二重引用符に囲まれているのでcharではありません。

特殊文字

次の文字列を印字すると

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

意外な結果になります。

```
some
im  ips
```

このようになったのは，いくつかの文字には特別の意味があるからです。これらはエスケープ\を前につけて用います。

- \nは改行です。
- \tはタブです。
- \\はバックスラッシュです（\はエスケープとして用いるため）。

上の例を文字通り印字するには，二つの方法があります。

- \\を\の代わりに使うか，
- 未加工（raw）文字列を使います。文法はr"…"で　（文字rを最初の引用符の前に置きます。）　）エスケープ文字が存在せず特別な意味もなく，全てがそのまま印字されます。

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

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

上に示したもの以外にも特殊文字があり，Nim manualに示されています。

文字列の結合

Nimの文字列は書き換え可能で，中身は変わり得ます。add関数で既存の文字列に別の文字列や文字を加える（付け足す）ことができます。元の文字列を変更したくないときは，文字列を&演算子で結合する（つなげる）と新しい文字列が返されます。

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 文字列を変更する場合はvarとして宣言します。

② 既存の文字列pに別の文字列を加えると，その文字列の値が変更されます。

③ 文字列にcharを加えることもできます。

④ 二つの文字列を連結すると，元の文字列を変更せずに新しい文字列ができます。

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

真偽

真偽（またはブール，bool）データ型は二つの値trueかfalseのどちらかです。ブール型は通常制御の流れ（次章参照）に用いられ，通常関係演算子の結果です。

ブール変数に通常用いられる名前の付け方は，はい/いいえ（真/偽）の質問，例えばisEmpty, isFinished, isMovingとして表すというものです。

関係演算子

関係演算子は，比較可能な二つの事柄の関係を試すものです。

二つの値はが等値かどうか比較するには==（二つの等号）が用いられます。これを以前に示した代入に用いられる=と混同しないようにしてください。

整数に対して定義されている全ての関係演算子を示します。

relationalOperators.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
```

文字や文字列を比較することもできます。

relationalOperators.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 全ての大文字は小文字より前。

② 文字列の比較は文字毎に行われます。最初の三つの文字は同一で，文字bは文字zよりも小さい。

③ 文字が同一でなければ，文字列の長さは比較されません。

④ 短い文字列は長い文字列よりも小さい。

```
true
false
true
true
true
```

論理演算子

論理演算子は一つまたはそれ以上の真偽値からなる式が真であるか試すために用いられます。

- 論理andがtrueを返すのは，両方ともtrueの場合のみ。
- 論理orがtrueを返すのは，少なくともどちらか一方がtrueのとき。
- 論理xorがtrueとなるのは一方が真で他方がそうではないとき。
- 論理notは真偽を反転させます。つまりtrue`をfalseにまたはその逆（一つの被演算子をとる唯一の論理演算子）

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
```

関係演算子と論理演算子を組み合わせて，より複雑な式を作ることができます。

例えば，(5 < 7) and (11 + 9 == 32 - 2*6)はtrue and (20 == 20)となり，true and trueなるので，最終結果はtrueとなります。

まとめ

この章はチュートリアルの中で最長でたくさんの範囲を扱いました。時間をとって各データ型を復習し，それぞれ何ができるか試してみてください。

型は一見制約のように見えますが，Nimコンパイラがコードを速く，偶発的におかしな動作をしないように確かめることを可能にするものです。これは大きなコードベースで役に立ちます。

基本型といくつかの演算を学んだので，Nimで簡単な計算ができます。知識を次の練習問題で試してみましょう。

練習問題

1. 書き換え不可の変数を作り，あなたの年齢（年）を格納してください。年齢を日数で印字してください（1年は365日とします）。
2. あなたの年齢が3で割り切れるか試してください。（ヒント: modを使います。）
3. 書き換え不可の変数を作り，あなたの身長をセンチメートルで格納してください。身長をインチで印字してください（1インチは2.54センチ）。
4. 管の直径が3/8インチです。直径をセンチメートルで表してください。
5. 書き換え不可の変数を二つ作り，あなたの姓と名を格納してください。変数fullNameを二つの変数を連結して作ってください。姓と名の間には空白文字を入れてください。あなたのフルネームを印字してください。
6. アリスは$400を15日間毎に稼ぎます。ボブは1時間あたり$3.14稼ぎ，1日に8時間，週7日働いています。30日後，アリスはボブよりも稼いでいるでしょうか（ヒント: 関係演算子を使います）。

制御の流れ

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

```
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
```

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

```
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 elifs. 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:

```
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:

```
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

```
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

```
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

```
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

```
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.

while.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

```
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

```
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

```
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:

```
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:

```
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:

```
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:

ufcs.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

```
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
```

Note that this procedure is here to demonstrate the result variable, and it is not 100% correct: if you would pass a sequence containing only negative numbers, this procedure would return 0 (which is not contained in the sequence).

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

```
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.)

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.
!!!!!
```

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

```
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.

interaction1.nim

```
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!
```

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

```
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!

原文のソースはGithub, 和訳のソースはGithubにあります。