Nim basics

Nim basics

Table of Contents

対象とする読者 4
読者として対象ではない 方 4
このチュートリアルの使い方 4
インストール 5
  Nimのインストール 5
  追加のツールのインストール 5
  インストールの確認 5
値の命名 7
  変数の宣言 7
  書き換え不可の代入 9
    Const 9
    Let 9
基本データ型 11
  整数 11
  浮動小数点数 12
    浮動小数点数と整数の変換 13
  文字 14
  文字列 14
    特殊文字 14
    文字列の結合 15
  真偽 16
    関係演算子 17
    論理演算子 18
  まとめ 20
  練習問題 20
流れの制御 21
  If文 21
    Else 22
    Elif 23
  Case 24
繰り返し 27
  Forループ 27
  Whileループ 30
  Breakとcontinue 30
  練習問題 31
コンテナ 33
  配列 33
  シーケンス 34
  要素指定と切り出し 35
  タプル 37
  練習問題 38
手続き 39
  手続きの宣言 39

手続きの呼び出し . . . 42
Result変数 . . . 44
前方宣言 . . . 45
練習問題 . . . 46
モジュール . . . 47
モジュールのインはポート . . . 47
自前のモジュール . . . 48
ユーザ入力の取り扱い . . . 50
ファイルからの読み込み . . . 50
ユーザ入力の読み取り . . . 51
数値の取り扱い . . . 52
練習問題 . . . 53
まとめ . . . 54
次の段階 . . . 54

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について探求するため，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，次のような構文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。構文は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）。

流れの制御

これまでのプログラムは全ての行が必ず実行されていました。流れを制御する文を使うと，コードの一部がある真偽条件が成り立つ場合にのみ実行させるようにすることができます。

プログラムを道路に例えるならば，流れの制御は分かれ道であり，その一つを条件により選択します。例）もし（if）卵がある値段以下なら買う。もし（if）雨が降っていれば傘を持っていく，さもなくば（else）サングラスを持っていく。

擬似コードで書くと次のようになります。

```
if eggPrice < wantedPrice:
  buyEggs

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

Nimの文法は，以下に示すようによく似ています。

If文

上に示したif文はプログラムの中で最も簡単な分岐です。

Nimの文法でif文は以下のように書きます。

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

① conditionは真偽型でなくてはなりません。真偽値または関係演算子，論理式やその組合せ。

② ifの行に続く全ての行は二つの空白文字で字下げし，同じブロックとして条件がtrueのときにのみ実行されます。

If文は入れ子にできます。すなわち，一つのifブロックの内側に別のif文を入れることができます。

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 最初の条件が真で二番目が偽なので中echoは実行されません。

② 二つの条件が真で両方の行が印字されます。

③ 最初の条件が偽なので，このブロックの内側の全ての行は無視され，何も印字されません。

④ if文の内側の論理and。

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

Else

Ifブロックの後に続くelseを使うと，if文の条件が偽であるときに実行される分岐を作ることができます。

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文がfalseのときだけブロックを実行するには，not演算子を使って条件を否定します。

Elif

Elifは"else if"の短縮形で，複数のif文をつなげることができます。

プログラムは真となるまで文を試します。その後の全ての文は無視されます。

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
```

gの場合gは三つの条件全てを満たしますが，最初の分岐だけが実行され，後の全ての分岐は実行が省略されます。

Case

Case文は複数の可能性から一つを選択する別の方法で，複数のelifのあるif文に似ています。case文は複数の真偽条件を取らず，異なる値とそれに分岐を対応させます。

次のようなif-elifブロックで書かれたコードは

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

はcase文で次のように書けます。

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

If文とは異なり，case文はとりうる全ての場合を包含しなくてはなりません。該当しない場合があるときはelse: discardを使うことができます。

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 必要のなのはhの三つの値だけですが，とりうる全ての場合（他の全ての文字）を含めるためにこの行が必要です。これがないとコンパイルが通りません。

```
You've chosen y
```

複数の値に対して同じ動作をさせる場合は，それぞれの分岐に対して複数の値を割り当てることもできます。

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
```

繰り返し

繰り返し（ループ）は，もう一つの流れの制御構造でコードの一部を複数回実行することを可能にするものです。

この章では二種類のループを学びます。

- forループ: 決まった回数の繰り返し
- whileループ: 条件が満たされる間繰り返し

Forループ

forループの構文は

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

伝統的にiがループ変数loopVariableの名前として使われていますが，なんでも構いません。この変数はループの内側だけで使えます。ループが終わった後は，ループ変数は破棄されます。

iterableは反復可能なオブジェクトです。これまでに述べた方のうち，文字列は反復可能オブジェクトです（いろいろな反復可能な型は次章に登場します）。

loop　bodyの全ての行は反復の度に実行されるため，コードの繰り返される部分を効率的に書くことができます。

Nimで（整数型）数値の範囲にわたって反復したいときは，iterableの構文は　start　..　finishでstartとfinishは数値です。このように書くと，startとfinishの間の数値をfinishを含めて反復します。既定の反復可能な範囲では，startはfinishよりも小さくなくてはなりません。

ある数値の手前まで（その数値を含まない）反復では..<を使います。

for1.nim

```
for n in 5 .. 9:  ①
  echo n

echo ""

for n in 5 ..< 9: ②
  echo n
```

① 反復を..を使って行う場合，範囲の始端・終端の両方が含まれます。

② 同じ範囲の反復を..<を使って行うと，終端の手前までで終端は含まれません。

```
5
6
7
8
9

5
6
7
8
```

1でない刻み幅を用いて数値の範囲について反復する場合，countupが用いられます。countupには，開始値，終了値（範囲に含まれる）と刻み幅を指定します。

for2.nim

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

① 0から16まで刻み幅4で小さい方から大きい方に数えます。終端（16）は範囲に含まれます。

```
0
4
8
12
16
```

startがfinishより大きい数値の範囲にわたって反復するときは，countdownと呼ばれる類似の関数を使います。大きい方から小さい方に数えているにも関わらず，刻み幅は正です。

for2.nim

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

echo ""

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

① 大きい数値から小さい数値に向かって反復するにはcountdownを使わなければなりません。（..演算子は開始する数値が終了する数値よりも小さい場合にのみ使えます。

② 大きい方から小さい方に数えるにも関わらず，刻み幅は正なくてはなりません。

```
4
3
2
1
0

-3
-5
-7
-9
```

文字列は反復可能なので，forループを使って文字列に含まれる文字を一つづつ反復していくことができます（この手の反復はfor-eachループとも呼ばれます）。

for3.nim

```
let word = "alphabet"

for letter in word:
  echo letter
```

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

反復カウンタ（0始まり）も必要な場合は，for <counterVariable>, <loopVariable> in <iterator>:という構文が使えます。これは，ある反復可能オブジェクトについて反復し，同時に別の反復可能オブジェクトの同じ位置の値を使うときに便利です。

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ループ

Whileループはif文に似ていますが，コードのブロックを条件が真の間実行し続けます。これはループが何回実行されるか事前に分からないときに使われます。

ループがある時点で必ず終了するようにして，無限ループにならないようにします。

while.nim

```
var a = 1

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

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

① この条件は新しいループに入る度に検査され，内側のコードが実行されます。

② incはaを一つ増やします。a = a + 1またはa += 1と書くのと同じです。

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

Breakとcontinue

Break文は通常ある条件が満たされた場合に先にループから出るために使います。

次の例では，break文を伴うif文がなければ，iが1000になるまで印字が継続します。break分があると，iが3になると（値を印字する前に）直ちにループを出ます

break.nim

```
var i = 1

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

```
1
2
```

continue文は直ちにループの次の反復を実行し，現在の反復の残り行は実行されません。次のコードで3と6が出力にないことに注意してください。

continue.nim

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

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

練習問題

1. コラッツ予想は有名な数学の問題で，ルールは単純です。まず数値を選びます。奇数なら3を加えて1を足します。もし偶数なら2で割ります。この手順を1になるまで続けます。例。5 → 奇数 → 3*5 + 1 = 16 → 偶数 → 16 / 2 = 8 → 偶数 → 4 → 2 → 1 → 終了!
   整数を（書き換え可能な変数として）選びコラッツ予想の各ステップを印字するプログラムを書いてください。（ヒント。divを割り算に使います。）
2. 書き換え不可の変数を作り，フルネームを格納します。Forループ書いて文字列を要素について反復し母音（a, e, i, o u）だけ印字してください（ヒント。case文を使って分岐毎に複数の値します。
3. Fizz　　buzzは子供の遊びで基本的なプログラミングの知識を試すためにも使われます。数は1から増える方向に数えます。数値が3の倍数なら　fizz，5の倍数なら_buzz_

で置き換え，もし数値が15（3と5両方）なら_fizzbuzz_で置き換えます。最初の数回は次のようになります。1, 2, fizz, 4, buzz, fizz, 7, …
Fizz　buzzの最初の30回を印字するプログラムを書いてください（ヒント。除算の順序のテストに注意してください。

4. 前の問題でインチをセンチメートルにまたはその逆に変換しました。複数の値の変換表を作ってください。表の例を示します。

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

コンテナ

コンテナは個々の要素の集まりを格納する型で，要素に対するアクセスができます。通常コンテナは反復可能で，繰り返しの章での文字列と同じように使うことができます。

例えば，日用品のリストは買いたい品目のコンテナで，素数のリストは数のコンテナです。擬似コードでは次のようになります。

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

配列

配列は最も単純なコンテナ型です。配列は一様，すなわち配列内の全ての要素が同じ型です。配列の長さは一定，つまり要素の数（正確には要素の最大の数）はコンパイル時に既知でなくてはなりません。したがって，配列を「長さが一定の一様コンテナ」と呼ぶことができます。

配列の宣言はarray[<length>, <type>]とします。ここでlengthは配列全体の大きさ（収容可能な要素の数）typeは配列全ての要素の型です。宣言を省略できるのは，長さと型が共に渡された要素から推定できるときです。

配列の要素は角括弧で囲みます。

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

① 値を与えると，bの長さと型はコンパイル時に既知です。配列aのように具体的に宣言することは正しいのですが，その必要はありません。

② この宣言では長さも型も推定できないので，エラーとなります。

③ 空の（後で要素を満たす）配列を宣言する正しい方法は長さと型を与え，要素の値を与えません。配列dは七つの文字列を可能できます。

配列の長さはコンパイル時に既知でなければならないので，次は動作しません。

```
const m = 3
let n = 5

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

① これがエラーとなるのはnがletを使って宣言されているので，その値はコンパイル時には分からない空です。配列の初期化にlengthとして使えるのはconstだけです。

シーケンス

シーケンスは配列に似たコンテナですが，長さはコンパイル時に既知である必要はなく，実行中に変えることができます。含まれる要素の型だけをseq[<type>]で指定します。シーケンスは一様，すなわちシーケンスの全ての要素は同じ型でなくてはなりません。

シーケンスの要素は@[と]で囲みます。

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

① 空のシーケンスの型は宣言しなくてはなりません。

② 空でないシーケンスの型は推定できます。この場合は，文字列を格納するシーケンスになります。

空のシーケンスを初期化する別の方法は，newSeq手続きを呼ぶことです。手続き呼び出しは次章で学びますが，ここではこのような方法もあることを覚えておきましょう。

```
var
  e = newSeq[int]() ①
```

① 型パラメタを角括弧の間に与えて，特定の型のシーケンスが返されるようにします。
よくあるミスは最後の()を忘れることです。これは必ず付けなくてはなりません。

シーケンスに要素を付け加えるにはadd関数を使います。文字列に対して行ったことと似ています。追加を行うには，シーケンスは書き換え可能である（varで定義する）必要があり，追加される要素はシーケンスと同じ型でなくてはなりません。

seq.nim

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

g.add('z')  ①
echo g

h.add(g)    ②
echo h
```

① 同一の型（char）である新しい要素を追加。

② 同一の型を含む別のシーケンスを追加。

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

既存のシーケンスに異なる型を渡そうとすると，エラーが発生します。

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

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

① intのシーケンスにfloatを渡そうとしています。

② charのシーケンスにintを渡そうとしています。

シーケンスの長さは変化するので長さを知る方法が必要です。長さはlen関数を使うと得られます。

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

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

```
3
4
```

要素指定と切り出し

要素指定により，インデックスによりコンテナから特定の要素を取り出すことができます。インデックスはコンテナ内の位置です。

Nimは他の多くのプログラミング言語と同様に0始まりのインデックスづけを採用しています。つまり，コンテナの最初の要素のインデックスは0で，二番目の要素のインデックスは1などとなります。

末尾からのインデックスには ^ を前に置きます。最後の要素（末尾から一つ目）のインデックスは ^1 です。

要素指定の構文は<container>[<index>]です。

indexing.nim

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

echo j[1]   ①
echo j[^1]  ②
```

① 0始まりの要素指定なでインデックス1の要素はb。

② 最後の要素を取得。

```
b
e
```

切り出しでは一度に連続した要素が得られます。切り出しは（forループの節で学んだ）範囲と同じ構文を使います。

start .. stopという構文を使うと両端は切り出しに含まれます。これに対し，start ..< stopではインデックスstopは切り出しには含まれません。

切り出しの構文は<container>[<start> .. <stop>]です。

indexing.nim

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

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

要素指定と切り出しは，既存の書き換え可能なコンテナと文字列に値を代入するときに使えます。

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 長さ5の配列のインデックスは0から4までです。配列のそれぞれの要素に値を代入しています。

② シーケンスの二番目の要素（インデックス1）に対して代入（書き換え）をしています。

③ 文字列のインデックス8と9の文字を書き換えています。

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

タプル

これまでに学んだ二つのコンテナはどちらも一様でした。これに対し，タプルは非一様データを格納します。つまり，タプルの要素は異なっていても構いません。配列同様にタプルは固定長です。

タプルの要素は丸括弧で囲みます。

tuples.nim

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

① タプルは異なる型のフィールドを含めることができます。この場合はstring，int，そしてcharです。

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

タプルのフィールドには区別するための名前を付けることができます。名前を付けると，インデックスでなく名前でタプルの要素にアクセスできます。

tuples.nim

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

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

① フィールドの値をフィールドのインデックスで変更。

② フィールドの値をフィールドの名前で変更。

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

練習問題

1. 10個の整数を格納できる空の配列を作ってください。
    - 数値10, 20, …, 100で配列を入れてください。（ヒント: ループを使います。）
    - 奇数のインデックスである配列の要素だけを印字してください。（値は20,　40, …）。
    - 偶数インデックスの要素を5枚してください。変更された配列を印字してください。
2. コラッツ予想の練習問題に再度取り組みます。ただし，今回は各段階で印字するのではなく，シーケンスに追加します。
    - 最初の数を選びます。面白い値には9, 19, 25や27などがあります。
    - 最初の数だけを要素とするシーケンスを作ります。
    - 前と同じロジックを使い，1にたどり着くまでシーケンスに要素を追加し続けます。
    - シーケンスの長さとシーケンス自体を印字します。
3. 2から100までの中で最も長いコラッツ数列を見つけてください。
    - 範囲内のそれぞれの数に対してコラッツ数列を計算します。
    - これまでの記録よりも現在のシーケンスが長ければ，現在の長さと最初の値を新しい記録として保存します。タプル(longestLength, startingNumber)または独立した二つの変数を使います。
    - 最長のシーケンスを与える最初の数とシーケンスの長さを印字します。

手続き

手続き，又は他のプログラミング言語で関数とも呼ばれており，特定の作業をするコードの一部を一つの単位にまとめたものです。このようにコードをまとめる利点は，手続きに書いてある内容のコードを再度書き直す代わりに手続きを呼び出すことができることです。

前の複数の章では，コラッツ予想にいくつかの方法で取り組みました。コラッツ予想のロジックを手続きにまとめていれば，同じコードをコラッツ予想の全ての練習問題で使うことができたはずです。

これまで多くの組込手続きを使ってきました。例えば，echoで印字し，addで要素をシーケンスに追加し，incで整数の値を増加させlenでコンテナの長さを取得するなど。ここでは，自前の手続きを作成し，利用する方法を理解します。

手続きを使う利点はいくつかあります。

- コードの重複を防ぐ。
- 何をするかに基づいてコードの断片に名前を付けることでコードを読みやすくする。
- 複雑な作業を単純な段階に分解する。

この節の最初に述べたように，手続きは他の言語で関数とも呼ばれています。この定義は数学での関数の定義を考えると，呼び誤りとも言えます。数学の関数は引数の組を取り（f(x,　y)のように。ここでfは関数でxとyはその引数），常に同一の入力に対して同一の答えを返します。

プログラムの手続きは，常に同じ値を返すとは限りません。時には，返り値がないこともあります。その理由は，コンピュータプログラムは状態を前に述べた変数に保存でき，手続きはこれを読み書きできるからです。Nimではfuncは現在数学的により正確な，副作用がないという制約が課された関数を表すために使うために確保されています。

手続きの宣言

自前の手続きを使う（呼び出す）前に，作成し何をするか定義する必要があります。

手続きはprocキーワードを用いて宣言され，手続き名と確固で囲まれた入力パラメタとのそれらの型が続き，最後の部分はコロンと手続きから返される値の型であり，次のようになります。

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

手続き本体は，==記号が後ろについた宣言の後に続いて，インデントしたブロックに書きます。

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 この宣言の手続きの名前はfindMaxで，二つのパラメタxとyをとり，int型を返します。

② 値を手続きから返すためにreturnキーワードを使います。

```
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 このechoLanguageRating手続きは与えられた名前を単におうむ返しするだけで，何も返さないので返り値の型は定義されていません。

既定ではどのパラメタも変更することはゆるさていないので，次のように書くとエラーが発生します。

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

var ourVariable = 10
changeArgument(ourVariable)
```

動作するようにするには，Nimと手続きを使うユーザが引数を変更できるように，変数として宣言する必要があります。

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

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

① 注目すべきは，今度はargumentが　var　intとして宣言されていて，単なるintではないことです。

```
15
20
```

当然手続に渡す名前も変数として宣言されていなくてはならず，constやletが付与されているものはエラーが生じます。

引数で渡すことは良い慣行ですが，手続きの外で宣言された変数と定数を使うことも可能です。

```
var x = 100

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

echoX()
echoX()
```

① ここで外部変数xにアクセスします。

② 変数として宣言されているので，値を更新することもできます。

```
100
101
```

手続きの呼び出し

手続きを宣言したら，呼び出しができます。多くのプログラミング言語で手続きや関数を呼びだす一般的な方法は名前を示し，次のように引数を括弧で囲むことです。

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

手続き呼び出しの結果は変数に保存することができます。

findMax手続きを上の例から呼び返り値を変数に保存するには，次のようにします。

callProcs.nim

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

echo a
echo b
echo c
```

① 関数findMaxの返り値はここでcと名付けられ，最初の二つの呼び出しの返り値を使って呼び出されます （findMax(987, 321)）。

```
987
321
987
```

Nimは他の多くの言語とは異なり，統一関数呼び出し構文が使えます。この構文を使うと，多様な呼び出し方が可能になります。

この構文では，最初の引数が関数名の前に書かれ，残りの引数を括弧の中に入れます。

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

既にこの構文を使って，要素を既存のシーケンスに追加しました（<seq>.add(<element>)）。この方がadd(<seq>, <element>)と書くよりも読みやすく，意図とが明確になります。引数の括弧を省略することも可能です。

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

このようなスタイルがecho手続きを呼び出すときや，len手続きを引数なしに呼び出すときに使われていることを見てきました。これら二つを組み合わせることも可能ですが

，見かけることはあまり多くありません。

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

統一呼び出し構文を使うと，複数の手続きをつなげるときに読みやすい書き方ができます。

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 複数のパラメタの型が同じであれば，型の宣言を簡潔に書くことができます。

② まずaにbを加え，演算の結果（2 + 3 = 5）multi手続きの最初の引数として与え，cを掛けます（5 * 4 = 20）。

③ まずcとbを掛けて，演算の結果（4 * 3 = 12）をplus手続きの最初の引数として与えaを足します（12 + 2 = 14）。

```
true
true
20
14
```

Result変数

Nimでは，値を返す全ての手続きはresult変数が暗黙に宣言され，（既定値で）初期化されます。手続きは，インデントされたブロックの末尾に到達したらreturn文がなくてもresult変数を返します。

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 返り値の型はintでresult変数はintの既定値0で初期化されます。

② 手続きの末尾に到達すると，resultの値が返されます。

```
33
```

この手続きはresult変数を説明するためのもので，100%正確ではありません。負の値だけを格納したシーケンスを渡すとこの手続きは（シーケンスに含まれない）0を返します。

注意!　Nimの古い版（Nim　0.19.0より前）文字列とシーケンスの既定値はnilだったので，返り値の型として使うときにはresult変数はs（""）または空シーケンス（@[]）として初期化する必要がありました。

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 Nimバージョン0.19.0以降ではこの行は不要。シーケンスは自動的に空シーケンスとして初期化されます。
Nimの古い版ではシーケンスを初期化しなければならず，この行がないとコンパイルエラー発生がしました（resultは既に暗黙に宣言されているので，varは使ってはなりません。）

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

手続きの中で別の手続きを呼び出すことができます。

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 ここも新しいNimの版では不要。

② 以前に宣言された手続きの呼び出し。返り値の型はboolでif文で使うことが可能。

③ 上述の手続きを呼び出す第3の方法。

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

前方宣言

この節の一番最初に述べたように，コードブロックを使わずに手続きを宣言することができます。その理由は手続きを呼び出す前に宣言する必要があるからです。次は動作しません。

```
echo 5.plus(10) # error      ①

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

① plusがまだ定義されていないので，エラーを生じます。

② ここでplusを定義していますが，使った後なのでNimはまだ知りません。

これを回避するには，前方宣言と呼ばれるものを使います。

```
proc plus(x, y: int): int    ①

echo 5.plus(10)              ②

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

① ここでNimにplusで続きが存在し，定義がこの通りであることを知らせます。

② 手続きを自由に使うことができ，これは動作します。

③ ここでplusが実装されます。当然前にした定義と一致していなければなりません。

練習問題

1. 与えられた名前に対して挨拶する（"Hello　<name>"を印字する）手続きを作ってください。名前のシーケンスを作り，この手続きを使ってそれぞれに人に挨拶してください。
2. 三つの値のうちで最大のものを返す手続きfindMax3を作ってください。
3. 2次元平面の点はtuple[x, y: float]で表されます。二つの点を受け取り，二つの点の合計となる新しい点を返す手続きを書いてください（xとyは独立に足します）。
4. "tick"と"tock"という単語を印字するtickとtockという手続きを作ってください。何回走ったか記録する大域変数を用意し，交互にカウンタが20に達するまで走らせてください。期待される出力は"tick"と"tock"が20回交互に繰り返されるというものです。（ヒント: 前方宣言を使います）。

無限ループに入ってしまったときは，Ctrl+Cを押してプログラムの実行を止めることができます。

全ての手続きを異なるパラメタで実行してください。

モジュール

これまでは，新しいNimのファイルを作ると既定で使える機能を学んできました。モジュールを使うと既定から拡張して，特定の目的のために用いる機能を使いすることができます。

よく用いられるNimのモジュールを挙げます。

- strutils: 文字列を扱う追加の機能
- sequtils: シーケンスに対する追加の機能
- math: 数学関数（対数，平方根 …), 三角関数（sin, cos, …）
- times: 時間を測定したり扱ったりする。

標準ライブラリやnimbleパッケージマネージャにはもっとたくさんあります。

モジュールのインはポート

モジュールをインポートして機能を全て使いたいときはimport <moduleName>をファイルに書きます。通常ファイルの先頭に書いて，コードが何を使うか分かるようにします。

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 strutilsをインポート。

② strutilsモジュールからsplitを使い，文字列を単語の列に分割します。

③ toUpperAsciiは全てのASCII文字を大文字にします。

④ repeatもstrutilsモジュールからで。文字や文字列全体を指定回数繰り返します。

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

他のプログラミング言語（特にPython）のユーザは，Nimkのインポートの動作は「誤り」に見えるかもしれません。その場合はこちらをご覧ください。

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 mathをインポート。

② 度をラジアンに変換するのにdegToRadを使います。

③ sinにはラジアンを与えます。結果を小数第二位で丸めるために（これもmath由来の）roundを使います（そうしないと結果は0.4999999999999999のようになるでしょう）。

④ Mathモジュールの ^ 演算子は数値の冪乗の計算につかいます。

```
0.5235987755982988
0.5
32
```

自前のモジュール

コードがプロジェクトのコードが多くなり，特定のことをする部品に分けることが適切であることがよくあります。二つのファイルを一つのフォルダの中に作り，firstFile.nimとsecondFile.nimと名付けた場合，一方から他方は次のようにインポートできます。

firstFile.nim

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

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

① plus手続きには，名前の後にアステリスク（*）　がついていることに注目してください。これはNimにこのファイルをインポートした別のファイルはこの手続きを使えることを伝えています。

② 対照的にこの手続きは，このファイルをインポートしたファイルからは見えません。

secondFile.nim

```
import firstFile          ①

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

① ここでfirstFile.nimをインポートします。ここには.nim拡張子は不要です。

② firstFileに定義されており見えるので，これは問題なく動作し15が印字されます。

③ しかしこれはエラーが発程します。minus手続きは名前の後ろにアステリスクが付いていないので見えないからです。

これら二つのファイルよりももっとあるときは一つ（またはそれ以上の）サブディレクトリに整理することができます。ディレクトリ構造が次のようであるとします。

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

secondFile.nimにその他の全てのファイルをインポートするには次のようにします。

secondFile.nim

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

ユーザ入力の取り扱い

これまでに学んだ項目（基本データ型とコンテナ，流れの制御，ループ）を使えば多数の簡単なプログラムが書けます。

この章ではプログラムをより対話型にする方法を学びます。そのためには，ファイルからデータを読み込んだり，ユーザに入力を求めたりする手段が必要となります。

ファイルからの読み込み

people.txtという名前のファイルがNimコードと同じディレクトリにあったといます。ファイルの中身は次のようなものです。

people.txt

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

プログラム中でファイルの中身を名前のリスト（シーケンス）として使います。

readFromFile.nim

```
import strutils

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

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

① ファイルの中身を読むにはreadFile手続きを使い，読むファイルのパスを与えます（Nimのプログラムと同じディレクトリにある場合はファイル名だけで十分です）。読み込み結果は複数行の文字列になります。

② 複数行の文字列を文字列のシーケンス（個々の文字列は一行の全ての内容を含みます）に分割するには，strutilsモジュールのsplitLinesを使います。

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

① 元のファイルには最後の改行（最後が空行）があり，ここにも残っています。

② 最後の改行のために，シーケンスは想定よりも長くなっています。

最後の改行の問題を解決するためには，strutilsのstrip手続きをファイルを読んだ後に使

います。これは，文字列の最初と最後の空白文字と呼ばれるものを除去します。空白文字は，スペースを作る，スペースや改行，タブなどが含まれます。

readFromFile2.nim

```
import strutils

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

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

① stripによって期待通りの結果が得られます。

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

ユーザ入力の読み取り

ユーザと対話するには，入力を求めて処理し，利用する必要があります。標準入力（stdin）から読むためreadLine手続にstdinを渡します。

interaction1.nim

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

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

① nameの型は文字列と推定されます。

```
Please enter your name:
①
```

① ユーザの入力を待ちます。名前を書いてEnterを押すとプログラムは再開します。

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

古い版のVS Codeだといつもの方法 （Ctrl+Alt+N）で動作しません。出力ウィンドウではユーザの入力ができないからです。ターミナルで例を実行する必要があります。
VS Codeの新しい版ではこのような制限はありません。

数値の取り扱い

ファイルやユーザ入力を読んだ結果は常に文字列になります。数値として使うには，文字列を数値に変換す必要があります。ここでもstrutilsモジュールを使い，parseIntで整数またはparseFloatで浮動小数点数にします。

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 文字列を整数に変換。このように書いた場合ユーザは有効な整数を入力するものと想定しています。ユーザが'79またはninety-threeと入力したらどうなるでしょうか。試してみてください。

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

Nimコードと同じディレクトリにファイルnumbers.txtがあり，次のような中身だとします。

numbers.txt

```
27.3
98.24
11.93
33.67
55.01
```

このファイルを読んで，与えられた数値の合計と平均を求めるには，次のようにします。

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 複数のモジュールをインポートします。strutilsはstripとsplitLines，sequtilsはmap，そしてmathはsumを提供します。

② 最後の改行を取り除き，行を分割して文字列のシーケンスを作ります。

③ mapは手続（この場合はparseFloat）をコンテナのそれぞれのメンバに適用します。それぞれの文字列を浮動小数点数にし，新しい浮動小数点数のシーケンスを返します。

④ mathモジュールのsumを使ってシーケンスの全ての要素の和を求めます。

⑤ sunNumsは浮動小数点数なので，シーケンスの長さを浮動小数点数に変換する必要があります。

```
226.15
45.23
```

練習問題

1. ユーザに身長と体重を尋ねてください。BMIインデックスを計算します。BMIの値と分類を報告ユーザにしてください。
2. コラッツ予想の練習問題で，今度はユーザに最初の数を聞くプログラムを作ってください。結果のシーケンスを印字してください。
3. 反転してほしい文字列を聞いてください。文字列を取り反転したものを返す手続きを作ってください。例えば，ユーザがNim-langと入れたら手続きがgnal-miNと返すようにします。（ヒント: 要素指定とcountdownを使います。）

まとめ

このチュートリアルを締めるときが来ました。チュートリアルが読者に役に立つもので，読者がプログラミングやNimプログラミング言語の第一歩を踏み出せたということを期待しています。

扱った内容は基礎であり上部を撫でただけにすぎませんが，読者は簡単なプログラムを書き簡単な作業や問題を解くことに十分なものです。Nimはもっと多くを提供しているので，引き続きその可能性を探求してください。

次の段階

Nimをチュートリアルで学ぶには

- Official Nim tutorial
- Nim by example

プログラミングの問題を解くには

- Advent of Code: 毎年12月に発表される面白い問題のシリーズ。古い問題（2015年以降）のアーカイブがあります。
- Project Euler: 主に数学の問題。

コード書きを楽しみましょう!

原文のソースはGithub, 和訳のソースはGithubにあります。