第I部

必要最小限のプログラミング

第1章　数の計算と関数

数式の計算は、コンピュータの発明以前から人間が行ってきた情報処理の1つであるが、プログラミングと共通する性質を多く持っている。まずは我々には馴染み深い数式の計算をコンピュータに行わせる方法を見ることで、プログラミングの世界への第一歩を踏み出そう。

1.1　コンピュータとの対話

本書では、irb と呼ばれるソフトウェアを用いて実際にプログラムを動かしながら様々な考え方が概念を学んでゆくことになる。irb はプログラミング言語 Ruby を対話的に使用するためのプログラミング環境である。対話的な環境とは、人間が質問や指示を1つ出すごとにコンピュータが即座に処理を行って返答するものなので、あたかも会話をするかのようにプログラミングを進めることができる。そのためプログラミングの学習やソフトウェア開発を試行錯誤しながら行うことに適している。

まずは irb を起動して簡単な数式の計算を行わせる方法を見る。なお、ここでは使用するコンピュータに irb がすでにインストールされていることを前提としている。インストール方法や OS による具体的な起動手順などは以下のページを参照せよ。

http://lecture.ecc.u-tokyo.ac.jp/johzu/joho-kagaku/

多くの OS では、irb を使って計算をさせる場合には次のような手順を踏むことになる:

ターミナルの起動 ⇒ irb の起動 ⇒ 数式の入力

まず最初に「ターミナル」を起動する。このソフトウェアは OS に対する命令をキーボードから入力するためのものである。

起動後はウインドウが開かれ、キーボードから入力できる状態になったことを示す**プロンプト**と呼ばれる文字列が表示される。

```
1 cm12345$ ▉
```

この画面では cm12345$がプロンプトであるが、使っている OS やコンピュータによって異なる文字列が表示される。なお ▉は文字が入力されてゆく位置を示す**カーソル**であり、これの形状も OS やコンピュータによって異なる。

OS ごとに異なるソフトウェアを参照する必要がある場合、本書では Mac OS X 日本語版における名称を用いる。「ターミナル」はオペレーティングシステムによって「コマンド プロンプト」や「Terminal」といった名前が付いている。

次にキーボードから irb Return と入力する。これは「irb という名前のソフトウェアを起動せよ」という OS に対する命令である。irb もまたキーボードから入力して計算を行うソフトウェアなので、今度は irb のプロンプトが表示される。なお、以降では Return を入力することを ↵ で表わす。

キーボードによっては Enter と書かれているかも知れない。

```
1 cm12345$ irb↵
2 irb(main):001:0> 
```

上ではキーボードからの入力に下線を引いてある。「irb(main):001:0>」という部分が irb のプロンプトである。

ここで適当な数式を入力すると、その式が計算され結果が表示される。例えば 1+1 ↵ と入力してみよう。

```
2 irb(main):001:0> 1+1 ↵
3 => 2
4 irb(main):002:0> 
```

上の面画の 3 行目が結果で、=>に続く「2」が計算結果である。次の行に再びプロンプトが表示され、さらに別の式が入力できる状態になっている。他にも式を入力して計算が行われることを確認してみよ。

実行画面やプログラムの行番号は、欄の左側に書かれている。

irb を終了するには、irb のプロンプトが表示されている状態で Control D を入力する。

Control あるいは Ctrl と書かれたキーを押しながら D を入力する。

```
4 irb(main):002:0> Control D
5 cm12345$ 
```

ターミナルから irb を起動した場合はターミナルのプロンプトが表示されるので、再び irb を起動したり、ターミナルを終了させることができる。

1.2　数式の計算

1.2.1　電卓がわり

まずはどのような数式の計算ができるか、例を見てゆこう。

```
1 irb(main):003:0> 7 - 2  # 引き算
2 => 5
3 irb(main):004:0> 7 * 2  # 掛け算
4 => 14
5 irb(main):005:0> 7 / 2  # 割り算の商
6 => 3
7 irb(main):006:0> 7 % 2  # 割り算の余り
8 => 1
```

```
irb(main):007:0> 7 ** 2 # 冪乗
=> 49
```

上の画面で数式の後ろに「# 引き算」などと書かれているが、この部分は説明のために書き加えた部分であり、実際には入力する必要はない。ただし # から後ろは無視されるので、入力しても問題はない。

ここに示した式の意味は説明するまでもないものがほとんどだろうが、いくつか注意が必要である。まず、$\times, \div$ のような記号や x^y のような書き方をキーボードから直接行うことができないため、*, /, %, ** などの記号を使っている。また、5 行目の割り算の結果は 3.5 とはならず、小数点以下を切り捨てた整数商になっている。

もう少し複雑な式の例も見ておこう。長い式を書いたときには、どこから計算するかによって結果が違ってしまうが、括弧を省略した場合は人間が手で計算するときと同様の、「乗除算は加減算よりも優先」などの決まりに従って計算される。

```
irb(main):009:0> 7 - 2 * 3
=> 1
irb(main):010:0> (7 - 2) * 3
=> 15
```

数の計算について一つだけ注意しなければいけないことがある。それは、整数どうしを / によって割ると結果は整数商になるが、Ruby では除数か被除数のどちらか一方でも実数であった場合は、結果も実数になるということである。

```
irb(main):012:0> 7.0 / 2
=> 3.5
irb(main):013:0> 7   / 2.0
=> 3.5
```

つまり Ruby では**整数 1 と実数 1.0 は違う計算結果を与えることがある**ので、特に実数の割り算を行いたい場合には注意せよ。

練習 1.1 (数式の計算) 次の計算を行う Ruby の式と、その結果を書け。

a) 56 と 97 と 33 の和

b) 47 引く 38

c) 35 掛ける 22

d) 34 を 15 で割った商

e) 34 を 15 で割った余り

f) 34 割る 15 (実数で)

g) 56 の 16 乗 (**を使って)

h) 56 の 16 乗 (**を使わずに)

i) 野球選手清原和博の Body Mass Index (BMI). 身長は 188 cm 体重は 104 kg とせよ。また BMI とは体重 (kg) を身長 (m) の自乗で割った値である。

(章末の練習問題: 1.3)

1.2.2　数学関数

三角関数や対数などの数学関数の計算には、**関数**を使った計算をする。

Ruby では**メソッド**と呼ばれているが、本書ではより一般的な名称である関数という言い方を用いる。

Ruby では数学関数を使う場合は `include(Math)` という命令を実行しておく必要がある。

```
19 irb(main):003:0> include(Math)      #数学関数を使う準備
20 => Object
21 irb(main):004:0> sqrt(2)
22 => 1.4142135623731
23 irb(main):005:0> cos(3.141592/3)
24 => 0.50000018867511
```

上の例の 19 行目が、数学関数を使うための準備である。21 行目は $\sqrt{2}$ の計算、23 行目は $\cos(\pi/3)$ の計算である。三角関数に与える角度は 0° から 360° を 0 から 2π に対応させたラジアンによって与えていることに注意せよ。

Ruby で使える代表的な数学関数には、以下のものがある。

関数の使い方	意味	関数の使い方	意味
`sin(x)`	$\sin x$	`exp(x)`	e^x
`cos(x)`	$\cos x$	`log(x)`	$\log x$
`tan(x)`	$\tan x$	`log10(x)`	$\log_{10} x$
`sqrt(x)`	$\sqrt{x}$		

(章末の練習問題: 1.4)

1.3　変数

1.3.1　値に名前をつける

ここまでで見た数式は、例えばBMIを求める式 `104 / (188 / 100.0) ** 2` のようにすべて数値だけを使って表わしていた。このような式は、変数を

使って一般化して「身長 h cm, 体重 w kg の BMI を $\frac{w}{(h/100.0)^2}$ と定める」のように書くことができる。

プログラミング言語でも、**変数**を使って数式を書くことができるので、その方法を見てみよう。変数を使うためには、まず変数を定義する。

```
irb(main):003:0> h=188.0
=> 188.0
irb(main):004:0> w=104.0
=> 104.0
```

上の `h=188.0` のような命令は `h` という名前の変数を定義して、その値を 188 にしている。これを**変数代入**あるいは単に**代入**という。

変数を使った式は次のようになる。

```
irb(main):006:0> w / (h/100.0) ** 2
=> 29.4250792213671
```

このように、数値のかわりに変数名を書くだけである。これを**変数の参照**という。

(章末の練習問題: 1.5 1.6 1.7)

1.3.2　変数を使うわけ *

数式 `104.0 / (188.0/100.0) ** 2` を変数を使って `w / (h/100.0) ** 2` のように書き換えても、やっている演算は変わらない。それにも関らず、変数を使って数式を書き直すことにはどのような理由があるのだろうか。

式の意味が理解しやすくなる

一つには `104.0 / (188.0/100.0) ** 2` という式は、たとえこれが BMI の計算だと知っていたとしても、その中の 104.0 や 188.0 という値が何を表わしているかは即座には分からない。しかし `w / (h/100.0) ** 2` であれば、`w` が体重 (weight) で `h` が身長 (height) であることは名前から容易に想像が付くだろう。つまり、変数を使うと式に現われる値の意味が分かりやすくなるということだ。

さらに Ruby を含むほとんどのプログラミング言語では、`w` のかわりに `weight`, `h` のかわりに `height` などのように、2 文字以上の名前の変数を使うことができる。変数が表わす値の意味をよりはっきりさせる必要がある場合には、`body_weight_in_pound` (ポンドで表わした体重) のように単語を下線記号 (_) でつなげた名前の変数を使うこともできる。この下線記号は、アルファベットと同様の扱いを受けるので、`body_weight_in_pound` で一つの変数になる。

値を変えた計算のやり直しができる

変数を使うと、式の一部の値を変化させた場合の計算、つまり計算のやり直しが簡単にできる。例えば同じ身長で体重が 10 kg 減ったの場合の BMI を求めることを考えよう。変数を使った式の場合、該当する変数 w を定義し直せば、あとは同じ式を計算するだけでよい。

```
7  irb(main):008:0> w=104.0-10
8  => 94.0
9  irb(main):009:0> w / (h/100.0) ** 2      #前と同じ式
10 => 26.5957446808511
```

なお irb では↑, ↓キーを押すと以前に入力した式が画面に現われる。9 行目では、実際には式を入力せず、↑キーを何回か押して 5 行目の式を表示させて計算させることができる。

1.4　関数の定義

1.4.1　BMI を求める関数

色々な人の BMI を計算しなければいけなくなったとしよう。数学であれば身長 height と体重 weight が与えられたときの BMI を

$$\mathrm{BMI}(\mathrm{height}, \mathrm{weight}) \equiv \frac{\mathrm{weight}}{(\mathrm{height}/100)^2} \tag{1.1}$$

と定義する、と言っておけばあとは $\mathrm{BMI}(188, 104)$ とか $\mathrm{BMI}(174, 119\times 0.454)$ のような式を書くことができる。

Ruby では関数を定義することでこれと同じことができる。

```
1  irb(main):003:0> def bmi(height,weight)
2  irb(main):004:1>     weight / (height/100.0) ** 2
3  irb(main):005:1> end
4  => nil
```

上の画面では def と end が関数を定義していることとその範囲を表わしていて、1 行目が式 1.1 中 ≡ の左側、2 行目が右側に対応する。

このようにして定義した関数は数学関数 (第 1.2.2 節) と同様に使うことができる。

```
5  irb(main):007:0> bmi(188.0,104.0)
6  => 29.4250792213671
7  irb(main):008:0> 1.1*bmi(174.0, 119.0 * 0.454)
8  => 19.6289470207425
```

練習 1.2 (関数の定義) 次のような関数を定義せよ。さらに、それを使った計算の例を示せ。

a) 平面上の 2 点 (x, y) と (u, v) の距離を求める `distance(x,y,u,v)`.

b) f フィート i インチをセンチメートルに変換する `feet_to_cm(f,i)`. ただし、1 フィート = 12 インチ = 30.48 cm である。

c) p ポンド o オンスをキログラムに変換する `pound_to_kg(p,o)`. 1 ポンド = 16 オンス = 0.4536 kg である。

(章末の練習問題: 1.8)

1.4.2　関数を使う関数

第 1.4.1 節で定義したような関数は、数学関数と同じように使えるので、別の関数の定義の中でも使うことができる。例えば、身長 f フィート i インチ、体重 p ポンド o オンスに対する BMI を計算する `bmi_yp(f,i,p,o)` を定義することを考えよう。すでに関数 `bmi`, `feet_to_cm` (練習 1.2b), `pound_to_kg` (練習 1.2c) は定義済みなので、これらを使って次のように定義すればよい。

```
1 irb(main):010:0> def bmi_yp(f,i,p,o)
2 irb(main):011:1>   bmi(feet_to_cm(f,i),
3 irb(main):012:2*       pound_to_kg(p,o))
4 irb(main):013:1> end
5 => nil
```

これでヤード・ポンド法で表わされた身長・体重からも BMI を求めることができる。例えば身長 5 フィート 11 インチ、体重 170 ポンドと書かれているシアトル・マリナーズのイチローであれば次のように計算される。

```
6 irb(main):015:0> bmi_yp(5,11,170,0)
7 => 23.7103429969605
```

(章末の練習問題: 1.9)

1.4.3　局所変数 *

三角形の 3 辺の長さ a, b, c が分かっているとき、その面積は Heron の公式により $\sqrt{s(s-a)(s-b)(s-c)}$ となることが知られている。ただし $s = \frac{1}{2}(a+b+c)$ である。

これを Ruby の関数として定義するときは、変数代入を定義の内側に書いて次のようにできる。

```
1 def heron(a,b,c)
2     s = 0.5*(a+b+c)
3     sqrt(s * (s-a) * (s-b) * (s-c))
4 end
```

ここでは 2 行目で変数 s に $\frac{1}{2}(a+b+c)$ の値を代入し、3 行目はその s を使った式になっている。変数 s は関数 heron の中だけで使われるものなので**局所変数**と呼ばれる。

上の定義の s を展開してしまえば、同じ計算を局所変数を使わずに定義することはできる。しかし局所変数を使えば、式がより簡単になり、局所変数の名前によって式の各部分の意味がより分かりやすくなり、また、同じ計算を何度も行うことが避けられるといった効果がある。

(章末の練習問題: 1.10 1.11)

1.5　ファイルに保存した関数定義

1.5.1　ファイルからの読み込み

キーボードから入力した関数定義は、irb を終了するとすべて消えてしまう。本格的なプログラムを作る場合には、テキストファイルにプログラムを書き、それを irb に読み込ませて実行する。

関数 bmi の例で手順を説明する。まずテキストエディタを使って、次のような内容を入力し、bmi.rb というファイル名で保存する。

この定義の中で、先頭の#で始まっている行は、何を定義しているかの覚え書きであり、プログラムとしては無視される。

```
1 # BMI of a person with height (cm) and weight (kg)
2 def bmi(height,weight)
3   weight / (height/100.0) ** 2
4 end
```

ファイル 1.1: bmi.rb

ファイルを保存するディリレクトリ (フォルダ) は、irb を起動したディレクトリにしておく。ファイル名に付ける.rb という拡張子は、内容が Ruby プログラムであることを示すためのものである。

定義をファイルに保存したら、irb に対してファイルから定義を読み込む命令を与える。

> 注意:　以下では、ファイルから定義を読み込めたことを確認するために、一度 irb を終了して、起動しなおしてから実行している。

```
irb(main):002:0> Control D                    # 一度irbを終了
cm12345$ irb                                  # irbを再起動
irb(main):003:0> load("./bmi.rb")             # ファイル読込
=> true
```

関数 load はファイルから関数定義を読み込むもので Ruby にはじめから用意されている。この関数を、ファイル名を"〜"で囲んだ引数とともに呼び出すと、そのファイル中にある関数定義があたかもキーボードから入力されたかのごとく読み込まれる。ファイル名の先頭には、現在のディレクトリを示す「./」を書く必要がある。

定義が読み込まれていれば、その後は bmi 関数を使うことができる。

```
irb(main):005:0> bmi(188.0, 104.0)
=> 29.4250792213671
```

もし読み込むファイルが irb を起動したディレクトリに保存されていなかったり、ファイル名が違っていた場合は下のような画面になる。この場合は、もう一度ファイル名やファイルのある場所を確認してみよう。

```
irb(main):009:0> load("./bmi.rb")
LoadError: no such file to load -- ./bmi.rb
        from (irb):9:in `load'
        from (irb):9
```

1.5.2　ファイルを読み込むファイル *

第 1.4.2 節で紹介したように、関数定義の中で他の関数を使うことがある。例えばフィートとポンドによる BMI 計算をする bmi_yp は bmi, feet_to_cm, pound_to_kg という 3 つの関数を使っている。このような bmi_yp の定義をファイルに書いてから読み込む場合には、bmi, feet_to_cm, pound_to_kg の定義も読み込んでおかなければならない。

そこで本書では、関数定義をファイルに書く際に、使っている関数を読み込む命令を一緒に書くことにする。つまりある関数 (bmi_yp) を定義するときに、別の関数 (bmi, feet_to_cm, pound_to_kg) を使っていたら、それらの関数が定義されているファイル (bmi.rb, yardpound.rb) を読み込む命令を、ファイルの先頭に書く。

このようにしておけば、bmi_yp を使う場合には、load("./bmi_yp.rb") という命令を実行するだけで必要な定義がすべて読み込まれるようになる。

```
1 load("./bmi.rb")
2 load("./yardpound.rb")
3
4 def bmi_yp(f,i,p,o)
5   bmi(feet_to_cm(f,i), pound_to_kg(p,o))
6 end
```

ファイル 1.2: bmi_yp.rb

1.5.3　誤りの指摘 (エラーメッセージ)

式・命令・定義に誤りがあると、irb は**エラーメッセージ**を表示する。ここまで読み進めた読者はすでに何度か経験済みであろう。誤りの種類は多岐に渡るためすべてを解説することはできないが、典型的な例だけでも見ておこう。

irb に対してキーボードから入力した式や命令が間違っていた場合は、即座にエラーメッセージが表示される。例えば bmi という関数名を誤って bm1 と書いてしまった場合は、次のようなメッセージが表示される。

```
1 irb(main):013:0> bm1(188.0,104.0)
2 NoMethodError: undefined method `bm1' for main:
     Object
3         from (irb):13
```

上の画面の 2 行目は、「メソッド (関数のこと) が見つからない誤り (NoMethodError): bm1 というメソッドは未定義 (undefined)」という意味である。3 行目はその誤りが起きた場所がキーボードからの入力 (irb) の 13 行目に該当する」ということを示している。なお、main:Object というのは変数やメソッドを定義している場所の名前であり、ここでは無視してよい。

またファイル中の定義に誤りがある場合には、それを読み込むときにエラーメッセージが表示されることがある。例えば次のような定義を書いたとしよう。

```
1 def bmi(height,weight)
2   weight / (height/100.0 ** 2
3 end
```

ファイル 1.3: mistake1.rb

この定義をよく見ると、2 行目の (が閉じていない。これを読み込むと次のようなエラーメッセージが表示される。

```
irb(main):003:0> load("./mistake1.rb")
SyntaxError: ./mistake1.rb:3: syntax error,
   unexpected kEND, expecting ')'
        from (irb):3
```

このエラーメッセージの意味は次のようになる。「文法誤り (SyntaxError): mistake1.rb というファイルの 3 行目付近での文法誤りで、) が現われるべき (expecting) ところに、予想外にも (unexpected) end が現われた」ということである。実際の誤りは mistake1.rb の 2 行目にあるのだが、3 行目まで読み込んだ時点でようやく) が抜けていることに気付いたのでこのようなメッセージになっている。このような場合には () の対応関係に問題があるらしいので、その点に注意してプログラムを確認する。

場合によっては、ファイルを読み込む時にはエラーが報告されず、読み込んだ関数を呼び出した時にエラーメッセージが表示されることもある。

```
def bmi(height,weight)
  weight / (hight/100.0) ** 2
end
```

ファイル 1.4: mistake2.rb

このファイル中の定義の誤りがどこだか分かるだろうか。ともかく読み込んでみよう。

```
irb(main):005:0> load("./mistake2.rb")
=> true
```

上のように関数 load を使って読み込んでもエラーメッセージは表示されない。ところが、読み込んだ関数 bmi を使ってみると次のようなメッセージが表示される。

```
irb(main):022:0> bmi(188.0,104.0)
NameError: undefined local variable or method `
   hight' for main:Object
        from ./mistake2.rb:2:in `bmi'
        from (irb):22
```

このメッセージは「名前間違い (NameError): hight という名前の局所変数 (local variable) またはメソッド は未定義 (undefined) である」という意味で、発見場所は上の画面の 5 行目に書かれている情報から./mistake2.rb というファイルの 2 行目の bmi という関数の中である、ということが分かる。

実際、mistake2.rb の定義の 2 行目を見てみると、height を hight と書き間違っていることが分かるだろう。

(章末の練習問題: 1.12)

1.5.4　定数関数

関数 load は関数定義を読み込むことができるが、代入命令によって定義された変数は読み込めない。例えば BMI の計算でいつも h=188.0 と w=104.0 という値を使うので、次のようなファイルを作って読み込んでも変数 h や w の値は読み込まれない。(正確には読み込まれるのだが消えてしまう。)

代入を関数 bmi の定義と一緒に書いておいて、そのファイルを load を使って読み込んでも h や w は定義されない。

```
1  # BMI of a person with height (cm) and weight (kg)
2  def bmi(height,weight)
3    weight / (height/100.0) ** 2
4  end
5  h=188.0          # これらの代入命令はloadされたときに
6  w=104.0          # は効果がない
```

ファイル 1.5: bmi.rb (bmi.rb に代入を追加した後—正しくない例)

代わりの方法として使えるのが**定数関数**を定義することである。例えば BMI の計算であれば、ファイル 1.6 の k_height, k_weight のように引数を 1 つも受け取らずに毎回同じ結果を返すような関数を定義しておく。

```
1  # BMI of a person with height (cm) and weight (kg)
2  def bmi(height,weight)
3    weight / (height/100.0) ** 2
4  end
5  def k_height() #K 選手の身長
6    188.0
7  end
8  def k_weight() #K 選手の体重
9    104.0
10 end
```

ファイル 1.6: bmi.rb (定数関数を追加した後—正しい例)

5 行目や 8 行目の最後にある () は、引数を受け取る変数名を書く所だが、1 つも引数を受け取らないのでこのようになっている。

このファイルを読み込めば、関数 k_height, k_weight を呼び出すことで具体的な数値を書く必要がなくなる。

```
1 irb(main):004:0> load("./bmi.rb")
2 => true
3 irb(main):005:0> k_weight()
4 => 104.0
5 irb(main):006:0> bmi(k_height(), k_weight())
6 => 29.4250792213671
```

これらの関数は引数を 1 つも受け取らないので、呼び出す際には k_height() のように、「空の引数」を書くことになる。

もちろん 188.0 と書いた方が k_height() と書くよりも短いので、このやり方ではキーボードの入力回数を減らしたことにはならない。しかし、名前の付いた定数関数を使うことによって具体的な値を覚えておく必要がなくなり、数値の書き間違いを防ぐこともできる。特に円周率やプランク定数のような桁数の多い数値を使う場合には、キーボードから何桁も入力することは大変なので、定数関数が活躍する。

1.6　定義のまとめ

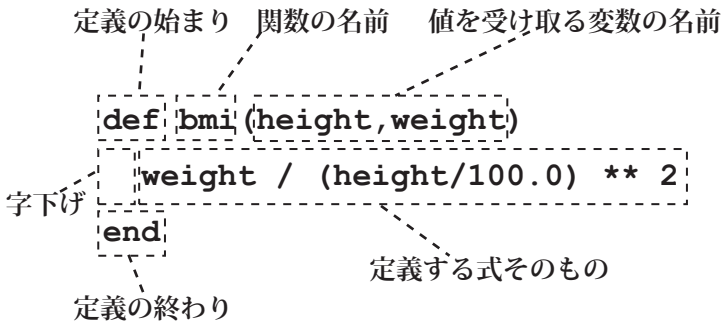
この章で紹介した式や命令をまとておこう。

include(Math)： cos や sqrt などの数学関数を使う前に、実行しておかなければいけない命令。

変数名 = **式**： 右辺の式を計算した値を、左辺に書かれた名前の変数にしまう**代入命令**。

関数名(式$_1$,..., 式$_n$)： 式$_1$,..., 式$_n$ を計算した値を、関数に渡す**関数呼び出し式**。関数に渡される式の値のことを**引数**(ひきすう)という。関数名 は sqrt のような予め定義されている関数でも、bmi のような自分で定義した関数でもよい。

def **関数名**(変数名$_1$,..., 変数名$_n$) ↵ 式 ↵ end： **関数定義**。少し複雑なので、具体例で説明する。



`def` と `end` は、定義の範囲を示している。**関数の名前**(ここでは `bmi`) は、変数名と同様、好きな名前を付けることができる。次の `(height,weight)` は、関数が値を受け取るために使う変数名である。

関数を「定義する式そのもの」では `height`, `weight` という変数を使うことができる。またこの式の先頭には、数文字の空白を入れて `def` と `end` の対応を見つけやすくしている。これは**字下げ**といって、必ずしも必要ではないのだが、見やすさのために読者も行うようにして欲しい。

`load("`**ファイル名**`")`: ファイルから関数定義を読み込む命令。`irb` を起動したディレクトリに置かれているファイル名を読み込む場合には `"./bmi.rb"` のように先頭に「`./`」を付ける。

1.7　章末問題

練習 1.3 (もっと数式の計算) 次の計算を行う Ruby の式と、その結果を書け。

a) 平成 16 年の 19 歳の日本人の平均身長・体重は以下の通りであった。それぞれの BMI を求めよ。

「平成 16 年国民健康・栄養調査報告」より。

	身長 (cm)	体重 (kg)
男	171.2	59.9
女	158.7	51.9

b) セ氏 22 度を華氏温度に変換した値。ただしセ氏温度 C と華氏温度 F は $F = \frac{9}{5}C + 32$ という関係

c) 華氏 50 度をセ氏温度に変換した値

d) 東京大学の授業料 535800 円の米ドル換算値。ただし 1 米ドルは 116.16 円とせよ

2008 年 8 月現在もの。`http://www.u-tokyo.ac.jp/stu01/e03_j.html` より。

e) Stanford 大学の授業料 12010 ドルの円換算値。

2008-9 年度のもの。`http://registrar.stanford.edu/students/finances/` より。

f) 中国におけるビッグマックの値段 11 元の日本円換算値。ただし 1 米ドルは 7.512 元とせよ。

g) 2005 年に米国に上陸したハリケーン Katrina の最大風速毎時 175 マイルの毎秒メートル換算値。1 マイルは 1609 メートルである。

h) 10 年後の日本の国債残高の 1 人あたりの額。ただし、日本が発行している普通国債の平均利率は 1.41%, 残高は 5414584 億円だとせよ。また、利息は複利法で加わり、追加発行も返済もしないとする。日本人の人口は 1.26 億人とする。

財務省「普通国債の利率別現在額と残高合計に占める割合（平成 20 年 3 月末）及び利率加重平均」より。

練習 1.4 (数学関数) 以下の計算を行う式と結果を書け。(注: 複数の関数を組み合わせなければいけない場合もある。)

a) $\sqrt{10},\ (\sqrt{2}\sqrt{5}), \sqrt{\sqrt{5}}\ \sqrt[12]{2}$

b) $\sin 30°,\ \cos 30°,\ \tan 30°$

c) $\log 1000,\ \log_{100} 100,\ \log_2 1000$

d) 2.7^{10} (** を使わずに)

e) テクネチウム 99m は、原子 1 個あたり 1 秒間に 0.0000320362394891 個の割合で崩壊する。半減期を求めよ。

練習 1.5 (変数の利用) 練習 1.3 で考えたそれぞれ式について、後から変化させることがありそうな値がどれかを考えよ。そのような値のかわりに変数を使って式を書き直せ。

練習 1.6 (代入) 次の計算を代入を使って行え。

a) $x = 10$, $y = x(x-3)$, $z = y(y-3)$ としたときの $z(z-3)$ の値。またもし変数を使わずに式を展開したらどのような式になるか。

b) 以下は 2 次方程式に関する一連の計算である。

   (a) $a = 3,\ b = 5, c = -7$ とする。

   (b) 二次方程式 $ax^2 + bx + c = 0$ の判別式の値を d とする。

   (c) p, q を $ax^2 + bx + c = 0$ の 2 つの解とする。

   (d) このときの $ap^2 + bp + c$ および $aq^2 + bq + c$ の値。

練習 1.7 (代入の右辺の式) 代入の右辺は `r=1/116.16` のように数式であってもよいし、その数式に変数が使われていてもよい。右辺が数式だった場合は、その式を計算した値が変数に代入される。

さて、米国 the Centers for Disease Control and Prevention の報告では 2002 年の 20 歳から 74 歳の米国人の平均身長、体重は以下の通りであった。

男女それぞれの BMI を計算せよ。このとき BMI の計算式自体は上と同じものになるように変数 h, w には cm, kg に換算した値を代入するように命令を書け。

	身長	体重
男	5 フィート 9.5 インチ	191 ポンド
女	5 フィート 4 インチ	164.3 ポンド

練習 1.8 (もっと関数の定義) 次のような関数を定義せよ。さらに、それを使った計算の例を示せ。

a) セ氏温度 c を華氏温度に変換する `celsius_to_fahrenheit(c)`.

b) 華氏温度 f をセ氏温度に変換する `fahrenheit_to_celsius(f)`.

c) 毎秒メートル (m/s) で表わされた速度 v を毎時マイルに変換する `ms_to_mph(v)`.

d) 上と逆の変換をする `mph_to_ms(v)`.

e) 米国では気温 (華氏 t 度) 風速 (v 毎時マイル) を加味した体感温度 (Wind Chill Index) を次のような式で定めている。

$$35.74 + 0.6215\mathtt{t} - 35.75(\mathtt{v}^{0.16}) + 0.4275\mathtt{t}(\mathtt{v}^{0.16})$$

この変換をする関数 `wind_chill_index(t,v)`. (例えば 2007 年の 10 月 20 日のニューヨークは華氏 20 度、風速 20 毎時マイルであった。)

f) 上で定義されている体感温度を、気温がセ氏、風速が m/s で与えて、結果もセ氏で求めたい。そのような関数 `wind_chill_index_celsius(t,v)`.

g) 二次方程式 $\mathtt{a}x^2 + \mathtt{b}x + \mathtt{c} = 0$ に関して

(a) 判別式 $\mathtt{b}^2 - 4\mathtt{ac}$ を求める `det(a,b,c)`.

(b) 解の 1 つ $\frac{-\mathtt{b}+\sqrt{\mathtt{b}^2-4\mathtt{ac}}}{2\mathtt{a}}$ を求める `solution1(a,b,c)`. (`det` を使って定義せよ。)

(c) もう 1 つの解 $\frac{-\mathtt{b}-\sqrt{\mathtt{b}^2-4\mathtt{ac}}}{2\mathtt{a}}$ を求める `solution2(a,b,c)`. (`solution1` と `solution2` の共通部分を 1 つの補助関数にできるか?)

(d) 二次関数 $f(\mathtt{x}) = \mathtt{a}\mathtt{x}^2 + \mathtt{b}\mathtt{x} + \mathtt{c}$ の値を求める `quadratic(a,b,c,x)`.

練習 1.9 (ヤード・ポンド法 BMI の完成) 練習 1.2b と練習 1.2c の定義を `yardpound.rb` というファイルに書け。新たに起動した `irb` で `load("./bmi_yp.rb")` を実行した後で `bmi_yp` を使った計算ができることを確めよ。

練習 1.10 (局所変数の有効範囲 *) 第 1.4.3 節で定義した関数 `heron` の外で変数 s が使われていた場合、関数呼び出しの後の s の値はどうなっているだろうか。次の命令と式を順に実行・計算し、関数定義の中と外で使われている変数 s の関係について説明を試みよ。

```
s = 123
heron(20,30,40)
s
```

練習 1.11 (局所変数と補助関数 *) ヒラメキの良い市谷くんは局所変数を定義せずに heron と同じ計算をする関数が定義できたと主張している。

青山さん:「疑わしいわね。s の定義を展開したんじゃないの?」

市谷くん:「そんなことはないさ。0.5*(a+b+c) という計算は一度しかやらないよ。補助的な関数を定義するのが鍵なんだ。」

市谷くんが定義した heron はどのようなものか、作ってみよ。

練習 1.12 (わざと間違ってみる) 第 1.5.3 節で紹介した他にも、誤りの種類は数多くある。実際にエラーメッセージを見てから誤りを探すことは時として簡単でない。練習のために、意図的に誤りを混入させたファイルを作り、どのようなエラーメッセージが表示されるかを観察しておけ。後日、同じようなエラーメッセージが表示されたときの原因究明に役立つはずである。例えば、次のような誤りを含んだファイルを作り、読み込ませてみよ。

a) ファイル 1.1 の 1 行目の # 記号を取り除いたもの。

b) 空白の挿入。定義の中の適当な位置に空白文字を入れてみよ。場所によっては誤りになるし、場所よってはならない。

c) 改行の挿入。定義の中の適当な位置で改行してみよ。場所によっては誤りになるし、場所よってはならない。

d) end の不足。定義の終わりを示す end を取り除いてみよ。できれば、複数の関数定義があるファイルで、最初の定義の end を取り除いてみよ。

e) いわゆる全角文字、つまり、日本語文字コードが混ざったもの。例えば空白を全角空白に置きかえたものや、数字を全角の数字に置きかえたものを試みよ。