ます。

- まず、Python というプログラミング言語の基本機能から学びます。これらの機能は、現代のすべてのプログラミング言語が備えているものですから、次にどの言語を使うことになったとしても、学んだことは活かせます。
- プログラミングについて系統的 / 方法論的に考える方法も学んでいきます。特に、複雑な問題を単純な問題に分割するためにはどうすればよいか、それら単純な問題の解決方法を組み合わせて複雑なアプリケーションを作るためにはどうすればよいかを学びます。
- プログラミングの生産性を引き上げるツールや大きな問題を解決するために役に立つツールを紹介します。



# 2章 Python入門

プログラムは、コンピュータが理解できるコマンドから構成されています。これらのコマンドは**文**(statement。ステートメントとも言います)と呼ばれ、コンピュータは文を**実行**(execute)します。この章では、Python の文の中でもっとも単純なものを説明し、それらを使った基本算術演算のしかたを明らかにします。この章の内容は、それ自体はあまり面白い話ではありませんが、この後で説明するあらゆるものの基礎となっていますので大切です。

## 2.1 全体の構図

プログラミングをしているときに何が起きるのかを理解するためには、コンピュータ上でプログラムがどのように実行されるかについて基礎的なことを理解しなければなりません。コンピュータ自体は、命令を実行したり算術演算をしたりすることができる**プロセッサ**(processor)、**ハードディスク**(hard drive)などのデータ格納場所、ディスプレイ、キーボード、ネットワークカードといったさまざまなハードウェア部品から組み立てられています。

これらの部品を相手にするために、すべてのコンピュータは、何らかの形で Microsoft Windows、Linux、Mac OS X のような**オペレーティングシステム**(operating system)を実行しています。オペレーティングシステム(OS)はプログラムの 1 つですが、ハードウェアに対する直接のアクセスが認められている唯一のプログラムだという点で、非常に特殊なプログラムでもあります。たとえば、コンピュータ上の他のプログラムが画面に絵を描いたり、キーボードで押されたキーを判別したり、ハードディスクからデータを取り出してきたりしようとするときには、OS に要求を送ります(図 2.1 参照)。

このようなやりかたは遠回りに感じられるかもしれませんが、こうすると、たとえばネットワークカードのモデルの違いについて注意しなければならないのは、OS を書く人だけになります。他のすべての人々(科学データを分析する人や 3D 仮想チャットルームを作る人)は、OS を経由して仕事をする方法だけを学べば、自分のプログラムをさまざまなハードウェアで正しく動作させることができます。

もっとも、今の説明は 25 年前のプログラマがしていたことです。今日では、プログラマとコンピュータのハードウェアにもう 1 つの階層を入れるのが普通です。Python、Java、Visual Basic

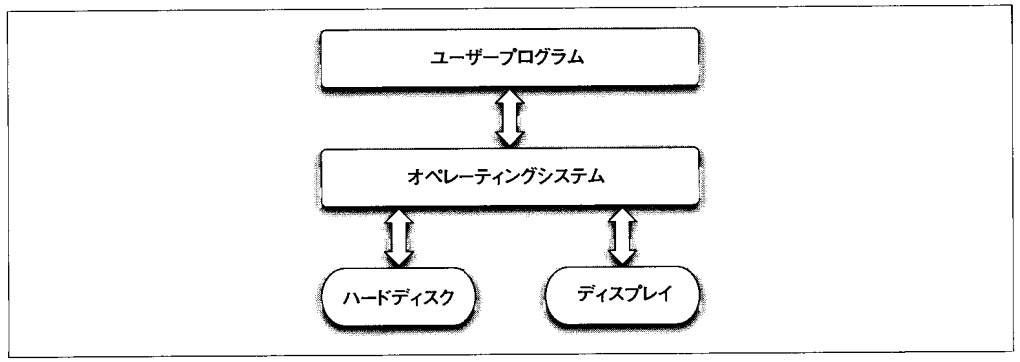


図2.1 オペレーティングシステムとのやり取り

などでプログラムを書く場合、それらのプログラムはOSのすぐ上で実行されるわけではありません。**インタープリタ**(interpreter)とか**仮想マシン**(virtual machine)といった他のプログラムがあなたのプログラムを受け取り、その中のコマンドをOSが理解できる言語に翻訳して実行します。こうすると、OSのすぐ上で直接プログラムを書く方法と比べて簡単、安全で移植性も高くなります。

しかし、インタープリタがあるだけでは不十分です。世界とやり取りするための手段が必要です。たとえば、**シェル**(shell)と呼ばれるキャラクタベースのプログラムを実行すれば、そのような手段が得られます(図2.2参照)。シェルは、キーボードからコマンドを読み出し、要求されたことを実行し、出力をテキストで表示します。これらすべてを1つのウィンドウの中で実行するのです。シェルの中には、OSとのやり取りのためのものもありますが、さまざまなプログラミング言語とやり取りするためのものもあります。この章では、Pythonシェルを使ってPythonの世界を探検していきます。

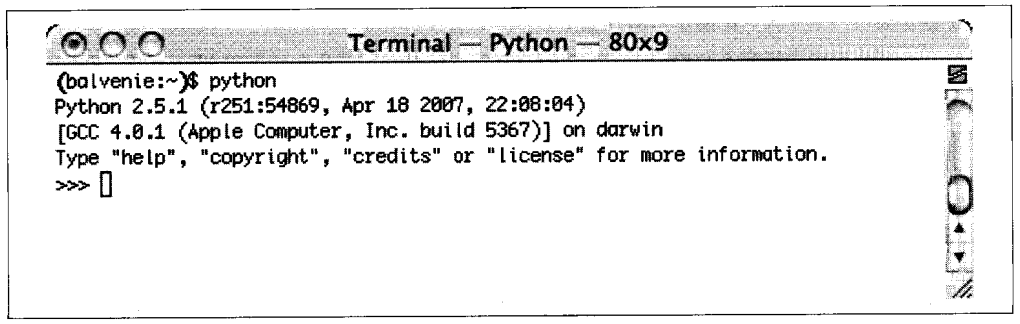


図2.2 Pythonシェル

IDE(integrated development environment:統合開発環境)は、Pythonとのやり取りのしかたとしてシェルよりも新しく、優れています。IDEは、Webブラウザ、ワープロ、描画プログラムと同じように、メニューやウィンドウを駆使した本格的なグラフィカルインターフェイスを完備しています。

プログラム演習用のIDEとして私たちが気に入っているのは、Wing 101というものです。これ



はプロ用のツールの「ライト」(フリー)バージョンです†。

Pythonにバンドルされている IDLE も、よい IDE です。私たちが Wing 101 を選んだのは、初心者プログラマ用に設計されているからで、IDLE も優れた開発環境です。

図2.3は、Wing 101のインターフェイスです。上部は、Pythonプログラムを書き込む編集ペーンです。下部の「Python Shell」と書かれている部分は、Pythonプログラムの断片を試すためのスペースです。上のペーンは「4章 モジュール」に進んだら使っていくことにして、さしあたりはシェルを使っていきます。

シェルの >>> の部分は、私たちに何かを入力せよと促している(prompt)ということで、**プロンプト**と言います。

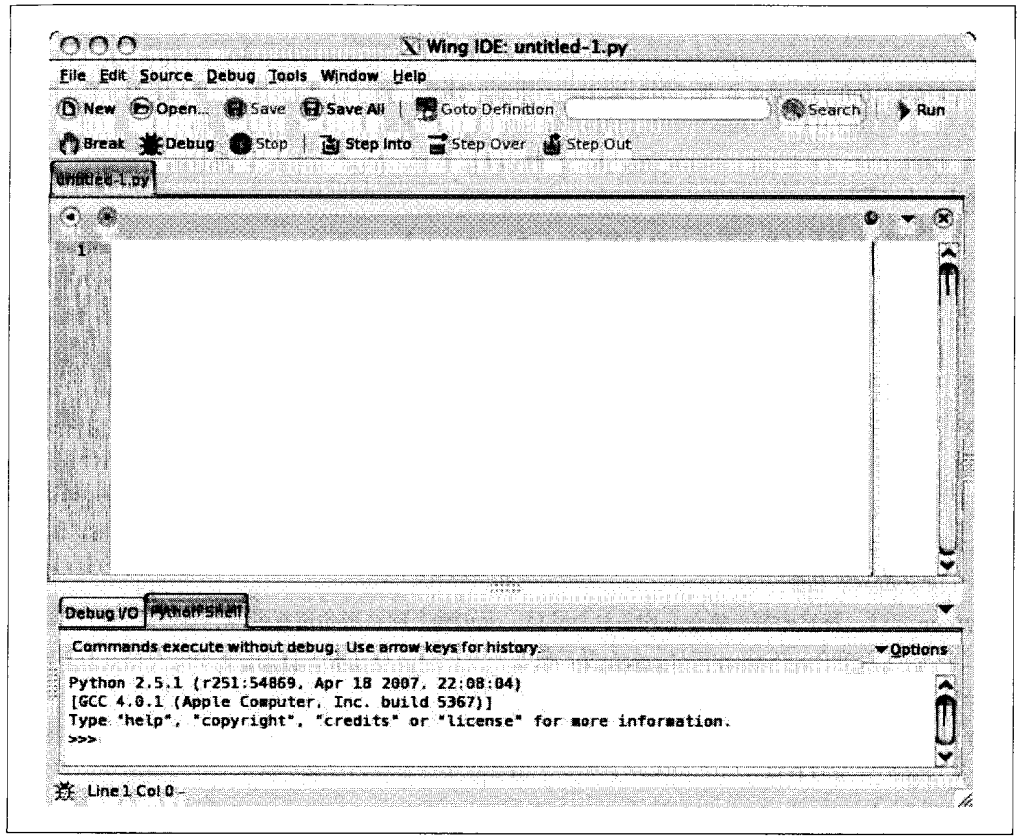


図2.3 Wing 101のインターフェイス

† 詳しくは、http://www.wingware.com を参照して下さい。

# 2.2 式

この章の最初に学んだように、Pythonのコマンドは**文**(statement)と呼ばれます。文には**式**(expression)と呼ばれるタイプのものがあります。3 + 4とか2 - 3 / 5といった数式は皆さんにもおなじみでしょう。このような数式は、2とか3 / 5といった値と、+や-のような**演算子**(operator)から構成されます。演算子は、**被演算子**(operand)をさまざまなやりかたで結合します。

Pythonは、プログラミング言語の常として、このような基本的な数式を**評価**(evaluate)できます。たとえば、次の式は、4と13を加算します。

basic/addition.cmd

```
>>> 4 + 13
17
```

式を評価すると、1つの結果が作られます。先ほどの式、4 + 13は、17という結果を生んでいます。

## 2.2.1 int型

4 + 13が17だと言われても、誰も驚きはしないでしょう。しかし、コンピュータは、いつもあなたが小学校で学んだやりかたで計算をするとは限りません。たとえば、17を10で割ったときにどうなるかを見てみましょう。

basic/int_div.cmd

```
>>> 17 / 10
1
```

1.7になると思ったはずですが、Pythonは1だと言っています。なぜでしょうか。Pythonではすべての値が**型**(type)を持っており、値を結合したときにどのようなふるまいになるかは値の型によって決まるからです。

Pythonでは、特定の型の値から作られた式は、元の値と同じ型の値を生み出します。たとえば、17と10は整数です。Pythonでは、int型の値という言い方をします。intをintで割ると、結果もintになります。

Pythonが整数式を四捨五入していないことに注意して下さい。四捨五入しているのなら、結果は2になっていなければなりません。しかし、結果は中間結果の**底値**(floor。その値よりも大きくない整数の中で最大のもの)になっています。割り算の余りが必要なら、Pythonの剰余(モジュロ)演算子(%)が使えます。

### Python 3.0の除算

Pythonの最新バージョン(Python 3.0および3.1)では、5 / 2は2ではなく2.5になります。しかし現在のところ、Python 3は旧バージョンほど普及していませんので、本書のサンプルでは「クラシックな」ふるまいの旧バージョンを使います。



basic/int_mod.cmd

```
>>> 17 % 10
7
```

負の被演算子を使ったときの%と/のふるまいには注意が必要です。Pythonは、整数の除算結果の底値を返しますから、結果は予想よりも1小さいかもしれません。

basic/neg_int_div.cmd

```
>>> -17 / 10
-2
```

剰余演算を使うと、結果の符号は第2被演算子の符号と同じになります。

basic/neg_int_mod.cmd

```
>>> -17 % 10
3
>>> 17 % -10
-3
```

## 2.2.2 float型

Pythonには、小数部を持つ数値を表現するfloatという別の型もあります。floatはfloating point(浮動小数点)という言葉の略で、数値の桁の中で小数点を左右に動かして得られる値を表現するという意味です。

2個のfloatによる式は、floatの結果を生み出します。

basic/float_div_intro.cmd

```
>>> 17.0 / 10.0
1.7
```

式の被演算子がintとfloatになっている場合、Pythonは自動的にintをfloatに変換します。次の2つの式が、上の式と同じ結果を返すのは、そのためです。

basic/float_division.cmd

```
>>> 17.0 / 10
1.7
>>> 17 / 10.0
1.7
```

浮動小数点数を書くときには、小数点の後の0を省略することもできます。

basic/float_division2.cmd

```
>>> 17 / 10.
1.7
>>> 17. / 10
1.7
```

しかし、プログラムが読みにくくなりますので、一般に、このスタイルはあまりよくないと考えられています。画面上の小数点は見落としやすく、簡単に「17.」が「17」に誤解されてしまいます。

## 2.3 型とは何か

これまで2つの型の数値を見てきましたが、ここで**型**(type)とはいったい何なのかをきちんと説明しておきましょう。コンピューティングにおける型とは、値の集合とその値を対象として実行される操作/演算の集合のことです。たとえば、int型は、...、-3、-2、-1、0、1、2、3、...という値と+、-、*、/、%という演算です(まだ紹介していない演算が他にもあります)。それに対し、84.2はfloat型の値の集合に含まれる要素ですが、int型の値の集合には含まれません。

算術演算はPythonよりも前に発明されており、intとfloatはまったく同じ演算子を持っています。表2.1は、これらの演算をさまざまな値に対して適用したときに何が起こるかを示したものです。

表2.1 算術演算

| 演算子 | 記号 | 例 | 結果 |
|---|---|---|---|
| - | 符号反転 | -5 | -5 |
| * | 乗算 | 8.5 * 3.5 | 29.75 |
| / | 除算 | 11 / 3 | 3 |
| % | 剰余 | 8.5 % 3.5 | 1.5 |
| + | 加算 | 11 + 3 | 14 |
| - | 減算 | (5 - 19) | -14 |
| ** | 指数(累乗) | 2 ** 5 | 32 |

### 2.3.1 有限精度

浮動小数点数は、中学で習った有理数とは少し異なります。たとえば、Python版の$\frac{1}{3}$を見てみましょう(結果の小数点以下が切り捨てられてしまいますので、被演算子に小数点を入れるのを忘れないで下さい)。

basic/rate.cmd

```
>>> 1.0 / 3.0
0.33333333333333331
```

最後の1はいったい何なのでしょうか。3でなければおかしいのではないでしょうか。この問題の原因は、現実のコンピュータが有限のメモリしか積んでいないことにあります。そのため、コンピュータが1つの数値について格納できる情報に制約がかかるのです。0.33333333333333331は、コンピュータが実際に格納できる値の中で、もっとも$\frac{1}{3}$に近い値なのです。



#### 数値の精度についてさらに

コンピュータは、値が何であれ、整数を格納するために同じ量のメモリを使います。つまり、-22984、-1、100000000は、どれも同じサイズのメモリを消費します。そのため、コンピュータが格納できるint値は、特定の範囲に含まれるものだけになっています。たとえば、今のデスクトップやラップトップなら、-2147483648から2147483647までの数値しか格納できません。

同じ理由から、コンピュータは実数の**近似値**(approximation)しか格納できません。たとえば、$\frac{1}{4}$は正確に格納できますが、すでに見たように、$\frac{1}{3}$は近似値でしか格納できません。メモリの使用量を増やすと、近似値が実際の値に近付きますが、近似値にしかならないという問題は解決できません。0.333...の後にいかに3をたくさん詰め込んだとしても、それでは正確な$\frac{1}{3}$にはならないのと同じです。

$\frac{1}{3}$と0.33333333333333331の差はわずかと言えるかもしれません。しかし、計算でその値を使うと、誤差は複合的に増えていきます。たとえば、このfloatを2つ足し合わせると、...6662という値になりますが、これは$\frac{2}{3}$の近似値として、0.666...よりもわずかながら精度が下がってしまいます。計算をすればするほど、丸め誤差(rounding error)は拡大します。特に、非常に大きな数値と非常に小さな数値を併用すると、誤差は大きくなります。たとえば、10,000,000,000と0.00000000001を加えたとします。最初の有効桁と最後の有効桁の間に20個の0が並ばなければならないところですが、この桁数はコンピュータが格納するには多すぎます。そこで、結果はちょうど10,000,000,000になってしまいます。これでは、加算をしなかったのと同じになってしまいます。銀行が顧客の預金残高の総計を計算するときに、このようなことが起きては困ってしまいます。

自分が書いたプログラムによって思いがけず痛い目にあわないようにするために、浮動小数点数にこのような問題があることを意識することはとても大切です。ただし、この問題の解決方法は、本書で説明できる範囲を越えています。実際、連続数学の近似値を求めるアルゴリズムの研究である**数値分析**(numerical analysis)は、コンピュータ科学と数学の下位分野の中でももっとも大きなものの1つです。

### 2.3.2 演算子の優先順位

それでは、intとfloatの知識を利用して、華氏表現の温度を摂氏表現に変換してみましょう。そのためには、華氏表記の温度から32を引き、$\frac{5}{9}$を掛けなければなりません。

basic/precedence.cmd

```
>>> 212 - 32.0 * 5.0 / 9.0
194.22222222222223
```

本当は100を期待していたのに、Pythonは結果が194.22222222222223[†]になると言っています。これは、*と/の優先順位が-よりも高いからです。つまり、式にこれらの演算子が含まれているとき、*と/は-と+よりも先に評価されるのです。そこで、私たちが実際に計算したのは、212 - ((32.0 * 5.0) / 9.0)だったのです。

優先順位は、中1の数学の授業で習ったように、式の一部をかっこで囲めば変更できます。

basic/precedence_diff.cmd

```
>>> (212 - 32.0) * 5.0 / 9.0
100.0
```

表2.2は、算術演算子の優先順位をまとめたものです。1 + 1.7 + 3.2 * 4.4 - 16 / 3のように複雑な式を書く場合は、特にかっこが必要とされない場合でも、かっこを入れると読みやすくなりますので、そうする習慣を付けるとよいでしょう。

表2.2 優先順位順に並べた算術演算子

| 演算子 | 意味 |
|---|---|
| ** | 指数(累乗) |
| - | 符号反転 |
| *、/、% | 乗算、除算、剰余 |
| +、- | 加算、減算 |

## 2.4 変数と代入文

ほとんどの電卓[††]には、メモリボタンが少なくとも1つ付いています。このボタンを押すと、後で使うときのために値を保存できます。Pythonでは、**変数**(variable)を使ってこれを行います。変数とは、値が結び付けられた名前にすぎません。変数名としては、英数字とアンダースコアを使うことができます。たとえば、X、species5618、degrees_celsiusは、どれも変数名として使えますが、777を変数名にすることはできません(数値と区別が付けられないので)。no-wayも変数名にできませんが、それは記号が含まれているからです。

新しい変数は、値を与えるだけで簡単に作れます。

basic/assignment.cmd

```
>>> degrees_celsius = 26.0
```

† これも浮動小数点数による近似値の1つです。
†† そして携帯電話、腕時計など。



この種の文は**代入文**(assignment statement)と呼ばれ、degrees_celsiusには26.0という値が**代入**(assign)されたと表現します。代入文は、次のように実行されます

1. =記号の右の式を評価します。
2. =記号の左にある変数に評価後の値を格納します。

図2.4は、代入文を実行した後の**メモリモデル**(memory model)を示しています。これは単純ですが、もっと複雑なメモリモデルも後で登場します。

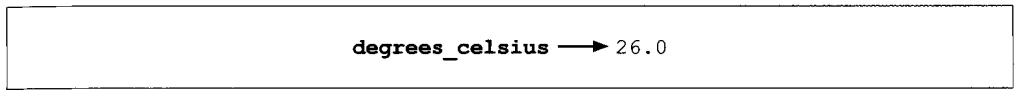


図2.4 変数と対応する値のメモリモデル

変数を作成したら、その値は他の計算でも使えます。たとえば、次のようにすれば、degrees_celsiusに格納されている温度と水の沸点の差を計算できます。

basic/variable.cmd

```
>>> 100 - degrees_celsius
74.0
```

式の中で変数の名前が使われているときには、Pythonは計算の中でその変数の値を使います。そのため、古い変数から新しい変数を作ることもできます。

basic/assignment2.cmd

```
>>> difference = 100 - degrees_celsius
```

変数名を単独で入力すれば、Pythonはその値を表示します。

basic/variable2.cmd

```
>>> difference
74.0
```

今何が起きたのでしょうか。私たちは、Pythonに演算子がまったく含まれていない非常に簡単な式を与えたのです。そこで、Pythonは式を評価して結果を出力したわけです。

Pythonに3の値を尋ねれば、当然の答えが返ってきます。

basic/simplevalue.cmd

```
>>> 3
3
```

変数が変数と呼ばれるのは、プログラムの実行とともに値が変わる可能性があるからです。たとえば、次のようにすれば、differenceに新しい値を代入できます。

basic/variable3.cmd

```
>>> difference = 100 - 15.5
>>> difference
84.5
```

このように新しい値を代入しても、値が書き換えられる前に変数を使って行った計算は変わりません。

basic/variable4.cmd

```
>>> difference = 20
>>> doubled = 2 * difference
>>> doubled
40
>>> difference = 5
>>> doubled
40
```

図2.5のメモリモデルが示すように、doubledに値を結び付けると、プログラムが明示的に上書きするまで、その値は変数に結び付いたままになります。differenceなどの他の変数を書き換えても、doubledは影響を受けません。

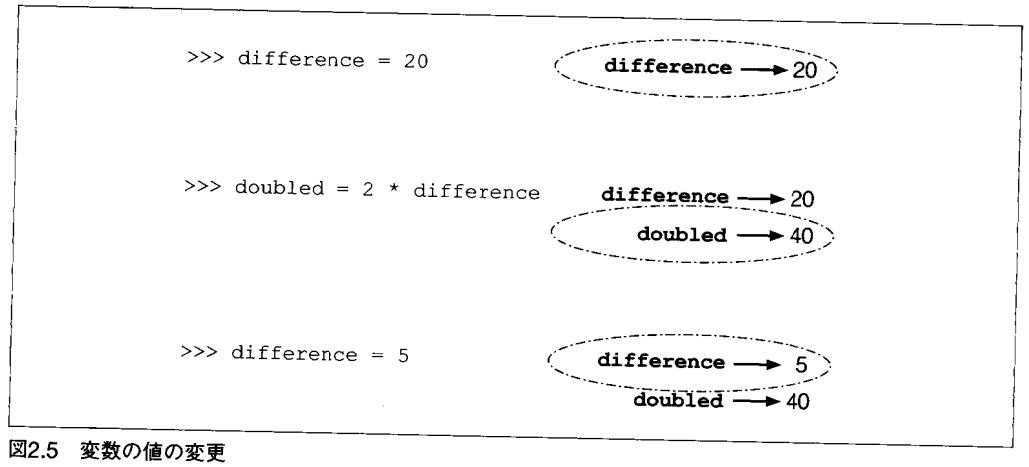


図2.5 変数の値の変更

さらに、代入文の左右両辺で変数を使うことさえできます。

basic/variable5.cmd

```
>>> number = 3
>>> number
3
>>> number = 2 * number
>>> number
```



```
6
>>> number = number * number
>>> number
36
```

こんなことをしても、数式では意味がありません。数値が自分の2倍と等しくなることはありません。しかし、Pythonの=は「～と等しい」という意味ではありません。=の意味は「～に値を代入する」です。

number = 2 * numberのような文を評価するとき、Pythonは次のことをします。

1. 現在numberに結び付けられている値を取得します。
2. その値に2を掛けて、新しい値を作ります。
3. その値をnumberに代入します。

### 2.4.1 複合演算子

前節の例では、number変数は代入文の両辺に使われていました。このような代入は非常によく行われるため、Pythonはこの種の演算の略記法を用意しています。

basic/variable6.cmd

```
>>> number = 100
>>> number -= 80
>>> number
20
```

**複合演算子**(combined operator)は、次のようにして評価されます。

1. =記号の右辺の式を評価します。
2. =記号に結び付けられている演算子を変数と式の計算結果に適用します。
3. =記号の左辺の変数に計算結果を代入します。

演算子が、右辺の式の評価**後**に適用されることに注意して下さい。

basic/variable7.cmd

```
>>> d = 2
>>> d *= 3 + 4
>>> d
14
```

表2.1のすべての演算子が略記法を持っています。たとえば、変数に同じ変数を掛けると、元の変数の自乗が得られます。

basic/variable8.cmd

```
>>> number = 10
>>> number *= number
>>> number
100
```

上の式は、次のものと同じです。

basic/variable9.cmd

```
>>> number = 10
>>> number = number * number
>>> number
100
```

## 2.5 ものごとがうまくいかないとき

変数は値を代入したときに作られると言いました。では、まだ作られていない変数を使おうとすると、どうなるのでしょうか。

basic/undefined_var.cmd

```
>>> 3 + something
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'something' is not defined
```

この結果はずいぶん暗号めいています。まず、初心者プログラマの立場から見たとき、エラーメッセージはPythonの数少ない弱点の1つです。最初の2行は、もっと長いプログラムを書き始めたときには必要不可欠なものになりますが、今の段階ではあまり役に立ちません。最後の行は、何がまずいのかを説明しています。somethingという名前が受け付けられなかったのです。

次のようなエラーメッセージを見かけることもあるでしょう。

basic/syntax_error.cmd

```
>>> 2 +
  File "<stdin>", line 1
    2 +
      ^
SyntaxError: invalid syntax
```

プログラミング言語（あるいは一般の言語）で認められるものとそうでないものを決める規則を**構文**（syntax）と呼びます。このメッセージが言おうとしているのは、私たちがPythonの構文規則に違反したということです。この場合、Pythonに対して2に何かを加えよと命令していますが、何を加えるのかを指示していないことが問題になっています。



## 2.6 関数の基礎

この章の前半で、私たちは華氏80度を摂氏に変換しました。数学者なら、これを$f(t) = \frac{5}{9}(t-32)$と書くところでしょう。ここで、$t$は摂氏に変換しようとしている華氏の温度です。華氏80度が摂氏では何度かを調べるには、$t$を80に置き換えて計算をします。すると、$f(80) = \frac{5}{9}(80-32)$、すなわち$26\frac{2}{3}$となります。

Pythonでも関数を書くことができます。数学者と同じように、プログラマは一般公式を定義することに慣れています。Pythonで変換関数を書くと、次のようになります（Enterキーを押して空行を追加し、関数定義が終わったことをPythonインタープリタに教えて下さい）。

basic/fahr_to_cel.cmd

```
>>> def to_celsius(t):
...     return (t - 32.0) * 5.0 / 9.0
...
```

数学の公式とは、次のような点が異なります。

- 関数定義は、Pythonの文の一種です。複雑ですが値に違いないものを持つ新しい名前を定義しています。
- Pythonに新しい関数を定義していることを知らせるために、defという**キーワード**（keyword）を使っています。
- 1時間後には忘れてしまいそうなfというような名前ではなく、to_celsiusのように意味のある名前を関数に付けています（これは、どうしてもそうしなければならないという要件ではありませんが、よいコーディングスタイルと考えられているものです）。
- 等号の代わりにコロン（:）を使っています。
- 関数の実際の公式を次の行で定義しています。その行はスペース4個分インデントされており、returnキーワードが付けられています。

新しい関数の定義に入ると、Pythonは自動的にドット3個のプロンプトを表示するようになります。通常の>>>プロンプトで大なり不等号を入力しないのと同じように、ここでもドットを入力したりはしません。Wing 101のようなスマートエディタを使っている場合には、エディタが自動的に関数**本体**（body）を必要なだけインデントします（メモ帳やPicoなどの簡単なテキストエディタではなく、Wing 101を使う理由の1つがこれです。スペースバーと親指の消耗を防いでくれるわけです）。

Pythonにto_celsius(80)、to_celsius(78.8)、to_celsius(10.4)を評価させると、次のようになります。

basic/fahr_to_cel_2.cmd

```
>>> to_celsius(80)
26.666666666666668
```

```
>>> to_celsius(78.8)
26.0
>>> to_celsius(10.4)
-12.0
```

これら3つの文は、関数を呼び出して何らかの仕事をさせることから、**関数呼び出し**（function call）と呼ばれます。関数を1度定義するだけで、何度でも呼び出すことができます。

関数定義の一般形式は、次の通りです。

```
def function_name(parameters):
    block
```

先ほど説明したように、defキーワードは、新しい関数を定義していることをPythonに知らせる役割を果たします。その次に関数名、0個以上の**引数**（parameter。仮引数とも言います）をかっこで囲んだものが続き、コロンで1行目が終わります。引数とは、関数が呼び出されるときに値が与えられる変数です（to_celsius関数のtを思い出して下さい）。たとえば、to_celsius(80)という関数呼び出しでは、tに80が代入されます。to_celsius(78.8)では78.8、to_celsius(10.4)では10.4がtに代入されます。これらの実際に渡される値を関数の**実引数**（argument）と言います。

関数が行う処理は、定義に含まれる文の**ブロック**（block）によって定義されます。to_celsiusのブロックは1つの文だけから構成されていましたが、後で見ていくように、もっと複雑な関数を構成するブロックには、文がいくつも含まれています。

最後に、return文の一般形式も見ておきましょう。

```
return expression
```

この文は、次のように実行されます。

1. returnキーワードの右側の式を評価します。
2. その値を関数の結果として使います。

関数**定義**と関数**呼び出し**の違いをはっきりと理解することが大切です。関数定義の場面では、Pythonは関数を記録しますが、実行しません。関数呼び出しでは、Pythonは関数の先頭行にジャンプし、実行を開始します（図2.6参照）。関数の実行が終了すると、Pythonは関数が呼び出された場所に戻ってきます。

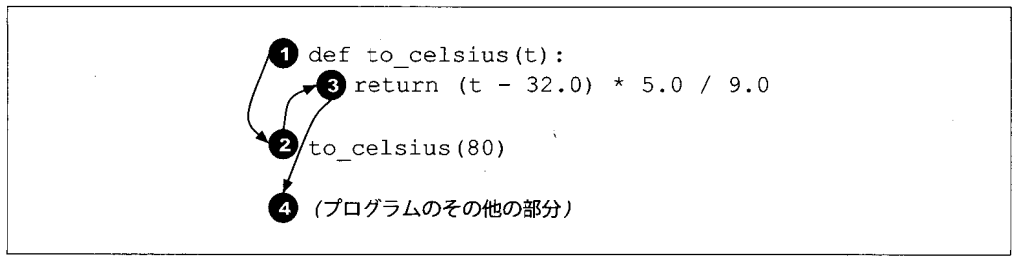


図2.6 関数の呼び出しフロー



## 2.6.1 ローカル変数

計算の中には複雑なものがあります。そのようなものは、手順を分解して別々のステップを作るとわかりやすいコードになります。ここでは、$ax^2 + bx + c$という多項式の評価を数ステップに分割してみましょう。

basic/multi_statement_block.cmd

```
>>> def polynomial(a, b, c, x):
...     first  = a * x * x
...     second = b * x
...     third  = c
...     return first + second + third
...
>>> polynomial(2, 3, 4, 0.5)
6.0
>>> polynomial(2, 3, 4, 1.5)
13.0
```

関数内で作られているfirst、second、thirdという変数は、**ローカル変数**（local variable。局所変数とも言います）と呼ばれます。これらの変数が存在するのは、関数の実行中だけです。関数が実行を終了すると、ローカル変数は消えてしまいます。そのため、関数の外からローカル変数にアクセスしようとしても、定義されていない変数にアクセスしようとするのと同じで、エラーになります。

basic/local_variable.cmd

```
>>> polynomial(2, 3, 4, 1.3)
11.280000000000001
>>> first
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'first' is not defined
>>> a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined
```

この例からもわかるように、関数の仮引数もローカル変数です。関数が呼び出されると、Pythonは呼び出しに与えられた実引数の値を関数の仮引数に代入します。関数が一定の個数の仮引数を取るものとして定義されている場合、呼び出しでは同じ数の実引数を渡さなければなりません†。

† 任意の個数の引数を取る関数の作り方については、後で学びます。

basic/matching_args_params.cmd

```
>>> polynomial(1, 2, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: polynomial() takes exactly 4 arguments (3 given)
```

変数の**スコープ**(scope)とは、変数にアクセスできるのがプログラム内のどこかということです。たとえば、ローカル変数のスコープは、変数が最初に定義された行から関数の末尾までです。

## 2.7 組み込み関数

Pythonには、よく行われる処理を実行する**組み込み関数**(built-in function)が多数含まれています。数値の絶対値を返すabsも、その1つです。

basic/abs.cmd

```
>>> abs(-9)
9
```

浮動小数点数をもっとも近い整数に丸めるroundもあります。

basic/round.cmd

```
>>> round(3.8)
4.0
>>> round(3.3)
3.0
>>> round(3.5)
4.0
```

ユーザー定義関数と同様に、Pythonの組み込み関数は、複数の引数を取ることがあります。たとえば、累乗関数のpowを使えば、$2^4$を計算できます。

basic/two_args.cmd

```
>>> pow(2, 4)
16
```

変数の型を変換する関数は、組み込み関数の中でももっとも役に立つものの1つです。型名のintやfloatは、まるで関数のように使うことができます。

basic/typeconvert.cmd

```
>>> int(34.6)
34
>>> float(21)
21.0
```



この例を見ると、浮動小数点数を整数に変換するときには、四捨五入ではなく切り捨てが使われることがわかります。

## 2.8 コーディングスタイルについて

心理学者の研究によれば、人間が同時に注意を向けられるものはごく少数だということです[Hoc04]。プログラムは非常に複雑になることがありますので、変数が何のためにあるのかを思い出しやすいような変数名を選ぶことが大切です。X1、X2、blahというようなものでは、次の週にプログラムを見たときには何も思い出せなくなっているでしょう。celsius、average、final_resultのような名前を使うべきです。

別の研究によれば、人間の脳はものの違いに自動的に気付いてしまうそうです。気付かないようにすることはできないというのです。ですから、テキストの中に統一が取れていない部分が多ければ多いほど、読むのに時間がかかるのです(もシ、テきすトガ此のやウニ書かレテいたラ、この章を読ムタめにどレクらイノ時間がかカルカ想像しテミて下サい)。ですから、統一の取れた変数名を使うことも大切です。

こういった規則は非常に重要なので、どの言語を使っている場合でも、メンバに特定のスタイルガイドに従うように求めているプログラミングチームは多数あります。これは、新聞社や出版社が見出しの組み方やリストの並べ方などについて方法を規定しているのと同じです。インターネットで**プログラミングスタイルガイド**(programming style guide)を検索すると、数百もの実例が見つかるはずです。

それと同時に、多くの人々がコードの「最良のスタイル」は何かを論じて何時間も浪費していることもわかるでしょう。この点について非常に頑迷な意見を持っている同級生がいる場合もあるはずです。もしそうであれば、その信念を裏付けるデータを尋ねてみるとよいでしょう。カンマの後ろにスペースを入れると、そうでないときと比べてプログラムが読みやすくなるということを立証する実地調査を知っているかどうかを確かめてみるのです。答えられないようなら、ぽんと背中をたたいて、彼らの好きにさせてやりましょう。

## 2.9 まとめ

この章では、次のことを学びました。

- オペレーティングシステムは、他のプログラムのためにコンピュータのハードウェアを管理するプログラムです。インタープリタや仮想マシンは、オペレーティングシステムの上に位置して、あなたのプログラムを実行するプログラムです。私たちのこれまでの経験では、複雑なシステムを作るための方法としては、このような階層化がもっとも優れています。
- プログラムは文から構成されています。文には、単純な式(すぐに評価されます)、代入文(新しい変数を作ったり、既存の変数の値を変更したりします)、関数定義(Pythonに新しい仕事のしかたを教えます)などの種類があります。

- Python のすべての値には、特定の型があります。値にどのような演算、処理を適用できるかは、型によって決まります。数値を表現する型は、int と float の 2 つです。
- 式は、決められた順序で評価されます。しかし、部分式をかっこで囲めば、その順序を変更することができます。
- 変数には、使う前に値を与えなければなりません。
- 関数が呼び出されると、その実引数の値が仮引数に代入され、関数内の文が実行され、値が返されます。関数の仮引数に代入される値と、関数内で作られるローカル変数の値は、関数が制御を返すと消えてしまいます。
- Python の文は、インデントを使ってブロックにまとめることができます。
- Python には、組み込み関数というあらかじめ定義されている関数があります。

## 2.10 練習問題

練習問題で自分の力を試してみましょう。

1. 次の式は、それぞれどのような値を返すでしょうか。Python に式を入力して答えを確かめて下さい。
   a) `9 - 3`
   b) `8 * 2.5`
   c) `9 / 2`
   d) `9 / -2`
   e) `9 % 2`
   f) `9 % -2`
   g) `-9 % 2`
   h) `9 / -2.0`
   i) `4 + 3 * 5`
   j) `(4 + 3) * 5`

2. 単項のマイナスは、数値の符号を反転させます。同じように単項のプラスもあります。たとえば、Python は +5 の意味を理解しています。x が -17 という値を持つとき、+x はどのような値を持つべきだと思いますか。符号には手を付けない方がよいのでしょうか、それとも絶対値関数と同じように、マイナス符号を取り除くべきでしょうか。Python シェルを使って、Python のふるまいを確かめて下さい。

3. a) 新しい変数 temp を作り、値 24 を代入して下さい。
   b) temp の値に 1.8 を掛け、32 を加えて、摂氏から華氏に変換して下さい。そして、計算結果の値を temp に代入して下さい。temp の新しい値はいくつですか。

4. a) 新しい変数 x を作り、値 10.5 を代入して下さい。
   b) 新しい変数 y を作り、値 4 を代入して下さい。



   c) x と y の合計を計算し、結果を x に代入して下さい。x と y の新しい値はいくつになりますか。

5. x の値が 3 の場合、`x += x - x` という文を Python が評価するときに起きることを箇条書きにして説明して下さい。

6. to_celsius という関数名には問題があります。元の単位が何だったのかに言及していませんし、動詞句になっていません（関数は積極的に何かをするものですから、多くの関数名は動詞句になっています）。元の単位は華氏であることを暗黙の前提としていますが、温度の単位としてはケルビンもありますし、他にも多数あります（それらについては「6.5 練習問題」を参照して下さい）。
   fahrenheit_to_celsius、あるいは convert_fahrenheit_to_celsius のような名前もあり得ますし、略して fahr_to_cel、もっと短く f2c。あるいはただの f にすることもできます。どの名前がもっともよいか、その理由を 1 段落くらいで説明して下さい。覚えやすさ、入力のしやすさ、読みやすさといったことを考えて下さい。また、普段使う言葉が英語ではない人々のことを考えるのも忘れてはいけません。

7. アメリカでは、1 ガロン当たりのマイル数で車の燃費を測ります。しかし、メートル法では、100km 当たりのリットル数で計算されるのが普通です。
   a) 1 ガロン当たりのマイル数から 100km 当たりのリットル数に変換する convert_mileage という関数を書いて下さい。
   b) 20、40MPG（マイルパーガロン）について、関数が正しい値を返すことをテストして下さい。
   c) 正しい値を調べるためにどのような方法を使いましたか。コンピュータの計算結果は、あなたが考えていた値とどのくらい近い値になっていましたか。

8. 仮引数と実引数の違いを説明して下さい。

9. a) km 単位の距離と車の燃費を表す値を引数として、その距離を移動するために必要なガソリンの量をリットル単位で返す liters_needed という関数を定義して下さい。liters_needed 関数は、前の問題で作った convert_mileage 関数を呼び出さなければなりません。
   b) liters_needed(150, 30) が 11.761938367442955、liters_needed(100, 30) が 7.84129224496197 を返すことを確かめて下さい。
   c) 100 と 30 を引数として liters_needed を呼び出したとき、convert_mileage の引数はいくつですか。
   d) liters_needed(100, 30) という関数呼び出しは convert_mileage 関数呼び出しを引き起こしますが、これら 2 つの関数のうち、先に実行を終了するのはどちらですか。

10. 本文では、組み込み関数として abs、round、pow、int、float を見てきました。これらの関数を使って、次のことをする式を書いて下さい。
    a) 3 の 7 乗を計算して下さい。

b) 34.7 を切り捨てによって整数に変換して下さい。
c) 34.7 を四捨五入によって整数に変換して下さい。
d) -86 の絶対値を取ってから、それを浮動小数点数に変換して下さい。



# 3章 文字列

数値はコンピューティングの基礎です。そもそも、コンピュータが発明されたのは、数値を操作するためだったくらいです。しかし、この世界には、住所、画像、音楽など、数値以外のデータが無数にあります。これらのデータはそれぞれ独自の型として表現でき、それらの型の操作方法を知ることは、プログラミングの能力の大きな部分を占めます。この章では、数値以外のデータの1つとして、この文に含まれる単語やDNAのらせん構造を表現するATGCの4字の並びなどのテキストを表現する型を紹介します。説明の過程で、プログラムをもう少し対話的にする方法も考えていきます。

## 3.1 文字列

コンピュータはもともと算術計算のために発明されたものかもしれませんが、今日のコンピュータが大半の時間を費やしているのは、テキストの処理です。デスクトップのチャットプログラムからGoogleに至るまで、コンピュータはテキストを作成、保存、検索し、こちらのテキストをあちらに動かすということを繰り返しています。

Pythonでは、テキストの塊は**文字列**(string)、すなわち**文字**(character。英字=letter、数字、記号を含む)の並び(シーケンス)として表現されます。文字シーケンスを格納するためのもっとも単純な型は、strです。この型は、北米のほとんどのキーボードで入力できるローマ字を格納できます。それに対し、unicodeというもう1つの型を使うと、漢字や化学記号、クリンゴン語に至るまで、ありとあらゆる文字を並べた文字列を格納できます。この章のサンプルでは、簡単な方の型であるstrを使っていきます。

Pythonでは、値をシングルクォートかダブルクォートで囲むことによって、その値が文字列だということを示します。

strings/string.cmd
```
>>> 'Aristotle'
'Aristotle'
>>> "Isaac Newton"
'Isaac Newton'
```