プログラミング体験実習の手引き

第二版 revision01
2017 年 11 月 8 日（水）

Lesson 1　プログラム

1.1　この体験実習では、プログラミングというものを体験してもらいます。

1.2　プログラミングというのは、「プログラム」と呼ばれる文書を書くということです。

1.3　「プログラム」というのは、コンピュータに実行してほしい動作を言葉で書きあらわしたもののことです。コンピュータが実行する動作は、すべて、人間がプログラムを書くことによって実現されています。

1.4　プログラムは、「ソフトウェア」とか「ソフト」とか「アプリ」などと呼ばれることもあります。この体験実習では、「プログラム」という言葉で統一することにします。

1.5　プログラムを書くためには、「プログラミング言語」と呼ばれる専用の言語を使う必要があります。

1.6　プログラミング言語には、たくさんのものがあります。この体験実習では、Python というプログラミング言語を使います。

Lesson 2　端末

2.1　プログラミングを始める前に、まず、「端末」と呼ばれるプログラムの使い方に慣れておきましょう。

2.2　「端末」というのは、「シェル」と呼ばれるプログラムを実行するための環境を作り出すプログラムのことです。

2.3　「シェル」というのは、「コマンド」と呼ばれるものを読み込んで、それを実行するプログラムです。

2.4　「コマンド」というのは、コンピュータに実行してほしいことを 1 行で書きあらわしたもののことです。つまり、1 行だけでできたプログラムのことだと考えることができます。

2.5　シェルは、コマンドが入力されるのを待っている状態だということを示すために、「プロンプト」と呼ばれる文字列を画面に表示します。

2.6　シェルに対して、プログラムの名前をコマンドとして入力すると、その名前のプログラムが起動します。たとえば、シェルに `cal` と入力すると、その名前のプログラムが起動して、今月のカレンダーが出力されます。

2.7　プログラムの名前のうしろに、「引数」と呼ばれる単語を書くことによって、プログラムが実行するべき動作を指定することもできます。たとえば、

```
cal 月 年
```

というコマンドを入力することによって、指定された年と月のカレンダーを出力させることができます。

Lesson 3　Python の言語処理系

3.1　人間が書いたプログラムをコンピュータに実行させるためには、「言語処理系」と呼ばれるプログラムが必要になります。ひとつの言語処理系が対応しているプログラミング言語はひとつだけですので、プログラムを実行するためには、そのプログラムを書くために使われているプログラミング言語に対応した言語処理系が必要になります。

3.2　Python の言語処理系は python という名前ですので、`python` というコマンドを入力することによって起動することができます。

3.3　引数なしで起動された python は、「インタラクティブシェル」と呼ばれます。これは、その名前のとおり、シェルのような動作をします。`>>>` というのが、インタラクティブシェルのプロンプトです。

3.4　インタラクティブシェルは、プログラムの断片を読み込んで、それを実行する、ということを繰り返します。

3.5　インタラクティブシェルは、`quit()` と入力することによって終了させることができます。

Lesson 4　式

4.1　実行することによって「値」と呼ばれる結果が得られる記述は、「式」と呼ばれます。

4.2　式を実行することを、式を「評価する」と言います。

4.3　インタラクティブシェルは、式が入力された場合、その式を評価して、得られた値を出力します。

4.4　たとえば、`5+3` という式を入力すると、インタラクティブシェルはそれを評価して、8 という値を出力します。

4.5　足し算をあらわす `+` のようなものを「演算子」と呼びます。

4.6　足し算は `+`、引き算は `-`、掛け算は `*`、割り算の商は `/`、整数の範囲での割り算の商は `//`、割り算のあまりは `%`、べき乗は `**` という演算子を使うことによって求めることができます。

Lesson 5　文字列

5.1　文字列を引用符で囲んだものを式として評価すると、その文字列が値として得られます。インタラクティブシェルで試してみてください。

5.2　`+` という演算子は、数値の足し算だけではなくて、文字列の連結にも使われます。たとえば、

```
'kitsune' + 'udon'
```

という式を評価すると、`kitsuneudon` という文字列が値として得られます。

5.3　`*` という演算子は、数値と数値との掛け算だけではなくて、文字列と数値との掛け算にも使われます。たとえば、

```
'w' * 30
```

という式を評価すると、`wwwwwwwwwwwwwwwwwwwwwwwwwwwwww` という文字列が値として得られます。

Lesson 6　関数

6.1　動作をあらわしているデータは、「関数」と呼ばれます。

6.2　関数に、それがあらわしている動作を実行させることを、関数を「呼び出す」と言います。

6.3　関数は、データを受け取って、それを処理して、その結果として得られたデータを返します。関数が受け取るデータは「引数」と呼ばれ、関数が返すデータは戻り値と呼ばれます。

6.4　たとえば、`len`という関数は、引数として文字列を受け取って、その長さ（文字数）を戻り値として返します。

6.5　関数は、

```
関数名(式, ··· )
```

という形の式を評価することによって呼び出すことができます。たとえば、

```
len('umiushi')
```

という式を評価することによって、`len`という関数を呼び出して、それに対して引数として`umiushi`を渡すことができます。このような式は、「関数呼び出し」と呼ばれます。

6.6　関数が戻り値として返したデータは、その関数を呼び出した関数呼び出しの値になります。

6.7　それでは、`len`という関数を呼び出す式をインタラクティブシェルに入力してみてください。

Lesson 7　組み込み関数とユーザー定義関数

7.1　動作を記述することによって関数を作ることを、関数を「定義する」と言います。

7.2　関数を定義するために書く記述は、「関数定義」と呼ばれます。

7.3　`len`のように、言語処理系に最初から組み込まれていて、関数定義を書かなくても使うことができる関数もあります。そのような関数は、「組み込み関数」と呼ばれます。

7.4　関数定義を書くことによって定義された関数は、「ユーザー定義関数」と呼ばれます。

7.5　関数を定義するというのは、単純な動作を組み合わせることによって複雑な動作を作ることだと考えることができます。

7.6　「単純な動作を組み合わせることによって複雑な動作を作る」というのが、プログラミングというものの本質です。

Lesson 8　関数定義の書き方

8.1　Python では、関数定義は、基本的には、

```
def 関数名(): return 式
```

と書きます。

8.2　関数を呼び出すと、その関数を定義した関数定義の中に書かれている式が評価さ

れて、その値が戻り値として返されます。

8.3 次の関数定義は、引数を何も受け取らないで、戻り値として 0 を返す、zero という関数を定義します。

```
def zero(): return 0
```

8.4 それでは、この関数定義をインタラクティブシェルに入力してみましょう。

8.5 1 行目を入力してエンターキーを押すと、... というプロンプトが表示されます。このプロンプトは、インタラクティブシェルが、2 行以上にまたがる記述の 2 行目以降を入力してほしいときに表示するものです。今は、2 行目以降の行を入力する必要がありませんので、何も入力しないでエンターキーを押してください。

8.6 それでは、入力した関数定義によって定義された関数を呼び出してみましょう。

```
zero()
```

という関数呼び出しをインタラクティブシェルに入力すると、zero が呼び出されて、0 という戻り値が出力されるはずです。

Lesson 9　仮引数

9.1 引数を受け取る関数を定義したいときは、「仮引数」と呼ばれるものを使います。

9.2 「仮引数」というのは、受け取った引数を参照するための名前のことです。

9.3 引数を受け取る関数を定義する関数呼び出しは、

```
def 関数名(仮引数, ··· ): return 式
```

と書きます。たとえば、

```
def namako(a, b, c): return 式
```

という関数定義によって定義された namako という関数を、

```
namako(37, 24, 61)
```

という関数呼び出しで呼び出したとすると、a は 37 を、b は 24 を、c は 61 を参照する名前になります。

9.4 次の関数定義は、引数として数値または文字列を受け取って、それに 10 を掛け算した結果を戻り値として返す、tentimes という関数を定義します。

```
def tentimes(a): return a*10
```

9.5 それでは、この関数定義をインタラクティブシェルに入力して、tentimes を呼び出してみましょう。

Lesson 10　テキストエディター

10.1 次に、関数定義をファイルに保存して、その関数定義をインタラクティブシェルに実行させる、という実習をします。

10.2 文字だけでできているデータは、「テキストデータ」と呼ばれます（単に「テキスト」と呼ばれることもあります）。

10.3 テキストを入力したり修正したりするために使われるプログラムは、「テキストエディター」と呼ばれます（単に「エディター」と呼ばれることもあります）。

10.4　人間が書くプログラムは、テキストデータです。したがって、人間がプログラムと書くときには、テキストエディターが使われます。

10.5　この体験実習では、gedit というテキストエディターを使います。gedit は、

```
gedit ファイル名 &
```

というコマンドで起動することができます。ちなみに、コマンドの末尾のアンパサンド (`&`) は、シェルに対して、プログラムの終了を待たないで次のプロンプトを出すように依頼する文字です。

10.6　それでは、次のコマンドで gedit を起動してください。

```
gedit star.py &
```

10.7　Python のプログラムを保存するファイルの名前には、`.py` という拡張子を付けることになっています。

10.8　gedit を使って、次の関数定義を入力して、ファイルに保存してください。

```
def star(a): return '*' * len(a)
```

10.9　この関数定義は、引数として文字列を受け取って、その文字列と同じ長さを持つアスタリスクの列を戻り値として返す、`star` という関数を定義します。

Lesson 11　ファイルの中のプログラムの実行

11.1　ファイルに格納されているプログラムをインタラクティブシェルに実行させたいときは、

```
from ファイル名 import *
```

という文をインタラクティブシェルに入力します。ただし、この中の「ファイル名」のところには、ファイル名から拡張子を取り除いた部分を書きます。

11.2　先ほど入力した関数定義は、`star.py` という名前のファイルに格納されていますので、

```
from star import *
```

という文をインタラクティブシェルに入力することによって、それを実行することができます。

11.3　`star` という関数が定義されたはずですので、それを呼び出してみましょう。

Lesson 12　真偽値

12.1　何かが成り立っているとき、それは「真」であると言います。逆に、何かが成り立っていないとき、それは「偽」であると言います。

12.2　真であるか偽であるかということをあらわすデータのことを「真偽値」と呼びます。

12.3　Python では、真は `True`、偽は `False` と書きあらわされます。

12.4　データとデータとの間に何らかの関係が成り立っているかどうかを調べたいときに使われる演算子は、「関係演算子」と呼ばれます。

12.5　関係演算子を使って関係が成り立っているかどうかを調べた結果は、真偽値で得

られます。

12.6 == という関係演算子は、二つのデータのあいだに「それらは等しい」という関係が成り立っているかどうかを調べます。

12.7 < という関係演算子は、二つのデータのあいだに「左のデータのほうが右のデータよりも小さい」という関係が成り立っているかどうかを調べます。

12.8 関係演算子としては、そのほかに、!=（等しくない）、>（より大きい）、<=（より小さいかまたは等しい）、>=（より大きいかまたは等しい）というものもあります。

Lesson 13　選択

13.1 どんなプログラミング言語にも、真偽値によって動作を選択する、ということを記述する方法があります。

13.2 Python では、「条件演算子」と呼ばれる演算子を使うことによって、選択を記述することができます。

13.3 条件演算子は、

式$_1$ `if` 式$_2$ `else` 式$_3$

という形の式を書くことによって使うことができます。この形の式を、「条件演算子式」と呼ぶことにしましょう。

13.4 条件演算子式を評価すると、まず最初に 式$_2$ が評価されます。そして、その値が真ならば、式$_1$ が評価されて、その値が条件演算子式の値になります。式$_2$ の値が偽ならば、式$_3$ が評価されて、その値が条件演算子式の値になります。

13.5 たとえば、

```
'equal' if 5 == 5 else 'not equal'
```

という式の値は `equal` で、

```
'equal' if 5 == 8 else 'not equal'
```

という式の値は `not equal` になります。

13.6 それでは、

```
gedit evenodd.py &
```

というコマンドで gedit を起動して、次の関数定義を入力してください。

```
def evenodd(a): return 'even' if a%2 == 0 else 'odd'
```

13.7 この関数定義は、引数として受け取った整数が偶数ならば `even` という文字列を戻り値として返して、奇数ならば `odd` という文字列を戻り値として返す、`evenodd` という関数を定義します。

13.8 入力ができたら、それを保存して、

```
from evenodd import *
```

とインタラクティブシェルに入力することによって、それを実行してください。

13.9 `evenodd` が正しく動作するか、確かめてみましょう。

Lesson 14　再帰

14.1　全体と同じものが一部分として含まれているという性質は、「再帰」と呼ばれます。

14.2　0 以上の整数について、n が 0 またはプラスの整数だとするとき、n から 1 までの整数をすべて乗算した結果、つまり、

$$n \times (n-1) \times (n-2) \times \cdots \times 1$$

という計算の結果は、n の「階乗」(factorial) と呼ばれて、$n!$ と書きあらわされます。ただし、$0!$ は 1 だと定義します。

14.3　たとえば、$5!$ は、

$$5 \times 4 \times 3 \times 2 \times 1$$

という計算をすればいいわけですから、120 ということになります。

14.4　定義されるべき概念が定義の中で使われているような概念の定義は、「再帰的な定義」と呼ばれます。

14.5　$n!$ という概念は、次のように再帰的に定義することができます。

$$\begin{cases} & 0! = 1 \\ n \geq 1 \text{ ならば} & n! = n \times (n-1)! \end{cases}$$

14.6　関数の定義も、再帰的に書くことができます。たとえば、階乗を求める関数の定義は、再帰的に書くことができます。

14.7　それでは、

```
gedit factorial.py &
```

というコマンドで gedit を起動して、次の関数定義を入力してください。

```
def factorial(n): return n * factorial(n-1) if n >= 1 else 1
```

14.8　この関数定義は、整数 n を受け取って $n!$ を返す、`factorial` という関数を定義します。

14.9　入力ができたら、それを保存して、

```
from factorial import *
```

とインタラクティブシェルに入力することによって、それを実行してください。

14.10　`factorial` が正しく動作するか、確かめてみましょう。