Python によるプログラミングの初歩

はじめに

「アルゴリズム入門」では、Python というプログラミング言語を用いてアルゴリズムを記述します。ここでは、講義で取り上げるアルゴリズムを理解できるように、初期の講義で利用する範囲で Python の簡単な導入部分を紹介します。

1 Python プログラムの実行

コンピュータ上で、プログラミング言語により記述されたアルゴリズムを読み込み、その記述通りに実行するソフトウェアを**言語処理系 (or 単に処理系)** と呼びます。最近では、処理系にプログラムを作成する機能を追加したソフトウェアも普及し、これらは**プログラム開発環境**と呼ばれています。

Python の処理系には、主に下記二つの動作モードがあります[1]。

対話モード: まずは処理系を起動させ、その上でプログラムを入力する。処理系はプログラムの入力に応じて 1 行ずつ実行する。

ファイル モード: プログラムを事前に作成し、それをファイルに保存しておいて、処理系から必要に応じて保存してあるプログラム ファイルを呼び出して実行する (プログラム ファイルの呼び出しは、処理系を起動する際の引数として渡す/ 処理系の起動後にメニューから読み込む等があります)。

講義で利用する IDLE あるいは JuPyter は、プログラムの実行だけではなく作成もできるので、開発環境と呼ばれています。また、どちらの開発環境も二つの動作モードで動かすことができます。

以下では、IDLE を例に説明します。IDLE は、それほど高機能ではないですが、本講義で取り上げるアルゴリズムの作成/実行には必要十分な機能があり、シンプルで使いやすいからです (要は、本講義は “アルゴリズム” に焦点を当てているので、それ以外はシンプルにしたい)。また、Python 以外の言語にも似たようなソフトウェアが提供されていることが多く、将来他の言語を学ぶ際にも応用し易いと考えるからです。

IDLE の対話モード:

“コマンド プロンプト (Windows の場合)” あるいは “ターミナル (Mac の場合)” から IDLE を起動後、表示されている “>>>” の後にプログラムを入力して Enter キー (or Return キー) を押すと、入力されたプログラムが実行されます。例えば、こんな感じ。

```
Python 3.6.5 |Anaconda, Inc.| (default, Mar 29 2018, 13:32:41) [MSC v.1900 64 bit (AMD64)]
on win32
Type "copyright", "credits" or "license()" for more information.
>>> 6 + 5
11
>>>
```

参考: Mac では Ctrl-P/N を、Windows では Alt-P/N を押すと過去の入力が再表示されます (“P” は最新の入力から古い方へ溯って行き、“N” は新しい方へ戻って来ます)。これをうまく編集することで、入力の効率を上げることができます。

[1] 本来 “ファイル モード” という名称は存在しないのですが、ここでは対話モードとの対比のために、この名称を使うことにします。

“>>>” は、**コマンド プロンプト**と呼ばれ、システム[2]が外部からの入力を待っている状態を示しています。この例では、

1. IDLE を起動後、IDLE はユーザからのプログラム入力を待っている。
2. ユーザが “6 + 5” を入力
3. IDLE がこれを実行し、計算結果 11 を表示
4. その後、IDLE はユーザからの次の入力を待っている。

という動作を表わしています。

IDLE のファイル モード:

IDLE では、起動後に表示されるメニューから「File」→「New File」を選ぶと、プログラムを作成するためのウィンドウが表示されます。ここで新しいプログラムを作成した後、プログラム作成ウィンドウにある「File」→「Save」or「Save As」より保存します。プログラムの保存では、ファイルの識別子に “.py” を付けるのが作法です (test.py とか)。プログラムの保存後、これもプログラム作成ウィンドウにある「Run」→「Run Module」を行なうことで、プログラムが読み込まれて実行されます。また、過去に作成したプログラムを読み込んで実行したい場合は、IDLE にある「File」→「Open」で行ないます。

注意)

IDLE で対話モードとファイル モードをそれぞれ使ってみると、両者の挙動が少し異なることに気付くかも知れません。例えば、上で挙げた “6 + 5” について考えてみます。対話モードのプロンプトから “6 + 5” と入力すると、IDLE は即座に “11” を表示します。では、ファイル モードで “6 + 5” とだけを記述したプログラム ファイルを作成し、これを実行した場合はどうでしょうか。IDLE は何も表示しません。例えば、こんな感じです。

```
Python 3.6.5 |Anaconda, Inc.| (default, Mar 29 2018, 13:32:41) [MSC v.1900 64 bit (AMD64)]
on win32
Type "copyright", "credits" or "license()" for more information.
>>>
>>> ================================ RESTART ================================
>>>
```

何故このような違いが生じるのでしょうか。

ここでは、取りあえず次のように考えておくことにします。対話モードでは、入力されたプログラムを 1 行ずつ (Enter キーを押される度に) 評価して、その結果を逐一表示します。評価とは、文や式の意味を認識して実行することです。例えば、「6 + 5」と書いてあれば計算式として認識し、その計算を行ないます。また、「“a”」と書いてあれば文字として認識し、実行します。文字の実行というものは意味を捉えにくいですが、ここでは “文字を読み上げる” と捉えておきましょう。さらに、「a」とだけ書いてあれば、これは変数 (データを入れる箱) だと認識し、(文字と同様) 変数の中身を読み上げます。このように対話モードでは、1 行毎に実行した結果を逐一表示しているわけです。これに対し、ファイル モードでは、ファイルとして記録されたプログラムの評価結果を逐一表示することはしません。何か表示する必要が生じた場合 (例えば、プログラムに誤りがあるとか、何らかの表示をする命令が記述されているとか) のみ表示します。大きなプログラムを実行する際は、こちらの方が余計な or 冗長な表示が減り都合の良い場合が多いです[3]。

IDLE では、両モードの使われ方を考慮して、このような仕様になっています。

---

[2]正確に言えば、IDLE というソフトウェアですが、複数の機能から構成されているソフトウェア/ハードウェア/ネットワークや、何らかの目的あるいはサービスのために、ソフトウェアやハードウェアを連携して構築した環境のことを総称して**システム**と呼ぶことがあります。コマンド プロンプトは、IDLE だけではなく様々なソフトウェアで使われているので、ここではシステムという言葉を使いました。

[3]例えば、数千行以上あるプログラムを実行する際、1 行毎にその結果を表示/確認するのではプログラムの挙動を示す情報が多過ぎるため、その把握が困難になります。これに対し、プログラムの要所毎に結果を表示/確認すれば、効率的にその挙動を把握できますね。

2　データ型/演算子

前節では、6 + 5 という式を例に使いました。IDLE は、これを計算式として評価し、その結果である 11 を表示しました。6 や 5 は数字であり、数字であれば加算 (+) を適用することが可能であるからです。では、式 6 + “A” はどうでしょうか (“A” は文字の A を表わします)。IDLE 上で実行してみましょう。

```
Python 3.6.5 |Anaconda, Inc.| (default, Mar 29 2018, 13:32:41) [MSC v.1900 64 bit (AMD64)]
on win32
Type "copyright", "credits" or "license()" for more information.
>>> 6 + "A"
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    6 + "A"
TypeError: unsupported operand type(s) for +: 'int' and 'str'
>>>
```

最後の行を見てみると、整数 (int) と文字 (str) に対して加算 (+) は実行できない、というエラーを表示しています。仮に、文字同士の加算を文字の連結だと拡大解釈したとしても (現に Python はこの拡大解釈をします)、整数と文字の加算は解釈が難しい (つまり存在しない) ですよね[4]。

このように、データが本来備えている特徴による分類を**データ型**と呼びます。データ型の利点の一つとして、プログラムの誤りを減らせることが挙げられます。例えば、計算式の項に “A” と明示されていれば、人間がプログラムの作成時にその誤りを発見することは容易です。しかし、6 + A (A は変数の A を表わします) のように変数が用いられると、変数の中身を事前に (人間がプログラムを目で追いながら) 調べることは難しくなるため、プログラムに誤りが入り易くなります。

以下に Python で利用できる基本データ型と演算子を挙げます。各データ型に、どの演算子が適用できるかは少々複雑なので、本講義で取り上げるアルゴリズムを扱う範囲では、常識的な判断をすればよいでしょう。

データ型:

Python で使う文字データは、文字列を「“ ”」(ダブル クォーテーション) あるいは「’ ’」(シングル クォーテーション) で囲みます。例えば、こんな感じ。

```
"abc"
'abc'
```

プログラミング言語によっては、ダブル クォーテーションとシングル クォーテーションとで扱いが異なるものもありますが、Python では両者に違いはありません。

数値データは、数値の種類や精度に応じて、以下のように記述します。

- 整数型: 24, -64 など。正負共に極めて桁数の大きな値は扱えません。
- 長整数型: 10000000000000000000L など。整数型では扱えない桁数の整数を扱います。末尾に付ける記号は “l” あるいは “L” のどちらでもよいですが、見易さを考えると “L” をお勧めします。
- 浮動小数点型: 1.414, 3.142e2, 2.718E-7 など。3.142e2 = 3.142×10^2 を意味します。指数表記を示す記号は、“e” あるいは “E” のどちらでもよいです。
- 8 or 16 進表現: 整数については、10 進表現の他に 8 進表現あるいは 16 進表現が可能です。8 進表現では数値の先頭に “0” を、16 進表現では “0x” を付けます。

Python には、文字列型/数値型以外の基本データとして、論理型があります (“論理” とは、文字や数に比べてイメージが湧かないですよね)。論理型の値は、「はい/いいえ」の二つしかありません。Python では、「はい or 真」を記号 “True” で、「いいえ or 偽」を記号 “False” で表します。論理型は複合文の制御などに用いられます。

[4]何かすっきりしない方は、例えば次のように考えてみると、よりはっきりするのではないでしょうか。整数と整数の加算 → 整数、文字と文字の加算 (連結) → 文字ですが、整数と文字の加算 → 整数 or 文字 or その他のどれでしょう。

演算子:

次は、データの処理についてです。例えば、数値計算や大小比較といったデータの処理では、演算子を用いた式を作成します。Python で式を作る際に利用できる主な演算子には、以下のようなものがあります。

代数演算子: +, −, *, /, % (割り算の余り), ** (べき乗), // (整数商)

ビット演算子: ~, &, |, ^, <<, >> (本講義では扱わない予定)

代入演算子: =, += (演算後に代入の意/以下同じ), − =, *=, /=, %=, **=, //=, &=, | =, ^=, <<=, >>=

比較演算子: ==, != (等しくない), <, >, <=, >=

論理演算子: and, or, not

また、上の各演算子には優先順位が設定されており、一つの式に複数種類の演算子を混在させた場合は、括弧などで優先順位を指定しない限り、下記の順番で演算が行なわれます。

優先順位が高い

```
+  -  ~                  (単項演算子)
**                       (代数演算子)
*  /  %  //              (〃)
+  -                     (〃)
<<  >>                   (ビット演算子)
&                        (〃)
^                        (〃)
|                        (〃)
<  <=  >  >=  ==  !=     (比較演算子)
not                      (論理演算子)
and                      (〃)
or                       (〃)
```

優先順位が低い

例えば、こんな感じ。

```
Python 3.6.5 |Anaconda, Inc.| (default, Mar 29 2018, 13:32:41) [MSC v.1900 64 bit (AMD64)]
on win32
Type "copyright", "credits" or "license()" for more information.
>>> 20 + 4 * 3 - 5
27
>>> (20 + 4) * (3 - 5)           # この式と上の式については、特に問題ないですよね。
-48
>>> a = 2
>>> b = 5
>>> c = 7
>>> a == b
False
>>> a + b == c
True
>>> (a == b) and (a + b == c)    # "代数 → 比較 → 論理" の順で評価しています。
False
>>> (a == b) or (a + b == c)
True
>>>
```

3 文

Python では通常、前節で説明した演算式は 1 行で書きます。複数の演算式を 1 行に書くこともできます。この場合は、演算式を ";" で区切ります。

しかし、本講義で取り扱う関数の定義/条件分岐/繰り返しなどは、1 行で書くことができません[5]。このようなプログラムは複合文と呼ばれ、次のような構造になります。

```
ヘッダ:
  文 1
  文 2; 文 3
  …
```

ヘッダは、内部の文 (この例では文 1, 文 2 など/これらを総称して**ブロック**と呼びます) をどのように実行するかといった制御を行ないます。また、ブロック内の文は、必ず字下げ (これを**インデント**と呼びます) をします[6]。次の例のように、字下げをしない場合、複合文と文 4 は別のプログラムとして処理されます (文 4 はヘッダによる制御の対象外となります)。

```
ヘッダ:
  文 1
  文 2; 文 3
  …
文 4
```

また、次のような記述は、複合文と明示しながらブロックがない (実際には複合文ではない) ので、文法エラーとなります。

```
ヘッダ:
文 1
  文 2; 文 3
  …
```

複合文の例として、BMI (Body Mass Index) の計算を行なう関数の定義を示します。

```
def bmi(height, weight):
  bmi_val = weight / height**2
  return(bmi_val)
```

このプログラムを作成し、IDLE 上で実行する際は少し注意が必要です。ファイル モードで実行する場合は、特に問題はありません。しかし、対話モードで実行する場合は、ブロックの終わりを示すために空行を入れる (Enter キーを空押しする) 必要があります。例えば、こんな感じ。

```
Python 3.6.5 |Anaconda, Inc.| (default, Mar 29 2018, 13:32:41) [MSC v.1900 64 bit (AMD64)]
on win32
Type "copyright", "credits" or "license()" for more information.
>>> def bmi(height, weight):
        bmi_val = weight / height**2
        return(bmi_val)
(ここに空行を入れている)
>>> bmi(1.8, 90)
27.777777777777775
>>>
```

[5] もちろん、何らかの区切り規則を用意して、紙テープへ書くように細長く書き続けることは理論上可能ですが、人間がプログラムの内容を理解することが困難になり、データ型の節で述べたようにプログラムへ誤りが入り易くなるため、現実的ではありません。

[6] 多くの言語では、"{" と "}" や "begin" と "end" によりブロックの境界を示しますが、Python ではインデントの有無により示すので、最初は少し戸惑うかも知れません。

もし、空行を入れなかった場合は、このようにエラーとなります。

```
Python 3.6.5 |Anaconda, Inc.| (default, Mar 29 2018, 13:32:41) [MSC v.1900 64 bit (AMD64)]
on win32
Type "copyright", "credits" or "license()" for more information.
>>> def bmi(height, weight):
        bmi_val = weight / height**2
        return(bmi_val)
bmi(1.8, 90)
SyntaxError: invalid syntax
>>>
```

4 コーディング規約

次に、実際にプログラムを作成する状況を想像してみましょう。一目で分かるような簡単なアルゴリズムであれば (例えば、二つの数字を比較して大きい方を決めるとか)、何も見ずにいきなり作成することもできますが、少々込み入ったアルゴリズムであれば、類似した既存のプログラムを参考にすることが多いでしょう (例えば、プログラミング関連の書籍には、数多くのコード例が掲載されていますね)。また、規模の大きいプログラムであれば、全てをゼロから作成するのではなく、もし既存のプログラムに流用できる部分があれば、流用した方が効率は良いですね。但し、流用に際しては、どんな改造が必要か、副作用はないか等をしっかり調査しなければなりません。このように、プログラムは一般に「書かれる」だけではなく、「読まれる」機会も非常に多いと言えます (もしかすると、読まれる方が多いかも)。つまり、プログラムとは、(上でも何度か述べましたが) プログラムを書く際に出来るだけ誤りが入らないようにするだけではなく、プログラムを読む際にも出来るだけ理解し易いように配慮する必要があるわけです。このような考えのもと、Python には、「書き方のお手本を示し、読み易いプログラムを書く」ための**コーディング規約 (PEP8)** が用意されています。

以下、簡単に Python のコーディング規約の一部を紹介します (これが全てではありません)。皆さんがプログラムを書く際は、できるだけこの規約に準拠して書くようにして下さい (その方が、誤りを発見し易くなります)。但し、あくまでこれは作法の話であり、ここから外れた場合は即文法誤りとなるわけではありません。要は、プログラムが読み易くなるように形を整えることが目的なので、自分はこうした方が読み易いという書き方があれば、プログラム全体をそれに合わせればよいのです。

行/インデント:

- インデント一つの幅は (半角) 空白 4 個にする。
- タブよりスペースを利用する (両者を混ぜるとエラーになる場合がある)。
- 1 行は 79 文字まで (それより長い場合は、例えば下記のように適当な場所で折り返す)。

```
# 関数呼び出しの例 1
hoge = function_name(var_one, var_two, var_three,
                     var_four, var_five, var_six)
# 関数呼び出しの例 2
hoge = function_name(
           var_one, var_two, var_three,
           var_four, var_five, var_six
       )
# 条件分岐の例
if (this_is_one_condition and
    that_is_another_condition):
    do_something()
```

また、折り返しに際しては、同じ扱いの部分 (上の例で言えば引数や条件) を縦に揃える。

- 扱いが異なる行はインデントの深さを変える (下記の例は、引数で折り返しているが、関数本体 (print) と引数が同じインデントになっているので良くない)。

```
# 関数定義の良くない例
def function_name(var_one, var_two, var_three,
                  var_four, var_five, var_six):
                  print(var_one)
```

空白:

- 複数の関数定義を書く場合は、各定義間に空白行を 2 行入れる。
- 複数の 1 行文をセミコロンにより結合しない。
- 括弧の外側および内側には、空白を入れない。
- カンマ, セミコロン, コロンの後ろに空白を一つ入れる。また、これらの前には空白を入れない。
- 2 項演算子の両側に空白を一つずつ入れる。
- 行末に、余分な空白は入れない。

命名規則:

- 関数名および変数名は全て小文字で表わし、読み易さに配慮して適宜アンダー スコア (“_”) を入れる。
- 定数名は全て大文字で表わし、読み易さに配慮して適宜アンダー スコア (“_”) を入れる。
- 単一の文字 “l” (小文字のエル)、“O” (大文字のオー)、“I” (大文字のアイ) を名前に使ってはならない。
- 既に Python の構文等で使われている文字列 (例えば if とか) を別途使用したい場合は、名前の最後にアンダー スコア (“_”) を付ける。

その他:

- 文字コードは UTF−8 を使用する。

5 コメント

最後に、プログラムのコメントについて説明します。コメントとはプログラムに関する注釈のことで、作成したプログラムの意図や動作を書き手が正しく理解する、あるいはこれらを正しく読み手に伝える役目があります。コメントを適切に記述することによって、プログラムの拡張や流用がし易くなります。また、プログラムがうまく動かない時には、その原因を突き止めるための手掛かりとなります。さらに、何らかの理由によりプログラムの一部を実行させたくない場合は、その部分をコメント扱いにすることで、一時的に実行を停止することもできます (これを “コメント アウト” と言います)。このように、コメントには大変重要な役割があり[7]、皆さんもできる限りコメントを入れるように是非心掛けて下さい。

Python のプログラムにコメントを入れる方法には、以下の二つがあります。

1 行コメント: 「#」は、コメント文が 1 行の場合に利用します。プログラム中に#がある場合、#から右にある文字は以後行末までコメント文として扱われ、処理系による実行対象となりません (例としては、上にあるプログラム例や実行例を参照して下さい)。

複数行コメント: 「’’’」(シングル クォーテーション 3 連続) あるいは「”””」(ダブル クォーテーション 3 連続) は、コメント文が連続して複数行ある場合 (あるいは、プログラムの一部をコメント アウトする場合) に利用します。プログラム中に’’’ あるいは ””” がある場合、これと次に現れる’’’ あるいは ””” とに挟まれた部分はコメント文として扱われ、処理系による実行対象となりません。例えば、こんな感じ。

[7] これを際立たせるために、別の章立てにしました

```
# 関数 bmi は処理系による実行対象
def bmi(height, weight):
  bmi_val = weight / height**2
  return(bmi_val)

# 関数 triarea は処理系による実行対象ではない (コメント アウトされている)。
"""
def triarea(w, h):
  s = (w * h) / 2.0
  return(s)
"""
```