A2-0 Python 入門

機械学習を対象としたプログラムの場合、一言で Python のコードといっても、実はこの後で紹介する NumPy や Pandas といった、標準的に利用されているライブラリの機能を使っていることが多いです。当記事では、その中でも元の Python そのものの文法について説明します。

基本的なデータ型

どのプログラミング言語でも、データ型が重要です。Python 入門も Python で標準的に扱えるデータ型の説明から始めます。

4 つの基本型

Python には様々なデータ型がありますが、その大部分は基本型と呼ばれるデータ型が構成要素です。そこで基本型について説明します。よく用いられる 4 つの基本型は次のとおりです。

整数型: 整数を表現するのに用いられます。
浮動小数点数型: 浮動小数点数を表現するのに用いられます。
文字列型: 文字列を表現するのに用いられます。
ブーリアン型: True または False の 2 種類の値(論理値)のみを取る変数型です。

それでは、コード A2-0-1 で個々の変数型定義を見ていきましょう。

コード A2-0-1 基本型変数の定義

```
# 整数型
# 数値表現が整数の場合、代入先変数は自動的に整数型になります。
a = 1

# 浮動小数点数型
# 数値表現に小数点が含まれると、代入先変数は自動的に浮動小数点型になります。
b = 2.0

# 文字列型
# 文字列はシングルクオート(')で囲みます。
# あるいはダブルクオート(")でもいいです。
c = 'abc'

# ブーリアン型
# True または False と取る変数の型です。
d = True
```

Java などの言語では、変数は宣言するときに型も明示的に指定します。Python では、そのようなことはなく、どの値が代入されたかにより、型が自動的に定まります。上のプログラムにより、変数 a は整数型に、変数 b は浮動小数点型に、変数 c は文字列型に、変数 d はブーリアン型に設定されることになります。

print 関数と type 関数

今、定義した a から d までの変数にどのような値が入っているか、またどの型が設定されているかを確認します。変数の内容を見るためには print 関数を、型を調べるためには type 関数を利用します。まずは、整数型の変数 a について調べてみましょう。実装はコード A2-0-2 になります。

コード A2-0-2 整数型の変数の値と型表示

```
# 整数型の変数 a の値と型表示
print(a)
print(type(a))
```

```
1
<class 'int'>
```

値は 1、型は「class 'int'」であることがわかりました。
次に浮動小数点数型の変数の値と型を表示します。実装はコード A2-0-3 です。

コード A2-0-3 浮動小数点数型の変数の値と型

```
# 浮動小数点数型の変数 b の値と型表示
print(b)
print(type(b))
```

```
2.0
<class 'float'>
```

浮動小数点数型の型は「class 'float'」となっていることがわかりました。
次に文字列型の変数の値と型を表示します。実装は A2-0-4 です。

コード A2-0-4 文字列型の変数の値と型表示

```
# 文字列型の変数 c の値と型表示
print(c)
print(type(c))
```

```
abc
<class 'str'>
```

文字列型の変数の型は「class 'str'」だとわかりました。
最後にブーリアン型の変数の値と型を表示します。実装はコード A2-0-5 になります。

コード A2-0-5 ブーリアン型の変数の値と型表示

```
# ブーリアン型の変数 d の値と型表示
print(d)
print(type(d))
```

```
True
<class 'bool'>
```

ブーリアン型の場合は、値は True か False かのいずれかを取ります。この 2 つの単語は Python では予約語[1]となっています。また、型は「class 'bool'」になっています。

リスト

リストは、複数の値を順番に並べて、全体を一つのまとまりとして扱うデータ構造[2]のことです。Java などのプログラミング言語で「配列」と呼ばれているデータ構造と同じものを指しています。

リストは、Python で複雑な処理を行う際最もよく利用されるデータ構造です。また、Python 固有の機能もいくつかあります。これらについて実習を進めていきましょう。

リスト定義

最初に、リストの定義から行います。リストは[]で囲まれた中にカンマ区切りで複数のデータを並べることで定義します。コード A2-0-6 で確かめてみましょう。

コード A2-0-6 リストの定義

```
# リストの定義
l = [1, 2, 3, 5, 8, 13]

# リスト変数 l の値と type
print(l)
print(type(l))
```

---

[1] プログラミング言語側ですでに利用しているため、変数名として利用できない単語をこう呼びます。

[2] 今まで「データ型」「データ構造」と呼んできた概念は厳密にいうと「クラス」と呼ばれるものに該当します。本書は Python の利用方法に焦点をおいているためクラスの厳密な説明は省略します。詳しく知りたい読者は別の参考書を参照してください。

```
[1, 2, 3, 5, 8, 13]
<class 'list'>
```

コード A2-0-6 では、リスト変数 l の定義をした後で、リスト全体を print 関数に渡しています。Python では、このような print 関数の使い方が可能で、結果は、リストの全要素を表示してくれます。また、type 関数の出力結果は「class 'list'」となっていることがわかります。

リスト要素数

プログラムでリストを扱っていると、そのリストの要素が全体でいくつあるのか知りたい場合がよくあります。その方法を示すのが、次のコード A2-0-7 です。

コード A2-0-7 リストの要素数

```
# リストの要素数
print(len(l))
```

```
6
```

Python ではこのような目的で len 関数が用意されています。この関数を呼び出すことで、リスト変数 l の要素数が 6 であることがわかりました。

リスト要素参照

リストに対する操作でよくあるのが、リストの特定の要素にアクセスする処理です。次のコード A2-0-8 にその方法が示されています。

コード A2-0-8 リストの要素参照

```
# リストの要素参照

# 最初の要素
print(l[0])

# 3 番目の要素
print(l[2])

# 最後の要素（こういう指定方法も可能）
print(l[-1])
```

```
1
3
13
```

特定の要素にアクセスする場合は l[2]のような形で参照します。最初の要素のインデックスは 0 です。なので、l[2]は 3 番目の要素を意味します。

Python 固有の参照方法もあります。それは 3 つめのコーディング例の l[-1]という書き方です。「-1」とは「リストの一番最後」を意味します。同様に l[-2]、l[-3]といった書き方も可能です。それぞれどの要素を指すのか、自分で考えた上で実際に試してみてください。

部分リスト参照 1

リストの参照方法には Python 独特の文法がいくつかあります。その典型的な例がこれから説明する部分リスト参照です。これは l[(インデックス 1):(インデックス 2)]のように、コロンを挟んで二つのインデックス値を指定し、その範囲内の要素をすべて参照する方法です。言葉の説明だけではわかりにくいので、次のコード A2-0-9 で確認しましょう。

コード A2-0-9 部分リスト参照 1

```
# リストの部分参照 1

# 部分リスト インデックス:2 以上 インデックス: 5 未満
print(l[2:5])

# 部分リスト インデックス:0 以上 インデックス: 3 未満
print(l[0:3])

# 開始インデックスが 0 の場合は省略可
print(l[:3])
```

```
[3, 5, 8]
[1, 2, 3]
[1, 2, 3]
```

コード A2-0-9 の最初の参照では l[2:5]という書き方をしています。この場合、3 番目の要素である l[2]から始まり、6 番目の要素である l[5]の一つ手前、つまり l[4]までの部分リストが参照される形になります。次の参照では l[0:3]という書き方なので、l[0]から始まり、l[3]の一つ手前、つまり l[2]までの部分リストになります。最後の例では開始要素の値が省略されています。この場合、開始要素は先頭要素つまり l[0]を意味するルールです。なので、l[0:3]と l[:3]は同じ結果を意味することになります。

部分リスト参照 2

部分リストの参照パターンをもう少しいろいろ見てみましょう。コード A2-0-10 がその実装です。

コード A2-0-10 部分リスト参照 2

```
# リストの部分参照 2

# 部分リスト インデックス:4 以上最後まで
# リストの長さを求める
n = len(l)
print(l[4:n])

# 最終インデックスが最終要素の場合は省略可
print(l[4:])

# 後ろから 2 つ
print(l[-2:])

#最初も最後も省略するとリスト全体になる
print(l[:])
```

```
[8, 13]
[8, 13]
[8, 13]
[1, 2, 3, 5, 8, 13]
```

今回は「5 番目の要素から最後の要素」までを参照対象にしたいとします。今まで習った話で、先頭のインデックスを 4 にすることはすぐわかりますが、問題は「最後」の要素の指定をどうするかです。最初のパターンは、配列の長さを返す len 関数を使って、最後の要素にあたるインデックスを調べる方法です。

l[(インデックス 1):(インデックス 2)]の書き方では、先頭の(インデックス 1)同様、後ろの(インデックス 2)も省略可能です。省略した場合は「最後の要素まで」という意味になります。2 番目のアクセスパターンはこの性質を利用してコーディングしたものです。確かに同じ結果が返ってきているので、実は、長さ n を調べる必要がなかったことがわかります。

最後の実装は、2 つめのパターンにもう一工夫加えたものです。「後ろから 2 つ目の要素」はインデックス値として-2 でも表すことができます。それで、l[-2:]という書き方をすると、「配列の後ろから 2 つ目から最後まで」という意味になるのです。この参照法は 5 章の実習でも出てくることになるので、しっかり覚えるようにしてください。

コード A2-0-10 の最後の例は l[:]と、先頭(インデックス 1)も終了(インデックス 2)も両方省略されています。この場合、もう想像がついたと思いますが、元のリスト全体を意味し

ます。リストのインデックスの場合、この書き方は(元の変数と同じなので)まったく意味がないのですが、この後で学ぶ NumPy では重要な意味を持つ場合が出てきます。その点は NumPy の解説で説明します。

タプル

リストとよく似たデータ構造として「タプル」と呼ばれるものがあります。厳密にはリストではないのですが、ここでまとめて説明することにします。コード A2-0-11 を見て下さい。ここにタプルの定義方法と典型的な利用方法を示しました。

コード A2-0-11 タプルの定義方法と典型的な利用方法

```
# タプルの定義
t = (1, 2, 3, 5, 8, 13)

# タプルの値表示
print(t)

# タプルの型表示
print(type(t))

# タプルの要素数
print(len(t))

# タプルの要素参照
print(t[1])
```

```
(1, 2, 3, 5, 8, 13)
<class 'tuple'>
6
2
```

コード A2-0-11 をみればわかるように、タプルとリストで異なるのはその定義方法です。リストが[] でデータを定義したのに対して、タプルは()でデータを定義します。また、type 関数で型表示をすると「class 'tuple'」と返ってきます。要素数を len 関数で調べるとか、要素の参照を t[1]のような形で行うところはリストとまったく同じでこの動きだけみるとリ

ストとの違いがわかりません。

しかし、リストとタプルの振る舞いで決定的に異なる点が一つあります。それは、次のコード A2-0-12 でわかります。

コード A2-0-12 タプルへの代入

```
t[1] = 1
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-12-d6b0ce29b2aa> in <module>
----> 1 t[1] = 1

TypeError: 'tuple' object does not support item assignment
```

コード A2-0-12 は、事前定義済みのタプル変数である t の特定の要素 t[1]を、別の値で書き換えようとしたものです。すると、上記のようなエラーになってしまいました。このように一度定義したタプルは、書き換えができないのです[3]。試してもらえばわかりますが、リスト変数の場合は、一度定義したリスト変数の特定の要素の値を書き換えることは可能です。

Python では、実はいろいろなところで、意識することなくタプルが使われています。その一例をコード A2-0-13 とコート A2-0-14 で示します。

コード A2-0-13 2 つの変数の組を新しい変数に代入

```
x = 1
y = 2
z = (x, y)
print(type(z))
```

```
<class 'tuple'>
```

[3] このような性質をプログラミング言語の用語で「イミュータブル」といいます。

コード A2-0-13 は変数 x と変数 y を(x, y) という形に表現して全体をまとめて新しい変数 z に代入しています。新しい変数 z の型は「class 'tuple'」です。これだけ見ると、なんの変哲もないコードに見えます。

次のコード A2-0-14 は、コード A2-0-13 で定義した新しい変数 z を利用した実装例です。

コード A2-0-14 2 つの変数に同時に値を代入

```
a, b = z
print(a)
print(b)
```

```
1
2
```

なんと、a と b という変数に対して同時に代入することができました。なぜこのようなことができたのでしょうか？

実は「a, b」と書くとこの式もタプルになっています。Python では、代入を意味する等式(=)の右辺と左辺が要素数が等しいタプル同士の場合、複数の変数へ同時に代入することができるのです。本書 3.3.4 項で説明している train_test_split という関数は、このようなタプルの性質を利用して、複数の変数に同時に値を返していることになります。

辞書

リストと並んで重要なデータ構造に辞書(プログラミング言語によってはハッシュテーブルと呼ばれる場合もあります)があります。Python での辞書の扱い方について簡単に説明しましょう。

辞書の定義

最初は辞書の定義方法です。コード A2-0-15 に具体例を示しました。

コード A2-0-15 辞書の定義

```
# 辞書の定義
my_dict = {'yes': 1, 'no': 0}

# print 文の結果
print(my_dict)

# type 関数の結果
print(type(my_dict))
```

```
{'yes': 1, 'no': 0}
<class 'dict'>
```

コード A2-0-15 の一番上が、辞書の定義方法です。JSON の書式と同じで、{}の囲みの中に、「キー値: 値」の形式の並びをカンマ区切りで記述します。

print 文の結果は、定義の内容がそのまま表示されます。また、type 関数で型を調べると「class 'dict'」であることがわかります。

辞書の参照

次のコード A2-0-16 で定義した辞書の参照方法を示します。

コード A2-0-16 辞書の参照方法

```
# キーから値を参照

# key= 'yes'で検索
value1 = my_dict['yes']
print(value1)

# key='no'で検索
value2 = my_dict['no']
print(value2)
```

```
1
0
```

コード A2-0-16 を見ればわかる通り、辞書からキーによる検索をする場合、my_dict['yes']のような書式で参照をします。リストの参照と似ていてややこしいのですが、[]の内部が整数でなく文字列の場合、辞書参照を意味することになります。

辞書の追加

一度作った辞書に新しい項目を追加する方法を示します。実装はコード A2-0-17 になります。

コード A2-0-17 辞書に新しい項目の追加

```
# 項目追加
my_dict['neutral'] = 2

# 結果確認
print(my_dict)
```

```
{'yes': 1, 'no': 0, 'neutral': 2}
```

今作った my_dict という辞書に新しい項目'neutral'を追加したい場合
my_dict['neutral'] = 2
のような書き方をすればいいです。再度、my_dict 全体を print 関数にかけることで、新し

い項目が追加されていることが確認できます。

制御構造

今まで、「基本データ型」「リスト型」「辞書型」というデータ型を中心に Python の文法を見てきました。プログラミング言語としては、データ型と並んで制御構造が重要です。代表的な制御構造である「ループ処理」「条件分岐(if 文)」「関数定義」について、順に説明します。

ループ処理

Python では、何通りかのループ処理の方法がありますが、一番典型的なのは、リストを引数として、その要素一つ一つに対して処理を行うパターンです。その実装例をコード A2-0-18 で示します。

コード A2-0-18 ループ処理

```
# ループ処理

# リストの定義
list4 = ['One', 'Two', 'Three', 'Four']

# ループ処理
for item in list4:
    print(item)
```

```
One
Two
Three
Four
```

事前に用意した list4 というリスト変数に対して for item in list4: という書き方をすると、そこから先の制御構造の内部では、リストの各要素を item という変数名で参照できます。コード A2-0-18 では一番単純に各要素を print 文で表示しただけですが、もっと複雑な処理もこの中で行えます。

ここで非常に重要な文法上の決まりがあります。それは、Python では制御構造の内部は

行のインデント[4]により規定されるというルールです。Java など他の言語でも、人間がプログラムを組むときは、見やすさ、わかりやすさから制御構造の内部をインデントすることはよく行われます。Python では、この慣用的なルールを言語の文法として規定してるのです。こうするメリットととして、他の言語で必要な制御構造の内部を示す括弧が不要になる点があります。この性質があるので、Python は他の言語よりシンプルにコードが書けるのです。一方で慣れないと、この文法を忘れてエラーを起こすので、この点は注意して下さい。

コード A2-0-18 の中で print 文だけ中途半端な場所から始まっているのは、このような深い意味があったのでした。

range 関数を使ったループ処理

コード A2-0-18 のようにループ処理対象のリストがすでにできあがっている場合のループ処理の方法はわかったと思います。では、Java などでよく行われる（for i =0, i < N, i++）のような、整数値インデックスでループ処理を回す場合はどうしたらいいのでしょうか？

ここでよく使われるのが range 関数です。まずは、その実装をコード A2-0-19 で見ていきましょう。

コード A2-0-19 range 関数を使ったループ処理

```
# range 関数と組み合わせ
for item in range(4):
    print(item)
```

```
0
1
2
3
```

コード A2-0-19 を見ると、range(4)の結果が[0, 1, 2, 3]というリストになっていることがわかると思います。range 関数とは、まさにこのようなリストを自動生成する関数です。

コード A2-0-19 では、range 関数の引数は 1 つだけでした。range 関数は引数を 2 つ、または 3 つ取ることもできます。それぞれどのような結果になるか、コード A2-0-20 とコード A2-0-21 で確認しましょう。

コード A2-0-20 引数 2 つの range 関数

[4] ブランクやタブを入れて行の開始地点を右に寄せることを「インデント」と呼びます。

```
# 引数 2 つの range 関数
for item in range(1, 5):
    print(item)
```

```
1
2
3
4
```

コード A2-0-20 の結果から、引数を 2 つ取る range 関数では、最初の引数は開始地点の整数値であることがわかります。

コード A2-0-21 引数 3 つの range 関数

```
# 引数 3 つの range 関数
for item in range(1, 9, 2):
    print(item)
```

```
1
3
5
7
```

コード A2-0-21 の結果から、引数を 3 つ取る range 関数の 3 つめの引数は、整数の増分値であることも確認できます。

条件分岐(if 文)

ループ処理と並んで重要な制御構造は if 文による条件分岐です。Python でどのような実装形式になるのか、次のコード A2-0-22 で確認してみましょう。

コード A2-0-22 if 文の実装例

```
# if 文のサンプル
for i in range(1, 5):
    if i % 2 == 0:
        print(i, 'は偶数です')
    else:
        print(i, 'は奇数です')
```

```
1 は奇数です
2 は偶数です
3 は奇数です
4 は偶数です
```

コード A2-0-22 では、先ほど説明した range 関数を使ったループ処理の内部に、if による分岐構造を入れて、二重に入れ子になった処理構造を持たせています。

今まで説明していなかった文法として「i % 2」があります。「これは整数 i を 2 で割ったあまり」を意味しています。あまりが 0 なら i は偶数、1 なら奇数なので、そのように分岐してメッセージを表示するプログラムになっています。

if による分岐の場合も処理構造の内部はインデントで示すルールは同じです。上の処理は単純なので、Java などどのプログラミング言語でも実装できるものですが、インデントを文法として使っている Python ほどシンプルにかける言語が他にないことも理解できると思います。

関数

プログラミング言語の制御構造として、ループ処理、条件分岐と並んで重要なものに関数があります。次のコード A2-0-23 では、Python での関数の定義方法と、呼び出しサンプルを示します。

コード A2-0-23 関数定義と関数呼び出し

```
# 関数の定義例
def square(x):
    p2 = x * x
    return p2

# 関数の呼出し例
x1 = 13
r1 = square(x1)
print(x1, r1)
```

```
13 169
```

Python で関数を定義する場合、行の頭に def と書いてその後
<関数名>(<引数の並び>):
という書き方をします。引数が 1 つもない関数もあり得て、その場合が def func(): のような書き方になります。
関数の場合もループ処理や if 文と同じで制御構造の内部はインデントされます。また、値を戻すときは、return <値> という形で戻します。

関数の呼び出し方法に関しては、特に難しい点はないので、解説は不要と思います。上のサンプルで、square という関数は、数値の引数 1 つを入力として、その 2 乗の結果を返す関数です。呼び出し例では、13 を引数としてその 2 乗の 169 が戻ってきているので、意図通りの動きになっていることがわかります。

Python では、前の説明した tuple の仕組みをうまく使うことで、複数の値を同時に返す関数を作ることも可能です。次のコード A2-0-24 では、その実例をお見せします。

コード A2-0-24 複数の値を戻す関数

```
#   関数の定義例 2
def squares(x):
    p2 = x * x
    p3= x * x * x
    return (p2, p3)

# 関数の呼出し例
x1 = 13
p2, p3 = squares(x1)
print(x1, p2, p3)
```

```
13 169 2197
```

これが、その関数 squares です。数値の引数 1 つを受け取って、2 乗を計算するところまでは前の関数と同じですが、もう一つ追加で 3 乗の値も計算しています。そして、return 文の中で(p2, p3)と 2 つの価のタプルを戻しています。こういう形の関数の場合、受け取る方も p2, p3 と 2 つの変数を用意しておくと、結果的に 2 つの変数の値を同時に受けることができます。Python ならではの便利な関数の定義の仕方ということができます。

その他の機能

ここまで、Python 独自の機能については、できるだけ詳しく説明しながら、Python の文法の基本的な部分をかけ足で説明してきました。ここでは、サンプルコードに出て来る、Python のそれ以外の利用法について、簡単に説明します。

ライブラリの導入

Python が機械学習に便利な理由の一つは、様々なライブラリが利用できる点にあります。本書の実行環境である、Google Colabolatry など Jupyter Notebook が動く環境でライブラリを利用する場合、2 段階の準備が必要なことがあります。

最初の段階は、ライブラリを Jupyter Notebook が稼働している OS 上にソフトウェアとして導入する段階です。NumPy や Pandas といった、機械学習で非常によく利用されるライブラリはすでに OS 上に導入ずみです。ところがあまり利用されないライブラリは、そのような状態になっていないことがあります。その場合、次のコード A2-0-25 のように、pip コマンドを使って導入を行います。

コード A2-0-25 ライブラリの導入

```
# 日本語化ライブラリ導入
!pip install japanize-matplotlib | tail -n 1
```

```
Requirement already satisfied: setuptools in /opt/anaconda3/lib/python3.7/site-
packages (from kiwisolver>=1.0.1->matplotlib->japanize-matplotlib)
(46.0.0.post20200309)
```

!pip と行頭に「!」がついているのは、Notebook 内から OS コマンドを発行する場合のルールです。コード A2-0-25 で導入している japanize-matplotlib とは、matplotlib というグラフ用ライブラリを日本語対応するためのもので、本書のサンプルコードでは標準的に利用しているため、サンプルコードの冒頭にかならずこのセルが含まれている形になっています。

ライブラリのインポート

OS 上にライプラリが導入済みであっても、すぐに利用可能なわけではありません。Notebook 上で利用するためには、もう一段階「インポート」という操作が必要です。その実装例を示しているのが、次のサンプルコード A2-0-26 になります。ちなみに、このコードは、当書籍のサンプルアプリ共通に定義されている共通処理の一部となります。

コード A2-0-26 ライプラリのインポート

```
# 必要ライブラリの import
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# matplotlib 日本語化対応
import japanize_matplotlib

# データフレーム表示用関数
from IPython.display import display
```

このコードを見ると import で始まっている行と、from で始まっている行の 2 通りがあ

ることがわかります。

その違いですが、import で始まっている import 文では、import したモジュールに含まれている関数がすべて利用可能になります。

逆に 「from x import y」 の形式の import 文は、「y という名前のライブラリのうち x という関数だけを利用する」という宣言になります。コード A2-0-26 の一番下の行の宣言により、「IPython.display」ライブラリの「display 関数」が利用可能ということになります。

また、「import pandas as pd」は、「pandas という名前のライブラリを pd という別名で利用可能にする」ということを意味します。コード A2-0-26 の冒頭 3 行の import 文は、機械学習のプログラミングで慣用的利用されている別名定義です。なので、いつもこの形で別名を宣言するものと理解して下さい。

ワーニング非表示

もう一つ、当書籍の共通処理の実装を説明します。コード A2-0-27 がその実装です。

コード A2-0-27 余分なワーニングの非表示

```
# 余分なワーニングを非表示にする
import warnings
warnings.filterwarnings('ignore')
```

ここでは、warnings という名前のライブラリをインポートし、その中で filterwarnings という関数を呼び出しています。Python では、ライブラリが日々バージョンアップをするなかで、「この関数が次のバージョンで使えなくなる予定です」という意味の警告メッセージを表示することがあります。このような警告メッセージを表示させないための設定がコード A2-0-27 になります。

数値の整形出力

本節の最後に数値の整形出力方法について説明します。Python では、今までの実習で見てきたとおり、最低限のデータの内容表示は、対象変数の型がリスト型や辞書型であったとしても print 関数に単に変数を渡すだけで十分です。一方で浮動小数点などの基本型に関しては、小数点以下の桁数指定など、より細かい単位で整形出力をしたい場合があります。そのような方法は、従来もいくつかあったのですが、Python 3.6 で新たに可能になった方式があり、短いコードで出力可能なので、本書の実習では標準的にこの方式を用いることにします。

具体的な実装例をコード A2-0-28 に示しました。

コード A2-0-28

```
# f 文字列の表示
a1 = 1.0/7.0
a2 = 123

str1 = f'a1 = {a1}　　a2 = {a2}'
print(str1)
```

```
a1 = 0.14285714285714285　　a2 = 123
```

f'xxx'と f で始まる文字列が python 3.6 から利用可能になった「f 文字列」と呼ばれる特殊な書式になります。その特徴はまず第一に変数の値を{変数名}の形で直接埋め込むことができる点です。コード A2-0-28 の str1 の式の中では「{a1}」と「{a2}」の形で 2 カ所の埋め込み定義があります。str1 を print 関数にかけることで、埋め込み結果を確認しています。

f 文字列の二つ目の特徴は、ここの埋め込み変数に対して、細かく書式指定ができる点となります。その具体例をコード A2-0-29 で示しました。

コード A2-0-29 f 文字列の詳細書式設定

```
# f 文字列の詳細オプション

# .4f  小数点以下 4 桁の固定小数点表示
# 04  整数を 0 詰め 4 桁表示
str2 = f'a1 = {a1:.4f}　a2 = {a2:04}'
print(str2)

# 04e  小数点以下 4 桁の浮動小数点表示
# #x  整数の 16 進数表示
str3 = f'a1 = {a1:.04e}　a2 = {a2:#x}'
print(str3)
```

```
a1 = 0.1429　a2 = 0123
a1 = 1.4286e-01　a2 = 0x7b
```

　最初の例では、浮動小数点型のデータに対して、
「固定小数点表示　小数点以下 4 桁」
の指定で表示してみました。また整数型に対しては、
「4 桁表示 0 詰め」を指定しています。
二番目の例では、浮動小数点型のデータに対して、
「浮動小数点表示　小数点以下 4 空」指定を、また整数型に対しては「16 進数表示」指定をしています。いずれも、想定通りの結果が得られていることがわかります。
　書式指定のより細かい文法のついては、下記のリンク先などの情報を参考にして下さい。
https://docs.python.org/ja/3/library/string.html#formatstrings