Python 入門　テキスト

2020 年 9 月版

明治大学
生田メディア支援事務室

はじめに

Python は、オブジェクト指向型のプログラミング言語の 1 つです。文法がとてもシンプルで可読性の高い言語です。最近では、機械学習、AI、科学技術計算、IoT といった単語と共に耳にすることが多いのではないでしょうか。Dropbox や Instagram、Pinterest といった Web サービスの開発にも使われています。Blender や Maya といった 3D モデリングツール、3D-CAD などの処理を Python で書くことも可能です。このように Python は幅広い分野で使われており、それに関連した豊富、かつ、高水準なライブラリが多数あります。マサチューセッツ工科大学の講義にも使われています※。

本テキストは、Python 入門者向けのテキストとなっています。まず Python の基本構文について説明します。その後、numpy と matplotlib という 2 つのモジュールについて実例を交えながら説明します。プログラミングは、実際に手を動かすことで理解が深まります。ただテキストを眺めるだけではなく、実際に自分でプログラムを書きながら読み進めることをおすすめします。

入門者向けではありますが、知っておくと良い情報や少し難しめの情報を「ステップアップコラム」として掲載しておきました。他のプログラミング言語に精通している方は、Python 特有のポイントに目を向けて学習を進めると、より多くの学びがあるかもしれません。余力のある方は、ステップアップコラムにも目を通してみてください。

Python のバージョンについて

Python は、1991 年にグイド・ヴァンロッサム氏によって生み出されてから、1994 年に Python1 系（Python1.○.○）、2000 年に Python2 系（2.○.○）、2008 年に Python3 系（3.○.○）がつくられ、徐々にバージョンアップしています。2020 年 1 月に 2 系のサポートが終了したことにより、現在サポートされているのは 3 系のみとなっています。

3 系同士であればバージョンによる大きな差はありませんが、2 系と 3 系との間には互換性がありません。つまり、2 系で作成した Python のプログラムは 3 系では動作しないことがあります。逆も同様です。2 系と 3 系ではプログラムの構文や標準で備わっている機能に違いがあるためです。Python スクリプトが、非互換でない機能だけで記述されている場合は動作します。

非互換のわかりやすい例として print 文が挙げられます。print 文は、2 系では print “文字列” と書きますが、3 系では print(“文字列”)と括弧つきで書きます。記述の違いに加えて、片方のバージョンでしか使えないオプションや関数があります。

また、デフォルトの文字コードにも違いがあります。2 系では ASCII ですが、3 系では UTF-8 がデフォルトの文字コードです。つまり、3 系では日本語を使用していても文字コードの設定が不要です。

本テキストでは、バージョンは 3 系を、OS は Windows 環境を前提としています。CentOS や macOS など別の OS を使っている方は、適宜読み替えてください。

※ 参考文献[4]を参照

目次

図一覧

表一覧

1. Python スクリプトを動かす

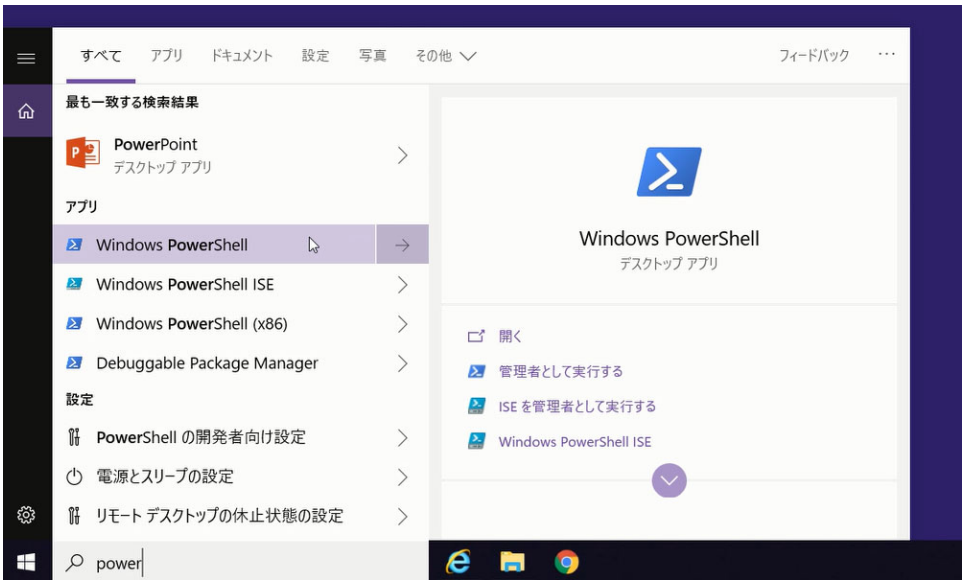
ここでは、対話モードを使った Python の簡単な使い方について説明します。次に、テキストエディタを使った Python ファイルの作り方及び、実行方法を説明します。最後に Jupyter Notebook の使い方を説明します。

1.1. 対話モードを使ってみよう

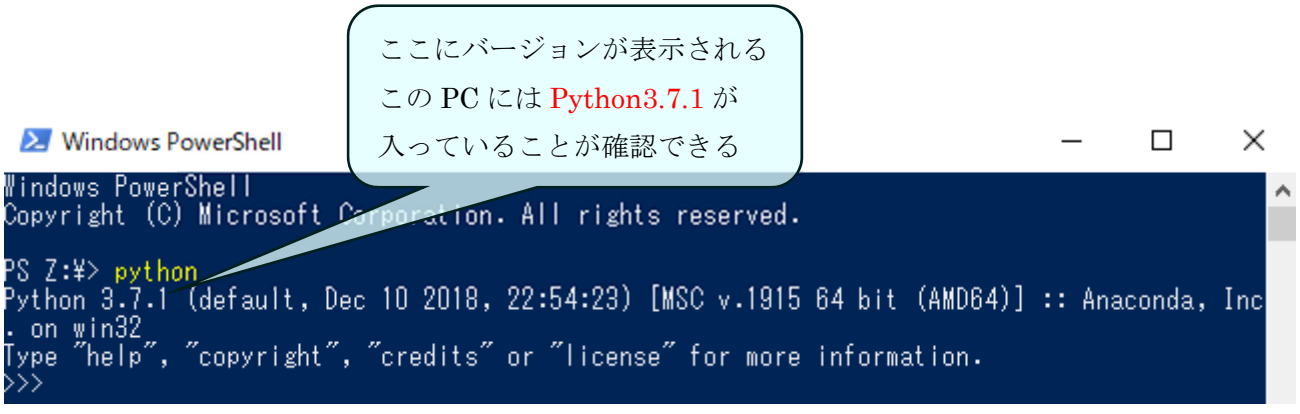
Python には、「対話モード（インタラクティブモード）」と呼ばれる実行形式があります。対話モードでは、Python スクリプトを入力すると、逐次実行されて、その結果をすぐに確認できます。入力に対する結果をすぐ得られるため、簡単な動作確認をするのに適した実行形式であると言えます。

対話モードを起動させてみましょう。まずは、Windows PowerShell を起動させます。左下の Windows マークを選択し、Power と入力、表示されたアプリの中に Windows PowerShell があるので起動します。

CentOS の場合は、画面左上にある「アプリケーション」→「システムツール」→「端末」で、macOS の場合は、「ターミナル」でそれぞれ代用可能です。



PowerShell が起動したら、「python」と入力しましょう。すると、対話モードが起動します。対話モードが起動すると、「>>>」が先頭に表示されます。



[実習 1] Python のバージョンを確認してみましょう

さっそく Python のプログラムを実行してみましょう。Hello World という文字を出力するだけの簡単なプログラムをまずは実行してみましょう。文字列の出力には、print 関数を利用します。

```
>>> print("Hello World")
Hello World
>>>
```

文字列の表示以外も処理可能です。Python の言語機能ほとんどすべてを使うことができます。試しに、四則演算をしてみましょう。

```
>>> 1 + 2
3
>>> 4 * 8
32
>>> 12 / 3
4.0
>>> 3 - 8
-5
>>>
```

対話モードでは、式を実行した結果が次の行に表示されます。そのため、print()を使用せずとも、計算結果が表示されています。対話モードを終了したい場合は、「exit()」と入力します。

対話モードを一度終了すると、入力されたデータはクリアされます。対話モードでは次章にて述べる変数や関数も定義できますが、対話モード終了時にこれらも同様にクリアされます。

1.2. help について

Python での対話モードでは help()コマンドが利用できます。このコマンドにより、各種情報のヘルプ表示をすることができます。例えば print 関数について知りたいときは「help("print")」と入力すれば見ることができます。

```
>>> help("print")
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='¥n', file=sys.stdout, flush=False)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file:  a file-like object (stream); defaults to the current sys.stdout.
    sep:   string inserted between values, default a space.
    end:   string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.
```

また、対話モードの中で「help()」と入力することで help のインタラクティブモードにすることができます。help のインタラクティブモードに入っている間は、先頭の「>>>」が「help>」になります。

```
>>> help()

Welcome to Python 3.7's help utility!

If this is your first time using Python, you should definitely check out
the tutorial on the Internet at https://docs.python.org/3.7/tutorial/.

Enter the name of any module, keyword, or topic to get help on writing
Python programs and using Python modules.  To quit this help utility and
return to the interpreter, just type "quit".

To get a list of available modules, keywords, symbols, or topics, type
"modules", "keywords", "symbols", or "topics".  Each module also comes
with a one-line summary of what it does; to list the modules whose name
or summary contain a given string such as "spam", type "modules spam".

help> print
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='¥n', file=sys.stdout, flush=False)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file:  a file-like object (stream); defaults to the current sys.stdout.
    sep:   string inserted between values, default a space.
    end:   string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.
```

ここから再び対話モードに戻るには「q」と入力します。

```
help> q

You are now leaving help and returning to the Python interpreter.
If you want to ask for help on a particular object directly from the
interpreter, you can type "help(object)".  Executing "help('string')"
has the same effect as typing a particular string at the help> prompt.
>>>
```

1.3. スクリプトファイルの実行

今回は、Z ドライブの直下に python フォルダを作成します。一旦、対話モードを終了して「mkdir python」と入力しましょう。

```
>>> exit()
PS Z:¥> mkdir python


    ディレクトリ: Z:¥


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----        [redacted]                        python
```

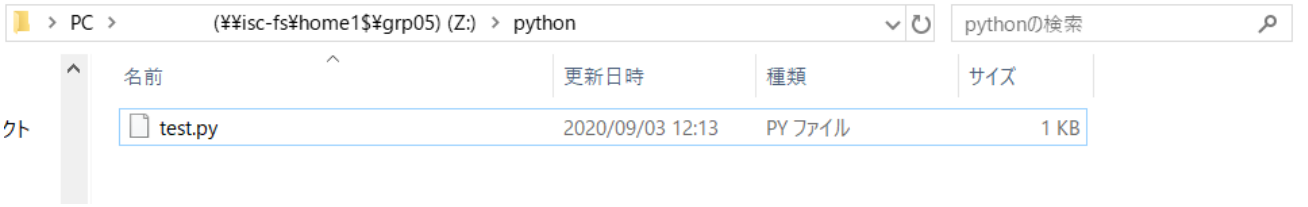
「cd python」と入力し、作成したフォルダに移動しておきましょう。

```
PS Z:¥> cd python
PS Z:¥python>
```

次に、秀丸エディタなど任意のテキストエディタを使用して「test.py」という py 拡張子ファイルを作成し、python フォルダに保存します。例として、先ほど対話モードで入力したのと同じ文字と計算結果を出力させてみましょう。

test.py

```
print("Hello World")
print("1 + 3 = " + str(1 + 3))
4 * 8
```



作成した Python ファイルを実行してみましょう。「python <実行するファイル名>」で実行できます。

```
PS Z:¥python> python test.py
```

実行結果は、次のように出力されます。

```
Hello World
1 + 3 = 4
```

print 関数を使用した箇所のみが出力されていることがわかります。対話モードと異なり、処理の結果は逐次出力されません。スクリプトで明示的に出力するようにした箇所でのみ出力されます。

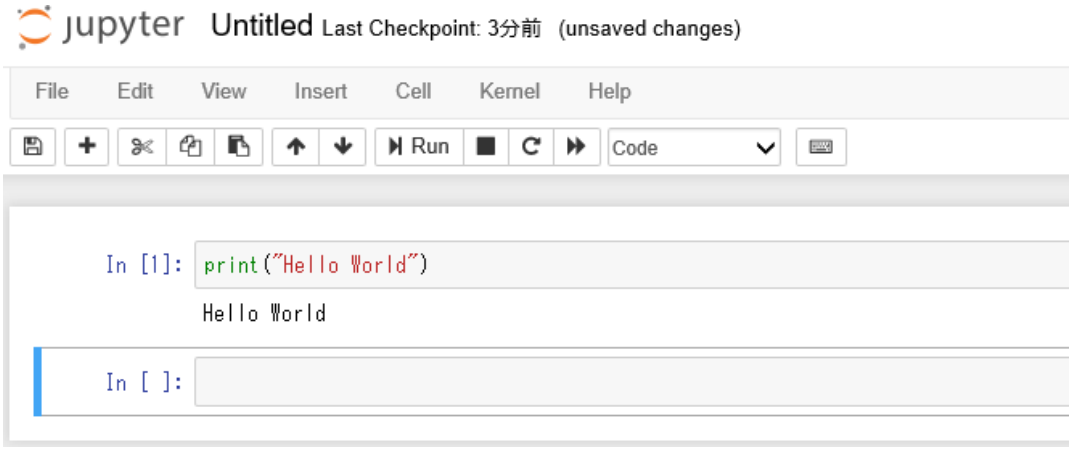
1.4. Jupyter Notebook について

Jupyter Notebook とは、オープンソースソフトウェア、オープンスタンダード、様々なプログラム言語などによるサービスを開発するためのもので、プログラムや説明の文章、実行結果等をまとめて管理することができます。統計モデリングやデータの視覚化、機械学習においてよく使われています。ブラウザで動作するため、他人とのプログラムの共有を容易に行うことができます(今回の演習では Jupyter Notebook を使用します)。

1.4.1. Jupyter Notebook の使い方

まずは、Jupyter Notebook を起動してみましょう Power Shell（または「端末」や「ターミナル」）で「jupyter notebook」とコマンド入力することで起動できます。新しいノートブックを作成してみます。画面右側の New ボタンから Python3 を選択します。選択されると新しいブラウザにノートブックが作成されます。ノートは.ipynb という拡張子のついたファイルとして作成されます。名前が『Untitled』になっているので名前を変えておきましょう。ノートブック上部の File から Rename を選択し『test』と変更します。

Hello World と出力する簡単なプログラムを実行してみましょう。実行は、Run というボタンで行います。また、キーボードの Shift キーを押しながら Enter キーを押すことでも同様の動作が得られます。このような主要なショートカットキーは、本節の最後に一覧で記します。



1.4.2. Cell について

ここで Cell について説明します。Jupyter Notebook では Cell という箱の中にコードを書き、順次 Cell を実行することで動かします。

Cell は＋ボタンで追加でき、ハサミのボタンで Cell を削除することができます。また、実行の方法は最初に行ったように Run ボタンで一つずつ実行しても良いのですが、一斉に実行する方法があります。ページ上部の Cell の欄から Run All を実行してみましょう。3 つの Cell を順番に実行することができると思います。ここで、同じ Cell を何度も実行していると左の Cell 番号が変わっていくのが分かると思います。また、Kernel の欄から Restart & Run All を選択することで、Cell 番号を 1 から振り直しつつ実行することもできます。

1.4.3. ノートの保存

Jupyter Notebook は、デフォルトでオートセーブ機能があります。手動でのセーブには File の欄から「Save and CheckPoint」を利用します。このアイコンを利用すると、保存と同時にその時点までのチェックポイントが作成されます。この機能で作成したチェックポイントにはいつでも戻ることが可能です。

1.4.4. ショートカットキー

Jupyter Notebook で使用できる主要なショートカットキーを表 1 に一覧で記します。ショートカットキーを活用することで、マウス操作を減らして作業をスムーズに進めることができるようになります。

表 1. Jupyter Notebook の主要ショートカットキー一覧

モード	ショートカットキー	機能
共通	Shift ＋ Enter	Cell を実行し、一つ下の Cell を選択
	Ctrl ＋ Enter	Cell を実行
	Ctrl ＋ S	ノートを保存
コマンドモード	Enter	編集モードにする
	↑, K	一つ上の Cell を選択
	↓, J	一つ下の Cell を選択
	A	一つ上に Cell を追加
	B	一つ下に Cell を追加
	X	選択中の Cell を切り取り
	C	選択中の Cell をコピー
	D ×２	選択中の Cell を削除
	Shift + M	Cell を結合
	Z	切り取りや削除した Cell を元に戻す
	O	選択中の Cell の実行結果を表示/非表示
	I ×２	実行中の Cell を停止
	0 ×２	すべての Cell の実行結果をリセット
	H	ショートカットキーを一覧表示（英語）
編集モード	Esc, Crtl ＋ M	コマンドモードにする
	Tab	コードを補完・インデントを入力
	Ctrl ＋]	インデントを入力
	Ctrl ＋ [	インデントを削除
	Ctrl ＋ A	すべて選択
	Ctrl ＋ Z	元に戻す
	Ctrl ＋ /	コメントアウト/解除

モードで「コマンドモード」と「編集モード」がありますが、Cell の枠が青色で Cell の追加や削除など Cell に対するショートカットキーを受け付ける状態をコマンドモードと呼び、Cell の枠が緑色で枠内に書かれたプログラムに対するショートカットキーを受け付ける状態を編集モードと呼びます。

なお、ショートカットキーは OS によって異なります。表 1 にも記した通り、コマンドモードでキーボードの H を押すとショートカットキーの一覧が表示されます。

2. Python の基本構文

2.1. 変数

プログラミングは、何かを計算処理することと、その計算結果を記憶しておくことの繰り返しで記述されます。変数とは、プログラミングで何かを記憶しておくための仕組みです。プログラム上で何かを記憶・取り出す際には変数を用います。

変数は、名前によって識別されます。Python では変数名に、英数字とアンダーバー「_」を使うことができます。なお、1 文字目はアンダーバー「_」、または、英字でなくてはなりません。この規則さえ守っていれば自由に名前をつけられます。

○ _variable, variable, variable1, Variable, VARIABLE
× 1, 11, 12345, 1variable

ただし、Python で使われているキーワードは名前にできません(and、 for、 if 等)。また、組み込み関数[5]とは異なる名前にした方が良いです。同じ名前にすると、組み込み関数が上書きされて動作しなくなります。

変数名のつけ方は、プログラミング言語ごとに異なる慣習があります。Python では変数名にスネークケースを用いるのが一般的です。スネークケースとは、小文字の英字と数字をアンダーバー「_」でつなぐ表記のことです。これは慣習であり、制約ではありません。変数名をスネークケース以外にしてもプログラムは動作します。ただし、慣習に従ってスネークケースとすることを推奨します。

2.1.1. 変数の定義

変数に値を割り当てるためには、「=」を用います。Python では、変数＝値 と書くだけで、変数に値を割り当てられます。変数に割り当てられている値を参照したい場合は、単に変数名だけを書きます。

```
In [1]: int_number = 100
        int_number

Out[1]: 100

In [2]: float_number = 100.
        float_number

Out[2]: 100.0

In [3]: name = "meijiro"
        name

Out[3]: 'meijiro'
```

複数の変数に同時に値を割り当てることもできます。意味のある単位でまとめて変数定義すると読みやすくなります。一方で、値が同じだからといってむやみにまとめて変数を定義すると、読みづらくなってしまう可能性があります。

```
In [1]: int_number, float_number, name = 100, 100., "meijiro"
        a = b = c = 100
```

値には、「型」があります。型は、値が文字列であるか数字であるか、もう少し詳しいものだと整数であるか、小数であるかといった情報を示すものです。型の詳細については後述します。変数宣言時に型を指定しなければならないプログラミング言語もありますが、Python は必要ありません。変数が宣言されるタイミングで、値を元に型が推測され、自動的に決定されます。このような性質を「動的型付け」と言います。動的型付けのおかげで、シンプルに変数を宣言できます。

2.1.2. 変数の型

Python では、型は表 2 のように分類できます。右辺の宣言方法によって型が決定します。表に型と宣言方法の組み合わせを示します。リスト、タプル、集合、辞書の詳細は、該当する章で説明します。

表 2. 変数の型について

型	宣言方法	例
int － 整数	整数、浮動小数点数（以下、小数と呼ぶ）、複素数の 3 種類を数値型といいます。数値をそのまま指定して宣言します。 ※虚数単位は i ではなく j を使います。	5
float － 浮動小数点数（実数の近似表現）		0.0012 1.2e-3
complex － 複素数		2 + 3j
str － 文字列	文字列を値として格納します。シングルクォート(')かダブルクォート(")で囲んで宣言します。	"Hello World" 'Hello World'
list － リスト	複数の要素を格納します。カンマ区切りで要素を列挙し、 **[]** で囲んで宣言します。	[1, 2, 3] [1, "a", 2, "b"] [1, [2, 3]]
tuple － タプル	複数の要素を格納します。一度作成した要素は変更できません。 **()** で囲んで宣言します。	(1, 2, 3) (1, "a", 2, "b") (1, (2, 3))
set － 集合	複数の要素を格納します。要素に順序はなく重複は除去されます。 **{}** で囲んで宣言します。	{1, 2, 3} {"Hello", "World", 1}
dict － 辞書	複数のキーと値のペアを格納します。キーと値のペアは key:value とし、 **{}** で囲んで宣言します。	{1: "Hello", 2: "World"} {"a": 1, "b": 2}
bool － 論理値	真・偽のいずれかの値を格納します。条件分岐などで使われます。	True (真) False (偽)

変数の型を調べたいときには、「type(変数)」とすることで現在の変数の型を調べることができます。

```
In [1]: type(5)
Out[1]: int

In [2]: type(3.)
Out[2]: float

In [3]: type([1,2,3])
Out[3]: list
```

2.1.3. 演算子

数値同士を足したり引いたりできるように、変数同士を足したり引いたりすることも可能です。こうした処理のことを演算と呼び、その演算を行わせるためのプログラム上の命令を演算子と呼びます。数値型と文字列型の演算子を紹介します。

表 3. 数値型と文字列型の演算子

数値型		文字列型	
演算子	処理	演算子	処理
x + y	足し算をする	s + t	文字列 s と t を結合する
x - y	引き算をする	s * x	文字列 s を x 回繰り返す
x * y	掛け算をする		
x / y	割り算をする		
x % y	割り算の余りを求める		
x ** y	累乗を計算する		
x // y	割り算の商（小数点切り捨て）を求める		

演算子の優先順序

1. **
2. *, /, %, //
3. +, -

数学の計算と同様に、() で囲んだ式が優先されます。

演算をすると、その結果が返されます。その結果をさらに変数に代入することもできます。演算子の具体的な使用例を次に示します。次の例だけでなく、表 3 の演算子を試して、挙動を確認しましょう。

```
In [1]: int_number = 10
        double_number = int_number * 2.0
        double_number
```
整数 × 小数
```
Out[1]: 20.0

In [2]: hello = "Hello"
        world = "World"
        hello_world = hello + world
        hello_world

Out[2]: 'HelloWorld'

In [3]: hello_n = hello * int_number
        hello_n
```
文字列 × 整数
```
Out[3]: 'HelloHelloHelloHelloHelloHelloHelloHelloHelloHello'
```

演算処理の結果、変数の型が変わることがあります。In[1]内の 2 行目のように整数と小数を乗算すると、結果は小数になります。また、文字列と整数を掛けると、文字列がその分繰り返されます。異なる型同士の演算を色々試してみると良いでしょう。

すべての型同士で演算可能なわけではありません。演算できない型同士の場合、エラーが発生します。例えば、文字列と整数は加算できないため、次のようなエラーが発生してしまいます。

```
In [1]: "1" + 2
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-1-42900e6a255b> in <module>
----> 1 "1" + 2

TypeError: can only concatenate str (not "int") to str
```

[実習 2] 整数×小数の計算を行い、答えを確認してみよう

2.1.4. 変数の型を変える

上記のような文字列と整数を足し合わせるような場合に、文字列を数値型に変換できると便利です。また、数値を文字列として扱いたい場合もあるかもしれません。その場合は、専用の関数を使います。関数は、処理をまとめて使い回しやすくしたものです。関数を呼び出すと、定義された処理が実行されます。詳しくは、2.9 関数で説明します。

「int(文字列)」や「float(文字列)」とすることで文字列を数値に、「str(数値)」とすることで数値を文字列に変換できます。「int(小数)」とすることで小数点以下を切り捨てることもできます。

このように変数やオブジェクトを別の型に変換することをキャスト（または型変換）といいます。

```
In [1]: int_number = 1
        string_number = "2"
        str(int_number) + string_number
Out[1]: '12'

In [2]: int_number + int(string_number)
Out[2]: 3

In [3]: float_number = 3.141592
        int(float_number)
Out[3]: 3
```

整数 → 文字列

文字列 → 整数

小数 → 整数

2.2. インデント

プログラミングにおいて、インデントとは行の頭にスペースなどで空白文字を入れて字下げを行うことを言います。

特に Python では同じ大きさの空白によってインデントされたまとまりをブロックとしてみなすので、インデントがとても重要になってきます。

findent.py

```
x = "test"

for i in range(5):
print(x)
```

```
In [1]: # findent.py

        x = "test"

        for i in range(5):
        print(x)
          File "<ipython-input-1-5570ec0b948b>", line 6
            print(x)
                ^
        IndentationError: expected an indented block
```

このように、正しくインデントされていない場合、インデントに関するエラーが表示されます。そこで、次のようにインデントをすることで正しく for 文（後述）のブロックを認識させることができます。

tindent.py

```
x = "test"

for i in range(5):
    print(x)
```

Python の公式コーディング規約では、半角スペース 4 文字分（Tab キー1 回分）のインデントを推奨しています。プログラム作成時にうまく動かないことがあれば、インデントのズレの可能性を考えましょう。

[実習 3] わざとインデントをずらしてみて、エラーが返ってくるか確認してみましょう

2.3. 条件分岐

「もし～ならば、この処理を行う」など条件に応じて処理を分岐させたい場合は、if 文を使用します。Python の if 文で出てくる言葉は「if」、「else」、「elif」の 3 種類です。

条件が 1 つの場合は if のみを、2 つの場合は if と else を、3 つ以上の場合は if と else の間に elif を必要な数だけ使用できます。4 つの条件に分岐させる場合の一般例は次の通りです。

if 条件 1**:**
　　条件 1 を満たす場合に実行したい処理
elif 条件 2**:**
　　条件 1 を満たさず条件 2 を満たす場合に実行したい処理
elif 条件 3**:**
　　条件 1, 2 を満たさず条件 3 を満たす場合に実行したい処理
else:
　　それ以外の場合（条件 1~3 を満たさない場合）に実行したい処理

条件の記述には、表 4 のような比較演算子を用いた式が使用できます。この比較演算子と条件分岐を用いることで、状況に応じた様々な処理が可能となります。

表 4. 比較演算子

比較演算子	比較内容
x == y	x と y が等しい
x != y	x と y が等しくない
x > y	x が y より大きい
x >= y	x が y 以上
x < y	x が y より小さい
x <= y	x が y 以下

比較演算子の結果は論理値で返されます。比較演算子は条件が正しい場合に True、正しくない場合に False を返します。そして if 文と elif 文は、比較演算子の結果が True の場合に実行されます。次に比較演算子の簡単な例を示します。

```
In [1]: 1 == 1
Out[1]: True

In [2]: 1 == 2
Out[2]: False

In [3]: 1 != 2
Out[3]: True

In [4]: 1 < 10
Out[4]: True

In [5]: 1 > 10
Out[5]: False
```

それでは、この比較演算子を使って if 文を記述してみましょう。Python ではインデントで if のブロックを構成します。C 言語や Java のようにブロックを構成するために中括弧{}を書く必要はありません。これは if 文だけでなく、for 文や while 文でも同様です。簡単な if-else の例を見てみましょう。

ifcondition.py

```
x = 1
if x == 1:
    print("x の値は 1 です")
else:
    print("x の値は 1 以外です")
```

elif を使えば、さらに細かく条件を指定できます。条件分岐の挙動を確認しやすくするために input 関数を利用しています。この関数はユーザの入力値を取得することができる関数です。取得した値を変数に割り当てることで、スクリプト内で入力値を扱えます。

elifcondition.py

```
x = int(input("整数を入力してください>>>"))
if x == 0:
    print("x は 0 です")
elif x < 0:
    print("x は負の値です")
else:
    print("x は正の値です")
```

[実習 4]　上記の elifcondition.py を作成し、実行させてみましょう

input 関数は、入力値を文字列として受け取るので、int()で整数に変換しています。値をいくつか入力してみて挙動を確認してみましょう。複雑な条件文を設定したい場合は、and または or を利用します。

and_or.py

```
x = int(input("整数を入力してください>>>"))
if 0 <= x and x < 10:
    print("0 以上、10 未満です")
elif x < 0 or 10 <= x:
    print("0 未満、もしくは、10 以上です")
```

上記の and のような条件文は、Python では次のようにシンプルに書けます。

```
if 0 <= x < 10:
    print("0 以上、10 未満です")
```

ステップアップコラム 「同値比較と同一比較」

この章で説明された == は、同値比較のための演算子です。あくまで値が同じであることを比較し、値が同じであれば、異なるオブジェクトであっても True を返します。一方で、 is は値が同じであってもオブジェクト自体が同一でないと True を返しません。これを端的に表したのが次の例です。

```
In [1]: a = [1,2,3]
        b = [1,2,3]

In [2]: a  == b
Out[2]: True

In [3]: a is b
Out[3]: False
```

いずれのオブジェクトも自身を一意に表す ID を持っています。ID が同じであれば、それらのオブジェクトは同一であるといえます。id() を使ってそれぞれのオブジェクトの ID を見てみると、オブジェクトが異なる ID を持っていることが確認できます。

```
In [1]: a = [1,2,3]
        b = [1,2,3]

In [2]: id(a)
Out[2]: 1764444117512

In [3]: id(b)
Out[3]: 1764443656328
```

None、True、False と同一であることを調べる場合には一般的に is を用いるのが良いと言われています。== はユーザが Python スクリプトを書くことで挙動を変えられるためです。

一方で、数値型や文字列型の値を比較する場合は、 == を用いる方が良いです。is を用いた場合、値によって結果が変わることがあるためです。値によって is の演算結果が変わってしまう例を次に示します。

```
In [1]: a = 123
        b = 123
        a is b
Out[1]: True
```

```
In [2]: a = 300
        b = 300
        a is b
Out[2]: False
```

2.4. リスト

リストは、複数のデータを順番に並べて保持できるオブジェクトです。

ある学部の学生 50 人の名前をプログラムで取り扱うケースを考えてみましょう。50 人の名前をそれぞれ変数として定義し、割り当てることもできますが、読みづらいコードとなってしまい、様々な面で非効率です。

inefficient.py

```
name1 = "John"
name2 = "Ewan"
name3 = "Jacob"
name4 = "David"
name5 = ...
# 名前の数だけ変数を用意するのは非効率
```

リストを使えば、複数のデータを 1 つの変数にまとめて保持できます。50 人の名前を書かなければいけない点は変わりませんが、nameN のような変数を書く必要はなくなりました。

efficient.py

```
names = ["John", "Ewan", "Jacob", "David", "Tom", ...]
```

リストのメリットは、データをひとまとめにすることだけではありません。リストは、データを効率的に操作するための多くの機能を持っています。本テキストでは、リストの作成・参照・追加といった基礎的な機能を取り扱います。オブジェクトであればすべてリストの要素にできます。Python は、すべてのものをオブジェクトとして取り扱います。つまり、Python で取り扱うものはすべてリストの要素にできるということです。

2.4.1. リストの作成

リストを作成するときは次のように[ ]で囲んで、要素をカンマ「,」で区切って並べます。

[オブジェクト 1, オブジェクト 2, ...]

リストの作成の具体例は次のとおりです。

```
In [1]: x = [1, 2, 3, 4, 5]
        x
Out[1]: [1, 2, 3, 4, 5]
```

リストの要素に型の制約はありません。つまり、異なる型の値を 1 つのリストに含めることができます。また、リスト自身もオブジェクトなので、要素として扱うこともできます。

```
In [1]: x = [1, "a", 2, "b", 3]
        x = [[1, 2], [2, 0], [3, 1]]
```

2.4.2. リストの要素の参照

要素を参照する場合は、要素の番号（インデックス）を整数で指定します。このようなオブジェクトをシーケンス型オブジェクトといいます。

Python のインデックスは 0,1,2,...と 0 番から数えます。インデックスには負の値も指定できます。その場合は、リストの後ろから要素を参照します。参照した値を変数に入れることもできます。

リストオブジェクト[インデックス]

```
In [1]: x = [2, 3, -5, 11, 7]
        x[1]
Out[1]: 3
```

```
In [2]: x[-1]
Out[2]: 7
```

リストの要素を複数取り出すこともできます。これはスライスと呼ばれています。開始インデックスからステップの値ずつ増やして、終了インデックスの範囲まで要素を取り出します。それぞれの項目は省略可能です。

リストオブジェクト[開始インデックス:終了インデックス:ステップ]

ここで、次の例のように[2:4]とすると、2 番の"c"と 3 番の"d"を表しており、4 番の"e"は対象にならない（終了インデックスは対象となる範囲に含まれない）ことに注意しましょう。開始インデックスだけでなく、ステップにも負の値を設定できます。例えば、[::-1]とすると、逆順のリストを取得できます。この挙動は非常に便利です。

```
In [1]: x = ["a", "b", "c", "d", "e"]
        x[2:4]
Out[1]: ['c', 'd']

In [2]: x[1:4:2]
Out[2]: ['b', 'd']

In [3]: x[2::]
Out[3]: ['c', 'd', 'e']

In [4]: x[::2]
Out[4]: ['a', 'c', 'e']

In [5]: x[::-1]
Out[5]: ['e', 'd', 'c', 'b', 'a']
```

len 関数を使えば、リストの長さを調べることができます。

```
In [1]: x = ["a", "b", "c", "d", "e"]
        len(x)
Out[1]: 5
```

len 関数はリストだけでなく、複数の要素を持つほとんどのデータに対して実行可能な関数です。後述するタプルや集合、辞書に対しても len 関数で長さを調べることができます。

リストの要素を並び替えることもできます。リストオブジェクト名に続けて「.sort()」と記述することで数値は小さい順に、文字列はアルファベット順に並び替えられます。逆順に並び替えたい場合は、「.reverse()」と記述します。

```
In [1]: x = [2, 3, -5, 11, 7]
        y = ["red", "blue", "yellow", "green"]
        x.sort()
        x
Out[1]: [-5, 2, 3, 7, 11]

In [2]: y.sort()
        y
Out[2]: ['blue', 'green', 'red', 'yellow']

In [3]: x.reverse()
        x
Out[3]: [11, 7, 3, 2, -5]
```

.sort()や.reverse()のように、変数名の後にピリオドでつながれたものをメソッドといいます。メソッドを呼び出すと、関数と同様に定義された処理が実行されます。詳しくは、「2.10 クラス」の中で説明します。

2.4.3. リストの要素の変更・追加・削除

リストに含まれる要素を変更したいときは、インデックスで変更したい要素を指定し、値を設定します。スライスでまとめて値を変更することもできます。

```
In [1]: x = [1, 2, 3, 4, 5, 6, 7, 8, 9]
        x[3] = "x"
        x

Out[1]: [1, 2, 3, 'x', 5, 6, 7, 8, 9]
```

```
In [2]: x[5:8] = ["a", "b", "c"]
        x

Out[2]: [1, 2, 3, 'x', 5, 'a', 'b', 'c', 9]
```

リストオブジェクト作成後に、要素を追加することもできます。append メソッドを使うとリストの最後に要素を追加できます。

```
In [1]: x = [1, 2, 3, 4, 5]
        x.append(10)
        x

Out[1]: [1, 2, 3, 4, 5, 10]
```

リストの途中に要素を追加するには、insert メソッドを使います。insert メソッドでは追加したい要素と何番目に追加したいかを指定します。

```
In [1]: x = [9, 8, 7, 6, 5]
        x.insert(3,0)
        x

Out[1]: [9, 8, 7, 0, 6, 5]
```

リストから要素を削除する方法もいくつかあります。remove メソッドを使うと、()の中に記述した要素を削除することができます。ここで、次の例における()の中の 3 は「3」という要素を表しており、3 番目の要素という意味ではないことに注意しましょう。リストの中に存在しない要素を削除しようとすると下の例のようにエラーメッセージが表示されます。

```
In [1]: x = [1, 2, 3, 4, 5]
        x.remove(3)
        x
Out[1]: [1, 2, 4, 5]
```

```
In [2]: x = ["a", "b", "c", "d", "e"]
        x.remove(3)
        x
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-2-5ec273c9089f> in <module>
      1 x = ["a", "b", "c", "d", "e"]
----> 2 x.remove(3)
      3 x

ValueError: list.remove(x): x not in list
```

○番目の要素を削除するには、pop メソッドを使います。

```
In [1]: x = ["a", "b", "c", "d", "e"]
        x.pop(3)
        x
Out[1]: ['a', 'b', 'c', 'e']
```

pop メソッドと似た方法で、del という関数を使ってもリストから要素を削除することができます。ただし、del はオブジェクトそのものを削除する関数です。そのため、後述するタプル、集合、辞書においても変数そのものを削除したいときは del 関数を使用できるので、覚えておいてください。

```
In [1]: x = [1, 2, 3, 4, 5]
        del x[2]
        x
Out[1]: [1, 2, 4, 5]

In [2]: del x[1:3]
        x
Out[2]: [1, 5]

In [3]: del x
        x
```

変数そのものを削除することもできる

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-3-1beca76b84e9> in <module>
      1 del x
----> 2 x

NameError: name 'x' is not defined
```

変数が削除されると、
変数が定義されていないことを表す

2.4.4. リストの要素の存在確認

リストにある値が含まれるかどうかは、in 演算子によって確認できます。リストに含まれている場合は True、そうでない場合は False となります。

確認したい値 **in** リストオブジェクト

```
In [1]: east_asias = ["Japan", "China", "North Korea", "South Korea", "Mongolia", "Taiwan"]
        "Japan" in east_asias
Out[1]: True

In [2]: "Singapore" in east_asias
Out[2]: False
```

要素の存在確認と条件分岐を組み合わせることで、要素が含まれる場合と含まれない場合で別々の処理をすることができます。

check_country.py

```
country = input("国名を入力してください>>>")
east_asias = ["Japan", "China", "North Korea", "South Korea",
"Mongolia", "Taiwan"]
if country in east_asias:
    print("東アジアの国です")
else:
    print("東アジアの国ではありません")
```

後述するタプル、集合、辞書においても in 演算子を使用できるので、ぜひ試してみてください。

[実習 5] 上記の check_country.py を作成し、Japan と America の 2 つを入力し結果を確認しましょう

ステップアップコラム 「None について」

Python では値が存在しない状態を None として定義しています。None は定数であり、None という状態で存在しています。オブジェクトがそもそも定義されていない場合は、別のエラーが発生します。つまり None とは、定義はされているが値が存在しない状態を表しています。

```
In [1]: print(y)
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-1-d9183e048de3> in <module>
----> 1 print(y)

NameError: name 'y' is not defined

In [2]: y = None
        print(y)
None
```

2.5. 繰り返し

「ある処理を 10 回行う」といった繰り返し処理は、プログラミングの至る所で現れます。繰り返し処理は for 文と while 文で実現できます。

2.5.1. for による繰り返し

for 文は、リストのような複数の値を持つ変数の要素を 1 つずつ取り出して、処理を実行します。

```
for 変数 in シーケンス型のオブジェクト:
    処理
else:
    終了後の処理
```

処理の部分では、for 文で指定した変数名で取り出した要素を参照できます。処理が終了したら、次の要素を取り出して同様に処理を実行します。シーケンス型のオブジェクトの最後の要素まで処理を終えたら、for ループは終了します。for ループが終了すると、else に書かれた終了後の処理が実行されます。簡単な例を見てみましょう。

forelse.py

```
for i in range(5):
    print(str(i) + "回目")
else:
    print("終わり")
```

range 関数については、本節の最後のステップアップコラムにて説明

else はオプションなので省略することができます。この例は次のように書いても同じ結果となります。

for.py

```
for i in range(5):
    print(str(i) + "回目")

print("終わり")
```

すべての処理が完了した時点で変数 i にアクセスしたい場合は、else でなければなりません。for-else の外で変数 i にアクセスできないためです。また、else に処理が書かれていると、終了処理であることが明確になります。else が書かれていない場合、それが for ループに関係する処理かどうかは、コードをきちんと読まない限りわかりません。

処理したいリストそのものを for 文に指定することで要素を 1 つずつ取り出すことができます。すべての要素を取り出し終えたら処理が終了する点は変わりません。

forlist.py

```
languages = ["Japanese", "English", "French"]
for lang in languages:
    print(lang)
```

```
In [1]: # forlist.py

        languages = ["Japanese", "English", "French"]
        for lang in languages:
            print(lang)

        Japanese
        English
        French
```

2.5.2. while による繰り返し

while も同様に繰り返し処理を実現しますが、条件のみを指定します。指定した条件を満たしている限り、処理に書かれた内容を実行し続けます。処理の部分に、while の条件が満たされなくなるような処理を書いておく、または、break 文で while ループを抜けるということをしなければ無限ループとなってしまいます。

while 条件**:**
    処理
else:
    終了後の処理

本テキストでは、 while の具体例については割愛します。リストなど要素の数だけ処理をしたい場合や特定の回数のみ繰り返したい場合は for 文、要素の数ではなく特定の条件下で同様の繰り返し処理をしたい場合は while 文と使い分けると良いでしょう。

2.5.3. 繰り返しの中止とスキップ

繰り返し処理をする過程で、ある条件のときに繰り返しを中断したり、以降の処理をせずに次の繰り返しに移ったりしたい場合があります。繰り返しを終了するには break を、次の繰り返しに移りループの最初から処理する場合には continue を利用します。次に簡単な例を示します。

break_continue.py

```
i = 0
while True:
    i += 1
    if i % 2 == 0:
        continue  # 繰り返しの最初に戻る
    elif i > 100:
        break  # 繰り返しを終了する
    print("i = {}".format(i))

print("100 より大きくなったので終了しました")
```

2 で割り切れる場合、つまり、偶数の場合は continue となるので、ループの最初の行に戻ります。奇数の場合は、print でその数字が出力されます。処理を繰り返し i が 100 より大きくなった場合に、break で繰り返し処理が終了します。繰り返し処理が終了すると、while の外にある print が実行されます。

ステップアップコラム 「range 関数によるリスト作成」

range 関数は、数字で指定したシーケンス型のオブジェクトを作るための関数です。for 文は与えられたシーケンスによってループ処理を行うため、次のように記述することができます。

for 変数 **in range**(始まりの数値,最後の数値,増加する量)**:**
for 変数 **in range**(始まりの数値,最後の数値)**:**
for 変数 **in range**(最後の数値)**:**

range 関数では、始まりの数値、最後の数値、増加する量の 3 つの値を整数で指定します。2 つしか記述しなかった場合は増加する量が 1 に、1 つしか記述しなかった場合は始まりの数値が 0、増加する量が 1 に自動的に設定されます。なお、リストの要素の参照と同様に最後の数値に指定した数値は範囲に含まれないことに注意しましょう。

また range 関数を使いリストを作成したい場合、list()で range 関数をリスト化する操作が必要です。

list(range(始まりの数値,最後の数値,増加する量)**)**

```
In [1]: a = range(5)
        a
Out[1]: range(0, 5)

In [2]: type(a)
Out[2]: range

In [3]: b = list(a)
        b
Out[3]: [0, 1, 2, 3, 4]

In [4]: type(b)
Out[4]: list

In [5]: c = list(range(5))
        c
Out[5]: [0, 1, 2, 3, 4]

In [6]: d = list(range(3,7))
        d
Out[6]: [3, 4, 5, 6]

In [7]: e = list(range(2,8,2))
        e
Out[7]: [2, 4, 6]
```

始まりの数値と増加する量が省略され、0 から開始して 1 ずつ増加する

増加する量が省略され、1 ずつ増加する

ステップアップコラム 「リスト内包表記」

Python は、リスト内包表記というリストを作成するための独自の方法を提供しています。

[式 for リストの要素 in シーケンス型のオブジェクト]

最終的にリスト内包表記は、すべての式の結果を要素として持つリストを返します。次にリスト内包表記の具体例を示します。

```
In [1]: square = [x**2 for x in range(5)]
        square

Out[1]: [0, 1, 4, 9, 16]
```

演算子の復習になりますが、**x**n** は、x を n 乗する式です。つまり、リストの要素を 2 乗して、新しいリストとして作成しています。上記内包表記は、次に示す for 文と同じ結果となります。

fordoubles.py

```
doubles = []
for v in range(5):
    doubles.append(v**2)
print(doubles)
```

for 文も同様にリストを作成しているのですが、内包表記に比べて記述量が多くなっているのがわかると思います。内包表記は、リストを生成していることが一目でわかります。また、リストを作成して append で要素を追加していくよりも多少高速です。

一方で、複雑な処理を無理に内包表記で行おうとすると、かえって読みづらいプログラムとなってしまいます。処理の複雑さに応じて、for 文を使うか内包表記を使うかを選択しましょう。

[実習 6] P24 の for.py を作成し、実行してみましょう

2.6. タプル

タプルは、リストと同じく複数の要素から構成される型で、要素が順に並んだシーケンス型オブジェクトです。ほとんどの点でリストと似た振る舞いをします。しかしタプルでは、一度作成したオブジェクトの要素を変更できません。これがリストとの大きな違いです。変更されると困る値をまとめて保持したい場合は、リストではなくタプルを使う方が良い選択です。

タプルを作成するときは次のように( )で囲んで、要素をカンマ「,」で区切って並べます。

(オブジェクト 1, オブジェクト 2, ...)

タプルの作成の具体例は次のとおりです。要素が 1 つだけの場合、最後にカンマを付けてから括弧を閉じます（カンマがないとただの数値として扱われます）。

```
In [1]: t = (1, 2, 3)
        t
Out[1]: (1, 2, 3)

In [2]: t = (1,)
        t
Out[2]: (1,)

In [3]: t = (1)
        t
Out[3]: 1
```

リストと同様にインデックスを指定して要素を参照できます。スライスによる参照も可能です。

```
In [1]: t = ("a", "b", "c", "d", "e")
        t[1]
Out[1]: 'b'

In [2]: t[2:4]
Out[2]: ('c', 'd')
```

タプルの要素が変更できないことを確認するために、インデックスで指定した要素に値を入れてみましょう。すると、「タプルは値の割り当てをサポートしていない」というエラーメッセージが表示されるとともに代入は失敗します。

```
In [1]: t = ("a", "b", "c", "d", "e")
        t[1] = "x"
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-1-952700967cf5> in <module>
      1 t = ("a", "b", "c", "d", "e")
----> 2 t[1] = "x"

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

タプルそのものの変更はできませんが、タプル同士を連結させて新しいタプルを生成することは可能です。タプルが制限しているのはあくまで自分自身の値を変えられないことであり、すべての操作が不可能だということではありません。

```
In [1]: t1 = (1, 2, 3)
        t2 = t1 + ("a", "b")
        t2
Out[1]: (1, 2, 3, 'a', 'b')
```

ステップアップコラム　「タプルの不変性」

値が変更できないというタプルの仕様を踏まえた上で、次の例を見てみましょう。

```
In [1]: t = (1, 2, [3, 4, 5])
        t[2] = [300, 400, 500]

        ---------------------------------------------------------------------------
        TypeError                                 Traceback (most recent call last)
        <ipython-input-1-afe20bca7b43> in <module>
              1 t = (1, 2, [3, 4, 5])
        ----> 2 t[2] = [300, 400, 500]

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

```
In [2]: t[2][1] = 400
        t

Out[2]: (1, 2, [3, 400, 5])
```

タプルの 3 番目の要素を変更しようとすると、エラーメッセージが表示されて割り当てに失敗します。しかし、3 番目の配列の中の要素を変更することには成功しています。タプルの要素への直接の割り当てはできませんが、タプルの直接の要素ではないリストの要素への割り当てはできてしまいます。つまり、タプルが不変であることを保証するのは、タプルの直接の要素に対してのみであるということです。

[実習 7]　タプルの普遍性を確認するために、t=(“a”,”b”,”c”,”d”,”e”)を用意し t[1]=”x”を実行しエラーが返ってくることを見てみましょう

2.7. 集合

集合は、数学の集合と同じような操作が可能な型です。リストと同じく複数の要素から構成されていますが、要素間に順序はありません。また、重複した要素は存在しません。

集合を作成するときは、次のように{ }で囲んで、要素をカンマ「,」で区切って並べます。

```
{オブジェクト 1, オブジェクト 2, ...}
```

```
In [1]: s = {1, 2, 3, 4, 5}
        s

Out[1]: {1, 2, 3, 4, 5}
```

作成した集合への要素の追加と削除は、add メソッドや remove メソッドで行えます。同じ値を繰り返し追加してみると、重複した要素が追加されないことを確認できます。

```
In [1]: s = {1, 2, 3, 4, 5}
        s.add(6)
        s
Out[1]: {1, 2, 3, 4, 5, 6}
```

```
In [2]: s.remove(1)
        s
Out[2]: {2, 3, 4, 5, 6}
```

集合同士は、論理演算できます。論理演算することで、集合同士で要素を追加したり、削除したりできます。a と b の 2 つの集合に対して、和集合、積集合、差集合をそれぞれ求めてみましょう。

calcset.py

```
a = {1, 2, 4, 6, 8}
b = {1, 3, 5, 7, 9}

print(a | b) # a と b の和集合
print(a & b) # a と b の積集合
print(a - b) # a と b の差集合
```

```
In [1]: # calcset.py

        a = {1, 2, 4, 6, 8}
        b = {1, 3, 5, 7, 9}

        print(a | b) # a と b の和集合
        print(a & b) # a と b の積集合
        print(a - b) # a と b の差集合

        {1, 2, 3, 4, 5, 6, 7, 8, 9}
        {1}
        {8, 2, 4, 6}
```

ある要素がある集合に属しているかの判定は、簡単に行えます。判定に用いる要素は集合でも構いません。その場合、部分集合であるかどうかを判定できます。

subset.py

```
a = {1, 2, 3, 4, 5}
b = {2, 3}

print({4} <= a) # 4 が a に含まれているか
print(a <= b) # a が b の部分集合であるか
print(a >= b) # b が a の部分集合であるか
```

```
In [1]: # subset.py

        a = {1, 2, 3, 4, 5}
        b = {2, 3}

        print({4} <= a) # 4 が a に含まれているか
        print(a <= b) # a が b の部分集合であるか
        print(a >= b) # b が a の部分集合であるか

        True
        False
        True
```

リストのようにインデックスを指定して要素にアクセスしようとするとエラーとなります。要素間に順序がないため、インデックスによる要素の取得はできません。

```
In [1]: s = {1, 2, 3}
        s[1]

        ---------------------------------------------------------------------------
        TypeError                                 Traceback (most recent call last)
        <ipython-input-1-d711d025981d> in <module>
              1 s = {1, 2, 3}
        ----> 2 s[1]

        TypeError: 'set' object does not support indexing
```

ステップアップコラム 「リスト、タプル、セットの相互変換」

これまで紹介した 3 つのデータ構造はそれぞれ容易に相互変換できます。list()、set()、tuple()を使うことで相互変換できます。あるリストに対して集合演算を使うために集合にしたり、変更不可能にするために一時的にタプルにしたりといった用途が考えられます。

```
In [1]: t = (1, 2, 3, 4, 5)
        l = [10, 20, 30 ,40, 50]
        s = {100, 200, 300, 400, 500}
        list(t)

Out[1]: [1, 2, 3, 4, 5]

In [2]: set(l)

Out[2]: {10, 20, 30, 40, 50}

In [3]: tuple(s)

Out[3]: (100, 200, 300, 400, 500)
```

2.8. 辞書

辞書は、リストと同じく複数の要素から構成され、集合と同じく要素間に順序のない型です。辞書型では、キーと値のペアで情報を保持します。

辞書を作成するときは次のように{ }で囲んで、キーと値をコロン「:」でつなげたものを要素としてカンマ「,」で区切って並べます。

```
{キー1: 値 1, キー2: 値 2, ...}
```

具体例として、キーが「apple」で値が「100」の要素と、キーが「orange」で値が「75」の要素の 2 つの要素からなる辞書を作成してみましょう。

```
In [1]: d = {"apple": 100, "orange": 75}
        d

Out[1]: {'apple': 100, 'orange': 75}
```

リストや別の辞書などをキーに指定することはできません。次のようなエラーメッセージが表示される型はキーとして使えないということを覚えておきましょう。

```
In [1]: d = {["Hello", "World"]: 1}
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-1-6548ca8b2175> in <module>
----> 1 d = {["Hello", "World"]: 1}

TypeError: unhashable type: 'list'
```

辞書ではインデックスの代わりにキーと呼ばれる識別子を使って要素を指定します。このようなオブジェクトをマッピング型オブジェクトといいます。

要素を取り出したいときは、次のようにキーを指定します。

```
In [1]: d = {"apple": 100, "orange": 75}
        d["apple"]

Out[1]: 100
```

このように、特定のキーと値がマッピングされるようなデータ構造であれば、辞書型で保存すると良いでしょう。ほとんどの場合において、リストとしてデータを定義するよりも高速に値にアクセスすることができます。

あるキーの値を変更したい場合は、次のようにキーと値を指定します。

```
In [1]: d = {"apple": 100, "orange": 75}
        d["apple"] = 80
        d

Out[1]: {'apple': 80, 'orange': 75}
```

存在しないキーを指定した場合は、エラーが発生します。ただし、存在しないキーに対して値を設定した場合は、そのキーと値の組み合わせが辞書に追加されます。

```
In [1]: d = {"apple": 100, "orange": 75}
        d["banana"]
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-1-b165d8b2e64e> in <module>
      1 d = {"apple": 100, "orange": 75}
----> 2 d["banana"]

KeyError: 'banana'
```

```
In [2]: d["banana"] = 100
        d
Out[2]: {'apple': 100, 'orange': 75, 'banana': 100}
```

作成した辞書から要素を削除するには、pop メソッドや del 関数を使います。

```
In [1]: d = {"apple": 100, "banana": 100, "orange": 75}
        d.pop("banana")
        d
Out[1]: {'apple': 100, 'orange': 75}
```

```
In [2]: del d["apple"]
        d
Out[2]: {'orange': 75}
```

辞書に対しても in 演算子は使用可能です。辞書に in 演算子を使用した場合、キーが存在するかどうかをチェックできます。in 演算子だけでは、値の存在チェックはできません。

```
In [1]: d = {"apple": 100, "banana": 100, "orange": 75}
        "apple" in d
Out[1]: True
```

```
In [2]: 100 in d
Out[2]: False
```

辞書は、keys 関数でキーの一覧を、values 関数で値の一覧をリストで取得できます。values 関数で取得したリストに対して in 演算子を使うことで、値の存在チェックが可能となります。

```
In [1]: d = {"apple": 100, "banana": 100, "orange": 75}
        d.keys()
Out[1]: dict_keys(['apple', 'banana', 'orange'])

In [2]: d.values()
Out[2]: dict_values([100, 100, 75])

In [3]: 100 in d.values()
Out[3]: True
```

2.9. 関数

関数とは、一連の処理をまとめたもののことを指します。関数を呼び出すことで、まとめられた一連の処理を実行できます。これまで使ってきた print や len も関数であり、これらは組み込み関数※と呼ばれています。汎用的な処理は組み込み関数として定義されていることが多いので、自分で何か処理を作る際は、まず組み込み関数として定義されていないか調べると良いでしょう。

Python では、def を使って関数を作成できます。下のように記述します。if 文や for 文と同様にインデントでブロックを形成します。処理の部分には、これまで学んだ処理を自由に書けます。引数は、オプションなので書かなくても結構です。

```
def 関数名(引数):
    処理
```

与えられた引数を出力するだけの簡単な関数を作成してみましょう。

def.py

```
def my_print(a):
    print(str(a) + "と入力されました")

my_print(128)
```

```
In [1]: # def.py

        def my_print(a):
            print(str(a) + "と入力されました")

        my_print(128)
        128と入力されました
```

※ 参考文献[5]を参照

関数は、処理の結果を呼び出し元に返すことができます。処理の結果は、return で返すことができます。return が実行されると、関数はその時点で処理を終了します。つまり、それ以降の処理は実行されません。試しに 2 つの値を足し合わせる関数を作ってみましょう。

returndef.py

```
def add(a, b=1):
    result = a + b
    print("a + b = {}".format(result))
    return result
    print("この行は実行されません")

result = add(3, 2)
print(result)
```

このスクリプトを実行すると、次のようになります。return のあとの print 文は実行されずに、result には、return で返された値が設定されていることがわかります。

```
In [1]: # returndef.py

        def add(a, b=1):
            result = a + b
            print("a + b = {}".format(result))
            return result
            print("この行は実行されません")

        result = add(3, 2)
        print(result)

        a + b = 5
        5
```

```
In [2]: result = add(3)
```

b=1 が自動的に設定される

```
        a + b = 4
```

この例における「b=1」のような引数をデフォルト引数と呼びます。関数呼び出し時にデフォルト引数が省略された場合は、デフォルト値が自動的に設定されます。

デフォルト引数はいくつでも設定できますが、後ろに通常の引数を記述するとエラーが発生します。

```
In [1]: def add(a, b=1, c):
            result = a + b + c
            print("a + b + c = {}".format(result))
            return result

          File "<ipython-input-1-4a0eb4c46a44>", line 1
            def add(a, b=1, c):
                       ^
        SyntaxError: non-default argument follows default argument
```

[実習 8]　returndef.py を作成し、実行してみましょう

2.10. クラス

Python は、オブジェクト指向をサポートしている言語なので、Java と同様にクラスを定義することができます。クラスは変数と関数をひとまとめにしたものです。Python が提供しているクラスを使用することや、開発者が自身でクラスを定義し、その定義を使用することもできます。本テキストでは、クラスの簡単な使い方のみを説明します。

2.10.1. クラスとインスタンス

Python が提供している Decimal クラスと Fraction クラスを例に、クラスの使い方を見てみましょう。

```
In [1]: from decimal import Decimal
        from fractions import Fraction

        Decimal
```

```
Out[1]: decimal.Decimal
```

```
In [2]: Fraction
```

```
Out[2]: fractions.Fraction
```

クラスは明確な役割を持ちます。Decimal は精確に小数を取り扱う、Fraction は分数を取り扱うという役割をそれぞれ持っています。

関数のように、括弧つきで呼び出すことで、そのクラスのインスタンスを生成できます。クラスは、ひな形であり、インスタンスは、そのひな形から作り出された実体のようなものです。多くの場合、クラスからインスタンスを生成し、そのインスタンスを使って処理をします。Decimal クラスと Fraction クラスは四則演算をサポートしているため、インスタンス同士の演算が可能です。

```
In [1]: from decimal import Decimal
        from fractions import Fraction

        d1 = Decimal("0.3333")
        d2 = Decimal("0.4444")

        d1 + d2
```

```
Out[1]: Decimal('0.7777')
```

```
In [2]: f1 = Fraction("3/7")
        f2 = Fraction("2/7")

        f1 + f2
```

```
Out[2]: Fraction(5, 7)
```

2.10.2. ユーザ定義クラスの作成

クラスは下のように定義します。また、クラスが持つ関数はメソッドと呼ばれます（以降、クラス内の関数を「メソッド」と呼びます)。

```
class クラス名():
    変数
    関数
```

クラス名には、一般的にキャメルケースが使用されます。キャメルケースは、単語の先頭 1 文字を大文字とします。スネークケースのように文字の区切りに記号を使いません。

Decimal や Fraction と同様に、インスタンス作成時に何らかの値を受け取り、その値を変数として持つクラスを作成します。Python ではインスタンス作成時に__init__メソッドが呼び出されるようになっているので、初期化処理は__init__メソッドに記述します。インスタンス作成時に引数が与えられていた場合、__init__メソッドにその引数が引き渡されます。

simpleclass.py

```
class Simple:
    def __init__(self, i="デフォルト値"):
        print("__init__メソッドが呼び出されました")
        self.instance_attr = i

    def instance_print(self):
        print("インスタンス変数:" + str(self.instance_attr))

# クラスを使った処理
simple_instance = Simple("test") # インスタンスの作成
simple_instance.instance_print()
simple_instance.instance_attr = "上書き" # インスタンス変数に値を割り当て
simple_instance.instance_print()
print(simple_instance.instance_attr) # インスタンス変数を直接呼び出す
```

上の Python スクリプトを実行すると、インスタンス作成時に与えた「test」という文字列が__init__メソッドに渡されていることがわかります。また、インスタンス作成時に__init__メソッドが自動的に呼び出されていることもわかります。

In [1]:
```
# simpleclass.py

class Simple:
    def __init__(self, i="デフォルト値"):
        print("__init__メソッドが呼び出されました")
        self.instance_attr = i

    def instance_print(self):
        print("インスタンス変数：" + str(self.instance_attr))

# クラスを使った処理
simple_instance = Simple("test") # インスタンスの作成
simple_instance.instance_print()
simple_instance.instance_attr = "上書き" # インスタンス変数に値を割り当て
simple_instance.instance_print()
print(simple_instance.instance_attr) # インスタンス変数を直接呼び出す
```

```
__init__メソッドが呼び出されました
インスタンス変数：test
インスタンス変数：上書き
上書き
```

変数と関数をクラスにまとめるだけであれば、クラスの使用にあたって何も身構えることはありません。__init__などの一部の特殊メソッドについて理解し、後は、変数や関数を今までどおり記述するだけです。

2.10.3. 第一引数の self

Simple クラスの instance_print の第 1 引数に self が指定されていますが、そのメソッドを呼び出す際に引数を与えていません。引数である self にはデフォルト値が設定されていないため、省略可能な引数でもありません。しかしながら、instance_print メソッドはエラーが発生していないのはなぜでしょうか。

エラーが発生しないのは、インスタンスがメソッドを呼び出す際に、self に自分自身を与えているためです。つまり、self にはメソッドを呼び出したインスタンス自身が自動的に渡されます。

ここで重要なのは、メソッドの第 1 引数にインスタンスが渡されるということです。self という名前であることは、それほど重要なことではありません。me でも myself でも構いません。しかし、特別な理由がない限り self を使うことを推奨します。self は、インスタンス自身を表す際によく用いられ、この名前をつけることが慣習となっているためです。

2.10.4. クラス変数とインスタンス変数

simpleclass.py の Simple クラスの変数はインスタンス変数と呼ばれており、インスタンスごとに独立して値を持ちます。一方で、クラスで共通の値を持たせたい場合があるかもしれません。これはクラス変数として定義することで実現できます。

classattr.py

```
class Simple:
    class_attr = "クラス変数"
    def __init__(self, i):
        self.instance_attr = i

s1 = Simple("s1")
s2 = Simple("s2")
print(Simple.class_attr)
print(s1.instance_attr)
print(s2.instance_attr)
```

クラス変数は、self 等をつけずに class の直下に定義します。定義されたクラス変数は、クラス名、クラス変数でアクセスできます。変数にアクセスするために、わざわざインスタンスを作成する必要はありません。

インスタンスごとに個別の値を割り当てる場合は、インスタンス変数とし、そうでない場合はクラス変数として定義しておくと良いでしょう。共通の値をインスタンス変数として定義しても動作には問題ありません。クラス変数として定義しておくと、メモリ効率が良くなるなど、いくつかの恩恵を得られます。クラス変数とインスタンス変数を意識して使い分けましょう。

3. Python のライブラリ活用

Python には、便利なライブラリが豊富に存在します※1。本章では、NumPy と matplotlib という 2 つのライブラリをピックアップして紹介します。これらは科学技術計算や統計解析といった分野を支える主要なライブラリです。簡単なサンプルを通して、いかに便利なライブラリであるかを実感してもらいたいと思います。ただし、どちらのライブラリも標準のライブラリではないので、使用する前にコマンド上で「pip install numpy」、「pip install matplotlib」をしておく必要があります。

3.1. 数値計算ライブラリ NumPy

NumPy は、演算処理を高速、かつ、簡単に行うためのライブラリです。科学技術計算や Web データの分析などの大規模なデータを取り扱う場合、多次元かつ大量のベクトル(= 配列)の演算が必要になります。Python でそのような演算を行うと処理に時間がかかってしまいますが、NumPy を用いれば高速に処理できます。NumPy は、科学技術計算の基礎ライブラリとして様々な演算機能を提供しているため、この次に紹介する matplotlib など多くのライブラリに活用されています。

大量のデータを効率的に処理するために、NumPy は ndarray というリスト風のオブジェクトを提供しています。ndarray は非常に多くの機能を有しているため、一部の機能だけを取り上げます。さらに詳しく知りたい場合は、公式ドキュメント等を参照してください※2。なお、以降のテキストで「配列」「array」と表現した場合、それらは ndarray のことを指します。

3.1.1. NumPy モジュールのインポート

NumPy は Python 標準の機能ではありません。したがって、NumPy の機能を使えるようにインポートしなければなりません。標準以外の機能を利用するには、 import 文を使います。NumPy の機能がまとめられたモジュールである NumPy をインポートして、ndarray のオブジェクトを作ってみましょう。

```
In [1]: import numpy

        a = numpy.array([0, 1, 2, 3, 4])
        a
Out[1]: array([0, 1, 2, 3, 4])

In [2]: type(a)
Out[2]: numpy.ndarray
```

Python の標準機能でない場合、事前のインポートを忘れないようにしましょう。事前にインポートせずに標準以外のモジュールを呼び出した場合、次のようなエラーが発生してしまいます。

※1 参考文献[6]を参照
※2 参考文献[7]を参照

```
In [1]: numpy.array([0, 1, 2, 3, 4])
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-1-6c5334d127fb> in <module>
----> 1 numpy.array([0, 1, 2, 3, 4])

NameError: name 'numpy' is not defined
```

import の際には別名を指定できます。別名の指定は、複数のモジュールで名前が重複した場合や、長いモジュール名を短くしたい場合に有用です。

NumPy は、import 時に np という別名を指定することが慣習となっています。変数の命名規則と同様に、これも慣習であり制約ではありません。従わなくても動作には何の問題もありません。

```
In [1]: import numpy as np

        np.array([0, 1, 2, 3, 4])
Out[1]: array([0, 1, 2, 3, 4])
```

対話モードの場合は終了時に情報がクリアされるため、対話モードを起動するたびに import でライブラリを読み込む必要があります。

3.1.2. 配列の作成

リスト型のオブジェクトを渡して配列を作成する方法が最もシンプルな方法です。リストが入れ子構造になっている場合は多次元配列として作成されます。

```
In [1]: import numpy as np

        np.array([0, 1, 2, 3, 4, 5])
Out[1]: array([0, 1, 2, 3, 4, 5])
```

```
In [2]: np.array([[0, 1], [2, 3], [4, 5]])
Out[2]: array([[0, 1],
               [2, 3],
               [4, 5]])
```

ndarray は、2 次元以上の配列の値をわかりやすく表示してくれます。リストで 2 次元以上の配列を作成したとしても、横並びに表示されるだけです。これも ndarray の利点の 1 つです。

同じ値を持つ m×n の配列も簡単に作成できます。ここでは、いくつかの関数を使用して様々な 2×3 の配列を生成しています。

```
In [1]: import numpy as np

        np.zeros([2, 3])

Out[1]: array([[0., 0., 0.],
               [0., 0., 0.]])

In [2]: np.full([2, 3], np.pi)

Out[2]: array([[3.14159265, 3.14159265, 3.14159265],
               [3.14159265, 3.14159265, 3.14159265]])

In [3]: np.random.rand(2, 3)

Out[3]: array([[0.89286015, 0.33197981, 0.82122912],
               [0.04169663, 0.10765668, 0.59505206]])
```

0.0~1.0 の一様乱数を 2×3 の配列で生成

シーケンス型のオブジェクトを生成できる range という関数があったように、NumPy にも arange 関数があります。range と同じように引数を指定できます。

```
In [1]: import numpy as np

        np.arange(10)

Out[1]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [2]: np.arange(4, 8)

Out[2]: array([4, 5, 6, 7])

In [3]: np.arange(1, 10, 3)

Out[3]: array([1, 4, 7])
```

冒頭で「リスト風のオブジェクト」と書いたとおり、ndarray はリストと同じような処理が可能です。一部の処理ついては、次の章で取り扱います。このテキストで取り上げる以外にも色々試してみましょう。

3.1.3. 要素の参照

NumPy の配列は、リストと同様にインデックスを指定してアクセスできます。NumPy は多次元配列の要素を参照しやすいように拡張されています。次にインデックスやスライスで要素にアクセスしている例を示します。

```
In [1]: import numpy as np

        a = np.random.rand(5, 5)
        a

Out[1]: array([[0.66959078, 0.90757465, 0.66792738, 0.07839758, 0.47551195],
               [0.93518839, 0.15130299, 0.02602744, 0.19179741, 0.0913012 ],
               [0.39544325, 0.7173842 , 0.42573716, 0.47124192, 0.17314569],
               [0.59481042, 0.01790533, 0.83170162, 0.22325554, 0.24177455],
               [0.80212384, 0.05251643, 0.43592669, 0.19057163, 0.22375085]])

In [2]: a[0, 2]

Out[2]: 0.6679273806561052

In [3]: a[0]        0 番目の行の要素すべてを参照

Out[3]: array([0.66959078, 0.90757465, 0.66792738, 0.07839758, 0.47551195])

In [4]: a[:, 2]     2 番目の列の要素すべてを（一次元配列で）参照

Out[4]: array([0.66792738, 0.02602744, 0.42573716, 0.83170162, 0.43592669])

In [5]: a[1:3, 1:4]     スライスを使用

Out[5]: array([[0.15130299, 0.02602744, 0.19179741],
               [0.7173842 , 0.42573716, 0.47124192]])
```

NumPy には、もう 1 つ、強力な要素の参照方法があります。インデックスを埋め込む部分に数値ではなく、条件式を埋め込むことにより、その条件に合う要素だけを列挙できます。多次元配列のうち、値が 0.5 以上の要素だけ取得してみましょう。

```
In [1]: import numpy as np

        a = np.random.rand(4, 4)
        a

Out[1]: array([[0.14897833, 0.6952221 , 0.97838966, 0.18802999],
               [0.63670753, 0.65882565, 0.01494436, 0.00911137],
               [0.73665984, 0.27975383, 0.37696378, 0.56158949],
               [0.22059939, 0.44989031, 0.0816356 , 0.29786276]])

In [2]: a[a > 0.5]

Out[2]: array([0.6952221 , 0.97838966, 0.63670753, 0.65882565, 0.73665984,
               0.56158949])
```

このように、NumPy は通常のリストよりも簡潔、かつ、高速に特定の要素を参照できる様々な機能を提供しています。

3.1.4. 要素の演算

配列に四則演算を行った場合、要素すべてに対して演算が行われます。

```
In [1]: import numpy as np

        a = np.random.rand(2, 2)
        a
```

```
Out[1]: array([[0.09071265, 0.12903915],
               [0.04265838, 0.7094841 ]])
```

```
In [2]: a + 3
```

```
Out[2]: array([[3.09071265, 3.12903915],
               [3.04265838, 3.7094841 ]])
```

```
In [3]: a - 1
```

```
Out[3]: array([[-0.90928735, -0.87096085],
               [-0.95734162, -0.2905159 ]])
```

```
In [4]: a * 5
```

```
Out[4]: array([[0.45356323, 0.64519574],
               [0.21329189, 3.54742048]])
```

```
In [5]: a / 2
```

```
Out[5]: array([[0.04535632, 0.06451957],
               [0.02132919, 0.35474205]])
```

配列同士を演算した場合、対応する要素同士が演算されます。配列同士を演算する場合は、基本的に要素数とその並びが同じでなければなりません。異なる要素数と並びの配列を演算すると、エラーが発生してしまいます。

```
In [1]: import numpy as np

        a1 = np.random.rand(2, 2)
        a2 = np.random.rand(2, 2)

        a1 + a2
```

```
Out[1]: array([[0.49306747, 1.31419748],
               [1.41557688, 1.03894633]])
```

```
In [2]: a3 = np.random.rand(3, 3)

        a1 + a3
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-2-4cd1d74ee05a> in <module>
      1 a3 = np.random.rand(3, 3)
      2
----> 3 a1 + a3

ValueError: operands could not be broadcast together with shapes (2,2) (3,3)
```

ただし、要素数と並びが異なる配列同士でも演算できる場合があります。

```
In [1]: import numpy as np

        a1 = np.random.rand(3, 5)
        a1

Out[1]: array([[0.70331198, 0.30787033, 0.80167852, 0.15940624, 0.18925079],
               [0.47286726, 0.11512468, 0.28373068, 0.61602944, 0.65212748],
               [0.42999794, 0.47110028, 0.24043035, 0.25370541, 0.31593092]])

In [2]: a2 = np.random.rand(3, 1)
        a2

Out[2]: array([[0.19726344],
               [0.42079531],
               [0.81579225]])

In [3]: a1 + a2

Out[3]: array([[0.90057542, 0.50513376, 0.99894196, 0.35666967, 0.38651423],
               [0.89366256, 0.53591998, 0.70452599, 1.03682475, 1.07292278],
               [1.24579019, 1.28689253, 1.0562226 , 1.06949766, 1.13172318]])
```

a1 のそれぞれの列に a2 が足される

これは、NumPy のブロードキャストという仕組みによるものです。ブロードキャスト可能な条件を満たす場合に、要素数と並びが異なる配列同士での演算が可能となります。ブロードキャスト可能条件の詳細については、ここでは割愛します※。

そのほかにも、転置行列を作成する T や要素の合計を求める sum 関数、行列の積を求める dot 関数などがあります。

```
In [1]: import numpy as np

        a1 = np.arange(9).reshape(3, 3)
        a1

Out[1]: array([[0, 1, 2],
               [3, 4, 5],
               [6, 7, 8]])

In [2]: a1.T

Out[2]: array([[0, 3, 6],
               [1, 4, 7],
               [2, 5, 8]])

In [3]: a1.sum()

Out[3]: 36

In [4]: a1.dot(np.arange(3))

Out[4]: array([ 5, 14, 23])
```

※ 参考文献[8]を参照

NumPy には多次元の配列を簡単に操作するための機能が多く提供されています。プログラミングする前に一度、公式ドキュメントを確認して同等の機能を持つ関数がないか確認し、「車輪の再発明」をすることのないよう心掛けましょう。

NumPy をもっと使ってみたいという場合は、「100 numpy exercises」という練習問題集があるのでそちらもぜひ挑戦してみてください※1。

3.2. グラフ描画ライブラリ matplotlib

matplotlib は、Python でグラフを描画するためのライブラリです。 matplotlib と NumPy を併用すれば、NumPy で高速に演算し、かつ、その演算結果を matplotlib でグラフとして描画するといったことができます。さらに詳しく知りたい方は、公式ドキュメントを参照してください※2。

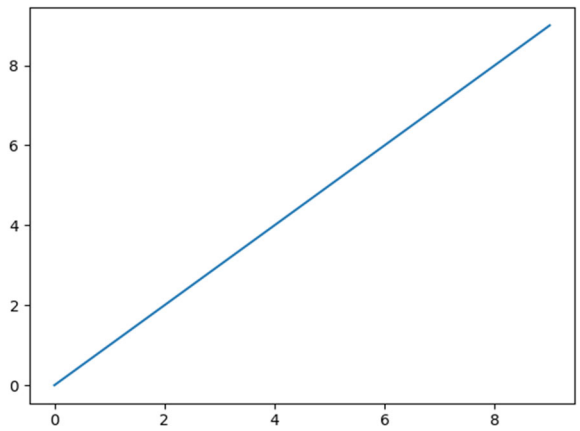
3.2.1. pyplot モジュールのインポート

matplotlib は、標準の機能ではないため、NumPy モジュールと同様にインポートしなければなりません。あるモジュールから特定のモジュールをインポートしたい場合は、from を使ったインポートが有用です。次に、matplotlib からグラフ描画に使う pyplot モジュールのみをインポートする例を示します。実行すると線形増加するグラフが表示されます。また、Jupyter Notebook 上でグラフを固定表示したい場合は、プログラムの冒頭に「%matplotlib inline」、拡大縮小したい場合は「%matplotlib notebook」、ノートの外部で表示したい場合（PyQt5 など）は、「%matplotlib qt」とそれぞれ記述する必要があります。

```
In [1]: %matplotlib qt

        from matplotlib import pyplot
        pyplot.plot(range(10))

Out[1]: [<matplotlib.lines.Line2D at 0x1c67fbc55c0>]
```



※1 参考文献[9]を参照
※2 参考文献[10]を参照

from を使わずにインポートした場合は、pyplot を使う際に毎回 matplotlib.pyplot と書かなければいけません。

```
In [1]: %matplotlib qt

        import matplotlib.pyplot
        matplotlib.pyplot.plot(range(10))

Out[1]: [<matplotlib.lines.Line2D at 0x2d8b12a66a0>]
```

短い記述でモジュールを呼び出したい場合は、from を使ったインポートをすると良いでしょう。モジュール名を省略するだけであれば、先述した as で別名をつける方法でも構いません。

3.2.2. グラフを描画してみる

NumPy と pyplot を用いて、x^2のグラフを作成してみましょう。pyplot モジュールの plot 関数でプロットし、show 関数でグラフを表示します。次の例では、慣例に従って pyplot を plt という別名でインポートしています。

quardraticfunc.py

```
import numpy as np
from matplotlib import pyplot as plt # 慣例に従い、plt に

x = np.linspace(-5, 5, 30) # -5 から 5 までの範囲を 30 分割した値の配列
y = np.power(x, 2)         # y = x の 2 乗

plt.plot(x, y) # x 軸と y 軸を描画する
plt.show()     # グラフを表示する
```

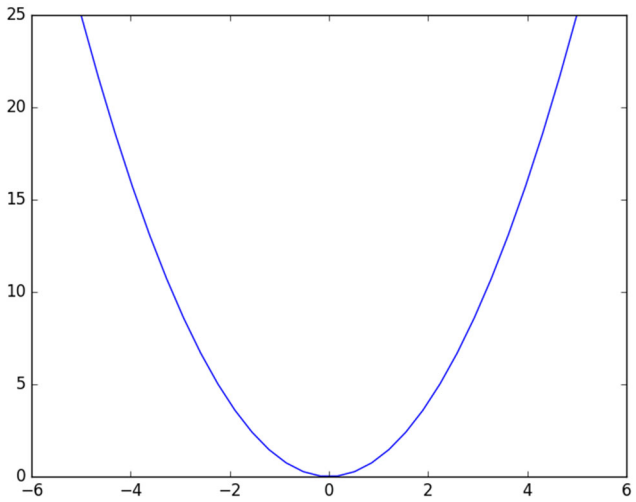


図 1. x^2のグラフ

凡例やタイトルなどグラフを作成する上で欠かせない要素は、matplotlib モジュールの関数を使えば簡単に追加できます。大学の環境では日本語が設定できないので、使用可能な文字は英数字のみとなります。また、show 関数を呼び出す前に plot 関数を複数回呼び出すことで、1 つのグラフに複数のプロットを表示させることができます。

trigonometric.py

```
import numpy as np
from matplotlib import pyplot as plt

x = np.linspace(-np.pi, np.pi)
plt.plot(x, np.sin(x), label="sin")
plt.plot(x, np.cos(x), label="cos")
plt.xlim(x.min(), x.max()) # x 軸の描画領域を値の最小値と最大値にする

plt.legend() # 凡例
plt.title(r"$¥sin(x)$ and $¥cos(x)$") # グラフのタイトルを数式スタイルで
plt.xlabel("X-Axis") # y 軸の名前
plt.ylabel("Y-Axis") # x 軸の名前
plt.show()
```

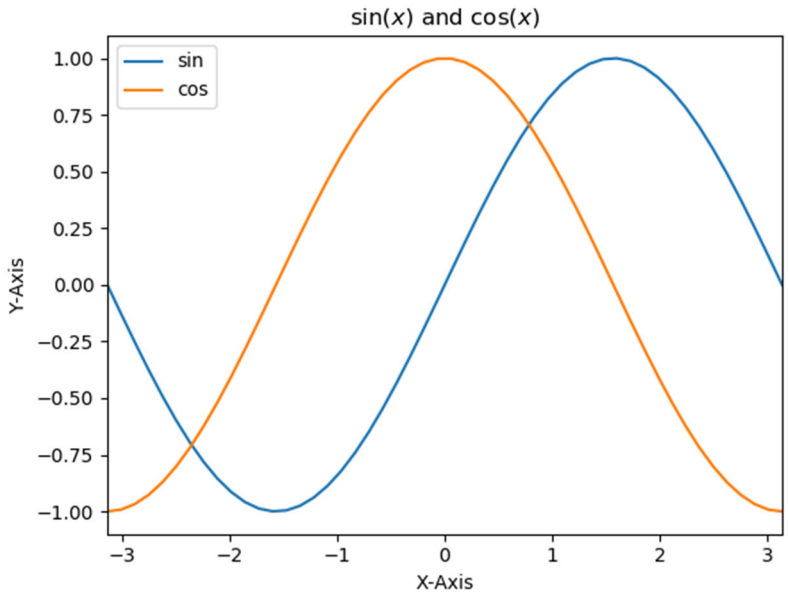


図 2. sin, cos のグラフ

matplotlib を使いこなせば、グラフの見た目をさらにカスタマイズしたり、棒グラフや散布図といった異なる種類のグラフを作成したりすることができます。グラフだけでなく、簡単なアニメーションを作ることもできます。詳しく知りたい方は、ぜひ公式ドキュメントを参照してください※。

※ 参考文献[10]を参照

3.3. CSV ファイル形式の統計データの読み込みと描画

ここでは、実際の統計データを読み込んでグラフで表示します。統計データは、kaggle※というサービスからダウンロードできます。kaggle は、専門家が統計や分析の最適なモデルを競い合うことを目的としたプラットフォームです。

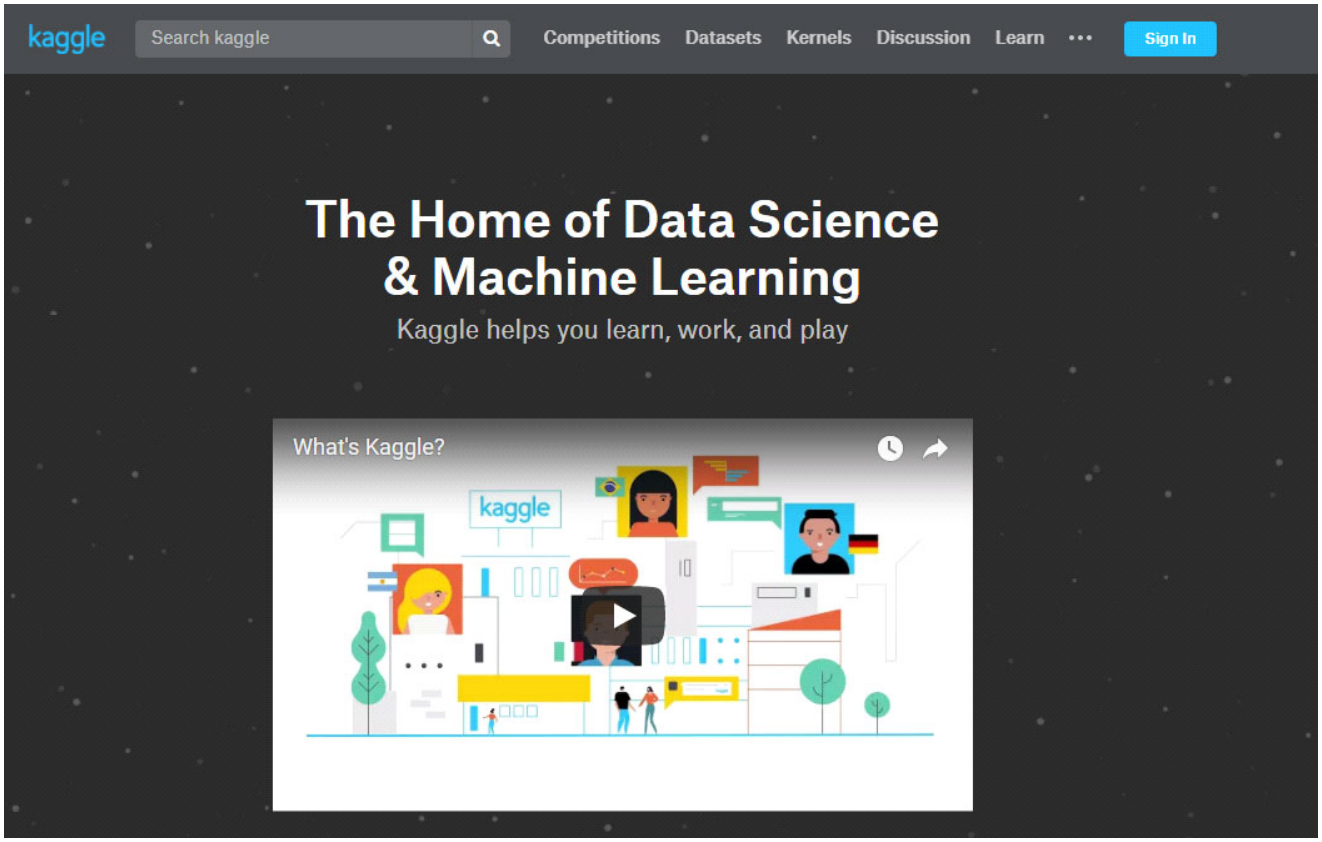


図 3. kaggle のトップページ

サービスの性質上、kaggle には様々な統計データが存在し、ログインさえ行えばそれらのデータをダウンロードできるようになっています。「データ分析や機械学習をしてみたいがサンプルデータがなくて困っている」という方にとっても有用なプラットフォームです。

ここでは例として、世界幸福度調査(2017 年度)の結果を使用します。横軸を国名、縦軸を幸福度とした棒グラフを作成します。日本のランクである 51 位までを取得し、グラフに描画します。元データは 156 個あるので、スライスして 51 個のデータとします。

データは、CSV ファイルとして保存されています。CSV ファイルは、NumPy モジュールの genfromtxt 関数、もしくは、loadtxt 関数を使うことで読み取れます。この例では、CSV ファイルの読み取り時に指定できるオプションの相性から genfromtxt 関数を利用しています。

※ 参考文献[11]を参照

csvload.py

```
  import numpy as np
  from matplotlib import pyplot as plt

  japan_rank = 51  # データの取得範囲
  x_tick = np.arange(japan_rank)  # x 軸のメモリを作成

  # csv ファイルの読み込み
  x, y = np.genfromtxt("世界幸福度調査 2017.csv", delimiter=",",
skip_header=1, usecols=(0, 2), dtype=str, unpack=True)

  plt.bar(x_tick, y[:japan_rank].astype(float), align="center",
label="hapiness")
  plt.xticks(x_tick, x[:japan_rank], rotation='vertical')
  plt.legend()
  plt.subplots_adjust(left=0.03, right=0.97, bottom=0.4, top=0.98)  # 描画領域を
微調整
  plt.xlim(-2, 52)
  plt.show()
```

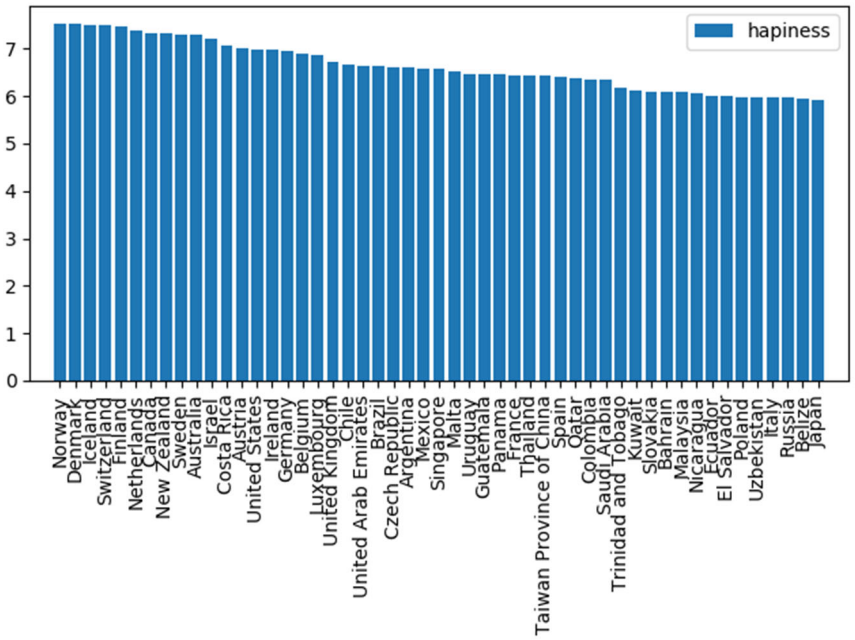


図 4. 世界幸福度調査のグラフ

4. さいごに

本テキストでは、Python の基礎構文について説明し、その後、NumPy や matplotlib といったライブラリを使ってデータを実際に可視化しました。Python という言語が簡潔で書きやすい言語であること、ライブラリを活用すれば簡単に処理できることを実感できたと思います。

「はじめに」で述べたように Python の活用分野は多岐に渡ります。そして、下の表のように活用分野に応じた様々なライブラリが提供されています。これは、ほんの一部であり、ほかにも様々な役立つライブラリが提供されています。

表 5. Python のライブラリ例

分野	ライブラリ	URL
数学・科学技術計算	NumPy	http://www.numpy.org/
	SciPy	https://www.scipy.org/
データの可視化	matplotlib	https://matplotlib.org/
	Altair	https://altair-viz.github.io/
機械学習・深層学習	TensorFlow	https://www.tensorflow.org/
	PyTorch	https://pytorch.org/
	Chainer	https://chainer.org/
	scikit-learn	http://scikit-learn.org/
Web アプリケーション	django	https://docs.djangoproject.com/
	Flask	http://flask.pocoo.org/
画像処理・解析	OpenCV	https://docs.opencv.org/trunk/index.html
	Pillow	https://pillow.readthedocs.io/
ゲーム開発	cocos2d	http://python.cocos2d.org/

Python を学習しておくことは、上記のライブラリを使えるようにする機会を得られると言っても過言ではありません。専門分野の知識は必要となりますが、Python を使えることで可能性を広げることができます。本テキストを読み終えた人が、1 人でも多く Python に興味を持ち、より深く学習・活用してくれることを期待しています。

5. 参考文献

◆ **図書**

[1] 柴田淳『みんなの Python：lightweight language Python definitive guide』、第 4 版、東京：SB クリエイティブ、2017 年

[2] 及川えり子『Python 超入門 ： モンティと学ぶはじめてのプログラミング』、東京：オーム社、2020 年

◆ **Web ページ**

[3] Python 公式ドキュメント
https://docs.python.org/ja/3/index.html (閲覧日: 2020 年 7 月 16 日)

[4] MITOPENCOURSEWARE Introduction to Computer Science and Programming in Python
https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-0001-introduction-to-computer-science-and-programming-in-python-fall-2016/ (閲覧日: 2020 年 7 月 16 日)

[5] 組み込み関数 – Python 3.8.5 ドキュメント
https://docs.python.jp/3/library/functions.html (閲覧日: 2020 年 7 月 28 日)

[6] GitHub - vinta/awesome-python: A curated list of awesome Python frameworks, libraries, software and resources
https://github.com/vinta/awesome-python (閲覧日: 2020 年 7 月 28 日)

[7] NumPy
http://www.numpy.org (閲覧日: 2020 年 7 月 28 日)

[8] Universal functions (ufunc) — NumPy v1.19 Manual
https://docs.scipy.org/doc/numpy/reference/ufuncs.html#broadcasting (閲覧日: 2020 年 7 月 28 日)

[9] GitHub - rougier/numpy-100: 100 numpy exercises (with solutions)
http://www.labri.fr/perso/nrougier/teaching/numpy.100 (閲覧日: 2020 年 7 月 28 日)

[10] Overview — Matplotlib 3.3.0 documentation
https://matplotlib.org/contents.html (閲覧日: 2020 年 7 月 28 日)

[11] Kaggle: Your Machine Learning and Data Science Community
https://www.kaggle.com/ (閲覧日: 2020 年 7 月 28 日)