
Python プログラミング入門

©2020–2022, 東京大学 数理・情報教育研究センター
(CC BY-NC-ND 4.0)

Apr 01, 2022

1	1-0. Colaboratory によるノートブックの使い方	2
1.1	Colaboratory の立ち上げ	2
1.2	ノートブックのアップロード	2
1.3	教材のオープン	5
1.4	ノートブックのダウンロード	5
1.5	ノートブックのアップロード (再び)	6
1.6	ノートブックの作成	7
1.7	ノートブックの操作	8
1.8	セル	8
1.9	セルの編集	8
1.10	練習	9
1.11	セルの挿入	9
1.12	セルの実行が止まらないとき	9
1.13	セルの操作	9
1.14	ノートブックの参照	9
2	1-1. 数値演算	10
2.1	簡単な算術計算	10
2.2	コメント	11
2.3	整数と実数	12
2.4	演算子の優先順位と括弧	13
2.5	算術演算子のまとめ	15
2.6	空白	15
2.7	エラー	15
2.8	数学関数 (モジュールのインポート)	16
2.9	練習	17
3	1-2. 変数と関数の基礎	18
3.1	変数	18
3.2	関数の定義と返値	20
3.3	ローカル変数	22
3.4	print	23
3.5	print と return	23
3.6	コメントと空行	24
3.7	関数の参照の書き方	24
3.8	▲グローバル変数	25
3.9	練習の解答	26
4	1-3. 論理・比較演算と条件分岐の基礎	28
4.1	if 文による条件分岐	28
4.2	様々な条件	29

4.3	真理値を返す関数	31
4.4	オブジェクト	32
4.5	None	32
4.6	▲条件として使われる他の値	33
4.7	▲再帰	33
4.8	練習の解答	34
5	1-4. テストとデバッグ	35
5.1	仕様・テスト・デバッグ	35
5.2	assert 文	36
5.3	エラーの分類	37
5.4	デバッグの具体例	39
5.5	コーディングスタイル	41
6	2-1. 文字列 (string)	43
6.1	文字列とインデックス	44
6.2	文字列とスライス	45
6.3	空文字列	46
6.4	文字列の検索	47
6.5	▲エスケープシーケンス	47
6.6	バックスラッシュの表示と入力	48
6.7	文字列の連結	49
6.8	文字列とメソッド	49
6.9	文字列の比較演算	53
6.10	練習	53
6.11	初心者によくある誤解 — 変数と文字列の混乱	54
6.12	練習	54
6.13	練習の解答	54
7	2-2. リスト (list)	56
7.1	リストとインデックス	57
7.2	練習	58
7.3	多重リスト	58
7.4	リストに対する関数・演算子・メソッド	59
7.5	破壊的（インプレース）な操作と非破壊的な生成	62
7.6	リストを操作するメソッドなど	63
7.7	リストと文字列の相互変換	66
7.8	タプル (tuple)	67
7.9	練習	69
7.10	多重代入	69
7.11	リストやタプルの比較演算	70
7.12	for 文による繰り返しとリスト・タプル	71
7.13	練習	73
7.14	for 文による繰り返しと文字列	73
7.15	練習	74
7.16	for 文によるリスト初期化の短縮記法	74
7.17	▲オブジェクトの等価性と同一性	75
7.18	練習の解答	76
8	2-3. 条件分岐	78
8.1	インデントによる構文	78
8.2	if … else による条件分岐	79
8.3	if … elif … else による条件分岐	80
8.4	練習	80
8.5	練習	81
8.6	▲複数行にまたがる条件式	81
8.7	if … elif … else における条件の評価	82
8.8	練習	82
8.9	or もしくは and で結合された条件の評価	82

8.10	▲ 3 項演算子 (条件式)	83
8.11	練習の解答	84
8.12	練習の解説	84
9	3-1. 辞書 (dictionary)	85
9.1	練習	87
9.2	辞書のメソッド	87
9.3	辞書とリスト	90
9.4	練習	91
9.5	練習の解答	91
10	3-2. 繰り返し	92
10.1	for 文による繰り返し	92
10.2	for 文による繰り返しと辞書	95
10.3	練習	96
10.4	range	96
10.5	練習	97
10.6	練習	98
10.7	range とリスト	98
10.8	for 文の入れ子	98
10.9	練習	100
10.10	練習	100
10.11	for 文の計算量	100
10.12	enumerate	102
10.13	in	102
10.14	while 文による繰り返し	103
10.15	制御構造と return 文	103
10.16	break 文	104
10.17	練習	104
10.18	continue 文	104
10.19	▲ for 文と while 文における else	105
10.20	pass 文	105
10.21	練習	106
10.22	練習	106
10.23	練習	106
10.24	練習	107
10.25	練習	107
10.26	練習	107
10.27	練習	108
10.28	練習	108
10.29	練習	109
10.30	練習の解答	109
10.31	練習の解説	111
10.32	練習の解答	111
11	3-3. 関数	114
11.1	関数の定義	114
11.2	引数	115
11.3	返値	115
11.4	複数の引数	116
11.5	変数とスコープ	116
11.6	▲キーワード引数	118
11.7	▲引数の初期値	119
11.8	▲可変長引数	119
11.9	▲辞書型の可変長引数	119
11.10	▲引数の順番	120
11.11	▲変数としての関数	120
12	4-1. ファイル入出力の基本	122

12.1	ファイルのオープン	122
12.2	ファイルのクローズ	123
12.3	行の読み込み	123
12.4	練習	124
12.5	ファイル全体の読み込み	124
12.6	練習	124
12.7	編集時のファイルの動作	125
12.8	ファイルに対する with 文	125
12.9	ファイルへの書き込み	125
12.10	練習	126
12.11	ファイルの読み書きにおける文字コード指定	127
12.12	改行文字の削除	128
12.13	練習の解答	129
13	4-2. イテレータ	130
13.1	next	130
13.2	for 文による繰り返しとファイルオブジェクト	131
13.3	練習	132
13.4	iter	132
13.5	練習	134
13.6	イテラブル	134
13.7	イテレータを返す enumerate	135
13.8	練習の解答	136
14	4-3. コンピュータにおけるファイルやディレクトリの配置	137
14.1	カレントワーキングディレクトリ	139
14.2	木構造によるデータ表現	140
15	5-1. モジュールの使い方	142
15.1	モジュールのインポート	142
15.2	from	143
15.3	as	144
15.4	練習	144
15.5	練習の解答	145
16	5-2. モジュールの作り方	146
16.1	モジュールファイル	146
16.2	自作モジュールの使い方	147
17	5-3. NumPy ライブラリ	148
17.1	配列の構築	148
17.2	配列要素を生成する構築関数	151
17.3	練習	152
17.4	配列要素の操作	153
17.5	要素毎の演算	155
17.6	よく使われる配列操作	157
17.7	配列の保存と復元	159
17.8	▲真理値配列によるインデックスアクセス	159
17.9	▲線形代数の演算	160
17.10	練習の解答	161
18	6-1. 内包表記	162
18.1	リスト内包表記	162
18.2	練習	163
18.3	練習	164
18.4	練習	164
18.5	内包表記の入れ子	164
18.6	練習	165
18.7	練習	165
18.8	▲条件付き内包表記	166

18.9	▲セット内包表記	166
18.10	▲辞書内包表記	166
18.11	▲ジェネレータ式	166
18.12	練習の解答	167
19	6-2. 高階関数	169
19.1	max	169
19.2	sorted	170
19.3	ラムダ式	170
19.4	リストからイテラブルへ	171
19.5	練習	171
19.6	map	172
19.7	練習	173
19.8	filter	173
19.9	練習	174
19.10	練習	174
19.11	練習の解答	174
20	6-3. クラス	176
20.1	クラス定義	176
20.2	初期化と属性	178
20.3	継承	179
20.4	特殊メソッド	179
20.5	継承による振舞いの改変	180
20.6	練習	181
20.7	▲with 文への対応	182
20.8	練習の解答	182
21	7-1. pandas ライブラリ	183
21.1	シリーズとデータフレーム	183
21.2	シリーズ (Series) の作成	184
21.3	データフレーム (DataFrame) の作成	184
21.4	CSV ファイルからのデータフレームの作成	185
21.5	データの参照	186
21.6	データの条件取り出し	188
21.7	列の追加と削除	188
21.8	行の追加と削除	189
21.9	データの並び替え	190
21.10	データの統計量	191
21.11	▲データの連結	191
21.12	▲データの結合	192
21.13	▲データのグループ化	192
21.14	▲欠損値、時系列データの処理	193
22	7-2. scikit-learn ライブラリ	194
22.1	機械学習について	194
22.2	教師あり学習	195
22.3	教師なし学習	195
22.4	データ	195
22.5	モデル学習の基礎	196
22.6	教師あり学習・分類の例	196
22.7	練習	197
22.8	教師あり学習・回帰の例	198
22.9	教師なし学習・クラスタリングの例	199
22.10	練習	201
22.11	教師なし学習・次元削減の例	201
22.12	練習の解答例	202
23	▲ Jupyter Notebook の使い方	204

23.1	セル	204
23.2	コマンドモード	205
23.3	編集モード	205
23.4	練習	205
23.5	(注意) Shift-Enter に反応がなくなったとき	206
24	▲セット (set)	207
24.1	セットの組み込み関数	208
24.2	練習	209
24.3	練習	209
24.4	集合演算	209
24.5	比較演算	210
24.6	セットのメソッド	210
24.7	練習	212
24.8	練習の解答	212
25	▲再帰	214
25.1	再帰関数の例：接頭辞リストと接尾辞リスト	214
25.2	再帰関数の例：べき乗の計算	215
25.3	再帰関数の例：マージソート	216
26	▲簡単なデータの可視化	219
26.1	matplotlib	219
26.2	折れ線グラフ	220
26.3	散布図	221
26.4	棒グラフ	221
27	▲CSV ファイルの入出力	223
27.1	CSV 形式とは	223
27.2	CSV ファイルの読み込み	224
27.3	CSV ファイルに対する for 文	225
27.4	CSV ファイルに対する with 文	225
27.5	CSV ファイルの書き込み	225
27.6	練習	233
27.7	練習	233
27.8	練習の解答	234
28	▲Bokeh ライブラリ	235
28.1	線グラフ	235
28.2	散布図	238
28.3	棒グラフ	239
28.4	ヒストグラム	239
28.5	ヒートマップ	240
28.6	グラフのファイル出力	240
29	▲Python スクリプトとコマンドライン実行	242
29.1	コマンドライン実行の具体例	243
29.2	コマンドライン引数	247
29.3	モジュールのコマンドライン実行	248
29.4	ソースファイル先頭部分にある宣言	249
29.5	練習の解答	250
30	▲Matplotlib ライブラリ	252
30.1	線グラフ	252
30.2	練習	259
30.3	練習	260
30.4	散布図	261
30.5	練習	262
30.6	棒グラフ	263
30.7	練習	263

30.8	ヒストグラム	264
30.9	練習	264
30.10	ヒートマップ	265
30.11	練習	266
30.12	グラフの画像ファイル出力	266
30.13	練習の解答	267
31	▲正規表現	270
31.1	正規表現の基本	270
31.2	練習	276
31.3	練習	277
31.4	練習	277
31.5	練習	278
31.6	練習	279
31.7	文字クラス	280
31.8	正規表現に関する基本的な関数	282
31.9	練習	284
31.10	練習	284
31.11	その他の反復演算	285
31.12	メタ文字	286
31.13	練習	288
31.14	練習	288
31.15	練習	289
31.16	練習	289
31.17	正規表現のエスケープシーケンス	290
31.18	正規表現に関する関数とメソッド	291
31.19	練習	293
31.20	練習	294
31.21	練習の解答	294
32	索引	299

▲で始まる項目は授業では扱いません。興味にしたがって学習してください。
ノートブック全体に▲が付いているものもありますので注意してください。

1-0. Colaboratory によるノートブックの使い方

Colaboratory によるノートブックの使い方について説明します。

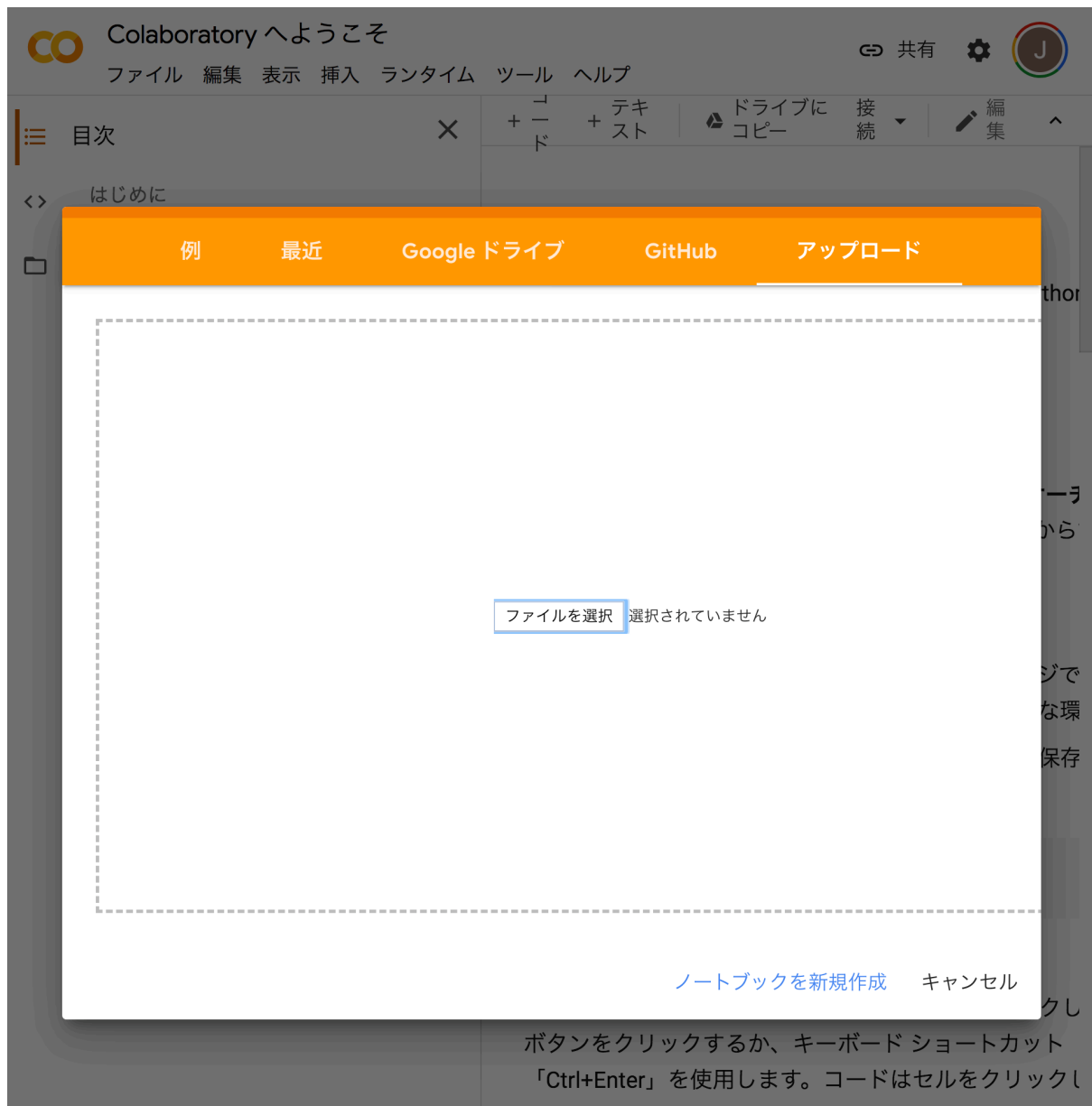
1.1 Colaboratory の立ち上げ

ブラウザに Google アカウント（個人でも ECCS でもどちらでもよい）でログインした後に、以下の URL を開いてください。

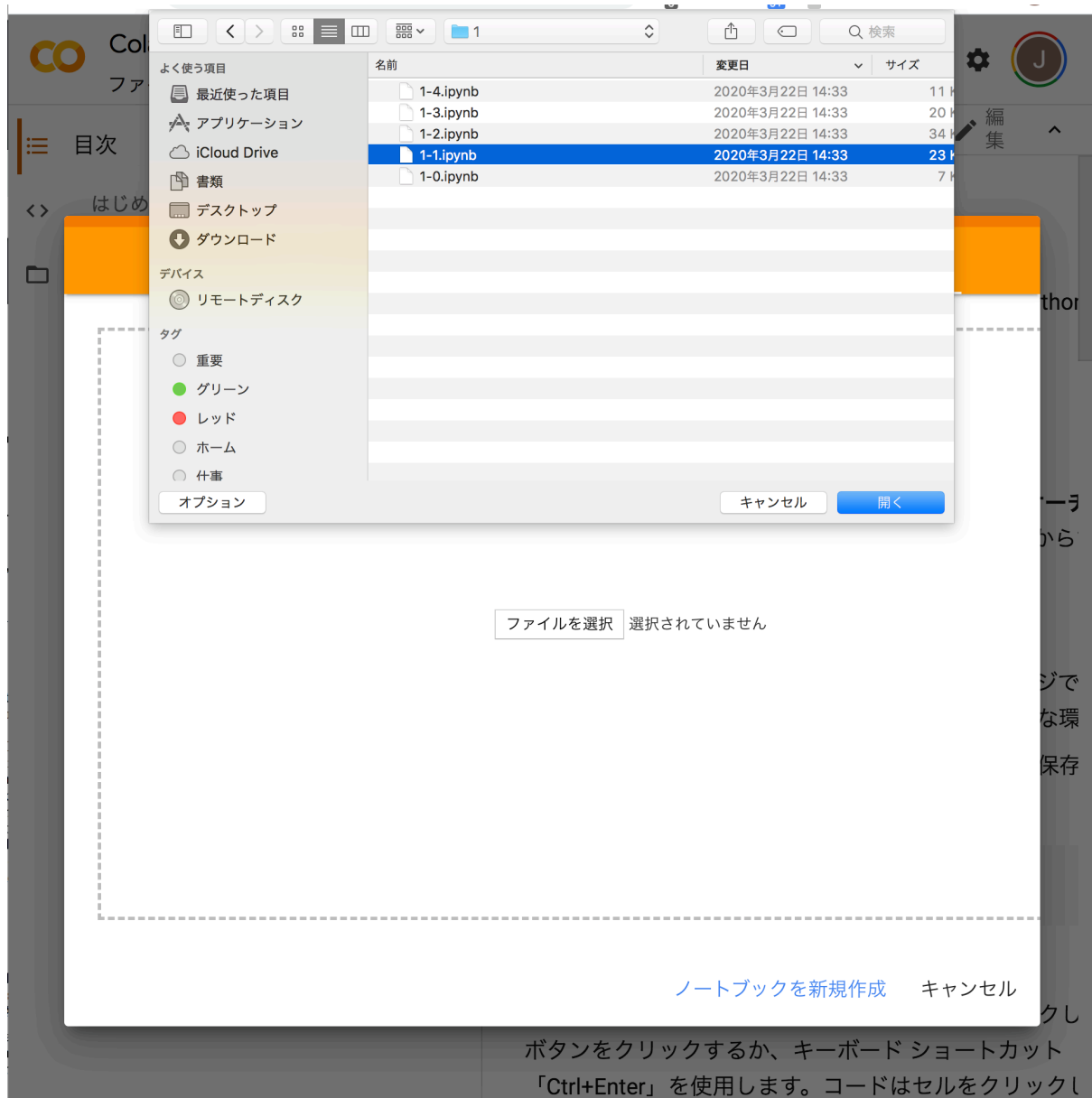
- <https://colab.research.google.com/>

1.2 ノートブックのアップロード

Colaboratory を立ち上げると、以下のようにノートブックを指定することが求められますので、「アップロード」のタブを選択してください。



そして、アップロードすべきファイルを指定してください。



指定したファイルは、いったん自分の Google Drive にアップロードされてから、Colaboratory によって開かれます。ブラウザでは以下のように表示されるでしょう。

1-1.ipynb - Colaboratory

colab.research.google.com/drive/1zs7wl-Knwzh8WPbADvCL019aTxzaA8uo

1-1.ipynb

ファイル 編集 表示 挿入 ランタイム ツール ヘルプ 最終保存: 9:40

+ コード + テキスト

接続 編集

1-1. 数値演算

数値演算について説明します。

参考

- <https://docs.python.org/ja/3/tutorial/introduction.html#using-python-as-a-calculator>
- <https://docs.python.org/ja/3/tutorial/modules.html>
- <https://docs.python.org/ja/3/library/numeric.html>

簡単な算術計算

CodeセルにPythonの式を入力して、プレイボタンを押すか、Shiftを押しながらEnterを押すと、式が評価され、その結果の値がセルの下に挿入されます。

1+1 の計算をしてみましょう。次のセルに 1+1 と入力して、Shiftを押しながらEnterを押してください。

```
[ ]
```

このようにして、電卓の代わりにPythonを使うことができます。+ は言うまでもなく**足し算**を表しています。

```
[ ] 7-2
```

```
[ ] 7*2
```

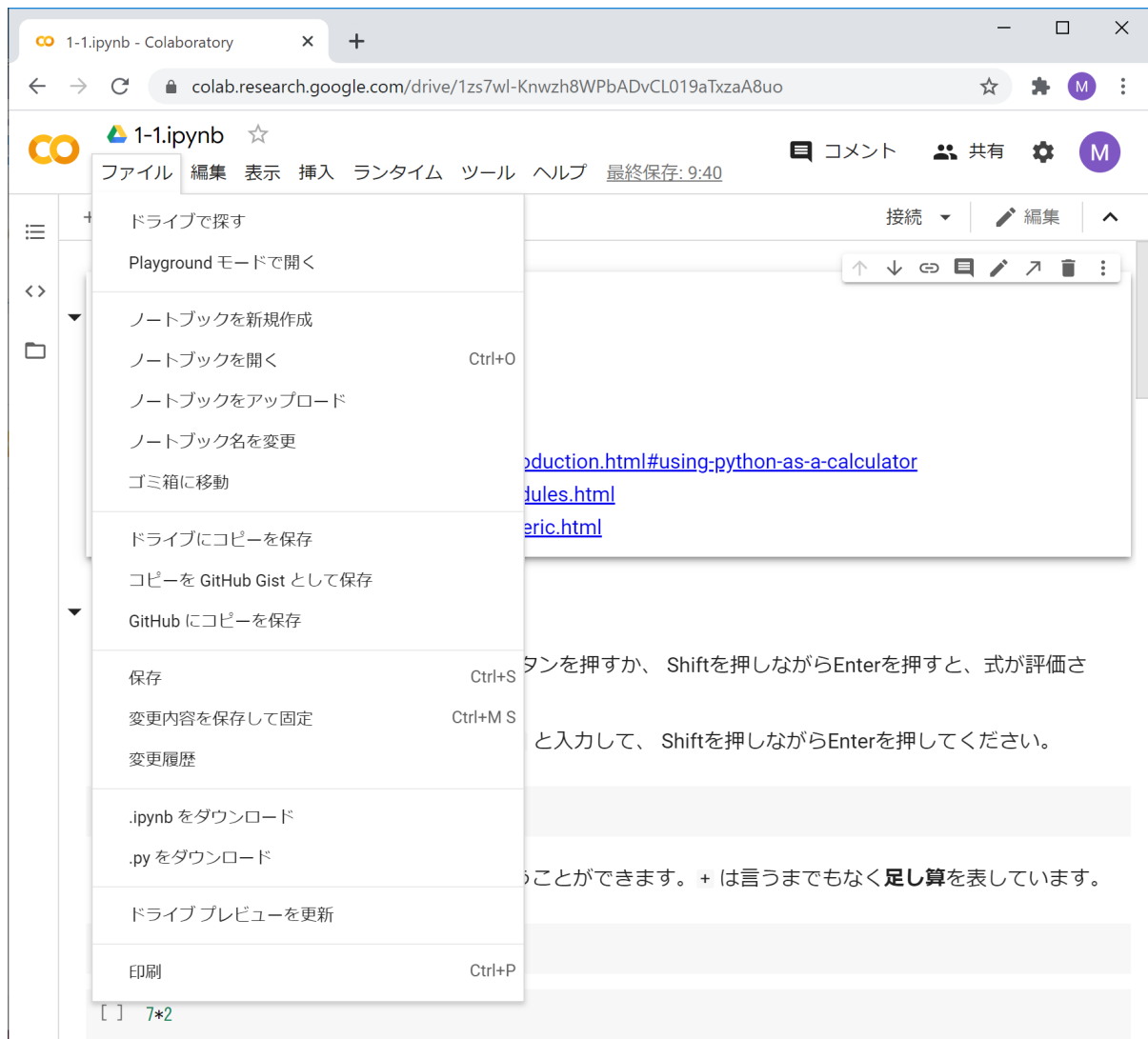
なお、ノートブックには `ipynb` という拡張子（エクステンション）が付いています。

1.3 教材のオープン

HTML 版の教材の「Open in Colab」をクリックしたり、Google Drive 上の教材を直接 Colaboratory でオープンした場合、指定したノートブックがオープンされますが、ノートブックを操作した結果は Google Drive 上に保存されません。ノートブックの上方にある「ドライブにコピー」のボタンを押せば、自分の Google Drive 上にノートブックのコピーが作られてオープンされます。ノートブックを操作した結果はコピーに保存されます。

1.4 ノートブックのダウンロード

Google Drive 上のノートブックをパソコンにダウンロードするには、Colaboratory のファイルメニューで「.ipynb をダウンロード」を選択します。



1.5 ノートブックのアップロード（再び）

最初のノートブックを開いた後に、別のノートブックを開くには、Colaboratory のファイルメニューで「ノートブックをアップロード」を選択してください。

The screenshot shows the Google Colaboratory web interface. The 'File' menu is open, displaying various options for creating and managing notebooks. The main workspace contains a welcome message and a code cell that demonstrates TensorFlow matrix addition.

File Menu Options:

- Python 2 のノートブックを新規作成
- Python 3 のノートブックを新規作成
- ドライブのノートブックを開く... (⌘/Ctrl+O)
- 最近のノートブックを開く... (⌘/Ctrl+Shift+L)
- ノートブックをアップロード...
- 名前を変更...
- ゴミ箱に移動
- ドライブにコピーを保存...
- GitHub にコピーを保存...
- 保存 (⌘/Ctrl+S)
- 変更内容を保存して固定 (⌘/Ctrl+M S)
- 変更履歴
- .ipynb をダウンロード
- .py をダウンロード
- ドライブ プレビューを更新
- 印刷 (⌘/Ctrl+P)

Main Editor Content:

Welcome to Colaboratory!

Colaboratory is a Google research project created to help disseminate machine learning education and research. It's a Jupyter notebook environment that requires no setup to use and runs entirely in the cloud. Colaboratory notebooks are stored in [Google Drive](#) and can be shared just as you would with Google Docs or Sheets. Colaboratory is free to use. For more information, see our [FAQ](#).

runtime support

Colaboratory also supports connecting to a Jupyter runtime on your local machine. For more information, see our [documentation](#).

Python 3

Colaboratory supports both Python2 and Python3 for code execution. When creating a new notebook, you'll have the choice between Python 2 and Python 3. You can also change the language associated with a notebook; this information will be written into the .ipynb file itself, and thus will be preserved for future sessions.

```
import sys
print('Hello, Colaboratory from Python {}'.format(sys.version_info[0]))
```

Hello, Colaboratory from Python 3!

TensorFlow execution

Colaboratory allows you to execute TensorFlow code in your browser with a single click. The example below adds two matrices.

$$\begin{bmatrix} 1. & 1. & 1. \\ 1. & 1. & 1. \end{bmatrix} + \begin{bmatrix} 1. & 2. & 3. \\ 4. & 5. & 6. \end{bmatrix} = \begin{bmatrix} 2. & 3. & 4. \\ 5. & 6. & 7. \end{bmatrix}$$

```
[ ] import tensorflow as tf
import numpy as np

with tf.Session():
    input1 = tf.constant(1.0, shape=[2, 3])
    input2 = tf.constant(np.reshape(np.arange(1.0, 7.0, dtype=np.float32), (2, 3)))
    output = tf.add(input1, input2)
    result = output.eval()
```

array([[2., 3., 4.],
 [5., 6., 7.]], dtype=float32)

Visualization

Colaboratory includes widely used libraries like [matplotlib](#), simplifying visualization.

```
[ ] import matplotlib.pyplot as plt
import numpy as np

x = np.arange(20)
y = [x_i + np.random.randn(1) for x_i in x]
a, b = np.polyfit(x, y, 1)
_ = plt.plot(x, y, 'o', np.arange(20), a*np.arange(20)+b, '-')
```

1.6 ノートブックの作成

また、ノートブックを新たに作成するには、Colaboratory のファイルメニューで「ノートブックを新規作成」を選択してください。Untitled0.ipynb という名前のノートブックが作られます。上方に表示されたタイトルをクリックすれば名前を変更することができます。

1.7 ノートブックの操作

ノートブックの上方のタイトルの下には、「ファイル」や「編集」などのメニュー、その下には「+ コード」と「+ テキスト」というボタンが表示されています。

Ctrl+s (Mac の場合は Cmd+s) を入力することによって、編集・操作中のノートブックを Google Drive のファイルにセーブできます。なお、ノートブックは適当なタイミングでオートセーブされます。ファイルメニューの右に「すべての変更を保存しました」と表示されていれば、Ctrl+s を入力する必要はありません。

以下の参考文献は、Jupyter Notebook に関する一般的な解説です。jupyter コマンドを起動してブラウザでノートブックを使うのと、Google Colaboratory によりノートブックを使うのでは、インタフェースが大幅異なっていることに注意してください。

- <https://jupyter.readthedocs.io/en/latest/>

1.8 セル

ノートブックはセルから成り立っています。

主に次の二種類のセルを使います。

- **Code セル (コードセル)**: Python のコードが書かれたセルです。Code セルの左端には [] と表示されています。Code セルの中のコードを実行するには、[] のところにマウスカーソルを移動してクリックします。[] のところにマウスカーソルを移動すると、●の中に▶が表示されます。これはプレイボタンを意味します。プレイボタンを押すとコードが実行され、その結果がセルの下部に挿入されます。(Shift を押しながら Enter を押しても実行できます。)
- **Markdown セル (テキストセル)**: 説明が書かれたセルです。このセル自身は Markdown セルです。

[]: 1+1

1.9 セルの編集

Code セル上のプレイボタンでないところにマウスカーソルを移動しクリックすると、Code セルが選択され、文字カーソルが表示されて、セルの編集が可能になります。Ctrl の付かない文字はそのまま挿入されます。

以下のような編集コマンドが使えます。

- Ctrl+c: copy
- Ctrl+x: cut
- Ctrl+v: paste
- Ctrl+z: undo
- ...

Code セルが選択されているとき、Shift+Enter (もしくは Shift+Return) を入力すると、セルの中のコードが実行されて、次のセルが選択されます。

1.10 練習

次の Code セルを選択して 10/3 と入力して実行してください。

[]:

Code セルの実行が終了し、別のセルが選択されると、セルの左端は [2] のようになり、[] の中に番号が入ります。この番号は、その Code セルが何番目に実行されたかを示すもので、Code セルが実行されるたびに 1 ずつ増えます。同じセルを続けて実行すれば、この番号は 1 ずつ増えるでしょう。

1.11 セルの挿入

Code セルを新たに挿入するには、ファイルメニューの下の「+ コード」ボタンを押します。現在選択されているセルの下に Code セルが挿入されます。

たとえば、この Markdown セルを選択してから、「+ コード」ボタンを押してみてください。この Markdown セルを選択するには、マウスカーソルをここに持って来てクリックすればよいです。説明の全体が四角で囲まれるはずです。

Markdown セルを新たに挿入するには、ファイルメニューの下の「+ テキスト」ボタンを押します。なお、ここでは Markdown セルの説明は行っていません。

1.12 セルの実行が止まらないとき

Code セルの左端の●の中に□が表示され、その周りをノの字が回り続けているならば、セルのコードは実行中です。いつまでたってもコードの実行が終了しない場合は、そのアイコンを押して、コードの実行を強制終了してください。●の中に□が表示されたアイコンはストップボタンを意味します。

たとえば、次のような例です。プレイボタンを押して実行中であることを確かめてから、ストップボタンを押してください。

[]:

```
while True:
    pass
```

1.13 セルの操作

セルを選択するとセルの右上に色々なボタンが表示されます。これらを押すことにより、セルの削除、セルの移動、コピーペーストなど、セルに対する各種の操作を行うことができます。

1.14 ノートブックの参照

Colaboratory が使用するノートブックは Google Drive 上にあります。右上の「共有」のボタンを押すことにより、ノートブックの共有設定を変えることができます。その上で、ノートブックが表示されているブラウザ上の URL を伝えることにより、教員や TA にノートブックを見せることができます。

[]:

1-1. 数値演算

数値演算について説明します。

参考

- <https://docs.python.org/ja/3/tutorial/introduction.html#using-python-as-a-calculator>
- <https://docs.python.org/ja/3/tutorial/modules.html>
- <https://docs.python.org/ja/3/library/numeric.html>
- <https://docs.python.org/ja/3/library/math.html>

2.1 簡単な算術計算

Code セルに Python の式を入力して、プレイボタンを押すか、Shift を押しながら Enter を押すと、式が評価され、その結果の値がセルの下に挿入されます。

1+1 の計算をしてみましょう。次のセルに 1+1 と入力して、Shift を押しながら Enter を押してください。

[]:

このようにして、電卓の代わりに Python を使うことができます。+ は言うまでもなく足し算を表しています。

[1]: 7-2

[1]: 5

[2]: 7*2

[2]: 14

[3]: 7**2

[3]: 49

- は引き算、* は掛け算、** はべき乗を表しています。

式を適当に書き換えてから、Shift を押しながら Enter を押すと、書き換えた後の式が評価されて、セルの下のはその結果で置き換わります。たとえば、上の 2 を 100 に書き換えて、7 の 100 乗を求めてみてください。

割り算はどうなるでしょうか。

```
[4]: 7/2
```

```
[4]: 3.5
```

```
[5]: 7//2
```

```
[5]: 3
```

Python では、割り算（除算）は / で表され、整数除算は // で表されます。// は小数部を切り捨てた整数値（商）を返します。

整数同士の // の結果は整数になります。

```
[6]: 7/1
```

```
[6]: 7.0
```

```
[7]: 7//1
```

```
[7]: 7
```

整数除算の余り（剰余）を求めたいときは、別の演算子 % を用います。

整数同士の % の結果は整数になります。

```
[8]: 7%2
```

```
[8]: 1
```

2.2 コメント

Python では一般に、コードの中に # が出現すると、それ以降、その行の終わりまでがコメントになります。コメントは行頭からも、行の途中からでも始めることができます。

プログラムの実行時には、コメントは無視されます。

```
[9]: # このように行頭に '#' をおけば、行全体をコメントとすることができます。
```

```
# 次のようにコード行に続けて直前のコードについての説明をコメントとして書くこともできます。
2**10 # 2 の 10 乗を計算します
```

```
[9]: 1024
```

```
[10]: # 次のようにコード行自体をコメントとすることで、その行を無視させる（コメントアウトする）こともよく行われます。
```

```
# 2**10 # 2 の 10 乗を計算します この行が「コメントアウト」された
2**12 # 実は計算したいのは 2 の 12 乗でした
```

```
[10]: 4096
```

2.3 整数と実数

Python では、**整数**と小数点のある数（**実数**）は、数学的に同じ数を表す場合でも、コンピュータの中で異なる形式で記憶されますので、表示は異なります。（実数は**浮動小数点数**ともいいます。）

```
[11]: 7/1
```

```
[11]: 7.0
```

```
[12]: 7//1
```

```
[12]: 7
```

しかし、以下のように、比較を行うと両者は等しいものとして扱われます。値同士が等しいかどうかを調べる `==` という演算子については、後で紹介します。

```
[13]: 7/1 == 7//1
```

```
[13]: True
```

`+` と `-` と `*` と `//` と `%` と `**` では、2つの数が整数ならば結果も整数になります。2つの数が実数であったり、整数と実数が混ざっていたら、結果は実数になります。

```
[14]: 2+5
```

```
[14]: 7
```

```
[15]: 2+5.0
```

```
[15]: 7.0
```

`/` の結果は必ず実数となります。

```
[16]: 7/1
```

```
[16]: 7.0
```

ここで、自分で色々と式を入力してみてください。以下に、いくつかセルを用意しておきます。足りなければ、Insert メニューを使ってセルを追加することができます。

```
[ ]:
```

```
[ ]:
```

```
[ ]:
```

2.3.1 実数のべき表示

```
[17]: 2.0**1000
```

```
[17]: 1.0715086071862673e+301
```

非常に大きな実数は、10 のべきとともに表示（**べき表示**）されます。 `e+301` は 10 の 301 乗を意味します。

```
[18]: 2.0**-1000
```

```
[18]: 9.332636185032189e-302
```

非常に小さな実数も、10 のべきとともに表示されます。 `e-302` は 10 の -302 乗を意味します。

2.3.2 いくらでも大きくなる整数

```
[19]: 2**1000
```

```
[19]: 1071508607186267320948425049060001810561404811705533607443750388370351051124936122493198378815695858
```

このように、Python では整数はいくらでも大きくなります。もちろん、コンピュータのメモリに納まる限りにおいてですが。

```
[20]: 2**2**2**2**2
```

```
[20]: 2003529930406846464979072351560255750447825475569751419265016973710894059556311453089506130880933348
```

2.3.3 整数と実数の間の変換

実数を整数に変換するには、`int` という関数を用います。（関数に関する一般的な説明は 1-2 を参照してください。）`int(x)` は、実数 `x` を（0 の方向に）切り下げた結果を返します。

```
[21]: int(2.9)
```

```
[21]: 2
```

```
[22]: int(-2.9)
```

```
[22]: -2
```

逆に、整数を実数に変換するには、`float` という関数を用います。`float(i)` は、整数 `i` を実数に変換した結果を返します。たとえば `i+0.0` としても、`i` を実数に変換できます。

```
[23]: float(2)
```

```
[23]: 2.0
```

```
[24]: 2+0.0
```

```
[24]: 2.0
```

2.4 演算子の優先順位と括弧

掛け算や割り算は足し算や引き算よりも先に評価されます。すなわち、掛け算や割り算の方が足し算や引き算よりも**優先順位**が高いと定義されています。

括弧を使って式の評価順序を指定することができます。

なお、数式 $a(b-c)$ 、 $(a-b)(c-d)$ は、それぞれ a と $b-c$ 、 $a-b$ と $c-d$ の積を意味しますが、コードでは、`a*(b-c)` や `(a-b)*(c-d)` のように積の演算子である `*` を明記する必要があることに注意してください。

また、数や演算子の間には、自由に空白を入れることができます。（後でもう一度説明します。）

```
[25]: 7 - 2 * 3
```

```
[25]: 1
```

```
[26]: (7 - 2) * 3
```

```
[26]: 15
```

```
[27]: 17 - 17//3*3
```

```
[27]: 2
```

```
[28]: 56 ** 4 ** 2
```

```
[28]: 9354238358105289311446368256
```

```
[29]: 56 ** 16
```

```
[29]: 9354238358105289311446368256
```

上の例では、 $4**2$ が先に評価されて、 $56**16$ が計算されます。つまり、 $x**y**z = x**(y**z)$ が成り立ちます。このことをもって、 $**$ は右に結合するといいます。

```
[30]: 16/8/2
```

```
[30]: 1.0
```

```
[31]: (16/8)/2
```

```
[31]: 1.0
```

上の例では、 $16/8$ が先に評価されて、 $2/2$ が計算されます。つまり、 $x/y/z = (x/y)/z$ が成り立ちます。このことをもって、 $/$ は左に結合するといいます。

$*$ と $/$ をまぜても左に結合します。

```
[32]: 10/2*3
```

```
[32]: 15.0
```

以上のように、演算子によって式の評価の順番が変わりますので注意してください。

ではまた、自分で色々と式を入力してみてください。以下に、いくつかセルを用意しておきます。

```
[ ]:
```

```
[ ]:
```

```
[ ]:
```

2.4.1 単項の + と -

$+$ と $-$ は、単項の演算子（**単項演算子**）としても使えます。（これらの演算子の後に 1 つだけ数書かれます。前と後の両方に数書かれる演算子は **2 項演算子** と言います。）

```
[33]: -3
```

```
[33]: -3
```

```
[34]: +3
```

```
[34]: 3
```

2.5 算術演算子のまとめ

算術演算子を、評価の優先順位にしたがって、すなわち結合力の強い順にまとめておきましょう。

単項の + と - は最も強く結合します。

次に、** が強く結合します。** は右に結合します。

その次に、2 項の * と / と // と % が強く結合します。これらは左に結合します。

最後に、2 項の + と - は最も弱く結合します。これらも左に結合します。

2.6 空白

既に $7 - 2 * 3$ のような例が出てきましたが、演算子と数の間や、演算子と変数（後述）の間には、空白を入れることができます。ここで空白とは、半角の空白のことで、英数字と同様に 1 バイトの文字コードに含まれているものです。

複数の文字から成る演算子、たとえば ** や // の間に空白を入れることはできません。エラーになることでしょう。

```
[35]: 7 **2
```

```
[35]: 49
```

```
[36]: 7* *2
```

```
Input In [36]
  7* *2
    ^
SyntaxError: invalid syntax
```

2.6.1 全角の空白

日本語文字コードである全角の空白は、空白とはみなされませんので注意してください。

```
[37]: 7  **2
```

```
Input In [37]
  7  **2
    ^
SyntaxError: invalid character in identifier
```

2.7 エラー

色々と試していると、エラーが起こることもあったでしょう。以下は典型的なエラーです。

```
[38]: 10/0
```

```
-----
ZeroDivisionError                                Traceback (most recent call last)
Input In [38], in <module>
----> 1 10/0

ZeroDivisionError: division by zero
```

このエラーは、ゼロによる割り算を行ったためです。実行時エラーの典型的なものです。

エラーが起こった場合は、修正して評価し直すことができます。上の例で、0 をたとえば 3 に書き換えて評価し直してみてください。

```
[39]: 10/

      Input In [39]
      10/
        ^
SyntaxError: invalid syntax
```

こちらのエラーは構文エラーです。つまり、入力がある Python の構文に違反しているため実行できなかったのです。

2.8 数学関数（モジュールのインポート）

```
[40]: import math
```

```
[41]: math.sqrt(2)
```

```
[41]: 1.4142135623730951
```

数学関係の各種の関数は、モジュール（ライブラリ）として提供されています。これらの関数を使いたいときは、上のように、`import` で始まる `import math` というおまじないを一度唱えます。そうしますと、`math` というライブラリが読み込まれて（インポートされて）、`math.` 関数名 という形で関数を用いることができます。上の例では、平方根を計算する `math.sqrt` という関数が用いられています。

もう少し例をあげておきましょう。`sin` と `cos` は `math.sin` と `math.cos` で求められます。

```
[42]: math.sin(0)
```

```
[42]: 0.0
```

```
[43]: math.pi
```

```
[43]: 3.141592653589793
```

`math.pi` は、円周率を値とする変数です。

変数については後に説明されます。

```
[44]: math.sin(math.pi)
```

```
[44]: 1.2246467991473532e-16
```

この結果は本当は 0 にならなければならないのですが、数値誤差のためにこのようになっています。

```
[45]: math.sin(math.pi/2)
```

```
[45]: 1.0
```

```
[46]: math.sin(math.pi/4) * 2
```

```
[46]: 1.414213562373095
```


2.9 練習

黄金比を求めてください。黄金比とは、5 の平方根に 1 を加えて 2 で割ったものです。約 1.618 になるはずです。

[]:

1-2. 変数と関数の基礎

変数と関数の基礎について説明します。

参考

- <https://docs.python.org/ja/3/tutorial/introduction.html#first-steps-towards-programming>
- <https://docs.python.org/ja/3/tutorial/controlflow.html#defining-functions>
- <https://docs.python.org/ja/3/library/functions.html#print>

3.1 変数

プログラミング言語における**変数**とは、値に名前を付ける仕組みであり、名前はその値を指し示すことになります。

```
[1]: h = 188.0
```

以上のように = を用いる構文によって、188.0 という値に h という名前が付きます。これを**変数定義**と呼びます。

定義された変数は、式の中で使うことができます。h という変数自体も式なので、h という式を評価することができ、変数が指し示す値が返ります。

```
[2]: h
```

```
[2]: 188.0
```

異なる変数は、いくらでも導入できます。たとえば、以下では w を変数定義します。

```
[3]: w = 104.0
```

ここで、h を身長 (cm)、w を体重 (kg) の意味と考えると、次の式によって BMI (ボディマス指数) を計算できます。

```
[4]: w / (h/100.0) ** 2
```

```
[4]: 29.425079221367138
```

なお、演算子 `**` の方が `/` よりも先に評価されることに注意してください。

変数という名前の通り、変数が指し示す値を変えることもできます。

```
[5]: w = 104.0-10
```

このように変数を再定義すれば、元々 `w` が指し示していた値 `104.0` を忘れて、新たな値 `94.0` を指し示すようになります。この後で、前と同じ BMI の式を評価してみると、`w` の値の変化に応じて、BMI の計算結果は変わります。

```
[6]: w / (h/100.0) ** 2
```

```
[6]: 26.595744680851066
```

なお、未定義の変数（たとえば BMI）を式の中で用いると、次のようにエラーが生じます。

```
-----
NameError                                Traceback (most recent call last)
<ipython-input-1-b910749d4383> in <module>
----> 1 BMI # 未定義の変数

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

次のセルの行頭にある `#` を削除して実行してみましょう。

```
[7]: # BMI # 未定義の変数
```

以降では、単純のため、変数が指し示す値を、変数の値として説明していきます。

3.1.1 代入文

変数定義に用いた `=` による構文を、Python では**代入文 (assignment statement)** と呼びます。そして、代入文を実行することを**代入 (assignment)** と言います。代入文は、`=` の左辺に右辺の式の評価結果の値を割り当てる文です。上記の例のように、左辺が変数の場合には、代入文は変数定義と解釈されます。

代入文は、右辺を評価した後に左辺に割り当てるという順番に従います。右辺に出現する変数が左辺に出て来てもかまいません。

```
[8]: w = w-10
```

上の代入文は、`w` の値を `10` 減らす操作となります。`=` は数学的な等号ではないことに注意してください。

もう一度 BMI を計算してみると、`w` の値が減ったことで、先と結果が変わります。

```
[9]: w / (h/100.0) ** 2
```

```
[9]: 23.766410140334994
```

注意： 数学における代入は、substitution（置換）であり、プログラミング言語における代入 (assignment) とは異なります。代入という単語よりも、assignment（割り当て）という単語で概念を覚えましょう。

3.1.2 累積代入文

上の例のように変数の値を減らす操作は、次のような**累算代入文 (augmented assignment statement)** を使って簡潔に記述することができます。

```
[10]: w -= 10
```

ここで、`-=` という演算子は、`-` と `=` を結合させた演算子で、`w = w - 10` という代入文と同じ意味になります。これは代入文と 2 項演算が複合したものであり、`-` に限らず、他の 2 項演算についても同様に複合した累算代入文が利用できます。たとえば、変数の値を増やすには `+=` という演算子を用いることができます。

```
[11]: w += 10
```

`=` も含めて、これらの演算子は**代入演算子**と呼ばれています。代入演算子によって変数の値がどのように変わるか、確かめてください。

```
[ ]:
```

3.2 関数の定義と返値

前述のように、変数の値が変わるたびに BMI の式を入力するのは面倒です。以下では、身長 `height` と体重 `weight` をもらって、BMI を計算する**関数 bmi** を定義してみましょう。関数を定義すると、BMI の式の再入力を省けて便利です。

次のような形式で、**関数定義**を記述できます。

関数定義など、複数行のコードセルには、**行番号**を振るのがよいかもしれません。行番号を振るかどうかは、コマンドモードでエルの文字（大文字でも小文字でもよいです）を入力することによって、スイッチできます。行番号があるかないかは、コードの実行には影響しません。

```
[12]: def bmi(height, weight):
      return weight / (height/100.0) ** 2
```

Python では、**関数定義**は、上のような形をしています。最初の行は以下のように `def` で始まります。

```
def 関数名 (引数, ...):
```

引数（ひきすう）とは、関数が受け取る値を指し示す変数のことです。**仮引数**（かりひきすう）ともいいます。

: 以降は関数定義の本体であり、関数の処理を記述する部分として以下の構文が続きます。

```
return 式
```

この構文は `return` で始まり、**return 文**と呼ばれます。`return` 文は、`return` に続く式の評価結果を、関数の呼び出し元に返して（これを**返値**と言います）、関数を終了するという意味を持ちます。この関数を、入力となる引数とともに呼び出すと、`return` の後の式の評価結果を返値として返します。

ここで、Python では、`return` の前に空白が入ることに注意してください。このような行頭の空白を**インデント**と呼びます。Python では、インデントの量によって、構文の**入れ子**を制御するようになっています。このことについては、より複雑な構文が出てきたときに説明しましょう。

上記では、`def` の後に続く `bmi` が関数名です。それに続く括弧の中に書かれた `height` と `weight` は、**引数**です。また、`return` の後に BMI の計算式を記述しているので、関数の呼び出し元には BMI の計算結果が返値として返ります。

では、定義した関数 `bmi` を呼び出してみましょう。

```
[13]: bmi(188.0, 104.0)
```

```
[13]: 29.425079221367138
```

第 1 引数を身長 (cm)、第 2 引数を体重 (kg) としたときの BMI が計算されていることがわかります。

関数呼び出しは演算式の種類なので、引数の位置には任意の式を記述できますし、関数呼び出し自体も式の中に記述できます。

```
[14]: 1.1*bmi(174.0, 119.0 * 0.454)
```

```
[14]: 19.628947020742505
```

もう 1 つ関数を定義してみましょう。

```
[15]: def felt_air_temperature(temperature, humidity):
      return temperature - 1 / 2.3 * (temperature - 10) * (0.8 - humidity / 100)
```

この関数は、温度と湿度を入力として、体感温度を返します。このように、関数名や変数名には `_` (アンダースコア) を含めることができます。アンダースコアで始めることもできます。

数字も関数名や変数名に含めることができますが、名前の最初に来てはいけません。

```
[16]: felt_air_temperature(28, 50)
```

```
[16]: 25.652173913043477
```

なお、`return` の後に式を書かないと、何も返されなかったことを表現するために、「何もない」ことを表す `None` という特別な値が返ります。(`None` という値は色々なところで現れることでしょう。)

`return` 文に到達せずに関数定義本体の最後まで行ってしまったときも、`None` という値が返ります。

3.2.1 予約語

Python での `def` や `return` は、関数定義や `return` 文の始まりを記述するための特別な記号であり、それ以外の用途に用いることができません。このように構文上で役割が予約されている語は、**予約語**と呼ばれます。Code セルの構文ハイライトで (太字緑色などで) 強調されるものが予約語だと覚えておけば大体問題ありません。

3.2.2 練習 ft_to_cm

`f` フィート `i` インチをセンチメートルに変換する関数 `ft_to_cm(f, i)` を定義してください。ただし、1 フィート = 12 インチ = 30.48 cm としてよい。

```
[17]: def ft_to_cm(f, i):
      ...
```

定義ができれば、次のセルを実行して、エラーがでないことを確認してください。

```
[18]: assert round(ft_to_cm(5, 2) - 157.48, 6) == 0
      assert round(ft_to_cm(6, 5) - 195.58, 6) == 0
```

```
-----
TypeError                                Traceback (most recent call last)
Input In [18], in <module>
----> 1 assert round(ft_to_cm(5, 2) - 157.48, 6) == 0
      2 assert round(ft_to_cm(6, 5) - 195.58, 6) == 0

TypeError: unsupported operand type(s) for -: 'NoneType' and 'float'
```

3.2.3 練習 quadratic

二次関数 $f(x) = ax^2 + bx + c$ の値を求める `quadratic(a,b,c,x)` を定義してください。

```
[19]: def quadratic(a, b, c, x):
      ...
```

定義ができれば、次のセルを実行して、エラーがないことを確認してください。

```
[20]: assert quadratic(1, 2, 1, 3) == 16
      assert quadratic(1, -5, -2, 7) == 12

-----
AssertionError                                Traceback (most recent call last)
Input In [20], in <module>
----> 1 assert quadratic(1, 2, 1, 3) == 16
      2 assert quadratic(1, -5, -2, 7) == 12

AssertionError:
```

3.3 ローカル変数

次の関数は、ヘロンの公式によって、与えられた三辺の長さに対して三角形の面積を返すものです。

```
[21]: import math

def heron(a,b,c):
    s = 0.5*(a+b+c)
    return math.sqrt(s * (s-a) * (s-b) * (s-c))
```

`math.sqrt` を使うために `import math` を行っています。

次の式を評価してみましょう。

```
[22]: heron(3,4,5)

[22]: 6.0
```

この関数の中では、まず、3 辺の長さを足して 2 で割った (0.5 を掛けた) 値を求めています。そして、その値を `s` という変数に代入しています。この `s` という変数は、この関数の中で代入されているので、この関数の中だけで利用可能な変数となります。そのような変数を**ローカル変数**と呼びます。

そして、`s` を使った式が計算されて `return` 文で返されます。ここで、関数定義のひとまとまりの本体であることを表すために、`s` への代入文も `return` 文も、同じ深さでインデントされていることに注意してください。

Python では、関数の中で定義された変数は、その関数のローカル変数となります。関数の引数もローカル変数です。関数の外で同じ名前の変数を使っても、それは関数のローカル変数とは「別もの」と考えられます。

`heron` を呼び出した後で、関数の外で `s` の値を参照しても、以下のように、`s` が未定義という扱いになります。

```
[23]: s

-----
NameError                                Traceback (most recent call last)
Input In [23], in <module>
----> 1 s

(continues on next page)
```

(continued from previous page)

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

以下では、`heron` の中では、`s` というローカル変数の値は 3 になりますが、関数の外では、`s` という変数は別もので、その値はずっと `100` です。

```
[24]: s = 100
heron(3,4,5)
```

```
[24]: 6.0
```

```
[25]: s
```

```
[25]: 100
```

3.4 print

上の例で、ローカル変数は関数の返値を計算するのに使われますが、それが定義されている関数の外からは参照することができません。

ローカル変数の値など、関数の実行途中の状況を確認するには、`print` という Python が最初から用意してくれている関数（**組み込み関数**）を用いることができます。この `print` を関数内から呼び出すことでローカル変数の値を確認できます。

`print` は任意個の引数を取ることができ、コンマ , の区切りには空白文字が出力されます。引数を与えずに呼び出した場合には、改行のみを出力します。

```
[26]: def heron(a,b,c):
      s = 0.5*(a+b+c)
      print('The value of s is', s)
      return math.sqrt(s * (s-a) * (s-b) * (s-c))
```

```
[27]: heron(1,1,1)

The value of s is 1.5
```

```
[27]: 0.4330127018922193
```

このように `print` 関数を用いて変数の値を観察することは、プログラムの誤り（**バグ**）を見つけ、修正（**デバッグ**）する最も基本的な方法です。これは 1-4 でも改めて説明します。

なお、以降の説明では、`print` 関数を呼び出して値を出力することを「**印字する**」と表現します。

3.5 print と return

関数が値を返すことを期待されている場合は、必ず `return` を使ってください。

関数内で値を印字しても、関数の返値として利用することはできません。

たとえば `heron` を以下のように定義すると、`heron(1,1,1) * 2` のような計算ができなくなります。

```
def heron(a,b,c):
    s = 0.5*(a+b+c)
    print('The value of s is', s)
    print(math.sqrt(s * (s-a) * (s-b) * (s-c)))
```

なお、

```
return print(math.sqrt(s * (s-a) * (s-b) * (s-c)))
```

のように書いても駄目です。print 関数は None という値を返しますので、これでは関数は常に None という値を返してしまいます。

3.6 コメントと空行

コメントについては既に説明しましたが、関数定義にはコメントを付加して、後から読んでもわかるようにしましょう。

コメントだけの行は空行（空白のみから成る行）と同じに扱われます。

関数定義の中に空行を自由に入れることができますので、長い関数定義には、区切りとなるところに空行を入れるのがよいでしょう。

```
[28]: # heron の公式により三角形の面積を返す
def heron(a,b,c): # a,b,c は三辺の長さ

    # 辺の合計の半分を s に置く
    s = 0.5*(a+b+c)
    print('The value of s is', s)

    return math.sqrt(s * (s-a) * (s-b) * (s-c))
```

3.7 関数の参照の書き方

関数は、

関数 heron は、三角形の三辺の長さをもって三角形の面積を返します。

というように、名前だけで参照することもあります。

heron(a,b,c) は、三角形の三辺の長さ a, b, c をもって三角形の面積を返します。

というように、引数を明示して参照することもあります。

ときには、

heron() は三角形の面積を返します。

のように、関数名に () を付けて参照することがあります。この記法は、heron が関数であることを明示しています。

関数には引数がゼロ個のものがありますが、heron() と参照するとき、heron は必ずしも引数の数がゼロ個ではないことに注意してください。

後に学習するメソッドという関数の親戚に対しても同様の記法が用いられます。

3.7.1 練習 qe_disc qe_solution1 qe_solution1

二次方程式 $ax^2 + bx + c = 0$ に関して以下のような関数を定義してください。

1. 判別式 $b^2 - 4ac$ を求める qe_disc(a,b,c)
2. 解のうち、大きくない方を求める qe_solution1(a,b,c)
3. 解のうち、小さくない方を求める qe_solution2(a,b,c)

ただし、qe_solution1 と qe_solution2 は qe_disc を使って定義してください。二次方程式が実数解を持つと仮定してよいです。


```
[29]: import math

def qe_disc(a, b, c):
    ...

def qe_solution1(a, b, c):
    ...

def qe_solution2(a, b, c):
    ...
```

定義ができれば、次のセルを実行して、エラーがでないことを確認してください。

```
[30]: assert qe_disc(1, -2, 1) == 0
assert qe_disc(1, -5, 6) == 1
assert round(qe_solution1(1, -2, 1) - 1, 6) == 0
assert round(qe_solution2(1, -2, 1) - 1, 6) == 0
assert round(qe_solution1(1, -5, 6) - 2, 6) == 0
assert round(qe_solution2(1, -5, 6) - 3, 6) == 0

-----
AssertionError                                Traceback (most recent call last)
Input In [30], in <module>
----> 1 assert qe_disc(1, -2, 1) == 0
      2 assert qe_disc(1, -5, 6) == 1
      3 assert round(qe_solution1(1, -2, 1) - 1, 6) == 0

AssertionError:
```

3.8 ▲グローバル変数

Python では、関数の中で代入が行われない変数は、グローバル変数とみなされます。

グローバル変数とは、関数の外（トップレベルもしくはモジュールレベルと呼ばれます）で定義される変数のことです。

グローバル変数は、関数の中から参照することができます。

```
[31]: g = 9.8
```

```
[32]: def force(m):
      return m*g
```

以上のように `force` を定義すると、`force` の中で `g` というグローバル変数を参照することができます。

```
[33]: force(104)
```

```
[33]: 1019.2
```

```
[34]: g = g/6
```

以上のように、`g` の値を変更してから `force` を実行すると、変更後の値が用いられます。

```
[35]: force(104)
```

```
[35]: 169.86666666666667
```

以下はより簡単な例です。

```
[36]: a = 10
def foo():
    return a
def bar():
    a = 3
    return a
```

```
[37]: foo()
```

```
[37]: 10
```

```
[38]: bar()
```

```
[38]: 3
```

```
[39]: a
```

```
[39]: 10
```

```
[40]: a = 20
```

```
[41]: foo()
```

```
[41]: 20
```

bar の中では a への代入があるので、a はローカル変数になります。ローカル変数の a とグローバル変数の a は別ものと考えてください。ローカル変数 a への代入があっても、グローバル変数の a の値は変化しません。foo 中の a はグローバル変数です。

```
[42]: def boo(a):
    return a
```

```
[43]: boo(5)
```

```
[43]: 5
```

```
[44]: a
```

```
[44]: 20
```

関数の引数もローカル変数の一種と考えられ、グローバル変数とは別ものです。

3.9 練習の解答

```
[45]: def ft_to_cm(f, i):
    return 30.48*f + (30.48/12)*i
```

```
[46]: def quadratic(a, b, c, x):
    return a*x*x + b*x + c
```

```
[47]: import math

def qe_disc(a, b, c):
    return b*b - 4*a*c

def qe_solution1(a, b, c):
```

(continues on next page)

(continued from previous page)

```
    return (-b - math.sqrt(qe_disc(a, b, c))) / (2*a)

def qe_solution2(a, b, c):
    return (-b + math.sqrt(qe_disc(a, b, c))) / (2*a)
```

1-3. 論理・比較演算と条件分岐の基礎

論理・比較演算と条件分岐の基礎について説明します。

参考

- <https://docs.python.org/ja/3/tutorial/controlflow.html>
- https://docs.python.org/ja/3/reference/compound_stmts.html
- <https://docs.python.org/ja/3/library/stdtypes.html>

4.1 if 文による条件分岐

制御構造については第2回と第3回で本格的に扱いますが、ここでは if による条件分岐 (if 文) の基本的な形だけ紹介します。

```
[1]: def bmax(a,b):  
    if a > b:  
        return a  
    else:  
        return b
```

上の関数 `bmax` は、2つの引数の大きい方（正確には小さくない方）を返します。

ここで if による条件分岐が用いられています。

```
if a > b:  
    return a  
else:  
    return b
```

`a` が `b` より大きければ `a` が返され、そうでなければ、`b` が返されます。

ここで、`return a` が、`if` より右にインデントされていることに注意してください。`return a` は、`a > b` が成り立つときのみ実行されます。

`else` は `if` の右の条件が成り立たない場合を示しています。`else:` として、必ず `:` が付くことに注意してください。

また、`return b` も、`else` より右にインデントされていることに注意してください。 `if` と `else` は同じインデントになります。

```
[2]: bmax(3,5)
```

```
[2]: 5
```

関数の中で `return` と式が実行されると、関数は即座に返りますので、関数定義の中のその後の部分は実行されません。

たとえば、上の条件分岐は以下のように書くこともできます。

```
if a > b:
    return a
return b
```

ここでは、`if` から始まる条件分岐には `else:` の部分がありません。条件分岐の後に `return b` が続いています。（`if` と `return b` のインデントは同じです。）

`a > b` が成り立っていれば、`return a` が実行されて `a` の値が返ります。したがって、その次の `return b` は実行されません。

`a > b` が成り立っていなければ、`return a` は実行されません。これで条件分岐は終わりますので、その次にある `return b` が実行されます。

なお、Python では、`max` という関数があらかじめ定義されています。（すなわち、`max` は組み込み関数です。）

```
[3]: max(3,5)
```

```
[3]: 5
```

4.2 様々な条件

`if` の右などに来る条件として様々なものを書くことができます。これらの条件には `>` や `<` などの比較演算子が含まれています。

```
x < y      # x は y より小さい
x <= y     # x は y 以下
x > y      # x は y より大きい
x >= y     # x は y 以上
x == y     # x と y は等しい
x != y     # x と y は等しくない
```

特に等しいかどうかの比較には `==` という演算子が使われることに注意してください。 `=` は代入の演算子です。

`<=` は小さいか等しいか、`>=` は大きい等しいかを表します。 `!=` は等しくないことを表します。

さらに、このような基本的な条件を、`and` と `or` を用いて組み合わせることができます。

```
i >= 0 and j > 0  # i は 0 以上で、かつ、j は 0 より大きい
i < 0 or j > 0    # i は 0 より小さいか、または、j は 0 より大きい
```

`i` が 1 または 2 または 3 である、という条件は以下のようになります。

```
i == 1 or i == 2 or i == 3
```

これを `i == 1 or 2 or 3` と書くことはできませんので、注意してください。

また、`not` によって条件の否定をとることもできます。

```
not x < y          # x は y より小さくない (x は y 以上)
```

比較演算子は、以下のように連続して用いることもできます。

```
[4]: 1 < 2 < 3
```

```
[4]: True
```

```
[5]: 3 >= 2 < 5
```

```
[5]: True
```

4.2.1 練習 absolute

数値 x の絶対値を求める関数 `absolute(x)` を定義してください。Python には `abs` という絶対値を求める組み込み関数が用意されていますが、それを使わずに定義してください。

```
[6]: def absolute(x):
     ...
```

定義ができれば、次のセルを実行して、エラーがでないことを確認してください。

```
[7]: assert absolute(5) == 5
     assert absolute(-5) == 5
     assert absolute(0) == 0
```

```
-----
AssertionError                                Traceback (most recent call last)
Input In [7], in <module>
----> 1 assert absolute(5) == 5
      2 assert absolute(-5) == 5
      3 assert absolute(0) == 0

AssertionError:
```

4.2.2 練習 sign

x が正ならば 1、負ならば -1、ゼロならば 0 を返す関数 `sign(x)` を定義してください。

```
[8]: def sign(x):
     ...
```

定義ができれば、次のセルを実行して、エラーがでないことを確認してください。

```
[9]: assert sign(5) == 1
     assert sign(-5) == -1
     assert sign(0) == 0
```

```
-----
AssertionError                                Traceback (most recent call last)
Input In [9], in <module>
----> 1 assert sign(5) == 1
      2 assert sign(-5) == -1
      3 assert sign(0) == 0

AssertionError:
```

4.3 真理値を返す関数

ここで、真理値を返す関数について説明します。

Python が扱うデータには様々な種類があります。数については既に見て来ました。

真理値とは、True または False のどちらかの値のことです。これらは変数ではなく、**組み込み定数**であることに注意してください。

- True は、正しいこと（真）を表します。
- False は、間違ったこと（偽）を表します。

実は、if の後の条件の式は、True か False を値として持ちます。

```
[10]: x = 3
```

```
[11]: x > 1
```

```
[11]: True
```

上のように、x に 3 を代入しておくと、x > 1 という条件は成り立ちます。したがって、x > 1 という式の値は True になるのです。

```
[12]: x < 1
```

```
[12]: False
```

```
[13]: x%2 == 0
```

```
[13]: False
```

そして、真理値を返す関数を定義することができます。

```
[14]: def is_even(x):
      return x%2 == 0
```

この関数は、x を 2 で割った余りが 0 に等しいかどうかという条件の結果である真理値を返します。

x == y は、x と y が等しいかどうかという条件です。この関数は、この条件の結果である真理値を return によって返しています。

```
[15]: is_even(2)
```

```
[15]: True
```

```
[16]: is_even(3)
```

```
[16]: False
```

このような関数は、if の後に使うことができます。

```
[17]: def is_odd(x):
      if is_even(x):
          return False
      else:
          return True
```

このように、直接に True や False を返すこともできます。

```
[18]: is_odd(2)
```

```
[18]: False
```

```
[19]: is_odd(3)
```

```
[19]: True
```

4.4 オブジェクト

Python における値（式の評価結果）は全て**オブジェクト**と総称されます。変数の値もオブジェクトです。したがって、数や真値もオブジェクトです。今後、文字列やリストなど、様々な種類のデータが登場しますが、それらは全てオブジェクトです。

今後、オブジェクトという用語がところどころで出て来ますが、オブジェクトとデータは同義と思って差し支えありません。正確には、式の評価結果や変数の値となるデータがオブジェクトです。

4.5 None

None というデータがあります。

セルの中の式を評価した結果が **None** になると、何も表示されません。

```
[20]: None
```

`print` で無理やり表示させると以下ようになります。

```
[21]: print(None)
```

```
None
```

None という値は、特段の値が何もない、ということを表すために使われることがあります。

条件としては、None は偽と同様に扱われます。

```
[22]: if None:
      print('OK')
      else:
      print('NG')
```

```
NG
```

`return` の後に式を書かないことがあります。

```
return
```

この場合、以下のように **None** が指定されているとみなされます。

```
return None
```

このような `return` 文を実行すると、関数の実行はそこで終了して **None** が返ります。

4.6 ▲条件として使われる他の値

True と False の他に、他の種類のデータも、条件としても用いることができます。

たとえば:

- 数のうち、0 や 0.0 は偽、その他は真とみなされます。
- 文字列では、空文字列 '' のみ偽、その他は真とみなされます。(文字列については 2-1 を参照。)
- 組み込み定数 None は偽とみなされます。(None については上記参照。)

```
[23]: if 0:
        print('OK')
      else:
        print('NG')
```

NG

```
[24]: if -1.1:
        print('OK')
      else:
        print('NG')
```

OK

4.7 ▲再帰

一般に、定義しようとするもの自身を定義の中で参照することを、**再帰**と言います。再帰による定義を再帰的定義と言います。

たとえば、数列の漸化式は再帰的定義と考えられます。実際に、 n 番目のフィボナッチ数を $\text{fib}(n)$ とおくと、 $\text{fib}(n)$ は次のような漸化式を満たします。

```
fib(n) = n   ただし n<2
fib(n) = fib(n-1) + fib(n-2)   ただし n>=2
```

この漸化式を用いて以下のように実際にフィボナッチ数を計算することができます。

```
fib(0) = 0
fib(1) = 1
fib(2) = fib(1) + fib(0) = 1 + 0 = 1
fib(3) = fib(2) + fib(1) = 1 + 1 = 2
fib(4) = fib(3) + fib(2) = 2 + 1 = 3
fib(5) = fib(4) + fib(3) = 3 + 2 = 5
...
```

この漸化式から、以下のように $\text{fib}(n)$ の再帰的定義が得られます。

```
[25]: def fib(n):
        if n < 2:
            return n
        else:
            return fib(n-1) + fib(n-2)
```

実際に、以下のように $\text{fib}(n)$ の値が求まります。

```
[26]: fib(10)
```

[26]: 55

4.8 練習の解答

```
[27]: def absolute(x):  
      if x < 0:  
          return -x  
      else:  
          return x
```

```
[28]: def sign(x):  
      if x < 0:  
          return -1  
      if x > 0:  
          return 1  
      return 0
```

1-4. テストとデバッグ

テストとデバッグについて説明します。

参考

- <https://docs.python.org/ja/3/tutorial/errors.html>

5.1 仕様・テスト・デバッグ

プログラムを書くときに、実現しようとしている事柄を**仕様**と呼びます。

対象のプログラムが仕様に適合しているかを、実際にプログラムを動作させて検査することを、**テスト**と呼びます。テストの際に、テスト対象に与える入出力ペアのことを、**テストケース**と呼びます。

書いたプログラムが仕様に適合しているかは、一般に自明ではありません。テストによって、仕様に反したプログラムの振舞いが、しばしば浮き彫りになります。仕様に反したプログラムの振舞いの原因を、**バグ**と呼び、それを取り除くことを**デバッグ**と呼びます。

プログラミングでは、典型的には

- 仕様を分析する
- プログラムを書く
- テストする
- デバッグする

という4つの行いを、必要に応じて繰り返すことになります。

5.2 assert 文

テストとデバッグに有用なのが、**assert 文**です。これは、`assert` の次に書かれた条件式が真であるべきだと仕様を宣言する文です。偽であった場合は、`AssertionError` が発生してプログラムがそこで停止します。

与えられた引数を二乗する関数 `square` を用いた具体例を示します。

```
[1]: def square(x):
      return x*x

x = -2
assert square(x) >= 0
```

この `assert` 文では、仕様として条件式 `square(x) >= 0` を宣言しています。`square` 関数が「二乗する」という仕様に沿っているなら、その条件式は真であるべきです。そして、実際 `square` はその仕様に適合しているので、ここでは `assert` 文が実行されても何も起きません。

しかし、`square` にバグがあった場合は、話が変わります。

```
[2]: def square(x):
      return x+x # バグがある

x = -2
assert square(x) >= 0
```

```
-----
AssertionError                                Traceback (most recent call last)
Input In [2], in <module>
      2     return x+x # バグがある
      4 x = -2
----> 5 assert square(x) >= 0

AssertionError:
```

上のセルを実行すると、`AssertionError` が生じます。

このように、`assert` 文は、それが存在する場所で、満たされていなければならない前提条件を記述するために用います。`assert` 文で停止したら、記述された前提条件に関わる部分にバグがあることが判明します。

テストケースは、テスト対象が満たすべき仕様という側面があるので、`assert` 文はテストにも用いられます。

```
[3]: def square(x):
      return x*x

assert square(2) == 4
assert square(-2) == 4
assert square(0) == 0
```

上の例では、`square` に対する3つのテストケースについて、`assert` 文でテストしています。テストケースが満たされた（つまり `assert` 文で停止しなかった）からと言って、テスト対象の `square` が正しいとは言えませんが、仕様への適合度が高いことから、尤もらしいとは言えます。

5.3 エラーの分類

不正なプログラムからは、様々なエラーが生じます。

エラーには大きく分けて、構文エラー・実行時エラー・論理エラーの3つがあります。以下では、それぞれの意味と、典型例を示します。

5.3.1 構文エラー

構文エラー (syntax error) とは、プログラムコードが、Python の構文に違反しているときに生じるエラーです。

Python における構文エラーの典型例として、

- クォートや括弧の閉じ忘れ
- コロンの付け忘れ
- インデントの崩れ
- 全角スペースの利用
- == の代わりに = を使う
- 変数の代わりに文字列を使う (Cf. 2-1 文字列)

などが挙げられます。

[4]: `print('This is the error) # クォートの閉じ忘れ`

```
Input In [4]
  print('This is the error) # クォートの閉じ忘れ
                                ^
SyntaxError: EOL while scanning string literal
```

[5]: `def f() # コロンの付け忘れ`
 `return 1`

```
Input In [5]
  def f() # コロンの付け忘れ
      ^
SyntaxError: invalid syntax
```

[6]: `def f():`
`return 1 # インデントの崩れ`

```
Input In [6]
  return 1 # インデントの崩れ
  ^
IndentationError: expected an indented block
```

[7]: `1 + 1 # 全角スペースの利用`

```
Input In [7]
  1 + 1 # 全角スペースの利用
    ^
SyntaxError: invalid character in identifier
```

上の例を実行するとわかるように、構文エラーがあると `SyntaxError` や `IndentationError` などが発生します。それに付随するエラーメッセージが、構文エラーの具体的内容とおおよその位置を説明してくれます。

構文エラーに直面した際は、エラーメッセージをよく読んで、原因を推察しましょう。上の例が示すように、エラーメッセージの説明は、必ずしも分かり易くないですが、原因の位置を絞りこむには有用です。

Python では、構文エラーが実行時に発生しているように見えますが、実際には、実行しようとするプログラムコードの解釈に失敗することでエラーが生じています。つまり、構文エラーは、プログラムの実行によって生じるエラーではなく、実行できなかったことで生じるエラーです。

5.3.2 実行時エラー

実行時エラー（runtime error）とは、プログラムを実行した際に生じるエラー全般を指します。簡単に言えば、プログラムを異常停止させるエラーです。

実行時エラーが生じる典型的な状況として、

- 存在しない名前の利用（変数名・関数名・メソッド名の誤植）
- グローバル変数のつもりでローカル変数を参照（Cf. 3-3 関数）
- ゼロによる除算
- 辞書に登録されていないキーに対する値を取得（Cf. 3-1 辞書）
- 存在しないファイルの読み込み（Cf. 4-1 ファイル入出力）
- `assert` 文における条件の不成立

などが挙げられます。

[8]: `undefined_variable` # 未定義の変数の参照

```
-----
NameError                                Traceback (most recent call last)
Input In [8], in <module>
----> 1 undefined_variable

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

[9]: `x = 1`
`def f():`
 `x = x # グローバル変数のつもりでローカル変数を参照`
`f()`

```
-----
UnboundLocalError                        Traceback (most recent call last)
Input In [9], in <module>
      2 def f():
      3     x = x # グローバル変数のつもりでローカル変数を参照
----> 4 f()

Input In [9], in f()
      2 def f():
----> 3     x = x

UnboundLocalError: local variable 'x' referenced before assignment
```

[10]: `1/0` # ゼロによる除算

```
-----
ZeroDivisionError                                Traceback (most recent call last)
Input In [10], in <module>
----> 1 1/0

ZeroDivisionError: division by zero
```

[11]: {'a': 1}['b'] # 登録されていないキーに対する値を参照

```
-----
KeyError                                         Traceback (most recent call last)
Input In [11], in <module>
----> 1 {'a': 1}['b']

KeyError: 'b'
```

[12]: open('non-existent.txt', 'r') # 存在しないファイルの読み込み

```
-----
FileNotFoundError                               Traceback (most recent call last)
Input In [12], in <module>
----> 1 open('non-existent.txt', 'r')

FileNotFoundError: [Errno 2] No such file or directory: 'non-existent.txt'
```

実行時エラーについては、送出される例外名（上の例では `NameError`・`UnboundLocalError`・`ZeroDivisionError`・`KeyError`・`FileNotFoundError`）が自己説明的であり、それに付随するエラーメッセージも、大抵原因を分かり易く説明してくれます。

実行時エラーに直面した際は、発生した例外名とエラーメッセージをよく読んで、エラーに関連する言語機能（例えば辞書やファイル）の仕組みを改めて確認しましょう。

5.3.3 論理エラー

論理エラー（logic error）とは、プログラムを実行できるが、意図したように動作しないことを意味します。これは、プログラムから発生するエラーではなく、プログラムを書いた人のエラーです。

バグと呼ばれるものの多くは、論理エラーです。したがって、デバッグでは、プログラムを書いた人の意図と、プログラムの振舞いを比較検証することになります。

`assert` 文は、仕様違反という論理エラーを、`AssertionError` という実行時エラーに変換していると見做すことができます。

5.4 デバッグの具体例

デバッグの具体的なシナリオを説明します。次の関数 `median(x, y, z)` は、`x` と `y` と `z` の中央値（真ん中の値）を求めようとするものです。ただし、`x` と `y` と `z` は相異なる数であると仮定します。

```
[13]: def median(x, y, z):
      if x > y:
          x = y
          y = x
      if z < x:
          return x
      if z < y:
          return z
```

(continues on next page)

(continued from previous page)

```

    return y

assert median(3, 1, 2) == 2
-----
AssertionError                                Traceback (most recent call last)
Input In [13], in <module>
      8         return z
      9         return y
--> 11 assert median(3, 1, 2) == 2

AssertionError:

```

このように、この median は間違っています。

さて、median は、ローカル変数の $x \cdot y \cdot z$ のいずれかを返す関数です。これらの変数の値が期待通りの値であるか、print を入れて印字し、観察してみましょう。

```

[14]: def median(x, y, z):
        print(x, y, z)
        if x > y:
            x = y
            y = x
        print(x, y, z)
        if z < x:
            return x
        if z < y:
            return z
        return y

assert median(3, 1, 2) == 2

3 1 2
1 1 2

```

```

-----
AssertionError                                Traceback (most recent call last)
Input In [14], in <module>
     10         return z
     11         return y
--> 13 assert median(3, 1, 2) == 2

AssertionError:

```

関数の入口にある最初の print では、期待通りに実引数となる $3 \cdot 1 \cdot 2$ が、 $x \cdot y \cdot z$ に代入されています。しかし、2 番目の print では、3 が消えて 1 が複製されています。このことから、この 2 つの print の間にある if 文が疑わしいことが分かります。

問題の if 文は、x と y の値を入れ替える意図があるものでした。その意図を正しく反映すると、次のようになります。

```

[15]: def median(x, y, z):
        print(x, y, z)
        if x > y:
            w = x
            x = y
            y = w
        print(x, y, z)
        if z < x:

```

(continues on next page)

(continued from previous page)

```

    return x
    if z < y:
        return z
    return y

assert median(3, 1, 2) == 2

3 1 2
1 3 2

```

期待通りに動きました。最後に、デバッグ用に導入した `print` は、`median` の仕様には含まれないので、きちんと消しましょう。

```

[16]: def median(x, y, z):
        if x > y:
            w = x
            x = y
            y = w
        if z < x:
            return x
        if z < y:
            return z
        return y

assert median(3, 1, 2) == 2

```

5.5 コーディングスタイル

実は、生じたバグを取る対処法よりも、そもそもバグが生じにくくする予防法の方が大切です。Python において特に重要視されているのが、**コーディングスタイル**、つまりコードの書き方です。読みにくい（可読性の低い）コードだと、些細なミスが生じやすく、また見つけにくいからです。

Python では **PEP8**（**非公式日本語訳**）と呼ばれる公式のスタイルガイドがあります。PEP8 には様々な側面でスタイルに関する規則があり、コードの可読性を高めることが強く推奨されています。ここまでに扱った言語の要素について、たとえば、

- インデントは半角スペースを4つで1レベル
- `= += ==` などの演算子の前後に半角スペースを1つ入れる
- `*` と `+` の複合式では `+` の前後に半角スペースを1つ入れる（例：`2*x + y`）
- 関数の開き括弧の前にスペースを入れない
- `l i o` を変数名として使わない
- 真理値の比較に `==` や `is` を使わない

などが代表的です。

PEP8 に基づいたコーディングスタイルの自動検査器もあります（参照：[pycodestyle](#)）。オンラインサービスもいくつか利用できる（例：[PEP8 online](#)）、適宜活用してみましょう。

PEP8 には陽に言及されていないものの、プログラミング一般に重要なこともあります。たとえば、

- 自己説明的でない“マジックナンバー”ではなく記号的に意味がわかる変数を使う
- 不要なコードは削除する
- 1つの関数では1つのタスクだけを処理する

などは、可読性を上げる代表的なポイントです。

勘違いはバグを引き起こします。自らが勘違いしないコードを書くことが肝要です。

[]:

CHAPTER 6

2-1. 文字列 (string)

文書処理などに必要な文字列について説明します。

参考

- <https://docs.python.org/ja/3/tutorial/introduction.html#strings>

Python が扱うデータには様々な種類がありますが、**文字列**はいくつかの文字の並びから構成されるデータです。Python は標準で多言語に対応しており、英語アルファベットだけではなく日本語をはじめとする多くの言語を取り扱えます。

文字列は、文字の並びをシングルクォート `'...'`、もしくはダブルクォート `"..."` で囲んで記述します。

以下の例では文字列をそれぞれ、変数 `word1`, `word2` に代入しています。

```
[1]: word1 = 'hello'
      word1
```

```
[1]: 'hello'
```

```
[2]: word2 = 'Hello'
      word2
```

```
[2]: 'Hello'
```

上の変数の値が確かに文字列であることは、組み込み関数 `type` によって確認できます。`type` は、任意のデータを引数として、そのデータの種別を返します。データの種別は、**データ型**もしくは**型**と呼ばれます。

```
[3]: type(word1)
```

```
[3]: str
```

`str` は文字列のデータ型を意味します。

```
[4]: type(word2)
```

```
[4]: str
```

`str` は組み込み関数としても用いられます。組み込み関数 `str` を使えば、任意のデータを文字列に変換できます。一般に、データ型は、そのデータ型への変換を行う関数として用いられることが多いです。

1-1 で学んだ数値を文字列に変換したい場合、次のように行います。

```
[5]: word3 = str(123)
word3
```

```
[5]: '123'
```

逆に、'123' という文字列を整数に変換するには、1-1 で述べた `int` という関数を用いることができます。（実は `int` は整数の型でもあります。）

```
[6]: i = int('123')
i
```

```
[6]: 123
```

関数 `float` を用いれば文字列を実数に変換できます。

```
[7]: f = float('123.4')
f
```

```
[7]: 123.4
```

文字列の長さは、組み込み関数 `len` を用いて次のようにして求めます。

```
[8]: len(word1)
```

```
[8]: 5
```

複数行にわたる文字列を記述するには、トリプルクォート（`'''...'''` もしくは `"""..."""`）を用いることができます。上記の参考 URL を参照してください。トリプルクォートはコメントとしても用いられます。なお、1-4 のコーディングスタイルのところで紹介したスタイルガイドの PEP8 では、トリプルクォートには `"""..."""` を使うのが適切と定められています。

6.1 文字列とインデックス

文字列はいくつかの文字によって構成されています。

文字列 `'hello'` の 3 番目の文字を得たい場合は、以下のような記法を用います。

```
[9]: 'hello'[2]
```

```
[9]: 'l'
```

文字列を値とする変数に対しても同様の記法を用います。多くの場合は変数に対してこの記法を用います。

```
[10]: word1 = 'hello'
word1[2]
```

```
[10]: 'l'
```

この括弧内の数値のことを**インデックス**と呼びます。インデックスは `0` から始まるので、ある文字列の `x` 番目の要素を得るには、インデックスとして `x-1` を指定する必要があります。

こうして取得した文字は、Python では長さが 1 の文字列として扱われます。（プログラミング言語によっては、文字列ではなく別の型のデータとして扱われるものもありますので注意してください。）

文字列に対して、インデックスを指定してその要素を変更することはできません。（次のセルはエラーとなります）Python のデータは、大きく、変更可能なものと変更不可能なものに分類できますが、文字列は変更不可能なデータです。したがって、文字列を加工する場合は、新たに別の文字列を作成します。

```
[11]: word1 = 'hello'
word1[0] = 'H'
```

```

-----
TypeError                                Traceback (most recent call last)
Input In [11], in <module>
      1 word1 = 'hello'
----> 2 word1[0] = 'H'

TypeError: 'str' object does not support item assignment

```

また、文字列の長さ以上のインデックスを指定することはできません。(次はエラーとなります)

```
[12]: word1[100]
```

```

-----
IndexError                                Traceback (most recent call last)
Input In [12], in <module>
----> 1 word1[100]

IndexError: string index out of range

```

インデックスに負数を指定すると、文字列を後ろから数えた順序に従って文字列を構成する文字を得ます。たとえば、文字列の最後の文字を取得するには、-1 を指定します。

```
[13]: word1[-1]
```

```
[13]: 'o'
```

まとめると文字列 `hello` の正負のインデックスは以下の表の関係になります。

インデックス	h	e	l	l	o
0 か 正	0	1	2	3	4
負	-5	-4	-3	-2	-1

6.2 文字列とスライス

スライスと呼ばれる機能を利用して、文字列の一部（部分文字列）を取得できます。

具体的には、取得したい部分文字列の先頭の文字のインデックスと最後の文字のインデックスに 1 を加えた値を指定します。たとえば、ある文字列の 2 番目の文字から 4 番目までの文字の部分文字列を得るには次のようにします。

```
[14]: digits1='0123456789'
      digits1[1:4]
```

```
[14]: '123'
```

文字列の先頭（すなわち、インデックスが 0 の文字）を指定する場合、次のように行えます。

```
[15]: digits1[0:3]
```

```
[15]: '012'
```

しかし、最初の 0 は省略しても同じ結果となります。

```
[16]: digits1[:3]
```

```
[16]: '012'
```

同様に、最後尾の文字のインデックスも、値を省略することもできます。

```
[17]: digits1[3:]
```

```
[17]: '3456789'
```

```
[18]: digits1[3:5]
```

```
[18]: '34'
```

スライスにおいても負数を指定して、文字列の最後の方から部分文字列を取得できます。

```
[19]: digits1[-4:-1]
```

```
[19]: '678'
```

スライスでは3番目の値を指定することで、とびとびの文字を指定できます。次のように `digits1[3:9:2]` と指定すると、インデックス 3 から 2 文字おきにインデックス 9 より小さい文字を並べた部分文字列を得ます。

```
[20]: digits1[3:9:2]
```

```
[20]: '357'
```

3 番目の値に `-1` を指定することもできます。これを使えば元の文字列の逆向きの文字列を得ることができます。

```
[21]: digits1[8:4:-1]
```

```
[21]: '8765'
```

6.3 空文字列

シングルクォート（もしくはダブルクォート）で、何も囲まない場合、長さ 0 の文字列（**空文字列**（くうもじれつ）もしくは、**空列**（くうれつ））となります。具体的には、下記のように使用します。

```
blank = ''
```

空文字列は、次のように、たとえば文字列中からある部分文字列を取り除くのに使用します。（`replace` は後で説明します。）

```
[22]: price = '2,980 円'
```

```
price.replace(',', '')
```

```
[22]: '2980 円'
```

文字列のスライスにおいて、指定したインデックスの範囲に文字列が存在しない場合、たとえば、最初に指定したインデックス `x` に対して、2 番目のインデックスの値に `x` 以下のインデックスの値を指定するとどうなるでしょうか？（ただし、2 つのインデックスは同じ符号を持つとし、スライスの 3 番目の値は用いないとします。）このような場合、結果は次のように空文字列となります（エラーが出たり、結果が `None` にはならないことに注意してください）。

```
[23]: digits1='0123456789'
```

```
print(' 空文字列 1 = ', digits1[4:2])
```

```
print(' 空文字列 2 = ', digits1[-1:-4])
```

```
print(' 空文字列 3 = ', digits1[3:3])
```

```
print(' 空文字列ではない = ', digits1[3:-1])
```

```
空文字列 1 =
```

```
空文字列 2 =
```

```
空文字列 3 =
```

```
空文字列ではない = 345678
```

6.4 文字列の検索

文字列 A が 文字列 B を含むかどうかを調べるには、`in` 演算子を使います。具体的には、次のように使用します。

```
文字列 B in 文字列 A
```

調べたい 文字列 B が含まれていれば `True` が、そうでなければ `False` が返ります。

```
[24]: 'lo' in 'hello'
```

```
[24]: True
```

```
[25]: 'z' in 'hello'
```

```
[25]: False
```

実際のプログラムでは文字列を値とする変数を用いることが多いでしょう。

```
[26]: word1 = 'hello'
      substr1 = 'lo'
      substr1 in word1
```

```
[26]: True
```

```
[27]: substr2 = 'z'
      substr2 in word1
```

```
[27]: False
```

`not in` 演算子は、`in` 演算子の逆を意味します。

```
[28]: word1 = 'hello'
      substr2 = 'z'
      substr2 not in word1
```

```
[28]: True
```

6.5 ▲エスケープシーケンス

文字列を作成するにはシングル `'` あるいはダブルクォート `"` で囲むと説明しました。これらの文字を含む文字列を作成するには、**エスケープシーケンス**と呼ばれる特殊な文字列を使う必要があります。

たとえば、下のように文字列に `'` を含む文字列を `'` で囲むと文字列の範囲がずれてエラーとなります。

```
[29]: non_escaped = 'This is 'MINE''
      non_escaped
```

```
Input In [29]
  non_escaped = 'This is 'MINE''
                  ^
SyntaxError: invalid syntax
```

エラーを避けるには、エスケープシーケンスで `'` を記述します、具体的には `'` の前に `\` と記述すると、`'` を含む文字列を作成できます。

```
[30]: escaped1 = 'This is \'MINE\''
      escaped1
```

```
[30]: "This is 'MINE'"
```

実は、シングルクォートで囲む代わりにダブルクォートを使えばエスケープシーケンスを使わずに記述できます。

```
[31]: doublequoted = "This is 'MINE'"
doublequoted
```

```
[31]: "This is 'MINE'"
```

他にも、ダブルクォートを表す `\"`、`\` を表す `\\`、改行を表す `\n` など、様々なエスケープシーケンスがあります。

```
[32]: escaped2 = "時は金なり\n\"Time is money\"\nTime is \\"
print(escaped2)
```

```
時は金なり
"Time is money"
Time is \
```

3 連のシングルクォート、もしくはダブルクォートを利用すれば、`\` や `\n` を使わずに記述できます。

```
[33]: triple_single_quated = '''時は金なり
'Time is money'
Time is \\'
print(triple_single_quated)
```

```
時は金なり
'Time is money'
Time is \
```

```
[34]: triple_double_quated = """時は金なり
'Time is money'
Time is \\'
print(triple_double_quated)
```

```
時は金なり
'Time is money'
Time is \
```

なお、プログラムの一部を無効に（コメントアウト）したいとき、3 連のクォートで囲んで文字列にしてしまうことがあります。

6.6 バックスラッシュの表示と入力

エスケープシーケンスの先頭にある文字は、バックスラッシュ `\` (Unicode U+005C) です。これは Python に限った話ではないですが、バックスラッシュは環境（正確にはフォント）によって見え方が異なります。Windows 上のフォントでは、円記号 `¥` として見えることが多いです。macOS 上のフォントでは、そのままバックスラッシュとして見えることが多いです。

JIS 配列キーボードでは、バックスラッシュキーがないことがあります。Windows 上では、円記号 `¥` キーでバックスラッシュが入力できます。macOS 上では、`Alt + ¥` キーでバックスラッシュが入力できます。ただし、IME 設定によっても入力方法は変わるので注意してください。

```
[35]: print('\n') # 改行文字 (バックスラッシュ + n)
print('¥n') # 改行文字でない (円記号 + n)
print('^^e2^^a7^^b5n') # 改行文字でない (Unicode U+29F5 のバックスラッシュ演算子 + n)
```



```
¥n
^^e2^^a7^^b5n
```

6.7 文字列の連結

+ 演算子を用いれば文字列同士を**連結**できます。この演算では新しい文字列が作られ、元の文字列は変化しません。

```
[36]: word1 = 'hello'
      word2 = ' world'
      text1 = word1 + word2
      text1
```

```
[36]: 'hello world'
```

* 演算子で文字列の繰り返し回数を指定できます。

```
[37]: word1 = 'hello'
      word1 * 3
```

```
[37]: 'hellohellohello'
```

6.8 文字列とメソッド

文字列に対する操作を行うため、様々な**メソッド**（関数のようなもの）が用意されています。

メソッドは必要に応じて (...) 内に引数を与え、以下のように使用します。

```
文字列. メソッド名 (式, ...)
# あるいは
文字列変数. メソッド名 (式, ...)
```

文字列には以下のようなメソッドが用意されています。

6.8.1 置換

replace メソッドは、指定した 部分文字列 **A** を、別に指定した 文字列 **B** で置き換えた文字列を作成します。この操作では、元の文字列は変化しません。具体的には、次のように使用します。

```
文字列.replace(部分文字列 A, 文字列 B)
```

```
[38]: word1 = 'hello'
      word1.replace('l', '123')
```

```
[38]: 'he123123o'
```

```
[39]: word1
```

```
[39]: 'hello'
```

6.8.2 練習

英語の文章からなる文字列 `str_engsentences` が引数として与えられたとき、`str_engsentences` 中に含まれる全ての句読点 (., ,, :, ;, !, ?) を削除した文字列を返す関数 `remove_punctuations` を作成してください。（練習の解答はこのノートブックの一番最後にあります。）

次のセルの ... のところを書き換えて `remove_punctuations(str_engsentences)` を作成してください。

```
[40]: def remove_punctuations(str_engsentences):
      ...
```

上のセルで解答を作成した後、以下のセルを実行し、実行結果が `True` になることを確認してください。

```
[41]: print(remove_punctuations('Quiet, uh, donations, you want me to make a donation to,
↳ the coast guard youth auxiliary?') == 'Quiet uh donations you want me to make a
↳ donation to the coast guard youth auxiliary')

False
```

6.8.3 練習

ATGC の 4 種類の文字から成る文字列 `str_atgc` が引数として与えられたとき、文字列 `str_pair` を返す関数 `atgc_bppair` を作成してください。ただし、`str_pair` は、`str_atgc` 中の各文字列に対して、`A` を `T` に、`T` を `A` に、`G` を `C` に、`C` を `G` に置き換えたものです。

次のセルの ... のところを書き換えて `atgc_bppair(str_atgc)` を作成してください。

```
[42]: def atgc_bppair(str_atgc):
      ...
```

上のセルで解答を作成した後、以下のセルを実行し、実行結果が `True` になることを確認してください。

```
[43]: print(atgc_bppair('AAGCCCCATGGTAA') == 'TTCGGGTACCATT')

False
```

6.8.4 検索

`index` メソッドにより、指定した部分文字列 `B` が文字列 `A` のどこに存在するか調べることができます。具体的には、次のように使用します。

```
文字列 A.index(部分文字列 B)
```

ただし、指定した部分文字列が文字列に複数回含まれる場合、最初のインデックスが返されます。また、指定した部分文字列が文字列に含まれない場合は、エラーとなります。

```
[44]: word1 = 'hello'
      word1.index('lo')
```

```
[44]: 3
```

```
[45]: word1.index('l')
```

```
[45]: 2
```

以下はエラーとなります。

```
[46]: word1.index('a')
```

```
-----
ValueError                                Traceback (most recent call last)
Input In [46], in <module>
----> 1 word1.index('a')

ValueError: substring not found
```

find メソッドも index と同様に部分文字列を検索し、最初に出現するインデックスを返します。
index との違いは、部分文字列が含まれない場合エラーとはならず -1 が返されることです。

```
[47]: word1 = 'hello'
      word1.find('a')
```

```
[47]: -1
```

6.8.5 練習

コロン (:) を 1 つだけ含む文字列 `str1` を引数として与えると、コロンの左右に存在する文字列を入れ替えた文字列を返す関数 `swap_colon(str1)` を作成してください。

次のセルの ... のところを書き換えて `swap_colon(str1)` を作成してください。

```
[48]: def swap_colon(str1):
      ...
```

上のセルで解答を作成した後、以下のセルを実行し、実行結果が `True` になることを確認してください。

```
[49]: print(swap_colon('hello:world') == 'world:hello')

False
```

6.8.6 数え上げ

count メソッドにより、指定した 部分文字列 B が 文字列 A にいくつ存在するか調べることができます。

文字列 A.count(部分文字列 B)

```
[50]: word1 = 'hello'
      word1.count('l')
```

```
[50]: 2
```

```
[51]: 'aaaaaaa'.count('aa')
```

```
[51]: 3
```

6.8.7 練習

ATGC の 4 種類の文字から成る文字列 `str_atgc` と塩基名 (A, T, G, C のいずれか) を指定する文字列 `str_bpname` が引数として与えられたとき、`str_atgc` 中に含まれる塩基 `str_bpname` の数を返す関数 `atgc_count` を作成してください。

次のセルの ... のところを書き換えて `atgc_count(str_atgc, str_bpname)` を作成してください。

```
[52]: def atgc_count(str_atgc, str_bpname):
      ...
```

上のセルで解答を作成した後、以下のセルを実行し、実行結果が `True` になることを確認してください。

```
[53]: print(atgc_count('AAGCCCCATGGTAA', 'A') == 5)
False
```

6.8.8 大文字・小文字

`lower`, `capitalize`, `upper` メソッドを用いると、文字列の中の英文字を小文字に変換したり、大文字に変換したりすることができます。

これらの操作では、元の文字列は変化しません。

```
[54]: upper_dna = 'DNA'
      upper_dna.lower() # 全ての文字を小文字にする
```

```
[54]: 'dna'
```

```
[55]: upper_dna
```

```
[55]: 'DNA'
```

```
[56]: lower_text = 'hello world!'
      lower_text.capitalize() # 先頭文字を大文字にする
```

```
[56]: 'Hello world!'
```

```
[57]: lower_text
```

```
[57]: 'hello world!'
```

```
[58]: lower_text.upper() #全ての文字を大文字にする
```

```
[58]: 'HELLO WORLD!'
```

```
[59]: lower_text
```

```
[59]: 'hello world!'
```

6.8.9 ▲空白文字の削除

半角スペース ' '・改行文字 '\n'・タブ文字 '\t'・全角スペース '　'などを総称して**空白文字**と呼びます。

`strip` メソッドを用いると、文字列の前後にある連続した空白文字を削除した文字列を取得できます。

```
[60]: ' abc\n'.strip()
```

```
[60]: 'abc'
```

```
[61]: ' a b c \n'.strip()
```

```
[61]: 'a b c'
```

左側の空白だけ削除する `lstrip` と右側の空白だけ削除する `rstrip` もあります。

```
[62]: ' abc\n'.lstrip()
```

```
[62]: 'abc\n'
```

```
[63]: ' abc\n'.rstrip()
```

```
[63]: ' abc'
```

6.9 文字列の比較演算

比較演算子、`==`, `<`, `>` などを用いて、2つの文字列を比較することもできます。

```
[64]: print('abc' == 'abc')
      print('ab' == 'abc')
```

```
True
False
```

```
[65]: print('abc' != 'abc')
      print('ab' != 'abc')
```

```
False
True
```

文字列の大小の比較は、いわゆる辞書式による比較で、文字列の最初の文字から順に比較して大小を決めます。片方がもう片方を拡張したものであれば、拡張した方を大きいとします。

```
[66]: print('abc' <= 'abc')
      print('abc' < 'abc')
      print('abc' < 'abd')
      print('ab' < 'abc')
```

```
True
False
True
True
```

6.10 練習

英語の文字列 `str_engsentences` が引数として与えられたとき、それが全て小文字である場合、`True` を返し、そうでない場合、`False` を返す関数 `check_lower` を作成してください。

次のセルの ... のところを書き換えて `check_lower(str_engsentences)` を作成してください。

```
[67]: def check_lower(str_engsentences):
      ...
```

上のセルで解答を作成した後、以下のセルを実行し、実行結果が全て `True` になることを確認してください。

```
[68]: print(check_lower('down down down') == True)
      print(check_lower('There were doors all round the hall, but they were all locked') == False)
```

```
False
False
```

6.11 初心者によくある誤解 — 変数と文字列の混乱

初心者によくある誤解として、変数と文字列を混乱する例が見られます。たとえば、文字列を引数に取る次のような関数 `func` を考えます（`func` は引数として与えられた文字列を大文字にして返す関数です）。

```
[69]: def func(str1):
      return str1.upper()
```

ここで変数 `str2` を引数として `func` を呼ぶと、`str2` に格納されている文字列が大文字になって返ってきます。

```
[70]: str2 = 'abc'
      func(str2)
```

```
[70]: 'ABC'
```

次のように `func` を呼ぶと上とは結果が異なります。次の例では変数 `str2`（に格納されている文字列 `abc`）ではなく、文字列 `'str2'` を引数として `func` を呼び出しています。

```
[71]: str2 = 'abc'
      func('str2')
```

```
[71]: 'STR2'
```

6.12 練習

コンマ (,) を含む英語の文章からなる文字列 `str_engsentences` が引数として与えられたとき、`str_engsentences` 中の一番最初のコンマより後の文章のみかならなる文字列 `str_res` を返す関数 `remove_clause` を作成してください。ただし、`str_res` の先頭は大文字のアルファベットとしてください。

次のセルの ... のところを書き換えて `remove_clause(str_engsentences)` を作成してください。

```
[72]: def remove_clause(str_engsentences):
      ...
```

上のセルで解答を作成した後、以下のセルを実行し、実行結果が `True` になることを確認してください。

```
[73]: print(remove_clause("It's being seen, but you aren't observing.") == "But you aren't_
      ↪observing.")
```

```
False
```

6.13 練習の解答

```
[74]: def remove_punctuations(str_engsentences):
      str1 = str_engsentences.replace('.', '') # 指定の文字を空文字に置換
      str1 = str1.replace(',', '')
      str1 = str1.replace(':', '')
      str1 = str1.replace('; ', '')
      str1 = str1.replace('!', '')
      str1 = str1.replace('?', '')
      return str1
      #remove_punctuations('Quiet, uh, donations, you want me to make a donation to the_
      ↪coast guard youth auxiliary?')
```

```
[75]: def atgc_bppair(str_atgc):
    str_pair = str_atgc.replace('A', 't') # 指定の文字に置換。ただし全て小文字
    str_pair = str_pair.replace('T', 'a')
    str_pair = str_pair.replace('G', 'c')
    str_pair = str_pair.replace('C', 'g')
    str_pair = str_pair.upper() # 置換済みの小文字の列を大文字に変換
    return str_pair
#atgc_bppair('AAGCCCCATGGTAA')
```

```
[76]: def swap_colon(str1):
    #コロンの位置を取得する # find でも OK
    col_index = str1.index(':')
    #コロンの位置を基準に前半と後半の部分文字列を取得する
    str2, str3 = str1[:col_index], str1[col_index+1:]
    #部分文字列の順序を入れ替えて結合する
    str4 = str3 + ':' + str2
    return str4
#swap_colon('hello:world')
```

```
[77]: def atgc_count(str_atgc, str_bpname):
    return str_atgc.count(str_bpname)
#atgc_count('AAGCCCCATGGTAA', 'A')
```

```
[78]: def check_lower(str_engsentences):
    if str_engsentences == str_engsentences.lower(): #元の文字列と小文字に変換した文字列を
        比較する
        return True
    return False
#check_lower('down down down')
#check_lower('There were doors all round the hall, but they were all locked')
```

```
[79]: def remove_clause(str_engsentences):
    int_index = str_engsentences.find(',')
    str1 = str_engsentences[int_index+2:]
    return str1.capitalize()
#remove_clause("It's being seen, but you aren't observing.")
```

```
[ ]:
```

2-2. リスト (list)

複数のデータを要素としてまとめて取り扱うデータとして、リストとタプルについて説明します。

参考

- <https://docs.python.org/ja/3/tutorial/introduction.html#lists>
- <https://docs.python.org/ja/3/tutorial/datastructures.html#tuples-and-sequences>

文字列を構成する要素は文字のみでしたが、**リスト**では構成する要素としてあらゆる型のデータを指定できます。他のプログラミング言語では、リストに相当するものとして**配列**（もしくは**アレイ**）や**ベクター**などがあります。

リストを作成するには、リストを構成する要素をコンマで区切り全体をかぎ括弧 [...] で囲みます。

以下のセルでは数値を要素とするリストを作成して、変数に代入しています。さらに、文字列と同様に組み込み関数 `type` を用いて、変数の値がリストであることを確認しています。

```
[1]: numbers = [0, 10, 20, 30, 40, 50]
      numbers
```

```
[1]: [0, 10, 20, 30, 40, 50]
```

```
[2]: type(numbers)
```

```
[2]: list
```

リストのデータ型は `list` です。（なお、後で見るように、`list` は他のデータをリストに変換する関数としても用いられます。）

次に文字列を構成要素とするリストを作成してみます。

```
[3]: fruits = ['apple', 'banana', 'chelly']
      fruits
```

```
[3]: ['apple', 'banana', 'chelly']
```

リストの要素としてあらゆる型のデータを指定でき、それらは混在してもかまいません。以下のセルでは、数値と文字列が混在しています。

```
[4]: numbers_fruits = [10, 'apple', 20, 'banana', 30]
      numbers_fruits
```



```
[4]: [10, 'apple', 20, 'banana', 30]
```

次のように、何も要素を格納していないリスト（空リスト）を作成できます。空リストはプログラム実行の途中結果を記録する場合などによく使われています。具体的な例については、後述する `append` メソッドの項を参照してください。

```
[5]: empty=[]
     empty
```

```
[5]: []
```

なお、`[]` を用いて空リストを作成するたびに、常に新しいオブジェクト（それまでに作られたオブジェクトとは同一でないオブジェクト）が生成されます。詳しくは「▲オブジェクトの等価性と同一性」を参照してください。

7.1 リストとインデックス

文字列の場合と同様に、インデックスを指定してリストの要素を取り出せます。リストの `x` 番目の要素を取得するには次のような記法を用います。インデックスは 0 から始まることに注意してください。

```
リスト [x-1]
```

```
[6]: abcd = ['a', 'b', 'c', 'd']
     abcd[2]
```

```
[6]: 'c'
```

文字列の場合とは異なり、リストは変更可能なデータです。すなわちインデックスで指定されるリストの要素は、代入によって変更できます。

```
[7]: abcd = ['a', 'b', 'c', 'd']
     abcd[2] = 'hello'
     abcd
```

```
[7]: ['a', 'b', 'hello', 'd']
```

文字列と同様に、スライスを使った範囲指定も可能です。

```
[8]: abcd = ['a', 'b', 'c', 'd']
     abcd[1:3]
```

```
[8]: ['b', 'c']
```

```
[9]: abcd = ['a', 'b', 'c', 'd']
     abcd[0:4:2]
```

```
[9]: ['a', 'c']
```

リストのスライスに対しては、代入も可能です。

```
[10]: abcd = ['a', 'b', 'c', 'd']
     abcd[1:3] = ['x', 'y', 'z']
     abcd
```

```
[10]: ['a', 'x', 'y', 'z', 'd']
```

7.2 練習

リスト `ln` を引数として取り、`ln` の偶数番目のインデックスの値を削除したリストを返す関数 `remove_evenindex` を作成してください（ただし、0 は偶数として扱うものとします）。

ヒント：スライスを使います。

以下のセルの ... のところを書き換えて `remove_evenindex(ln)` を作成してください。

```
[11]: def remove_evenindex(ln):
      ...
```

上のセルで解答を作成した後、以下のセルを実行し、実行結果が全て `True` になることを確認してください。

```
[12]: print(remove_evenindex(['a', 'b', 'c', 'd', 'e', 'f', 'g']) == ['b', 'd', 'f'])
      print(remove_evenindex([1, 2, 3, 4, 5]) == [2, 4])

False
False
```

7.3 多重リスト

リストの要素としてリストを指定することもできます。リストを要素とするリストは多重リストと呼ばれます。次は二重リストの例です。

```
[13]: lns = [[1, 2, 3], [10, 20, 30], ['a', 'b', 'c']]
```

多重リストの要素指定は複数のインデックスで行います。前の例で外側の `[]` で示されるリストの 2 番目の要素のリスト、すなわち `[10, 20, 30]` の最初の要素は次のように指定します。

```
[14]: lns[1][0]
```

```
[14]: 10
```

3 番目のリストそのものを取り出したいときは、次のように指定します。

```
[15]: lns[2]
```

```
[15]: ['a', 'b', 'c']
```

以下のようにリストの要素として、リストを値とする変数を指定することもできます。

```
[16]: lns2 = [lns, ['x', 1, [11, 12, 13]], ['y', [100, 120, 140]] ]
      lns2
```

```
[16]: [[1, 2, 3], [10, 20, 30], ['a', 'b', 'c']],
      ['x', 1, [11, 12, 13]],
      ['y', [100, 120, 140]]]
```

```
[17]: lns2[0]
```

```
[17]: [[1, 2, 3], [10, 20, 30], ['a', 'b', 'c']]
```

7.4 リストに対する関数・演算子・メソッド

7.4.1 リストの要素数

組み込み関数 `len` はリストの長さ、すなわち要素数、を返します。

```
[18]: numbers = [0, 10, 20, 30, 40, 50]
      len(numbers)
[18]: 6
```

7.4.2 `max` と `min`

リストを引数とする関数は色々あります。関数 `max` は、数のリストが与えられると、その中の最大値を返します。同様に関数 `min` はリストの中の最小値を返します。

```
[19]: numbers = [30, 50, 10, 20, 40, 60]
      max(numbers)
[19]: 60
```

```
[20]: numbers = [30, 50, 10, 20, 40, 60]
      min(numbers)
[20]: 10
```

`max` と `min` は文字列のリストに対しても適用できます。文字列の比較は、いわゆる辞書順で行われます。

```
[21]: characters = ['e', 'd', 'a', 'c', 'f', 'b']
      min(characters)
[21]: 'a'
```

7.4.3 `sum`

関数 `sum` は、数のリストが与えられると、その要素の総和を返します。

```
[22]: numbers = [30, 50, 10, 20, 40, 60]
      sum(numbers)
[22]: 210
```

```
[23]: sum([])
[23]: 0
```

7.4.4 リストと演算子

演算子 `+` によってリストの連結、`*` によって連結における繰り返し回数を指定することができます。

```
[24]: numbers = [0, 10, 20, 30, 40, 50]
      numbers + ['a', 'b', 'c']
[24]: [0, 10, 20, 30, 40, 50, 'a', 'b', 'c']
```

```
[25]: numbers*3
```

```
[25]: [0, 10, 20, 30, 40, 50, 0, 10, 20, 30, 40, 50, 0, 10, 20, 30, 40, 50]
```

要素が全て同じ値（たとえば、0）のリストを作る最も簡単な方法は、この * 演算子を使う方法です。

```
[26]: zero10 = [0] * 10
zero10
```

```
[26]: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

* は + を繰り返して行うのと同じ結果を返します。たとえば、`x*3` は `x+x+x` と同じ結果を返します。
`x` が多重リストのとき、`x` の要素であるリストが `y` の中に複数回現れます。

```
[27]: x = [[0, 1], [2, 3]]
y = x*3
y
```

```
[27]: [[0, 1], [2, 3], [0, 1], [2, 3], [0, 1], [2, 3]]
```

このとき、`x` の要素が変更されると、`y` の中では複数箇所に変化が起こるので、注意してください。

```
[28]: x[0][0] = 99
y
```

```
[28]: [[99, 1], [2, 3], [99, 1], [2, 3], [99, 1], [2, 3]]
```

これは、`y` の中の複数箇所にあるオブジェクトが同一だからです。詳しくは、以下の「▲オブジェクトの等価性と同一性」を参照してください。

演算子 `in` は、左辺の要素がリストに含まれれば `True` を、それ以外では `False` を返します。

```
[29]: 10 in numbers
```

```
[29]: True
```

リストに対する `in` 演算子は、論理演算 `or` を簡潔に記述するのに用いることもできます。たとえば、

```
a1 == 1 or a1 == 3 or a1 == 7:
```

は

```
a1 in [1, 3, 7]:
```

と同じ結果を得られます。`or` の数が多くなる場合は、`in` を用いた方がより読みやすいプログラムを書くことができます。

```
[30]: a1 = 1
print(a1 == 1 or a1 == 3 or a1 == 7, a1 in [1, 3, 7])
a1 = 3
print(a1 == 1 or a1 == 3 or a1 == 7, a1 in [1, 3, 7])
a1 = 5
print(a1 == 1 or a1 == 3 or a1 == 7, a1 in [1, 3, 7])
```

```
True True
True True
False False
```

`in` 演算子は、左辺の要素がリストに含まれるかどうかを、リストの要素を最初から順に調べることで判定しています。したがって、リストの長さに比例した時間がかかります。つまり、リストの長さが大きければ、それなりの時間がかかることに注意してください。

`not in` 演算子は、`in` 演算子の逆を意味します。

```
[31]: 10 not in numbers
```

```
[31]: False
```

```
[32]: 11 not in numbers
```

```
[32]: True
```

7.4.5 指定した要素のインデックス取得と数えあげ

`index` メソッドは引数で指定した要素のインデックスの番号を返します。文字列には `index` に加えてこれと似た `find` メソッドもありましたが、リストでは使えません。

```
[33]: numbers = [0, 10, 20, 30, 40, 50]
      numbers.index(20)
```

```
[33]: 2
```

`count` メソッドは指定した要素の数を返します。

```
[34]: all20 = [20]*3
      all20.count(20) # 指定した要素のリスト内の数
```

```
[34]: 3
```

7.4.6 並べ替え (sort メソッド)

`sort` メソッドはリスト内の要素を並べ替えます。引数に何も指定しなければ昇順でとなります。

```
[35]: numbers = [30, 50, 10, 20, 40, 60]
      numbers.sort()
```

```
[36]: numbers
```

```
[36]: [10, 20, 30, 40, 50, 60]
```

```
[37]: characters = ['e', 'd', 'a', 'c', 'f', 'b']
      characters.sort()
      characters
```

```
[37]: ['a', 'b', 'c', 'd', 'e', 'f']
```

`reverse = True` オプションを指定すれば、要素を降順に並べ替えることもできます。（これは3.3のキーワード引数と呼ばれるものですが、ここでは天下りの、並べ替えの方法を指定する情報と理解しておいてください。）

```
[38]: numbers = [30, 50, 10, 20, 40, 60]
      numbers.sort(reverse = True)
      numbers
```

```
[38]: [60, 50, 40, 30, 20, 10]
```

7.4.7 並べ替え (sorted 組み込み関数)

関数 `sorted` ではリストを引数に取って、そのリスト内の要素を昇順に並べ替えた結果をリストとして返します。

```
sorted(リスト)
```

```
[39]: numbers = [30, 50, 10, 20, 40, 60]
      sorted(numbers)
```

```
[39]: [10, 20, 30, 40, 50, 60]
```

文字列の比較は、いわゆる辞書順で行われます。

```
[40]: characters = ['e', 'd', 'a', 'c', 'f', 'b']
      sorted(characters)
```

```
[40]: ['a', 'b', 'c', 'd', 'e', 'f']
```

`sorted` においても、`reverse = True` と記述することで要素を降順に並べ替えることができます。

```
[41]: numbers = [30, 50, 10, 20, 40, 60]
      sorted(numbers, reverse=True)
```

```
[41]: [60, 50, 40, 30, 20, 10]
```

ついでですが、多重リストをソートするとどのような結果が得られるか確かめてみてください。

```
[42]: lns = [[20, 5], [10, 30], [40, 20], [30, 10]]
      lns.sort()
      lns
```

```
[42]: [[10, 30], [20, 5], [30, 10], [40, 20]]
```

7.5 破壊的 (インプレース) な操作と非破壊的な生成

上記では、`sort` メソッドと `sorted` 関数を紹介しましたが、両者の使い方が異なることに気が付きましたか？

具体的には、`sort` メソッドでは元のリストが変更されています。一方、`sorted` 関数では元のリストはそのままになっています。もう一度確認してみましょう。

```
[43]: numbers = [30, 50, 10, 20, 40, 60]
      numbers.sort()
      print('sort メソッドの実行後の元のリスト:', numbers)
      numbers = [30, 50, 10, 20, 40, 60]
      sorted_numbers = sorted(numbers)
      print('sorted 関数の実行後の元のリスト:', numbers)
```

```
sort メソッドの実行後の元のリスト: [10, 20, 30, 40, 50, 60]
```

```
sorted 関数の実行後の元のリスト: [30, 50, 10, 20, 40, 60]
```

このように、`sort` メソッドは元のリストを変更してしまいます。このような操作を**破壊的**あるいは**インプレース (in-place)** であるといいます。

一方、`sorted` 関数は新しいリストを生成し元のリストを破壊しません、このような操作は**非破壊的**であるといいます。

`sorted` 関数を用いた場合、その返回值 (並べ替えの結果) は新しい変数に代入して使うことができます。

一方、`sort` メソッドはリストを返さないためそのような使い方はできません。

```
[44]: numbers = [30, 50, 10, 20, 40, 60]
      numbers1 = sorted(numbers)
      print('sorted 関数の返回值:', numbers1)

      numbers = [30, 50, 10, 20, 40, 60]
      numbers2 = numbers.sort()
      print('sort メソッドの返回值:', numbers2)

sorted 関数の返回值: [10, 20, 30, 40, 50, 60]
sort メソッドの返回值: None
```

7.6 リストを操作するメソッドなど

ここからはリストを操作するためのメソッドなどを紹介していきます。

メソッドや組み込み関数が破壊的であるかどうかは、一般にその名称などからは判断できません。それぞれ破壊的かどうか理解してから利用しなければなりません。

なお、次の `append` メソッド以外は、必要に応じて参照すればよく、それ以降タプルの項まで飛ばして構いません。

7.6.1 リストに要素を追加する

`append` メソッドはリストの最後尾に指定した要素を付け加えます。

```
リスト.append(追加する要素)
```

```
[45]: numbers = [10, 20, 30, 40, 50]
      numbers.append(100)
      numbers
```

```
[45]: [10, 20, 30, 40, 50, 100]
```

`append` は、上述した空のリストと組み合わせて、あるリストから特定の条件を満たす要素のみからなる新たなリストを構成する、というような状況でしばしば用いられます。たとえば、リスト `numbers1 = [10, -10, 20, 30, -20, 40, -30]` から 0 より大きい要素のみを抜き出したリスト `positives` は次のように構成することができます。

```
[46]: numbers1 = [10, -10, 20, 30, -20, 40, -30]
      positives = [] # 空のリストを作成する
      positives.append(numbers1[0])
      positives.append(numbers1[2])
      positives.append(numbers1[3])
      positives.append(numbers1[5])
      positives
```

```
[46]: [10, 20, 30, 40]
```

7.6.2 ▲リストにリストの要素を追加する

`extend` メソッドはリストの最後尾に指定したリストの要素を付け加えます。

```
リスト.extend(追加するリスト)
```

```
[47]: numbers = [10, 20, 30, 40, 50]
      numbers.extend([200, 300, 400, 200]) # numbers += [200, 300, 400, 200] と同じ
      numbers
```

```
[47]: [10, 20, 30, 40, 50, 200, 300, 400, 200]
```

7.6.3 ▲リストに要素を挿入する

`insert` メソッドはリストのインデックスを指定した位置に新しい要素を挿入します。

```
リスト.insert(インデックス, 新しい要素)
```

```
[48]: numbers = [10, 20, 30, 40, 50]
      numbers.insert(1, 1000)
      numbers
```

```
[48]: [10, 1000, 20, 30, 40, 50]
```

7.6.4 ▲リストから要素を削除する

`remove` メソッドは指定した要素をリストから削除します。

```
リスト.remove(削除したい要素)
```

ただし、指定した要素が複数個リストに含まれる場合、一番最初の要素が削除されます。また、指定した値がリストに含まれない場合はエラーとなります。

```
[49]: numbers = [10, 20, 30, 40, 20]
      numbers.remove(30) # 指定した要素を削除
      numbers
```

```
[49]: [10, 20, 40, 20]
```

```
[50]: numbers.remove(20) # 指定した要素が複数個リストに含まれる場合、一番最初の要素を削除
      numbers
```

```
[50]: [10, 40, 20]
```

```
[51]: numbers.remove(100) # リストに含まれない値を指定するとエラー
```

```
-----
ValueError                                Traceback (most recent call last)
Input In [51], in <module>
----> 1 numbers.remove(100)

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


7.6.5 ▲リストからインデックスで指定した要素を削除する

pop メソッドはリストから指定したインデックスを削除し、その要素を返します。

```
リスト.pop(削除したい要素のインデックス)
```

```
[52]: numbers = [10, 20, 20, 30, 20, 40]
      print(numbers.pop(3))
      print(numbers)
```

```
30
[10, 20, 20, 20, 40]
```

インデックスを指定しない場合、最後尾の要素を削除して返します。

```
リスト.pop()
```

```
[53]: ln = [10, 20, 30, 20, 40]
      print(ln.pop())
      print(ln)
```

```
40
[10, 20, 30, 20]
```

7.6.6 ▲リスト要素を削除する

del 文は指定するリストの要素を削除します。具体的には以下のように削除したい要素をインデックスで指定します。

del も破壊的であることに注意してください。

```
del リスト [x]
```

```
[54]: numbers = [10, 20, 30, 40, 50]
      del numbers[2]
      numbers
```

```
[54]: [10, 20, 40, 50]
```

スライスを使うことも可能です。

```
del リスト [x:y]
```

```
[55]: numbers = [10, 20, 30, 40, 50]
      del numbers[2:4]
      numbers
```

```
[55]: [10, 20, 50]
```

7.6.7 ▲リストの要素を逆順にする

`reverse` メソッドはリスト内の要素の順序を逆順にします。

```
[56]: characters = ['e', 'd', 'a', 'c', 'f', 'b']
      characters.reverse()
      characters

[56]: ['b', 'f', 'c', 'a', 'd', 'e']
```

7.6.8 ▲ `copy`

リストを複製します。すなわち、`ln` の値がリストであるとき、`ln.copy()` は `ln` と同じ長さのリストを新たに作って、`ln` の要素を新しいリストに同じ順番で格納して、その新しいリストを返します。

複製されたリストに変更を加えたとしても、もとのリストは影響を受けません。

```
[57]: numbers = [10, 20, 30, 40, 50]
      numbers2 = numbers.copy()
      del numbers[1:3]
      numbers.reverse()
      print(numbers)
      print(numbers2)

[50, 40, 10]
[10, 20, 30, 40, 50]
```

一方、代入を用いた場合には影響を受けることに注意してください。

```
[58]: numbers = [10, 20, 30, 40, 50]
      numbers2 = numbers
      del numbers[1:3]
      numbers.reverse()
      print(numbers)
      print(numbers2)

[50, 40, 10]
[50, 40, 10]
```

7.7 リストと文字列の相互変換

文字列は変更不可能である一方、リスト変更可能です。そのため、文字列処理をする際は、文字列のリストに一旦変換してから、変更を加えて、文字列に変換することが典型的です。ここでは、文字列とリストの相互変換の方法を示します。

まず、文字列 `s` を `list` 関数に渡すと、`s` を文字単位で区切ったリストが得られます。

```
[59]: list('abc123')

[59]: ['a', 'b', 'c', '1', '2', '3']
```

文字単位ではなく、指定された文字列で区切ってリストにする際は、`split` メソッドを使います。

```
[60]: 'banana'.split('n')

[60]: ['ba', 'a', 'a']
```

```
[61]: 'A and B and C'.split(' and ')
```

```
[61]: ['A', 'B', 'C']
```

`split` を無引数で呼び出すと、連続した空白文字を区切りと見做します。

```
[62]: 'A  B\nC '.split()
```

```
[62]: ['A', 'B', 'C']
```

逆に、文字のリストを連結して 1 つの文字列にする際は、`join` メソッドを次のように使います。

```
接合点に挿入する文字列.join(文字列のリスト)
```

```
[63]: ''.join(['a', 'b', 'c', '1', '2', '3'])
```

```
[63]: 'abc123'
```

```
[64]: 'n'.join(['ba', 'a', 'a'])
```

```
[64]: 'banana'
```

なお、`join` の引数は、文字列のリストだけでなく、文字列のタプルでも問題ありません。タプルについては、次で述べます。

7.7.1 練習

email アドレス `email` とドメイン名 `domain` を引数に取って、`email` のドメイン名を `domain` に置き換える関数 `change_domain(email, domain)` を作成してください。なお、email アドレスのドメイン名とは、`'@'` で区切られた右側の部分を意味します。

次のセルの ... のところを書き換えて `change_domain(email, domain)` を完成させてください。

```
[65]: def change_domain(email, domain):
      ...
```

上のセルで解答を作成した後、以下のセルを実行し、実行結果が `True` になることを確認してください。

```
[66]: print(change_domain('spam@utokyo-ipp.org', 'ipp.u-tokyo.ac.jp') == 'spam@ipp.u-tokyo.
      ↪ac.jp')
```

```
False
```

7.8 タプル (tuple)

タプルは、リストと同じようにデータの並びであり、あらゆる種類のデータを要素にできます。ただし、リストと違ってタプルは一度設定した要素を変更できません（文字列も同様でした）。すなわち、タプルは変更不可能なデータです。したがって、リストの項で説明したメソッドの多く、要素を操作するものは適用できないのですが、逆にいうと、作成した後で要素を変更する必要がない場合は、タプルの方が実装の効率がよいので、リストよりもタプルを使うべきです。

たとえば、関数が複数の値をリストにして返し、呼び出し側がすぐにリストをばらばらにして値を取り出すような場合は、リストよりもタプルを使うべきです。また、平面上の点を表そうとするとき、`x` 座標と `y` 座標を別々に変化させる必要がなければ、`(3, 5)` のようなタプルを使うのが自然です。このように、タプルを作成するには数学におけるのと同様に要素を丸括弧 (...) で囲みます。

例を見ましょう。

```
[67]: x = 3
      y = 5
      point = (x, y)
```

```
[68]: point
```

```
[68]: (3, 5)
```

```
[69]: type(point)
```

```
[69]: tuple
```

```
[70]: numbers3 = (1, 2, 3)
      numbers3
```

```
[70]: (1, 2, 3)
```

実は、丸括弧なしでもタプルを作成できます。

```
[71]: numbers3 = 1,2,3
      numbers3
```

```
[71]: (1, 2, 3)
```

要素が1つだけの場合は、`t = (1)`ではなく、次のようにします。

```
[72]: onlyone = (1,)
      onlyone
```

```
[72]: (1,)
```

`t = (1)`だと、`t = 1`と同じになってしまいます。

```
[73]: onlyone = (1)
      onlyone
```

```
[73]: 1
```

何も要素を格納していないタプル（空タプル）は `()` で作成できます。

```
[74]: empty = ()
      empty
```

```
[74]: ()
```

リストや文字列と同様に、インデックスや組み込み関数を使った操作が可能です。

```
[75]: numbers3 = (1, 2, 3)
      numbers3[1] # インデックスの指定による値の取得
```

```
[75]: 2
```

```
[76]: len(numbers3) # lenはタプルを構成する要素の数
```

```
[76]: 3
```

```
[77]: numbers3[1:3] # スライス
```

```
[77]: (2, 3)
```

上述しましたが、一度作成したタプルの要素を後から変更することはできません。したがって以下のプログラムはエラーとなります。

```
numbers3 = (1, 2, 3)
numbers3[1] = 5
```

組み込み関数 `list` を使って、タプルをリストに変換できます。（`list` はリストのデータ型でもあります。）

```
[78]: numbers3 = (1, 2, 3)
      list(numbers3)
```

```
[78]: [1, 2, 3]
```

組み込み関数 `tuple` を使って、逆にリストをタプルに変換できます。（`tuple` はタプルのデータ型でもあります。）

```
[79]: numbers2 = [1, 2]
      tuple(numbers2)
```

```
[79]: (1, 2)
```

7.9 練習

整数の要素からなるリスト `ln` を引数として取り、`ln` に含まれる要素を逆順に格納したタプルを返す関数 `reverse_totuple` を作成してください。

以下のセルの ... のところを書き換えて `reverse_totuple(ln)` を作成してください。

```
[80]: def reverse_totuple(ln):
      ...
```

上のセルで解答を作成した後、以下のセルを実行し、実行結果が `True` になることを確認してください。

```
[81]: print(reverse_totuple([1, 2, 3, 4, 5]) == (5, 4, 3, 2, 1))
```

```
False
```

7.10 多重代入

多重代入では、左辺に複数の変数などを指定してタプルやリストの全ての要素を一度の操作で代入することができます。

```
[82]: numbers = [0, 10, 20, 30, 40]
      [a, b, c, d, e] = numbers
      b
```

```
[82]: 10
```

以下のようにしても同じ結果を得られます。

```
[83]: a, b, c, d, e = numbers
      b
```

```
[83]: 10
```

多重代入は文字列に対しても実行可能です。

```
[84]: a, b, c, d, e = 'hello'
      d
```

```
[84]: 'l'
```

タプルに対しても実行可能です。

```
[85]: numbers3 = (1, 2, 3)
      (x,y,z) = numbers3
      y
```

```
[85]: 2
```

以下のように様々な書き方が可能です。

```
[86]: x,y,z = numbers3
      print(y)
      (x,y,z) = (1, 2, 3)
      print(y)
      x,y,z = (1, 2, 3)
      print(y)
      (x,y,z) = 1, 2, 3
      print(y)
      x,y,z = 1, 2, 3
      print(y)
```

```
2
2
2
2
2
2
```

多重代入を使うことで、2つの変数に格納された値の入れ替えを行う手続きはしばしば用いられます。

```
[87]: x = 'apple'
      y = 'pen'
      x, y = y, x
      print(x, y) #w = x; x = y; y = w と同じ結果が得られる
```

```
pen apple
```

7.11 リストやタプルの比較演算

数値などを比較するのに用いた比較演算子を用いて、2つのリストやタプルを比較することもできます。

```
[88]: print([1, 2, 3] == [1, 2, 3])
      print([1, 2] == [1, 2, 3])
```

```
True
False
```

```
[89]: print((1, 2, 3) == (1, 2, 3))
      print((1, 2) == (1, 2, 3))
```

```
True
False
```

```
[90]: print([1, 2, 3] != [1, 2, 3])
      print([1, 2] != [1, 2, 3])
```

```
False
True
```

```
[91]: print((1, 2, 3) != (1, 2, 3))
      print((1, 2) != (1, 2, 3))
```

```
False
True
```

大小の比較は、いわゆる辞書式による比較で、リストやタプルの最初の要素から順に比較して大小を決めます。片方がもう片方を拡張したものであれば、拡張した方を大きいとします。

```
[92]: print([1, 2, 3] <= [1, 2, 3])
      print([1, 2, 3] < [1, 2, 3])
      print([1, 2, 3] < [1, 2, 4])
      print([1, 2] < [1, 2, 3])
```

```
True
False
True
True
```

```
[93]: print((1, 2, 3) <= (1, 2, 3))
      print((1, 2, 3) < (1, 2, 3))
      print((1, 2, 3) < (1, 2, 4))
      print((1, 2) < (1, 2, 3))
```

```
True
False
True
True
```

7.12 for 文による繰り返しとリスト・タプル

きまった操作の繰り返しはコンピュータが最も得意とする処理のひとつです。リストのそれぞれの要素にわたって操作を繰り返したい場合は **for 文** を用います。

リスト `ls` の要素全てに対して、実行文 を繰り返すには次のように書きます。

```
for value in ls:
    実行文
```

`for` で始まる行の `in` の後に処理対象となるリスト `ls` が、`in` の前に変数 `value` が書かれます。

`ls` の最初の要素、すなわち `ls[0]` が `value` に代入され 実行文 が処理されます。実行文 の処理が終われば、`ls` の次の要素が `value` に代入され、処理が繰り返されます。このようにして、`ls` の要素に対する処理が `len(ls)` 回繰り返されると、`for` 文の処理が終了します。

ここでの `in` の働きは、先に説明したリスト要素の有無を検査する `in` とは異なることに、そして、`if` 文と同様、実行文 の前にはスペースが必要であることに注意してください。

次に具体例を示します。3つの要素を持つリスト `ls` から1つずつ要素を取り出し、変数 `value` に代入しています。実行文 では `value` を用いて取り出した要素を参照しています。

```
[94]: ls = [0,1,2]

      for value in ls:
          print('For loop:', value)
```

```
For loop: 0
For loop: 1
For loop: 2
```

`in` の後に直接リストを記述することもできます。

```
[95]: for value in [0,1,2]:
      print('For loop:', value)
```

```
For loop: 0
For loop: 1
For loop: 2
```

実行文の前にスペースがないとエラーが出ます。

```
[96]: for value in [0,1,2]:
      print('For loop:', value)
```

```
Input In [96]
      print('For loop:', value)
      ^
IndentationError: expected an indented block
```

エラーが出れば意図した通りにプログラムが組めていないのにすぐ気が付きますが、エラーが出ないために意図したプログラムが組めていないことに気が付かないことがあります。たとえば、次のような内容を実行しようとしていたとします。

```
[97]: for value in [0,1,2]:
      print('During for loop:', value)
      print('During for loop, too:', value)
```

```
During for loop: 0
During for loop, too: 0
During for loop: 1
During for loop, too: 1
During for loop: 2
During for loop, too: 2
```

後者の print の行のスペースの数が間違っていると、次のような結果になる場合がありますので注意してください。

```
[98]: for value in [0,1,2]:
      print('During for loop:', value)
      print('During for loop, too:', value) #この行のスペースの数が間違っていたがエラーは出ない
```

```
During for loop: 0
During for loop: 1
During for loop: 2
During for loop, too: 2
```

タプルの要素にまたがる処理もリストと同様に行えます。

```
[99]: for value in (0,1,2):
      print('For loop:', value)
```

```
For loop: 0
For loop: 1
For loop: 2
```

以下はリストに対する for 文の典型例です。numbers は数のリストとします。

```
[100]: numbers = [0,1,2,3,4,5]
```

以下のようにして、このリストの要素の自乗から成るリストを求めることができます。


```
[101]: squares1 = []
       for x in numbers:
           squares1.append(x**2)
       squares1
```

```
[101]: [0, 1, 4, 9, 16, 25]
```

squares1 には最初に空リストが代入されます。そして、numbers の各要素の自乗がこのリストに次々と追加されます。

7.13 練習

整数の要素からなるリスト ln を引数として取り、ln の要素の総和を返す関数 sum_list を作成してください。

以下のセルの ... のところを書き換えて sum_list(ln) を作成してください。

```
[102]: def sum_list(ln):
       ...
```

上のセルで解答を作成した後、以下のセルを実行し、実行結果が全て True になることを確認してください。

```
[103]: print(sum_list([10, 20, 30]) == 60)
       print(sum_list([-1, 2, -3, 4, -5]) == -3)
```

```
False
False
```

7.14 for 文による繰り返しと文字列

for 文を使うと文字列全体にまたがる処理も可能です。文字列 str1 をまたがって一文字ずつの繰り返し処理を行う場合は次のように書きます。ここで、c には取り出された一文字（の文字列）が代入されています。

```
for c in str1:
    実行文
```

str1 で与えられる文字列を一文字ずつ大文字で出力する処理は以下のようになります。

```
[104]: str1 = 'Apple and pen'
       for c in str1:
           print(c.upper())
```

```
A
P
P
L
E
```

```
A
N
D
```

```
P
E
N
```

7.15 練習

ATGC の 4 種類の文字から成る文字列 `str_atgc` が引数として与えられたとき、次のようなリスト `list_count` を返す関数 `atgc_countlist` を作成してください。ただし、`list_count` の要素は、各塩基 `bp` に対して `str_atgc` 中の `bp` の出現回数と `bp` の名前を格納した（長さ 2 の）リストとします。

ヒント：文字列 'ATGC' に対する繰り返しを用いることができます。

以下のセルの ... のところを書き換えて `atgc_countlist(str_atgc)` を作成してください。

```
[105]: def atgc_countlist(str_atgc):
        ...
```

上のセルで解答を作成した後、以下のセルを実行し、実行結果が `True` になることを確認してください。

```
[106]: print(sorted(atgc_countlist('AAGCCCCATGGTAA')) == sorted([[5, 'A'], [2, 'T'], [3, 'G'],
↪ [3, 'G'], [4, 'C']]))
```

```
-----
TypeError                                Traceback (most recent call last)
Input In [106], in <module>
----> 1 print(sorted(atgc_countlist('AAGCCCCATGGTAA')) == sorted([[5, 'A'], [2, 'T'],
↪ [3, 'G'], [4, 'C']]))

TypeError: 'NoneType' object is not iterable
```

7.16 for 文によるリスト初期化の短縮記法

先に、リストの要素の自乗から成るリストを求める例を説明しました。

```
[107]: numbers = [0,1,2,3,4,5]

squares1 = []
for x in numbers:
    squares1.append(x**2)
squares1
```

```
[107]: [0, 1, 4, 9, 16, 25]
```

詳しくは 6-1 で説明されますが、**内包表記**を用いて書き換えると、以下のように 1 行で書くことができます。

```
[108]: squares2 = [x**2 for x in numbers]
squares2
```

```
[108]: [0, 1, 4, 9, 16, 25]
```

すなわち、

[`x` を含む式 for `x` in リストを返す式]

という式は、“リストを返す式” が返したリストの各要素を `x` に代入して “`x` を含む式” を計算し、その結果をリストにして返します。もちろん、変数は `x` でなくてもよいです。

```
[109]: [y**2 for y in numbers]
```

```
[109]: [0, 1, 4, 9, 16, 25]
```

7.17 ▲オブジェクトの等価性と同一性

1-3 で、Python における値はオブジェクトと総称されますと述べました。ここでは、オブジェクトの等価性と同一性について説明します。

既に見てきたように、演算子 `==` を用いて**オブジェクトの等価性**を判定できます。

```
[110]: a = []
       b = []
```

このとき、`a` と `b` の値はどちらも空リストなので、以下のように `a` の値と `b` の値は等価です。

```
[111]: a == b
[111]: True
```

しかし、`[` で始まり `]` で終わる式を評価すると、必ず新しいリスト（オブジェクト）が作られて返されるので、`a` と `b` の値は同一ではありません。

オブジェクトの同一性は演算子 `is` を用いて判定できます。

```
[112]: a is b
[112]: False
```

リストの要素はオブジェクトなので、要素ごとに等価性と同一性が定まります。

例として、`a` と `b` を要素とするリスト `c` を作ります。

```
[113]: c = [a, b]
       c
```

```
[113]: [[], []]
```

```
[114]: c[0] is c[1]
[114]: False
```

`a` を変化させてみましょう。

```
[115]: a.append(1)
       a
```

```
[115]: [1]
```

すると `c` は以下ようになります。

```
[116]: c
[116]: [[1], []]
```

ここで、`a` と `b` は等価でなくなりました。

```
[117]: a == b
[117]: False
```

次に、`b` を要素として二重に含むリスト `d` を作ります。

```
[118]: d = [b, b]
       d
```

```
[118]: [[], []]
```

```
[119]: d[0] is d[1]
```

```
[119]: True
```

bを変化させてみましょう。

```
[120]: b.append(1)
b
```

```
[120]: [1]
```

すると d は以下ようになります。

```
[121]: d
```

```
[121]: [[1], [1]]
```

演算子 == でリストを比較すると、要素まで見て等価性を判定します。

```
[122]: print(a, b)
a == b
```

```
[1] [1]
```

```
[122]: True
```

演算子 == は、要素の比較も == で行います。

```
[123]: print(c, d)
c == d
```

```
[[1], [1]] [[1], [1]]
```

```
[123]: True
```

一方、オブジェクトの同一性は変化しません。

```
[124]: a is b
```

```
[124]: False
```

== の否定形は != で、is の否定形は is not です。not x == y は x != y と書けます。not x is y は x is not y と書けます。is not はこれで 1 つの演算子なので注意してください。

```
[125]: c != d
```

```
[125]: False
```

```
[126]: a is not b
```

```
[126]: True
```

7.18 練習の解答

```
[127]: def remove_evenindex(ln):
        ln2 = ln[1::2]
        return ln2
#remove_evenindex(['a', 'b', 'c', 'd', 'e', 'f', 'g'])
```

```
[128]: def change_domain(email, domain):
        return '@'.join(email.split('@')[0], domain))
```

```
[129]: def reverse_totuple(ln):  
        ln.reverse()  
        tup = tuple(ln)  
        return tup  
#reverse_totuple([1, 2, 3, 4, 5])
```

```
[130]: def sum_list(ln):  
        int_sum = 0  
        for value in ln:  
            int_sum += value  
        return int_sum  
#sum_list([10, 20, 30])
```

```
[131]: def atgc_countlist(str_atgc):  
        list_count = []  
        for value in 'ATGC':  
            int_bpcnt = str_atgc.count(value)  
            list_count.append([int_bpcnt, value])  
        return list_count  
#atgc_countlist('AAGCCCCATGGTAA')
```

```
[ ]:
```

2-3. 条件分岐

制御構造のうち**条件分岐**について説明します。

参考:

- <https://docs.python.org/ja/3/tutorial/controlflow.html#if-statements>

if で始まり条件分岐を行う制御構造によって、条件に応じてプログラムの動作を変えることができます。ここではまず「インデント」について説明し、そのあとで条件分岐について説明します。

8.1 インデントによる構文

条件分岐の前に、Python の**インデント**（行頭の空白、字下げ）について説明します。Python のインデントは実行文をグループにまとめる機能を持ちます。

プログラム文はインデントレベル（深さ）の違いによって異なるグループとして扱われます。細かく言えば、インデントレベルが進む（深くなる）とプログラム文はもとのグループの下に位置する別のグループに属するものとして扱われます。逆に、インデントレベルが戻る（浅くなる）までプログラム文は同じグループに属することになります。

具体例として、第 1 回で定義した関数 `bmax()` を使って説明します:

```
[1]: def bmax(a,b):  
    if a > b:  
        return a  
    else:  
        return b  
  
print('Hello World')  
  
Hello World
```

この例では 1 行目の関数定義 `def bmax(a,b):` の後から第 1 レベルのインデントが開始され 5 行目まで続きます。すなわち、5 行目までは関数 `bmax` を記述するプログラム文のグループということです。

次に、3 行目の一行のみの第 2 レベルのインデントの実行文は、if 文（if による条件分岐）の論理式 `a > b` が `True` の場合にのみ実行されるグループに属します。そして、4 行目の `else` ではインデントが戻されています。5 行目から再び始まる第 2 レベルの実行文は 2 行目の論理式が `False` の場合に実行されるグループに属します。

最後に、7行目ではインデントが戻されており、これ以降は関数 `bmax()` の定義とは関係ないことがわかります。

Python ではインデントとして半角スペース 4 つが広く利用されています。本教材でもこの書式を利用します。1-4 のコーディングスタイルのところで紹介したスタイルガイドの PEP8 でも、半角スペース 4 つが推奨されています。

Code セルでは行の先頭で Tab を入力すれば、自動的にこの書式のインデントが挿入されます。また、インデントを戻すときは Shift-Tab が便利です。なお、Colaboratory では、Tab を入力すると半角スペース 2 つのインデントが挿入されます。

8.2 if … else による条件分岐

これまで関数 `bmax` を例にとって説明しましたが、一般に if 文では、式が真であれば if 直後のグループが、偽であれば else 直後のグループが、それぞれ実行されます。（真であった場合、else 直後のグループは実行されません。）

```
if 式:
    このグループは「式」が真のときにのみ実行される
else:
    このグループは「式」が偽のときにのみ実行される
```

また、else は省略することができます。省略した場合、「式」が真のときに if 直後のグループが実行されるのみになります。

```
if 式:
    このグループは「式」が真のときにのみ実行される
このグループは常に実行される
```

条件が複雑になってくると、if 文の中にさらに if 文を記述して、条件分岐を入れ子（ネスト）にすることがあります。この場合は、インデントはさらに深くなります。

そして、下の 2 つのプロラムの動作は明らかに異なることに注意が必要です。

```
if 式 1:
    このグループは「式 1」が真のときにのみ実行される
    if 式 2:
        このグループは「式 1」「式 2」が共に真のときにのみ実行される
        if 式 3:
            このグループは「式 1」「式 2」「式 3」が全て真のときにのみ実行される
            このグループは「式 1」と「式 2」が共に真のときにのみ実行される
        このグループは「式 1」が真のときにのみ実行される
    このグループは常に実行される
```

```
if 式 1:
    このグループは「式 1」が真のときにのみ実行される
このグループは常に実行される
if 式 2:
    このグループは「式 2」が真のときにのみ実行される（「式 1」には影響されない）
このグループは常に実行される
if 式 3:
    このグループは「式 3」が真のときにのみ実行される（「式 1」「式 2」には影響されない）
このグループは常に実行される
```

8.3 if … elif … else による条件分岐

ここまでで if … else 文について紹介しましたが、複数の条件分岐を続けて書くことができる elif を紹介します。

たとえばテストの点数から評定（優、良、可、…）を計算したい場合など、「条件 1 のときは処理 1、条件 1 に該当しなくても条件 2 であれば処理 2、更にどちらでもない場合、条件 3 であれば処理 3、…」という処理を考えます。if … else による文のみでこの処理を行う場合、次のようなプログラムになってしまいます：

```
if 式 1:
    「式 1」が真のときにのみ実行するグループ
else:
    if 式 2:
        「式 1」が偽 かつ 「式 2」が真のときにのみ実行するグループ
    else:
        if 式 3:
            「式 1」「式 2」が偽 かつ 「式 3」が真のときにのみ実行するグループ
        else:
            ...
```

このような場合には、以下のように elif を使うとより簡潔にできます：

```
if 式 1:
    このグループは「式 1」が真のときにのみ実行される
elif 式 2:
    このグループは「式 1」が偽 かつ 「式 2」が真のときにのみ実行される
elif 式 3:
    このグループは「式 1」「式 2」が偽 かつ 「式 3」が真のときにのみ実行される
else:
    このグループは「式 1」「式 2」「式 3」がいずれも偽のときにのみ実行される
```

if … elif … else では、条件は上から順に評価され、式が真の場合、直後の実行文グループのみが実行され終了します。その他の場合、すなわち全ての条件が False のときは、else 以降のグループが実行されます。

なお、elif もしくは else 以降を省略することも可能です。

8.4 練習

関数 `exception3(x,y,z)` の引数は以下の条件を満たすとします。

- `x` と `y` と `z` の値は整数です。
- `x` と `y` と `z` のうち、2 つの値は同じで、もう 1 つの値は他の 2 つの値とは異なるとします。

その異なる値を返すように、以下のセルの … のところを書き換えて `exception3(x,y,z)` を定義してください。

```
[2]: def exception3(x,y,z):
    ...
```

次のセルで動作を確認してください。

```
[3]: print(exception3(1,2,2))
print(exception3(4,2,4))
print(exception3(9,3,9))
```



```
None
None
None
```

8.5 練習

関数 `exception9(a)` の引数は以下の条件を満たすとします。

- 引数 `a` には、長さが 9 のリストが渡されます。
- このリストの要素は整数ですが、1 つの要素を除いて、残りは要素の値は全て同じとします。

その 1 つの要素の値を返すように、以下のセルの ... のところを書き換えて `exception9(a)` を定義してください。

```
[4]: def exception9(a):
      ...
```

次のセルで動作を確認してください。

```
[5]: print(exception9([1,2,2,2,2,2,2,2,2]))
      print(exception9([4,4,4,4,4,2,4,4,4]))
      print(exception9([9,9,9,9,9,9,9,9,3]))
```

```
None
None
None
```

8.6 ▲複数行にまたがる条件式

複雑な条件式では複数行に分割した方が見やすい場合もあります。ここでは、式を複数行にまたがって記述する 1 つの方法を示します。1 つ目は、丸括弧で括られた式を複数の行にまたがって記述する方法です。2 つ目は、行末にバックスラッシュ `\` を置く方法です。

```
[6]: ### 丸括弧で括る方法
      x, y, z = (-1, -2, -3)
      if (x < 0 and y < 0 and z < 0 and
          x != y and y != z and x != z):
          print('x, y and z are different and negatives.')
```

行末にバックスラッシュ(`\`)を入れる方法

```
x, y, z = (-1, -2, -3)
if x < 0 and y < 0 and z < 0 and \
    x != y and y != z and x != z:
    print('x, y and z are different and negatives.')
```

```
x, y and z are different and negatives.
x, y and z are different and negatives.
```

8.7 if … elif … else における条件の評価

if と elif による条件分岐では、if あるいは elif に続く条件式が True の場合、それ以降の elif に続く条件式の評価は行われません。

以下のプログラムで `x` を 3, 0, -4 とした際に何が表示されるかを予想したのちに実行してみましょう。

特に、`x = -4` としたときの動作に注意してください。（`x is zero.` は表示されません。）

```
[7]: x = 3 # example: 3, 0, -4

if x > 0:
    print('x is greater than zero.')
elif x < 0:
    print('x is less than zero, but x will be 0')
    x = 0
else:
    print('x is zero.')

print(x)

x is greater than zero.
3
```

8.8 練習

以下のプログラムはプログラマの意図どおりに動作しません。print の出力内容から意図を判断して条件分岐を書き換えてください。

```
[8]: x = -1
if x < 3:
    print('x is larger than or equal to 2, and less than 3')
elif x < 2:
    print('x is larger than or equal to 1, and less than 2')
elif x < 1:
    print('x is less than 1')
else:
    print('x is larger or equal to 3')

x is larger than or equal to 2, and less than 3
```

8.9 or もしくは and で結合された条件の評価

if 文に与える条件が or もしくは and で結合されている場合、条件は左から順に評価され、不要（以降の式を評価するまでもなく自明）な評価は省かれます。

たとえば、`if a == 0 or b == 0:` において 最初の式 `a == 0` が True の場合、式全体の結果が True となることは自明なので、二番目の式 `b == 0` を評価することなく続く実行文グループが実行されます。

逆に、`if a == 0 and b == 0:` において、最初の式が False の場合、以降の式は評価されることなく処理が進みます。

以下のセルで示す例の 1 行目で、`x` の値を 0, -4 に変更し、表示される内容を予想し、予想通りか確認してください。

```
[9]: x = 10          # del x のエラーを抑制するため
     y = 10
```

```
del x          # x を未定義に
```

```
if x > 5 or y > 5:
    print("'x' or 'y' is larger than 5")
```

```
-----
NameError                                Traceback (most recent call last)
```

```
Input In [9], in <module>
```

```
2 y = 10
```

```
4 del x          # x を未定義に
```

```
----> 6 if x > 5 or y > 5:
      7     print("'x' or 'y' is larger than 5")
```

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

```
[10]: x = 10
      y = 10          # del y のエラーを抑制するため
```

```
del y          # y を未定義に
```

```
if x > 5 or y > 5:
    print("'x' or 'y' is larger than 5")
```

```
'x' or 'y' is larger than 5
```

8.10 ▲ 3 項演算子（条件式）

Python では以下のように if ... else を 1 行に書くこともできます。

```
[11]: x = 0
      sign = 'positive or zero' if x >= 0 else 'negative'
      print(sign)
```

```
positive or zero
```

これは、以下と等価です。

```
[12]: x = 0
      if x >= 0 :
          sign = 'positive or zero'
      else:
          sign = 'negative'
      print(sign)
```

```
positive or zero
```

8.11 練習の解答

以下は解答例です。これ以外にも様々な解答があり得ます。

```
[13]: def exception3(x,y,z):
      if x==y:
          return z
      elif x==z:
          return y
      else:
          return x
```

```
[14]: def exception9(a):
      x = a[0] + a[1] + a[2]
      y = a[3] + a[4] + a[5]
      z = a[6] + a[7] + a[8]
      if x==y:
          return exception3(a[6], a[7], a[8])
      elif x==z:
          return exception3(a[3], a[4], a[5])
      else:
          return exception3(a[0], a[1], a[2])
```

8.12 練習の解説

最後の練習では、条件文の順番を修正する必要があります。条件は上から順に処理され、式が真の場合にその「直後の実行文グループのみ」が処理されます。

```
[15]: x = -1
      if x < 1:
          print('x is less than 1')
      elif x < 2:
          print('x is larger or equal to 1, and less than 2')
      elif x < 3:
          print('x is larger or equal to 2, and less than 3')
      else:
          print('x is larger or equal to 3')

x is less than 1
```

```
[ ]:
```

3-1. 辞書 (dictionary)

キーと値を対応させるデータ構造である辞書について説明します。

参考

- <https://docs.python.org/ja/3/tutorial/datastructures.html#dictionaries>

辞書は、**キー (key)** と **値 (value)** を対応づけるデータです。キーとしては、文字列・数値・タプルなどの変更不可能なデータを使うことができますが、変更可能なデータであるリスト・辞書を使うことはできません。（辞書も変更可能なデータです。）一方、値としては、変更の可否にかかわらずあらゆる種類のデータを指定できます。

たとえば、文字列 'apple' をキーとし値として数値 3 を、'pen' をキーとして 5 を対応付けた辞書は、次のように作成します。

```
[1]: ppap = {'apple' : 3, 'pen' : 5}
ppap
```

```
[1]: {'apple': 3, 'pen': 5}
```

```
[2]: type(ppap)
```

```
[2]: dict
```

辞書の キー 1 に対応する値を得るには、リストにおけるインデックスと同様に、

```
辞書 [キー 1]
```

とします。

```
[3]: ppap = {'apple' : 3, 'pen' : 5}
ppap['apple']
```

```
[3]: 3
```

辞書に登録されていないキーを指定すると、エラーになります。

```
[4]: ppap['orange']
```

```
-----
KeyError
```

```
Traceback (most recent call last)
```

```
(continues on next page)
```

(continued from previous page)

```
Input In [4], in <module>
----> 1 ppap['orange']

KeyError: 'orange'
```

キーに対する値を変更したり、新たなキーと値を登録するには代入文を用います。

```
[5]: ppap = {'apple' : 3, 'pen' : 5}
      ppap['apple'] = 10
      ppap['pinapple'] = 7
      ppap
[5]: {'apple': 10, 'pen': 5, 'pinapple': 7}
```

上のようにキーから値は取り出せますが、値からキーを直接取り出すことはできません。また、リストのようにインデックスを指定して値を取得することはできません。

```
[6]: ppap[1]

-----
KeyError                                Traceback (most recent call last)
Input In [6], in <module>
----> 1 ppap[1]

KeyError: 1
```

キーが辞書に登録されているかどうかは、演算子 `in` を用いて調べることができます。

```
[7]: ppap = {'apple': 3, 'pen': 5}
      'apple' in ppap
[7]: True

[8]: 'banana' in ppap
[8]: False
```

組み込み関数 `len` によって、辞書に登録されている要素、キーと値のペア、の数が得られます。

```
[9]: ppap = {'apple': 3, 'pen': 5}
      len(ppap)
[9]: 2
```

`del` 文によって、登録されているキーの要素を削除することができます。具体的には、次のように削除します。

```
del 辞書 [削除したいキー]
```

```
[10]: ppap = {'apple' : 3, 'pen' : 5}
       del ppap['pen']
       ppap
[10]: {'apple': 3}
```

空のリストと同様に空の辞書を作ることができます。このような空のデータは繰り返し処理でしばしば使われます。

```
[11]: empty_d = {}
      empty_d
```

```
[11]: {}
```

9.1 練習

リスト `list1` が引数として与えられたとき、`list1` の各要素 `value` をキー、`value` の `list1` におけるインデックスをキーに対応する値とした辞書を返す関数 `reverse_lookup` を作成してください。

以下のセルの ... のところを書き換えて `reverse_lookup(list1)` を作成してください。

```
[12]: def reverse_lookup(list1):
      ...
```

上のセルで解答を作成した後、以下のセルを実行し、実行結果が `True` になることを確認してください。

```
[13]: print(reverse_lookup(['apple', 'pen', 'orange']) == {'apple': 0, 'orange': 2, 'pen': 1})
      ↪ 1})
False
```

9.2 辞書のメソッド

辞書のメソッドを紹介しておきます。

9.2.1 キーを指定して値を得るメソッド

`get` メソッドは、引数として指定したキーが辞書に含まれてる場合にはその値を取得し、指定したキーが含まれていない場合には `None` を返します。`get` を利用することで、エラーを回避し、登録されているかどうかかわからないキーを使うことができます。先に説明したキーを括弧、`[...]`、で指定する方法では、辞書にキーが存在しないとエラーとなりプログラムの実行が停止してしまいます。

```
[14]: ppap = {'apple' : 3, 'pen' : 5}
      print('キー apple に対応する値 = ', ppap.get('apple'))
      print('キー orange に対応する値 = ', ppap.get('orange'))
      print('キー orange に対応する値 (エラー) = ', ppap['orange'])
```

```
キー apple に対応する値 = 3
キー orange に対応する値 = None
```

```
-----
KeyError                                Traceback (most recent call last)
Input In [14], in <module>
      2 print('キー apple に対応する値 = ', ppap.get('apple'))
      3 print('キー orange に対応する値 = ', ppap.get('orange'))
----> 4 print('キー orange に対応する値 (エラー) = ', ppap['orange'])

KeyError: 'orange'
```

また、`get` に 2 番目の引数を与えると、その引数の値を「指定したキーが含まれていない場合」に `get` が返す値とすることができます。

```
[15]: ppap = {'apple' : 3, 'pen' : 5}
      print('キー apple に対応する値 = ', ppap.get('apple', -1))
      print('キー orange に対応する値 = ', ppap.get('orange', -1))
```

```
キー apple に対応する値 = 3
キー orange に対応する値 = -1
```

9.2.2 ▲キーがない場合に登録を行う

setdefault メソッドは、指定したキーが辞書に含まれてる場合には、対応する値を返します。キーが含まれていない場合には、2 番目の引数として指定した値を返すと同時に、キーに対応する値として登録します。

```
[16]: ppap = {'apple' : 3, 'pen' : 5}
print('キー apple に対応する値 = ', ppap.setdefault('apple', 7))
print('setdefault("apple", 7) を実行後の辞書 = ', ppap)
print('キー orange に対応する値 = ', ppap.setdefault('orange', 7))
print('setdefault("orange", 7) を実行後の辞書 = ', ppap)
```

```
キー apple に対応する値 = 3
setdefault("apple", 7) を実行後の辞書 = {'apple': 3, 'pen': 5}
キー orange に対応する値 = 7
setdefault("orange", 7) を実行後の辞書 = {'apple': 3, 'pen': 5, 'orange': 7}
```

上のような setdefault を用いた手続きを、[...] を用いて書き換えるとたとえば次のようになります。

```
[17]: ppap = {'apple' : 3, 'pen' : 5}
if 'apple' not in ppap:
    ppap['apple'] = 7
print('キー apple に対応する値 = ', ppap['apple'])
print('実行後の辞書 = ', ppap)
if 'orange' not in ppap:
    ppap['orange'] = 7
print('キー orange に対応する値 = ', ppap['orange'])
print('実行後の辞書 = ', ppap)
```

```
キー apple に対応する値 = 3
実行後の辞書 = {'apple': 3, 'pen': 5}
キー orange に対応する値 = 7
実行後の辞書 = {'apple': 3, 'pen': 5, 'orange': 7}
```

9.2.3 ▲キーを指定した削除

pop メソッドは指定したキーおよびそれに対応する値を削除し、削除されるキーに対応付けられた値を返します。

```
[18]: ppap = {'apple' : 3, 'pen' : 5}
print(ppap.pop('pen'))
print(ppap)
```

```
5
{'apple': 3}
```


9.2.4 ▲全てのキーと値の削除

`clear` メソッドは全てのキーと値を削除します。その結果、辞書は空となります。

```
[19]: ppap = {'apple' : 3, 'pen' : 5}
      ppap.clear()
      ppap
```

```
[19]: {}
```

9.2.5 キーの一覧を得る

`keys` メソッドはキーの一覧を返します。これはリストのようなものとして扱うことができ、`for` ループと組み合わせて繰り返し処理で利用されます（3-2 を参照してください）。以下のように、`keys` メソッドが返した結果に関数 `list` を適用すると、通常のリストになります。

```
[20]: ppap = {'apple' : 3, 'pen' : 5}
      list(ppap.keys())
```

```
[20]: ['apple', 'pen']
```

9.2.6 値の一覧を得る

`values` メソッドはキーに対応する全ての値の一覧を返します。これもリストのようなものとして扱うことができます。

```
[21]: list(ppap.values())
```

```
[21]: [3, 5]
```

9.2.7 キーと値の一覧を得る

`items` メソッドはキーとそれに対応する値をタプルにした一覧を返します。これもタプルを要素とするリストのようなものとして扱うことができ、`for` ループなどで活用します（3-2 を参照してください）。

```
[22]: list(ppap.items())
```

```
[22]: [('apple', 3), ('pen', 5)]
```

9.2.8 ▲辞書を複製する

`copy` メソッドは辞書を複製します。リストの場合と同様に一方の辞書を変更してももう一方の辞書は影響を受けません。

```
[23]: ppap = {'apple': 3, 'pen': 5, 'orange': 7}
      ppap2 = ppap.copy()
      ppap['banana'] = 9
      print(ppap)
      print(ppap2)

{'apple': 3, 'pen': 5, 'orange': 7, 'banana': 9}
{'apple': 3, 'pen': 5, 'orange': 7}
```

9.2.9 ▲ keys, values, items の返値

keys, values, items メソッドの一連の説明では、返値を「リストのようなもの」と表現してきました。通常のリストとどう違うのでしょうか？

次の例では、ppap の keys, values, items メソッドの返値をそれぞれ ks, vs, itms に代入し、print でそれぞれの内容を表示させています。

次いで、ppap に新たな要素を加えたのちに、同じ変数の内容を表示させています。1, 2 回目の print で内容が異なることに注意してください。もとの辞書が更新されると、これらの内容も動的に変わります。

```
[24]: ppap = {'apple': 3, 'pen': 5, 'orange': 7}
      ks = ppap.keys()
      vs = ppap.values()
      itms = ppap.items()
      print(list(ks))
      print(list(vs))
      print(list(itms))
      ppap['kiwi'] = 9
      print(list(ks))
      print(list(vs))
      print(list(itms))

['apple', 'pen', 'orange']
[3, 5, 7]
[('apple', 3), ('pen', 5), ('orange', 7)]
['apple', 'pen', 'orange', 'kiwi']
[3, 5, 7, 9]
[('apple', 3), ('pen', 5), ('orange', 7), ('kiwi', 9)]
```

9.3 辞書とリスト

冒頭で述べたように、辞書では値としてあらゆる型のデータを使用できます。すなわち、次のように値としてリストを使用する辞書を作成可能です。リストの要素を参照するには数字インデックスをさらに指定します。

```
[25]: numbers = {'dozens': [10, 20, 40], 'hundreds': [100, 101, 120, 140]}
      print(numbers['dozens'])
      print(numbers['dozens'][1])

[10, 20, 40]
20
```

逆に、辞書を要素とするリストを作成することもできます。

```
[26]: ppap = {'apple': 3, 'pen': 5}
      pets = {'cat': 3, 'dog': 3, 'elephant': 8}
      ld = [ppap, pets]
      print(ld[1])
      print(ld[1]['dog'])

{'cat': 3, 'dog': 3, 'elephant': 8}
3
```

9.4 練習

辞書 `dic1` と文字列 `str1` が引数として与えられたとき、以下のように `dic1` を変更する関数 `handle_collision` を作成してください。ただし、`dic1` のキーは整数、キーに対応する値は文字列を要素とするリストとします。

1. `dic1` に `str1` の長さ `n` がキーとして登録されていない場合、`str1` のみを要素とするリスト `ls` を作成し、`dic1` にキー `n`、`n` に対応する値 `ls` を登録します。
2. `dic1` に `str1` の長さ `n` がキーとして登録されている場合、そのキーに対応する値（リスト）に `str1` を追加します。

以下のセルの ... のところを書き換えて `handle_collision(dic1, str1)` を作成してください。

```
[27]: def handle_collision(dic1, str1):
      ...
```

上のセルで解答を作成した後、以下のセルを実行し、実行結果が `True` になることを確認してください。

```
[28]: dic1_orig = {3: ['ham', 'egg'], 6: ['coffee', 'brandy'], 9: ['port wine'], 15: [
      ↪ 'curried chicken']}
      dic1_result = {3: ['ham', 'egg', 'tea'], 6: ['coffee', 'brandy'], 9: ['port wine'],
      ↪ 15: ['curried chicken']}
      handle_collision(dic1_orig, 'tea')
      print(dic1_orig == dic1_result)

False
```

9.5 練習の解答

```
[29]: def reverse_lookup(list1):
      dic1 = {} # 空の辞書を作成する
      for value in list1:
          dic1[value] = list1.index(value)
      return dic1
      #reverse_lookup(['apple', 'pen', 'orange'])
```

```
[30]: def handle_collision(dic1, str1):
      if dic1.get(len(str1)) is None:
          ls = [str1]
      else:
          ls = dic1[len(str1)]
          ls.append(str1)
          dic1[len(str1)] = ls
      #handle_collision({3: ['ham', 'egg'], 6: ['coffee', 'brandy'], 9: ['port wine'], 15: [
      ↪ 'curried chicken']}, 'tea')
```

```
[ ]:
```

CHAPTER 10

3-2. 繰り返し

制御構造のうち繰り返しについて説明します。

参考:

- <https://docs.python.org/ja/3/tutorial/controlflow.html#for-statements>
- <https://docs.python.org/ja/3/tutorial/controlflow.html#the-range-function>
- <https://docs.python.org/ja/3/tutorial/introduction.html#first-steps-towards-programming>
- <https://docs.python.org/ja/3/tutorial/controlflow.html#break-and-continue-statements-and-else-clauses-on-loops>
- <https://docs.python.org/ja/3/tutorial/controlflow.html#pass-statements>

繰り返しを行う制御構造 **for** や **while** によって、同じ処理の繰り返しを簡単にプログラムすることができます。

10.1 for 文による繰り返し

2-2 で、リストと文字列に対する **for** 文の繰り返しについて説明しました。Python における **for** 文の一般的な文法は以下のとおりです。

```
for 変数 in 文字列・リスト・辞書など:  
    実行文
```

if 文と同様、実行文のインデントは深くなっていることに注意してください。

for 文では **in** 以降に与えられる、文字列・リスト・辞書などにわたって、実行文のグループを繰り返します。一般に繰り返しの順番は文字列・リスト・辞書などに要素が現れる順番で、要素は **for** と **in** の間の変数に代入されます。

リストの場合、リストの要素が最初から順番に取り出されます。以下に具体例を示します。関数 **len** は文字列の長さを返します。

```
[1]: words = ['dog', 'cat', 'mouse']  
for w in words:  
    print(w, len(w))  
print('finish')
```

```
dog 3
cat 3
mouse 5
finish
```

このプログラムで、for 文には3つの文字列で構成されるリスト **words** が与えられています。リストの要素は変数 **w** に順番に代入され、文字列とその長さが印字されます。そして、最後の要素の処理がおわれば for 文の繰り返し（**ループ**）を抜け、完了メッセージを印字します。

次は文字列に対する for 文の例です。文字列を構成する文字が先頭から一文字ずつ文字列として取り出されます。

```
[2]: word = 'supercalifragilisticexpialidocious'
for c in word:
    print(c)
```

```
s
u
p
e
r
c
a
l
l
i
f
r
a
g
i
l
i
s
t
i
c
e
x
p
i
a
l
i
d
o
c
i
o
u
s
```

組み込み関数 **ord** は与えられた文字の番号（コード）を整数として返します。組み込み関数 **chr** は逆に与えられた整数をコードとする文字を返します。

```
[3]: print(ord('a'))
print(ord('b'))
print(ord('z'))

print(chr(97))
```

```
97
98
122
a
```

上で確認しているように、文字 'a', 'b', 'z' のコードはそれぞれ 97, 98, 122 です。文字のコードは 'a' から 'z' までは連続して 1 ずつ増えていきます。

これを用いて以下のように英小文字から成る文字列の中の各文字の頻度を求めることができます。

```
[4]: height = [0] * 26
    for c in word:
        height[ord(c) - ord('a')] += 1

    print(height)

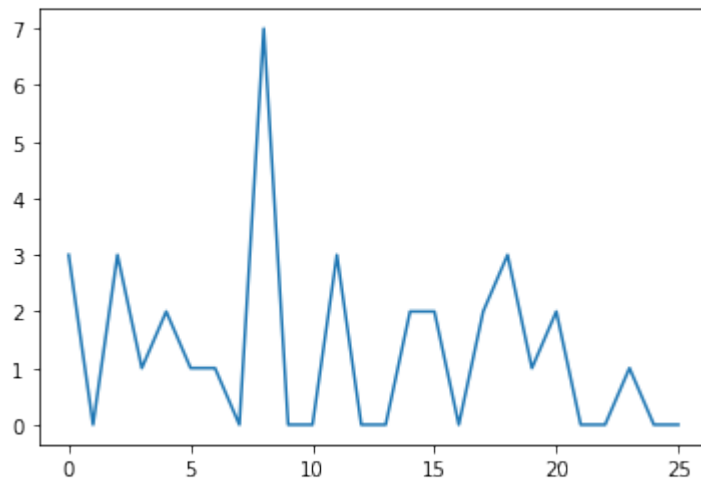
[3, 0, 3, 1, 2, 1, 1, 0, 7, 0, 0, 3, 0, 0, 2, 2, 0, 2, 3, 1, 2, 0, 0, 1, 0, 0]
```

height を視覚化してみましょう。詳しくは、付録の 5-matplotlib を参照してください。

```
[5]: import matplotlib.pyplot as plt

    plt.plot(height)

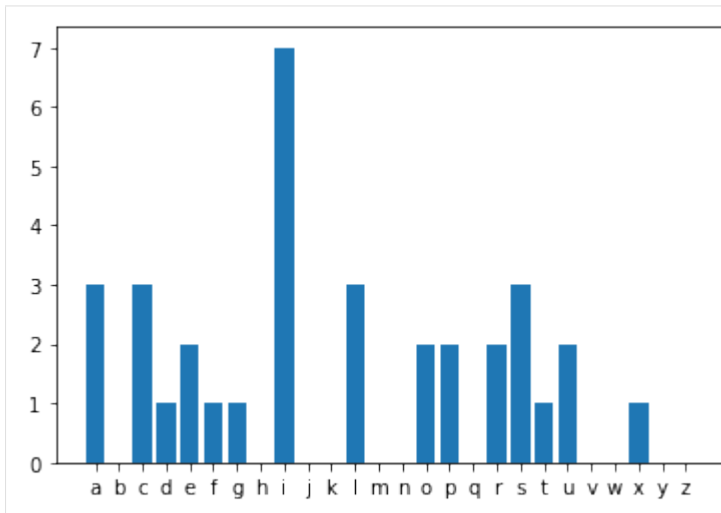
[5]: [<matplotlib.lines.Line2D at 0x7f190315fc10>]
```



```
[6]: left = list(range(26)) # range 関数については以下を参照してください。
    labels = [chr(i + ord('a')) for i in range(26)] # 内包表記については 6-1 を参照ください。

    plt.bar(left,height,tick_label=labels)

[6]: <BarContainer object of 26 artists>
```



10.2 for 文による繰り返しと辞書

辞書の要素にわたって操作を繰り返したい場合も for 文を用います。辞書 `dic1` の全てのキーを変数 `key` に代入しながら、実行文を繰り返すには次のように書きます。

```
for key in dic1.keys():
    実行文
```

for の行の `in` の右辺に辞書のキー一覧を返す `keys` メソッドが使われています。

次の例では、キーを 1 つずつ取り出し、`key` に代入した後、`key` に対応する値を参照しています。

```
[7]: dic1 = {'cat': 3, 'dog': 3, 'elephant': 8}
for key in dic1.keys():
    print('key:', key, ', value:', dic1[key])
```

```
key: cat , value: 3
key: dog , value: 3
key: elephant , value: 8
```

`values` メソッドを使えば（キーを使わずに）値を 1 つずつ取り出すこともできます。

```
[8]: dic1 = {'cat': 3, 'dog': 3, 'elephant': 8}
for value in dic1.values():
    print('value:', value)
```

```
value: 3
value: 3
value: 8
```

`items` メソッドを使えばキーと値を一度に取り出すこともできます。次の例では、`in` の左辺に複数の変数を指定し多重代入を行っています。

```
[9]: dic1 = {'cat': 3, 'dog': 3, 'elephant': 8}
for key, value in dic1.items():
    print('key:', key, 'value:', value)
```

```
key: cat value: 3
key: dog value: 3
key: elephant value: 8
```

実は、辞書の `items` でなくとも、タプルのリストもしくはリストのリストに対しても、同様に複数の変数を指定することができます。

```
[10]: list1 = [[0, 10], [1, 20], [2, 30]]
      for i, j in list1:
          print(i, j)

0 10
1 20
2 30
```

10.3 練習

辞書 `dic1` が引数として与えられたとき、次のような辞書 `dic2` を返す関数 `reverse_lookup2` を作成してください。ただし、`dic1` のキー `key` の値が `value` である場合、`dic2` には `value` というキーが登録されており、その値は `key` であるとします。また、`dic1` は異なる 2 つのキーに対応する値は必ず異なるとします。

以下のセルの ... のところを書き換えて `reverse_lookup2` を作成してください。

```
[11]: def reverse_lookup2(dic1):
      ...
```

上のセルで解答を作成した後、以下のセルを実行し、実行結果が `True` になることを確認してください。

```
[12]: print(reverse_lookup2({'apple': 3, 'pen': 5, 'orange': 7}) == {3: 'apple', 5: 'pen', 7: 'orange'})

False
```

10.4 range

特定の回数の繰り返し処理が必要なときは、`range` 関数を用います。

```
for value in range(j):
    実行文
```

これによって実行文を `j` 回実行します。具体例を見てみましょう。

```
[13]: for value in range(5):
      print('Hi!')

Hi!
Hi!
Hi!
Hi!
Hi!
```

さて、`for` と `in` の間の `value` は変数ですが、`value` には何が入っているのか確認してみましょう。

```
[14]: for value in range(5):
      print(value)

0
1
2
3
4
```


すなわち、`value` は 0 ～ 4 を動くことがわかります。

この `value` の値を用いることでリスト `ln` の要素を順番に用いることもできます。回数としてリストの長さ `len(ln)` を指定します。

```
[15]: ln = ['e', 'd', 'a', 'c', 'f', 'b']
      for value in range(len(ln)):
          print(ln[value])
```

```
e
d
a
c
f
b
```

`range()` 関数は:

1. 引数を 1 つ与えると 0 から引数までの整数列を返します。このとき引数の値は含まれないことに注意してください。
2. 引数を 2 つあるいは 3 つ与えると:
 - 最初の引数を数列の開始 (start)、2 番目を停止 (stop)、3 番目を数列の刻み (step) とする整数列を返します。
 - 3 番目の引数は省略可能で、既定値は 1 となっています。
 - 2 番目の引数の値は含まれないことに注意してください。

以下の例は、0 から 9 までの整数列の総和を計算、印字するプログラムです:

```
[16]: s = 0
      for i in range(10):
          s = s + i

      print(s)
```

```
45
```

以下の例は、1 から 9 までの奇数の総和を計算、印字するプログラムです。

```
[17]: s = 0
      for i in range(1,10,2):
          s = s + i

      print(s)
```

```
25
```

10.5 練習

引数で与えられる 2 つの整数 `x, y` 間 (`x, y` を含む) の整数の総和を返す関数 `sum_n` を `for` 文を利用して作成してください。たとえば、`sum_n(1,3)` の結果は $1 + 2 + 3 = 6$ となります。

以下のセルの ... のところを書き換えて `sum_n` を作成してください。

```
[18]: def sum_n(x, y):
      ...
```

上のセルで解答を作成した後、以下のセルを実行し、実行結果が `True` になることを確認してください。

```
[19]: print(sum_n(1, 3) == 6)
```

```
False
```

10.6 練習

整数 `int_size` を引数として取り、長さが `int_size` であるリスト `ln` を返す関数 `construct_list` を作成してください。ただし、`ln` の `i` 番目の要素は `i` とします (`i` は 0 以上 `int_size-1` 以下の整数)。

以下のセルの ... のところを書き換えて `construct_list(int_size)` を作成してください。

```
[20]: def construct_list(int_size):
```

```
    ...
```

上のセルで解答を作成した後、以下のセルを実行し、実行結果が `True` になることを確認してください。

```
[21]: print(construct_list(10) == [0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
False
```

10.7 range とリスト

`range` 関数は整数列を返しますが、リストを返さないことに注意してください。これは、繰り返し回数の大きな `for` 文などで大きなリストを作ると無駄が大きくなるためです。

`range` 関数を利用して整数列のリストを生成するには、以下のように `list` を関数として用いて、明示的にリスト化する必要があります。

```
[22]: seq_list = list(range(5))
print(seq_list)
```

```
[0, 1, 2, 3, 4]
```

10.8 for 文の入れ子

`for` 文を多重に入れ子（ネスト）して使うこともよくあります。まずは次の例を実行してみてください。

```
[23]: list1 = [['a', 'b', 'c'], ['d', 'e', 'f'], ['g', 'h', 'i'], ['j', 'k', 'l']]
```

```
for i in range(4):
    for j in range(3):
        print('list1 の', i + 1, ' 番目の要素 (リスト) の', j + 1, ' 番目の要素 =',
              ↪ list1[i][j])
```

```
list1 の 1 番目の要素 (リスト) の 1 番目の要素 = a
list1 の 1 番目の要素 (リスト) の 2 番目の要素 = b
list1 の 1 番目の要素 (リスト) の 3 番目の要素 = c
list1 の 2 番目の要素 (リスト) の 1 番目の要素 = d
list1 の 2 番目の要素 (リスト) の 2 番目の要素 = e
list1 の 2 番目の要素 (リスト) の 3 番目の要素 = f
list1 の 3 番目の要素 (リスト) の 1 番目の要素 = g
list1 の 3 番目の要素 (リスト) の 2 番目の要素 = h
list1 の 3 番目の要素 (リスト) の 3 番目の要素 = i
list1 の 4 番目の要素 (リスト) の 1 番目の要素 = j
```

(continues on next page)

(continued from previous page)

```
list1 の 4 番目の要素 (リスト) の 2 番目の要素 = k
list1 の 4 番目の要素 (リスト) の 3 番目の要素 = l
```

$i = 0$ のときに、2 番目 (内側) の for 文において、 j に 0 から 2 までの値が順に代入されて、各場合に `print` が実行されます。その後、2 番目の for 文の実行が終わると、1 番目 (外側) の for 文の最初に戻って、 i の値に新しい値が代入されて、 $i = 1$ になります。その後、再度 2 番目の for 文を実行することになります。このときに、この 2 番目の for 文の中で j には再度、0 から 2 までの値が順に代入されることになります。

決して、「最初に $j = 2$ まで代入したから、もう 2 番目の for 文は実行しない」という訳ではないことに注意してください。一度 for 文の実行を終えて、再度同じ for 文 (上の例でいうところの 2 番目の for 文) に戻ってきた場合、その手続きはまた最初からやり直すことになるのです。

以下のプログラムは、変数 C に組み合わせの数をリストのリストとして求めます。

$C[i][j]$ は、 i 個から j 個を選ぶ組み合わせの数になります。

```
[24]: C = [[1]]
      for i in range(100):
          C.append([1]+[0]*i+[1])
          for j in range(i):
              C[i+1][j+1] = C[i][j] + C[i][j+1]

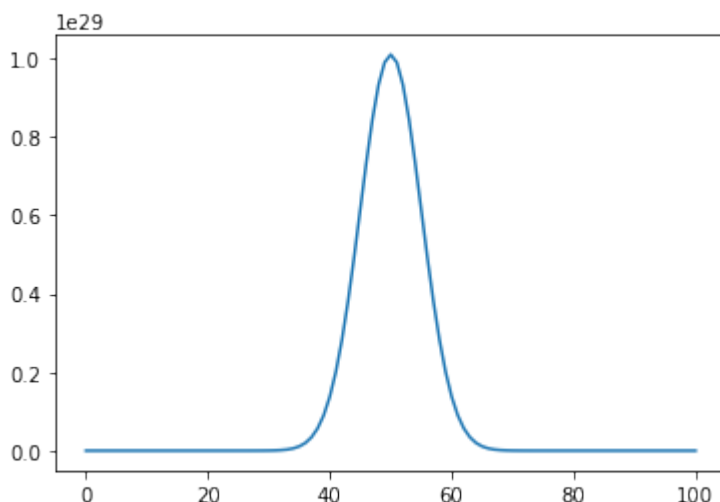
      C[:10]
```

```
[24]: [[1],
      [1, 1],
      [1, 2, 1],
      [1, 3, 3, 1],
      [1, 4, 6, 4, 1],
      [1, 5, 10, 10, 5, 1],
      [1, 6, 15, 20, 15, 6, 1],
      [1, 7, 21, 35, 35, 21, 7, 1],
      [1, 8, 28, 56, 70, 56, 28, 8, 1],
      [1, 9, 36, 84, 126, 126, 84, 36, 9, 1]]
```

$C[100]$ を視覚化してみましょう。

```
[25]: plt.plot(C[100])
```

```
[25]: [<matplotlib.lines.Line2D at 0x7f190300c430>]
```



10.9 練習

次のような関数 `sum_lists` を作成してください。

- `sum_lists` はリスト `list1` を引数とします。
- `list1` の各要素はリストであり、そのリストの要素は数です。
- `sum_lists` は、`list1` の各要素であるリストの総和を求め、それらの総和を足し合せて返します。

以下のセルの ... のところを書き換えて `sum_lists` を作成してください。

```
[26]: def sum_lists(list1):
      ...
```

上のセルで解答を作成した後、以下のセルを実行し、実行結果が `True` になることを確認してください。

```
[27]: print(sum_lists([[20, 5], [6, 16, 14, 5], [16, 8, 16, 17, 14], [1], [5, 3, 5, 7]]) == 158)
      False
```

10.10 練習

リスト `list1` と `list2` が引数として与えられたとき、次のようなリスト `list3` を返す関数 `sum_matrix` を作成してください。

- `list1, list2, list3` は、3つの要素を持ちます。
- 各要素は大きさ3のリストになっており、そのリストの要素は全て数です。
- `list3[i][j]` (ただし、`i` と `j` は共に、0以上2以下の整数) は `list1[i][j]` と `list2[i][j]` の値の和になっています。

以下のセルの ... のところを書き換えて `sum_matrix` を作成してください。

```
[28]: def sum_matrix(list1, list2):
      ...
```

上のセルで解答を作成した後、以下のセルを実行し、実行結果が `True` になることを確認してください。

```
[29]: print(sum_matrix([[1,2,3],[4,5,6],[7,8,9]], [[1,4,7],[2,5,8],[3,6,9]]) == [[2, 6, 10],
      [6, 10, 14], [10, 14, 18]])
      False
```

10.11 for 文の計算量

山口和紀編「情報」第2版（東京大学出版会）には、「**計算量**とは、アルゴリズムをもとにしたプログラムの実行時間を見積もるための指標である。」と書かれています。さらに、「この見積もりは**計算量のオーダー**と呼ばれる非常におおまかな尺度で考える。」と書かれています。（太字は引用時に追加しました。）

たとえば、リストに対する for 文

```
for x in リスト:
    要素 x に対する処理
```

では、「要素に対する処理」が要素の数だけ実行されます。この処理の時間が一定であるとする、要素の数を n としたとき、全体の処理には n に比例する時間がかかります。このことを、**オーダー n** といって、 $O(n)$ と書きます。一方、

```
for x in リスト:
    for y in 同じリスト:
        要素の組み合わせ (x,y) に対する処理
```

という二重のループでは、要素の組み合わせに対する処理が一定時間で終わるとしても、ループの中でループが実行されるので、全体の処理には、 n^2 に比例する時間がかかります。このことを $O(n^2)$ と書きます。 n が 10 倍になったとき、一重のループの実行時間は 10 倍にしかありませんが、二重ループの実行時間は 100 倍になります。 n が 100 倍になったときは、前者は 100 倍ですが後者は 10000 倍になります。

二重ループが明らかでないこともあります。以下の関数は、リストとして与えられたデータの平均と分散を計算するものです。

```
[30]: def average(d):
      s = 0
      for x in d:
          s = s + x
      return s/len(d)

      def variance(d):
          s = 0
          for x in d:
              s = s + (x-average(d))**2
          return s/len(d)
```

ガウス分布から 100 個のデータと 10000 個のデータを生成して分散を計算してみましょう。

```
[31]: import random
      d100 = []
      for i in range(100):
          d100.append(random.gauss(0,10))
      d10000 = []
      for i in range(10000):
          d10000.append(random.gauss(0,10))
```

```
[32]: variance(d100)
```

```
[32]: 102.75403310124948
```

```
[33]: variance(d10000)
```

```
[33]: 99.86713428119583
```

10000 個の場合は相当に時間がかかることがわかります。これは、`variance` の `for` 文の中で `average` を呼んでいるためです。見かけ上は一重ループなのですが、`average` の中にもループがあるため、二重ループと同じ時間がかかります。したがって、10000 個の場合は、100 個の場合に比べて 10000 倍時間がかかります。

ローカル変数を用いて `variance` の定義を書き直してみましょう。

```
[34]: def variance(d):
      av = average(d)
      s = 0
      for x in d:
          s = s + (x-av)**2
      return s/len(d)
```

```
[35]: variance(d100)
```

```
[35]: 102.75403310124948
```

```
[36]: variance(d10000)
```

```
[36]: 99.86713428119583
```

10000 個の場合でも一瞬で実行が終わったことでしょう。この場合、一重のループを 2 回実行しているだけだからです。

10.12 enumerate

for 文の繰り返し処理では、要素の順序を把握したいことがあります。これまで学んだ方法では以下のように書けます。

```
i = 0
for val in some_list:
    print(i, val)
    # 繰り返させたい処理
    i += 1
```

Python では `enumerate()` 関数が用意されており、上のプログラムは以下のように書き換えることができます。

```
for i, val in enumerate(some_list):
    # 繰り返させたい処理
```

2 つの変数 `i`, `val` が指定されています。`i` には `0, 1, 2, …` が順に代入されます。`val` にはリストの要素が順に代入されます。

たとえば、リストの要素をキー、そのインデックスを値とする辞書が欲しい場合は、以下のように書くことができます。

```
[37]: words = ['dog', 'cat', 'mouse']
mapping = {}
for i, w in enumerate(words):
    mapping[w] = i

print(mapping)          # {'dog': 0, 'cat': 1, 'mouse': 2} が得られる。
{'dog': 0, 'cat': 1, 'mouse': 2}
```

10.13 in

Python では for ループでリストを展開する `in` とは別に、2-2 で説明したように、リスト内の要素の有無を検査する `in` 演算子と `not in` 演算子が定義されています。以下のように、`if` 文の条件に `in` が出現した場合、`for` 文とは動作が異なるので注意してください。

```
colors = ['red', 'green', 'blue']
color = 'red'

if color in colors:
    # do something
```

10.14 while 文による繰り返し

while 文では while の後の条件式が False となるまで、実行文グループを繰り返します。

下記のプログラムでは、 $\sum_{x=1}^{10} x$ が total の値となります。

```
[38]: x = 1
total = 0
while x <= 10:
    total += x
    x += 1

print(x, total)

11 55
```

条件式が False になったときに、while 文から抜けているので、終了後の x の値が 11 になっていることに注意してください。なお、上の例を for 文で実行する場合には以下のようになります。

```
[39]: total = 0
for x in range(11):
    total += x

print(x, total)

10 55
```

10.15 制御構造と return 文

return 文は 1-2 で説明したように関数を終了し、値を返す（返値）機能を持ちます。if, for, while といった制御構造の中で return 文が実行された場合、ただちに関数の処理を終了し、その後の処理は行われません。

以下の関数 simple_lsearch は与えられたリスト、lst に myitem と等しいものがあれば True を、なければ False を返します。

- 2 行目の for 文で lst の各要素に対して繰り返しを実行するように指定されています。
- 3 行目の if 文で要素 item が myitem と等しい場合、4 行目の return True でただちに関数を終了しています。
- for 文で全てのリスト要素に対してテストが終わり、等しいものがない場合は、5 行目の return False が実行されます。

```
[40]: def simple_lsearch(lst, myitem):
    for item in lst:
        if item == myitem:
            return True
    return False
```

10.16 break 文

break 文は、for 文もしくは while 文の実行文グループで利用可能です。break 文は実行中のプログラムで最も内側の繰り返し処理を中断し、そのループを終了させる目的で利用されます。以下のプログラムは、初項 256、公比 1/2、の等比級数の和を求めるものです。ただし、総和が 500 をこえれば打ち切られます。

```
[41]: x = 256
total = 0
while x > 0:
    if total > 500:
        break          # 500 を超えれば while ループを抜ける
    total += x
    x = x // 2          # // は少数点以下を切り捨てる除算

print(x, total)

4 504
```

10.17 練習

文字列 str1 と str2 が引数として与えられたとき、str2 が str1 を部分文字列として含むかどうか判定する関数 simple_match を作成してください。具体的には、str2 を含む場合、その部分文字列が開始される str1 のインデックスを返値として返してください。str2 を含まない場合、-1 を返してください。ただし、simple_match の中で文字列のメソッドやモジュール（正規表現など）を使ってはいけません。

以下のセルの ... のところを書き換えて simple_match を作成してください。

```
[42]: def simple_match(str1, str2):
    ...
```

上のセルで解答を作成した後、以下のセルを実行し、実行結果が True になることを確認してください。

```
[43]: print(simple_match('location', 'cat') == 2)
print(simple_match('soccer', 'cat') == -1)
print(simple_match('category', 'cat') == 0)
print(simple_match('carpet', 'cat') == -1)

False
False
False
False
```

10.18 continue 文

continue 文は break 文同様に、for および while ループの実行文グループで利用可能です。continue 文は実行中のプログラムで最も内側の繰り返し処理を中断し、次のループの繰り返しの処理を開始します。

下記のプログラムでは、colors リストの 'black' は印字されませんが 'white' は印字されます。

```
[44]: colors = ['red', 'green', 'blue', 'black', 'white']
for c in colors:
    if c == 'black':
        continue
    print(c)
```



```
red
green
blue
white
```

10.19 ▲ for 文と while 文における else

for 文および while 文では `else` を書くこともできます。この実行文グループは、ループの最後に一度だけ実行されます。

```
[45]: colors = ['red', 'green', 'blue', 'black', 'white']
      for c in colors:
          if c == 'black':
              continue
          print(c)
      else:
          print('')
```

```
red
green
blue
white
```

for 文および while 文の `else` ブロックの内容は `continue` で終了したときは実行されますが、一方で `break` でループを終了したときは実行されません。

10.20 pass 文

Python では空の実行文グループは許されていません。一方で、空白のコードブロックを用いることでプログラムが読みやすくなる場合があります。たとえば以下の、`if ... elif ... else` プログラムはエラーとなります。

```
[46]: x = -1
      if x < 0:
          print('x is positive')
      elif x == 0:
          # IndentationError: expected an indented block
      elif 0 < x < 5:
          print('x is positive and smaller than 5')
      else:
          print('x is positive and larger than or equal to 5')
```

```
Input In [46]
elif 0 < x < 5:
^
```

```
IndentationError: expected an indented block
```

なにもしない `pass` 文を用いて、以下のように書き換えることで正常に実行されます。

```
[47]: x = -1
      if x < 0:
          print('x is positive')
      elif x == 0:
```

(continues on next page)

(continued from previous page)

```
# no error
pass
elif 0 < x < 5:
    print('x is positive and smaller than 5')
else:
    print('x is positive and larger than or equal to 5')
x is positive
```

10.21 練習

以下のプログラムでは 1 秒おきに print が永久に実行されます。

```
from time import sleep

while True:
    print('Yeah!')
    sleep(1)
```

10 回 print が実行された後に while 文を終了するように書き換えてください。実行中のセルを停止させるには、ストップボタンが使えます。

[]:

10.22 練習

英語の文章からなる文字列 `str_engsentence` が引数として与えられたとき、`str_engsentence` 中に含まれる 3 文字以上の全ての英単語を要素とするリストを返す関数 `collect_engwords` を作成してください。ただし、同じ単語を重複して含んでいて構いません。

以下のセルの ... のところを書き換えて `collect_engwords(str_engsentence)` を作成してください。

```
[48]: def collect_engwords(str_engsentence):
      ...
```

上のセルで解答を作成した後、以下のセルを実行し、実行結果が `True` になることを確認してください。

```
[49]: print(collect_engwords('Unfortunately no, it requires something with a little more,
      ↪kick, plutonium.') == ['Unfortunately', 'requires',
      'something', 'with', 'little', 'more', 'kick', 'plutonium'])
False
```

10.23 練習

2 つの同じ大きさのリストが引数として与えられたとき、2 つのリストの奇数インデックスの要素を入れ替えて、その結果得られる 2 つのリストをタプルにして返す関数 `swap_lists` を作成してください（ただし、0 は偶数として扱うものとします）。与えられたリストは破壊しても構いません。

以下のセルの ... のところを書き換えて `swap_lists(ln1, ln2)` を作成してください。

```
[50]: def swap_lists(ln1, ln2):
      ...
```

上のセルで解答を作成した後、以下のセルを実行し、実行結果が `True` になることを確認してください。

```
[51]: print(swap_lists([1, 2, 3, 4, 5], ['a', 'b', 'c', 'd', 'e'])) == ([1, 'b', 3, 'd', 5],
↪ ['a', 2, 'c', 4, 'e']))
```

False

10.24 練習

文字列 `str1` を引数として取り、`str1` の中に含まれる大文字の数を返す関数 `count_capitalletters` を作成してください。

以下のセルの ... のところを書き換えて `count_capitalletters(str1)` を作成してください。

```
[52]: def count_capitalletters(str1):
      ...
```

上のセルで解答を作成した後、以下のセルを実行し、実行結果が `True` になることを確認してください。

```
[53]: print(count_capitalletters('Que Ser^^c3^^a1, Ser^^c3^^a1') == 3)
```

False

10.25 練習

長さが 3 の倍数である文字列 `str_augc` が引数として与えられたとき、`str_augc` を長さ 3 の文字列に区切り、それらを順に格納したリストを返す関数 `identify_codons` を作成してください。

以下のセルの ... のところを書き換えて `identify_codons(str_augc)` を作成してください。

```
[54]: def identify_codons(str_augc):
      ...
```

上のセルで解答を作成した後、以下のセルを実行し、実行結果が `True` になることを確認してください。

```
[55]: print(identify_codons('CCCCGGCACCT') == ['CCC', 'CCG', 'GCA', 'CCT'])
```

False

10.26 練習

正の整数 `int1` が引数として与えられたとき、`int1` の値の下桁から 3 桁毎にコンマ (,) を入れた文字列を返す関数 `add_commas` を作成してください。ただし、数の先頭にコンマを入れる必要はありません。

以下のセルの ... のところを書き換えて `add_commas` を作成してください。

```
[56]: def add_commas(int1):
      ...
```

上のセルで解答を作成した後、以下のセルを実行し、全ての実行結果が `True` になることを確認してください。

```
[57]: print(add_commas(14980) == '14,980')
      print(add_commas(3980) == '3,980')
      print(add_commas(298) == '298')
      print(add_commas(1000000) == '1,000,000')
```

```
False
False
False
False
```

10.27 練習

リスト `list1` が引数として与えられ、次のような文字列 `str1` を返す関数 `sum_strings` を作成してください。

`list1` は `k` 個の要素を持つとします（ただし、`k` は正の整数）。`list1` の要素が文字列でなければ文字列に変換してください。その上で、`list1` の 1 番目から `k-2` 番目の各要素の後ろにコンマとスペースからなる文字列 `,` を加え、`k-1` 番目の要素の後ろには、`' and '` を加え、1 番目から `k` 番目までの要素を繋げた文字列を `str1` とします。

以下のセルの ... のところを書き換えて `sum_strings` を作成してください。

```
[58]: def sum_strings(list1):
      ...
```

上のセルで解答を作成した後、以下のセルを実行し、実行結果が `True` になることを確認してください。

```
[59]: print(sum_strings(['a', 'b', 'c', 'd']) == 'a, b, c and d')
      print(sum_strings(['a']) == 'a')
      print(sum_strings([1, 2, 3]) == '1, 2 and 3')
```

```
False
False
False
```

10.28 練習

辞書 `dic1` と長さ 10 以下の文字列 `str1` が引数として与えられたとき、以下のように `dic1` を変更する関数 `handle_collision2` を作成してください。ただし、`dic1` のキーは、1 以上 10 以下の整数、キーに対応する値は文字列とします。

1. `dic1` に `str1` の長さ `n` がキーとして登録されていない場合、`dic1` に キー `n`、`n` に対応する値 `str1` を登録します。
2. `dic1` に `str1` の長さ `n` がキーとして登録されている場合、`i` の値を `n+1, n+2, ...` と 1 つずつ増やしていき、`dic1` にキーとして登録されていない値 `i` を探します。キーとして登録されていない値 `i` が見つかった場合、その `i` をキー、`i` に対応する値として `str1` を登録してください。ただし、`i` を 10 まで増やしても登録されていない値が見つからない場合は、`i` を 1 に戻した上で `i` を増やす作業を続行してください。
3. 2 の手順によって、登録可能な `i` が見つからなかった場合、`dic1` を変更しません。

以下のセルの ... のところを書き換えて `handle_collision2(dic1, str1)` を作成してください。

```
[60]: def handle_collision2(dic1, str1):
      ...
```

上のセルで解答を作成した後、以下のセルを実行し、実行結果が `True` になることを確認してください。

```
[61]: dic1_orig = {6: 'Styles', 4: 'Link', 7: 'Ackroyd'}
      handle_collision2(dic1_orig, 'Big Four')
      print(dic1_orig == {6: 'Styles', 4: 'Link', 7: 'Ackroyd', 8: 'Big Four'})
```

(continues on next page)

(continued from previous page)

```
dic1_orig = {6: 'Styles', 4: 'Link', 7: 'Ackroyd', 8: 'Big Four', 10: 'Blue Train', 9:
'End House'}
handle_collision2(dic1_orig, 'Edgware')
print(dic1_orig == {6: 'Styles', 4: 'Link', 7: 'Ackroyd', 8: 'Big Four', 10: 'Blue
↪Train', 9: 'End House', 1: 'Edgware'})
dic1_orig = {6: 'Styles', 4: 'Link', 7: 'Ackroyd', 8: 'Big Four', 10: 'Blue Train', 9:
'End House', 1: 'Edgware', 2: 'Orient', 3: 'Three Act', 5: 'Clouds'}
handle_collision2(dic1_orig, 'ABC')
print(dic1_orig == {6: 'Styles', 4: 'Link', 7: 'Ackroyd', 8: 'Big Four', 10: 'Blue
↪Train', 9: 'End House', 1: 'Edgware', 2: 'Orient', 3: 'Three Act', 5: 'Clouds'})
```

False
False
True

10.29 練習

整数を最初の要素、文字列をその次の要素とするリスト（これを子リストと呼びます）を要素とするリスト `list1` が引数として与えられたとき、次のような辞書 `dic1` を返す関数 `handle_collision3` を作成してください。

- 各子リスト `list2` に対して、`dic1` のキーは `list2` の最初の要素である整数とし、そのキーに対応する値は次の要素である文字列とします。
- 2 つ以上の子リストの最初の要素が同じ整数である場合、`list1` においてより小さいインデックスを持つ子リストの文字列を、その整数のキーに対応する値とします。

以下のセルの ... のところを書き換えて `handle_collision3(list1)` を作成してください。

```
[62]: def handle_collision3(list1):
      ...
```

上のセルで解答を作成した後、以下のセルを実行し、実行結果が `True` になることを確認してください。

```
[63]: print(handle_collision3([[3, 'Richard III'], [1, 'Othello'], [2, 'Tempest'], [3,
↪'King John'], [4, 'Midsummer'], [1, 'Lear']])) == {1: 'Othello', 2: 'Tempest', 3:
↪'Richard III', 4: 'Midsummer'})
```

False

10.30 練習の解答

```
[64]: def reverse_lookup2(dic1):
      dic2 = {} #辞書を初期化する
      for key, value in dic1.items():
          dic2[value] = key
      return dic2
      #reverse_lookup2({'apple':3, 'pen':5, 'orange':7})
```

```
[65]: def sum_n(x,y):
      sum = 0
      for i in range(x, y + 1):
          sum = sum + i
      return sum
      #sum_n(1,3)
```

```
[66]: def construct_list(int_size):
    ln = int_size * [0]
    for i in range(int_size):
        ln[i] = i
    return ln
#construct_list(10)
```

```
[67]: def sum_lists(list1):
    total = 0
    for list2 in list1: # for j in range(len(list1)):と list2 = list1[j] としてもよい
        #print(list2)
        for i in range(len(list2)):
            #print(i, list2[i])
            total += list2[i]
    return total
```

```
[68]: def sum_matrix(list1, list2):
    list3 = [[0,0,0],[0,0,0],[0,0,0]] #結果を格納するリストを初期化する（これがない場合も
    #試してみてください）
    for i in range(3):
        for j in range(3):
            list3[i][j] += list1[i][j] + list2[i][j]
            #print(i, j, list1[i][j], '+', list2[i][j], '=', list3[i][j])
    return list3
#sum_matrix([[1,2,3],[4,5,6],[7,8,9]], [[1,4,7],[2,5,8],[3,6,9]])
```

```
[69]: def simple_match(str1, str2):
    for i in range(len(str1)-len(str2)+1):
        j = 0
        while j < len(str2) and str1[i+j] == str2[j]: #str1 と str2 が一致している限り
            #ループ（ただし、j が str2 の長さ以上にならないようにする）#この条件がないと…？
            j += 1
        if j == len(str2): #str2 の最後まで一致しているとその条件が成立
            return i
    return -1
#for 文による別解
#def simple_match(str1, str2):
#    for i in range(len(str1)-len(str2)+1):
#        #print('i=', i)
#        fMatch = True#マッチ判定
#        for j in range(len(str2)):
#            #print('j=', j, 'str1[i+j]=', str1[i+j], ' str2[j]=', str2[j])
#            if str1[i+j] != str2[j]:#str2 が終了する前に一致しない箇所があるかどうか
#                fMatch = False
#                break
#        if fMatch:
#            return i
#    return -1
#print(simple_match('location', 'cat') == 2)
#print(simple_match('soccer', 'cat') == -1)
#print(simple_match('category', 'cat') == 0)
#print(simple_match('carpet', 'cat') == -1)
```

10.31 練習の解説

下のセルは、繰り返し回数として `count` 変数を利用した解答例です。回数を理解しやすくするため `print()` 関数で `count` 変数も印字しています。

```
[70]: from time import sleep

count = 0
while True:
    print('Yeah!', count)
    count += 1
    if(count >= 10):
        break
    sleep(1)
```

```
Yeah! 0
Yeah! 1
Yeah! 2
Yeah! 3
Yeah! 4
Yeah! 5
Yeah! 6
Yeah! 7
Yeah! 8
Yeah! 9
```

10.32 練習の解答

```
[71]: def collect_engwords(str_engsentences):
    list_punctuation = ['.', ',', ':', ';', '!', '?']
    for j in range(len(list_punctuation)): #list_punctuation 中の文字列（この場合、句読点）を空文字列に置換する
        str_engsentences = str_engsentences.replace(list_punctuation[j], '')
        #print(str_engsentences)
    list_str1 = str_engsentences.split(' ')
    list_str2 = []
    for j in range(len(list_str1)):
        if len(list_str1[j]) >= 3:
            list_str2.append(list_str1[j])
    return list_str2
#collect_engwords('Unfortunately no, it requires something with a little more kick,
↳plutonium.')
```

```
[72]: def swap_lists(ln1, ln2):
    for j in range(len(ln1)):
        if j % 2 == 1:
            ln1[j], ln2[j] = ln2[j], ln1[j]
    return ln1, ln2
#swap_lists([1, 2, 3, 4, 5], ['a', 'b', 'c', 'd', 'e'])
```

```
[73]: def count_capitalletters(str1):
    int_count = 0
    for i in range(len(str1)):
        str2 = str1[i].upper()
        str3 = str1[i].lower()
```

(continues on next page)

(continued from previous page)

```

    if str1[i] == str2 and str2 != str3: #前者の条件で大文字であることを、後者の条件で
句読点などでないことを判定する
        int_count += 1
    return int_count
#count_capitalletters('Que Ser^^c3^^a1, Ser^^c3^^a1')

```

```

[74]: def identify_codons(str_augc):
    str_codons = []
    int_codonnum = int(len(str_augc)/3)
    for i in range(int_codonnum):
        str_codons.append(str_augc[i*3: i*3+3])
    return str_codons
#identify_codons('CCCCGGCACCT')

```

```

[75]: def add_commas(int1):
    list1 = list(str(int1)) #文字列に変換し、更にそれを1文字ずつリストに格納する
    str1 = ''
    ccnt = 1 #3の倍数の位を調べるのに使う
    for i in range(len(list1)-1, -1, -1): #1の位の値から、大きい方の位の値に向かって処理
        行う
        str1 = list1[i] + str1
        if ccnt % 3 == 0 and i != 0: #3の倍数の位の前であり、一番大きい位でないならば
            str1 = ',' + str1 #コンマをうつ
            ccnt += 1
    return str1
#print(add_commas(14980) == '14,980')
#print(add_commas(2980) == '2,980')
#print(add_commas(298) == '298')
#print(add_commas(1000000) == '1,000,000')

```

```

[76]: def sum_strings(list_str):
    str1 = ''
    for i in range(len(list_str)):
        if i < len(list_str) - 2: #後ろから3番目までの要素
            str1 = str1 + str(list_str[i]) + ', '
        elif i == len(list_str) - 2: #後ろから2番目の要素
            str1 += str(list_str[i]) + ' and '
        else: #一番後ろの要素
            str1 += str(list_str[i])
    return str1
#sum_strings(['a', 'b', 'c', 'd'])
#sum_strings(['a'])

```

```

[77]: def handle_collision2(dic1, str1):
    n = len(str1)
    for i in range(n, 11):
        if dic1.get(i) is None: # == None でもよい
            dic1[i] = str1
            return
    for i in range(1, n):
        if dic1.get(i) is None: # == None でもよい
            dic1[i] = str1
            return

```



```
[78]: def handle_collision3(list1):
      dic1 = {} # 空の辞書を作成する
      for i in range(len(list1)):
          list2 = list1[i]
          if dic1.get(list2[0]) is None: # == None でもよい
              dic1[list2[0]] = list2[1]
      return dic1
#handle_collision3([[3, 'Richard III'], [1, 'Othello'], [2, 'Tempest'], [3, 'King John'],
↪[4, 'Midsummer'], [1, 'Lear']])
```

```
[ ]:
```

3-3. 関数

関数について改めて説明します。

参考:

- <https://docs.python.org/ja/3/tutorial/controlflow.html#defining-functions>

11.1 関数の定義

関数は処理（手続きの流れ）をまとめた再利用可能なコードです。関数には以下の特徴があります。

- 名前を持つ。
- 手続きの流れを含む。
- 返値（明示的あるいは非明示的に）を返す。

`len()` や `sum()` などの組み込み関数は関数の例です。

まず、関数の定義を試みましょう。関数を定義するには `def` を用います。

```
[1]: # 'Hello' を表示する関数 greeting
def greeting():
    print('Hello')
```

関数を定義したら、それを呼び出すことができます。

```
[2]: # 関数 greeting を呼び出し
greeting()

Hello
```

関数定義の一般形は以下の通りです。

```
def 関数名 (引数):
    関数本体
```

1 行名はヘッダと呼ばれ、**関数名**はその関数を呼ぶのに使う名前、**引数**はその関数へ渡す変数の一覧です。変数がない場合もあります。

関数本体はインデントした上で、処理や手続きの流れを記述します。

11.2 引数

関数を定義する際に、ヘッダの括弧の中に関数へ渡す変数の一覧を記述します。これらの変数は関数のローカル変数となります。ローカル変数とはプログラムの一部（ここでは関数内）でのみ利用可能な変数です。

```
[3]: #引数 greeting_local に渡された値を表示する関数 greeting
def greeting(greeting_local):
    print(greeting_local)
```

関数を呼び出す際に引数に値を渡すことで、関数は受け取った値を処理することができます。

```
[4]: #関数 greeting に文字列 'Hello' を渡して呼び出し
greeting('Hello')
```

```
Hello
```

このようにして引数に渡される値のことを、**実引数**（**argument**）と呼ぶことがあります。実引数に対して、ここまで説明してきた引数（ローカル変数である引数）は、**仮引数**（**parameter**）と呼ばれます。参考：公式 FAQ：実引数と仮引数の違いは何ですか？

実引数のことを引数と呼ぶこともありますので、注意してください。

11.3 返値

関数は受け取った引数を元に処理を行い、その結果の**返値**（1-2 で説明済み）を返すことができます。

返値は、**return** で定義します。関数の返値がない場合は、**None** が返されます。**return** が実行されると、関数の処理はそこで終了するため、次に文があっても実行はされません。また、ループなどの繰り返し処理の途中でも **return** が実行されると処理は終了します。関数の処理が最後まで実行され、返値がない場合は最後に **return** を実行したことと同じになります。

return の後に式がない場合は、**None** が返されます。（**return** が実行されずに関数の最後まで来たときも同様です。）**return** を式なしで実行することで、関数の処理を途中で抜けることができます。また、このような関数は、与えられた配列を破壊的に変更するなど、呼び出した側に何らかの変化を及ぼす際にも用いられます。

```
[5]: #引数 greeting_local に渡された値を返す関数 greeting
def greeting(greeting_local):
    return greeting_local

#関数 greeting に文字列 'Hello' を渡して呼び出し
greeting('Hello')
```

```
[5]: 'Hello'
```

```
[6]: #入力の平均を計算して返す関数 average
def average(nums):
    #組み込み関数の sum() と len() を利用
    return sum(nums)/len(nums)

#関数 average に数字のリストを渡して呼び出し
average([1,3,5,7,9])
```

```
[6]: 5.0
```

関数の返値を変数に代入することもできます。

```
[7]: #関数 greeting の返値を変数 greet に代入
      greet = greeting('Hello')
      greet
```

```
[7]: 'Hello'
```

11.4 複数の引数

関数は任意の数の引数を受け取ることができます。複数の引数を受け取る場合は、引数をコンマで区切ります。これらの引数名は重複しないようにしましょう。

```
[8]: #3つの引数それぞれに渡された値を表示する関数 greeting
      def greeting(en, fr, de):
          print(en + ', ' + fr + ', ' + de)

      #関数 greeting に3つの引数を渡して呼び出し
      greeting('Hello', 'Bonjour', 'Guten Tag')
```

```
Hello, Bonjour, Guten Tag
```

関数は異なる型であっても引数として受け取ることができます。

```
[9]: #文字列と数値を引数として受け取る関数 greeting
      def greeting(en, number, name):
          #文字列に数を掛け算すると、文字列を数の回だけ繰り返すことを指定します
          print(en*number+' '+name)

      #関数 greeting に文字列と数値を引数として渡して呼び出し
      greeting('Hello',3, 'World')
```

```
HelloHelloHello,World
```

11.5 変数とスコープ

関数の引数や関数内で定義される変数はローカル変数のため、それらの変数は関数の外からは参照できません。

```
[10]: #引数 greeting_local に渡された値を表示する関数 greeting
      def greeting(greeting_local):
          print(greeting_local)

      greeting('Hello')

      #ローカル変数（関数 greeting の引数）greeting_local を参照
      greeting_local
```

```
Hello
```

```
-----
NameError                                Traceback (most recent call last)
Input In [10], in <module>
      5 greeting('Hello')
      7 #ローカル変数（関数 greeting の引数）greeting_local を参照
----> 8 greeting_local

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

一方、変数が**グローバル変数**であれば、それらの変数は関数の外からも中からも参照できます。グローバル変数とはプログラム全体、どこからでも利用可能な変数です。

```
[11]: #グローバル変数 greeting_global の定義
greeting_global = 'Hello'

#グローバル変数 greeting_global の値を表示する関数 greeting
def greeting():
    print(greeting_global)

greeting()

#グローバル変数 greeting_global を参照
greeting_global

Hello
```

```
[11]: 'Hello'
```

グローバル変数と同じ名前の変数を関数内で定義すると、それは通常はグローバル変数とは異なる、関数内のみで利用可能な**ローカル変数**の定義として扱われます。グローバル変数と同じ名前の引数を用いる場合も同様です。

```
[12]: #グローバル変数 greeting_global と同じ名前の変数に値を代入する関数 greeting
def greeting():
    greeting_global = 'Bonjour'
    print(greeting_global)

greeting()

#変数 greeting_global を参照
greeting_global

Bonjour
```

```
[12]: 'Hello'
```

しかし、グローバル変数と同名のローカル変数を定義することは、一般に注意が必要です。何故なら、ローカル変数としての定義を含む関数内では、同名のグローバル変数を参照できないからです。たとえば、次のコードは、Hello と Bonjour が順に印字することを期待するかもしれませんが、

```
[13]: def greeting():
        print(greeting_global) # 最初の参照
        greeting_global = 'Bonjour' # ローカル変数の定義
        print(greeting_global)

greeting()

-----
UnboundLocalError                                Traceback (most recent call last)
Input In [13], in <module>
      3     greeting_global = 'Bonjour' # ローカル変数の定義
      4     print(greeting_global)
----> 6     greeting()

Input In [13], in greeting()
      1 def greeting():
----> 2     print(greeting_global) # 最初の参照
      3     greeting_global = 'Bonjour' # ローカル変数の定義
      4     print(greeting_global)
```

(continues on next page)

(continued from previous page)

```
UnboundLocalError: local variable 'greeting_global' referenced before assignment
```

最初の `greeting_global` の参照でエラーになります。これは、関数内に `greeting_global` の定義があると、その関数内どの場所でも `greeting_global` がローカル変数として参照されるためです。最初の参照時には、ローカル変数の `greeting_global` が未定義なので、エラーが生じます。

このように、グローバル変数と同じ名前のローカル変数を使おうとするのは間違いの元です。グローバル変数と名前が衝突しないように、ローカル変数を定義しましょう。

11.5.1 ▲ global 宣言

関数内ではグローバル変数が更新されないのが基本です。しかし、どうしても関数内でグローバル変数を更新したいときには、`global` 宣言を使って更新したいグローバル変数を指定します。

[14]: #グローバル変数 `greeting_global` に値を代入する関数 `greeting`

```
def greeting():
    global greeting_global
    greeting_global = 'Bonjour'
    print(greeting_global)
```

```
greeting()
```

```
##変数 greeting_global を参照
greeting_global
```

```
Bonjour
```

[14]: 'Bonjour'

`global` 宣言された変数名は、関数内で常にグローバル変数として参照されます。これを濫用すると間違いの元になるので、原則として利用しないようにしましょう。

11.6 ▲キーワード引数

上記の一般的な引数（位置引数とも呼ばれます）では、事前に定義した引数の順番に従って、関数は引数を受け取る必要があります。

キーワード付き引数（**キーワード引数**）を使うと、関数は引数の変数名とその値の組みを受け取ることができます。その際、引数は順不同で関数に渡すことができます。

[15]: #文字列と数値を引数として受け取る関数 `greeting`

```
def greeting(en, number, name):
    print(en*number+' '+name)
```

```
#関数 greeting に引数の変数名とその値の組みを渡して呼び出し
greeting(en='Hello', name='Japan', number=2)
```

```
HelloHello, Japan
```

位置引数とキーワード引数を合わせて使う場合は、最初に位置引数を指定する必要があります。

[16]: #位置引数とキーワード引数を組み合わせた関数 `greeting` の呼び出し

```
greeting('Hello', name='Japan', number=2)
```

```
HelloHello, Japan
```

11.7 ▲引数の初期値

関数を呼び出す際に、引数が渡されない場合に、**初期値**を引数として渡すことができます。

初期値のある引数に値を渡したら、関数はその引数の初期値の代わりに渡された値を受け取ります。

初期値を持つ引数は、位置引数の後に指定する必要があります。

```
[17]: #引数の初期値（引数の変数 en に対する Hello）を持つ関数 greeting
def greeting(name, en='Hello'):
    print(en+' ', '+name)

#引数の初期値を持つ関数 greeting の呼び出し
greeting('World')

Hello, World
```

11.8 ▲可変長引数

仮引数の前に * を付けて関数を定義すると、複数の引数をタプルとして受け取ることができます。呼び出す側は、引数の個数を変えることができます。一般に、個数が可変の引数は**可変長引数**と呼ばれます。

```
[18]: #可変長の引数を受け取り、それらを表示する関数 greeting
def greeting(*args):
    print(args)

#可変長の引数を受け取る関数 greeting に複数の引数を渡して呼び出し
greeting('Hello', 'Bonjour', 'Guten Tag')

('Hello', 'Bonjour', 'Guten Tag')
```

リストやタプルの要素を可変長引数として関数に渡す場合は、* をリストやタプルの前につけて渡します。

```
[19]: #リスト型オブジェクト greeting_list を関数 greeting に渡して呼び出し
greeting_list = ['Hello', 'Bonjour', 'Guten Tag']
greeting(*greeting_list)

('Hello', 'Bonjour', 'Guten Tag')
```

11.9 ▲辞書型の可変長引数

仮引数の前に ** を付けて関数を定義すると、複数のキーワード引数を辞書として受け取ることができます。辞書として受け取られる引数は、**辞書型の可変長引数**と呼ばれます。

```
[20]: #可変長のキーワード引数を受け取り、それらを表示する関数 greeting
def greeting(**kwargs):
    print(kwargs)

#可変長のキーワード引数を受け取る関数 greeting に複数の引数を渡して呼び出し
greeting(en='Hello', fr='Bonjour', de='Guten Tag')

{'en': 'Hello', 'fr': 'Bonjour', 'de': 'Guten Tag'}
```

辞書の各キーと値を複数のキーワード引数として関数に渡す場合は、** をその辞書の前につけて渡します。

```
[21]: #辞書型オブジェクト greeting_dict を関数 greeting に渡して呼び出し
greeting_dict = {'en': 'Hello', 'fr': 'Bonjour', 'de': 'Guten Tag'}
greeting(**greeting_dict)

{'en': 'Hello', 'fr': 'Bonjour', 'de': 'Guten Tag'}
```

11.10 ▲引数の順番

位置引数、初期値を持つ引数、可変長引数、辞書型の可変長引数は、同時に指定することができますが、その際、これらの順番で指定する必要があります。

```
def 関数名(位置引数, 初期値を持つ引数, 可変長引数, 辞書型の可変長引数)
```

```
[22]: #位置引数、初期値を持つ引数、可変長引数、辞書型の可変長引数
#それぞれを引数として受け取り、それらを表示する関数 greeting
def greeting(greet, en='Hello', *args, **kwargs):
    print(greet)
    print(en)
    print(args)
    print(kwargs)

#可変長引数へ渡すリスト
greeting_list = ['Bonjour']

#辞書型の可変長引数へ渡す辞書
greeting_dict = {'de': 'Guten Tag'}

#関数 greeting に引数を渡して呼び出し
greeting('Hi', 'Hello', *greeting_list, **greeting_dict)

Hi
Hello
('Bonjour',)
{'de': 'Guten Tag'}
```

11.11 ▲変数としての関数

関数は変数でもあります。既存の変数と同じ名前の関数を定義すると、元の変数はその新たな関数を参照するものとして変更されます。一方、既存の関数と同じ名前の変数を定義すると、元の関数名の変数はその新たな変数を参照するものとして変更されます。

```
[23]: #グローバル変数 greeting_global の定義と参照
greeting_global = 'Hello'
type(greeting_global)
```

```
[23]: str
```

```
[24]: #グローバル変数 greeting_global と同名の関数の定義
#変数 greeting_global は関数を参照する
def greeting_global():
    print('This is the greeting_global function')

type(greeting_global)
```



```
[24]: function
```

4-1. ファイル入出力の基本

ファイル入出力の基本について説明します。

参考

- <https://docs.python.org/ja/3/tutorial/inputoutput.html#reading-and-writing-files>

12.1 ファイルのオープン

ファイルから文字列を読み込んだり、ファイルに書き込んだりするには、まず、`open()` という関数によってファイルをオープンする（開く）必要があります。

```
[1]: f = open('sample.txt', 'r')
```

変数 `f` には、ファイルを読み書きするためのデータが入ります。これを**ファイルオブジェクト**と呼びます。

'sample.txt' はファイル名で、そのファイルの絶対パス名か、このノートブックからの相対パス名を指定します。

ここでは、sample.txt という名前のファイルがこのノートブックと同じディレクトリにあることを想定しています。

たとえば、novel.txt というファイルが、ノートブックの 1 段上のディレクトリ（このディレクトリが入っているディレクトリ）にあるならば、'../novel.txt' と指定します。ノートブックの 1 段上のディレクトリに置かれている data というディレクトリにあるならば、'../data/novel.txt' となります（4-3 にもう少し詳しい解説があります）。

'r' はファイルをどのモードで開くかを指しており、'r' は**読み込みモード**を意味します。このモードで開いたファイルに書き込みすることはできません。

よく使われるモードは、次の 3 種類です。

引数 | モード

'r' | 読み込み 'w' | 書き込み 'a' | 追記

モードの引数がなかった場合は、'r' であると解釈されます。書き込みについては後でも説明します。

モードの詳細は[公式ドキュメント](#)を参照。

12.2 ファイルのクローズ

ファイルオブジェクトを使い終わったら、原則として、`close()` メソッドを呼び出して、クローズする（閉じる）必要があります。

```
[2]: f.close()
```

`close()` を呼び出さずに放置すると、そのファイルがまだ使用中だと認識されてしまいます。これは、同じファイルを利用しようとする他のプログラムの働きを阻害します。（個室のトイレをイメージしてください。）

`close()` の呼び出しは重要ですが、忘れがちなものでもあります。後述する `with` 文を使うのが安全です。

12.3 行の読み込み

ファイルオブジェクトには、`readline()` というメソッドを適用することができます。ファイルから新たに1行を読んで文字列として返します。この「1行」というのは、正確には、ファイルの先頭もしくは改行文字の次の文字から、ファイルの終わりもしくは改行文字までの文字列です。1行は必ずしも改行文字で終わらないという点に注意して下さい。

ファイルの終わりに来たとき、`readline()` は `''` という空文字列を返します。

以下のようにして `readline()` を使ってファイルを行単位で読んでみましょう。

ファイルを読み終わると空文字列が返ることを確認してください。

```
[3]: f = open('sample.txt', 'r')
```

```
[4]: f.readline()
```

```
[4]: 'Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor
↳incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud
↳exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.\n'
```

```
[5]: f.readline()
```

```
[5]: 'Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu
↳fugiat nulla pariatur.\n'
```

```
[6]: f.readline()
```

```
[6]: 'Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt
↳mollit anim id est laborum.\n'
```

```
[7]: f.readline()
```

```
[7]: ''
```

```
[8]: f.close()
```

`readline()` メソッドの呼び出しは、ファイルオブジェクトを消費します。改めて読み出したいときには、再度オープンして新しいオブジェクトを使ってください。

12.4 練習

文字列 `name` をファイル名とするファイルの最後の行を文字列として返す関数 `last_line(name)` を定義してください。

```
[9]: def last_line(name):
    ...
```

上のセルで解答を作成した後、以下のセルを実行し、実行結果が `True` になることを確認してください。

```
[10]: print(last_line('sample.txt')== "Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.\n")
False
```

12.5 ファイル全体の読み込み

ファイル全体を一括で読み込んで、1つの文字列を取得したいときには、`read()` メソッドを利用します。

```
[11]: f = open('sample.txt', 'r')
f.read()
[11]: 'Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.\nDuis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur.\nExcepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.\n'
```

一度 `read()` を呼ぶと、ファイルの終端に達するので、それ以降は空文字列を返します。

```
[12]: f.read()
[12]: ''
```

```
[13]: f.close()
```

`read()` メソッドは、内部的には `readline()` メソッドを呼んでいます。したがって、`read()` メソッドも同様にファイルオブジェクトを消費します。

12.6 練習

文字列 `name` をファイル名とするファイルをオープンして、`read()` メソッドによってファイル全体を文字列として読み込み、その文字数を返す関数 `number_of_characters(name)` を作成してください。

注意：`return` する前にファイルをクローズすることを忘れないようにしてください。

```
[14]: def number_of_characters(name):
    ...
```

上のセルで解答を作成した後、以下のセルを実行し、実行結果が `True` になることを確認してください。

```
[15]: print(number_of_characters('sample.txt') == 446)
False
```

12.7 編集中のファイルの動作

プログラムでファイルを開くと、そのプログラム内でそのファイルを閉じるまでは、他のプログラムでそのファイルを編集することはできません。

下のセルを実行した後で、Windows ならエクスプローラ、macOS なら Finder で上のファイルを探して、削除してみてください。「ファイルを閉じてから再実行してください。(Windows の場合)」といったメッセージが出て、削除ができないはずです。

```
[16]: f = open('test.txt', 'r')
```

下のセルを実行した後だと削除できます。

```
[17]: f.close()
```

12.8 ファイルに対する with 文

ファイルのオブジェクトは、with 文に指定することができます。

```
with ファイルオブジェクト as 変数:
    ...
```

with の次には、open によってファイルをオープンする式を書きます。

また、as の次には、ファイルのオブジェクトが格納される変数を書きます。

with 文は処理後にファイルのクローズを自動的にやってくれますので、ファイルに対して close() を呼び出す必要がありません。

```
[18]: with open('sample.txt', 'r') as f:
        print(f.read())
```

```

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor
↳incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud
↳exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.
Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu
↳fugiat nulla pariatur.
Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt
↳mollit anim id est laborum.
```

12.9 ファイルへの書き込み

ファイルへの書き込みは、print 関数を使って行えます。file 引数に書き込み先のファイルオブジェクトを指定します。file は 3.3 で説明されているキーワード引数と呼ばれる引数ですので、以下のように file=... という形で指定します。

```
[19]: with open('print-test.txt', 'w') as f:
        print('hello\nworld', file=f)
```

文字列の中の \n は改行文字を表します。\\n はエスケープシーケンス (2-1 に説明があります) の一種です。エスケープシーケンスには、この他に、復帰文字を表す \\r やタブを表す \\t などがあります。

ファイルの読み書きのモードとしては、書き込みモードを意味する 'w' を指定しています。既に同じ名前のファイルが存在する場合は上書きされます (以前の内容はなくなります)。ファイルがない場合は、新たに作成されます。

'a' を指定すると、ファイルが存在する場合、既存の内容の後に追記されます。ファイルがない場合は、新たに作成されます。

print 関数は、デフォルトで、与えられた文字列の末尾に改行文字を加えて印字します。末尾に加える文字は、end 引数で指定できます。

```
[20]: with open('print-test.txt', 'a') as f:
      print('hello', 'world\n', end='', file=f) # 改行文字を加えない
```

また、複数の印字対象を渡すと、デフォルトで、空白文字で区切って印字します。この区切り文字は、sep 引数で指定できます。

```
[21]: with open('print-test.txt', 'a') as f:
      print('hello', 'world', sep=', ', file=f) # 'hello, world' が印字される
```

この他にも、ファイルオブジェクトには、より原始的な書き込み用メソッドが用意されています。write() メソッドは、与えられた1つの文字列を単に書き込みます。次に示すように、write() メソッドと read() メソッドは、対で使うことが良くあります。

```
[22]: with open('sample.txt') as src, open('sample.txt.bak', 'w') as dst:
      dst.write(src.read())
```

このコードは、sample.txt を sample.txt.bak にコピーします。

12.10 練習

2つのファイル名 infile, outfile を引数として、infile の半角英文字を全て大文字にした結果を outfile に書き込む file_upper(infile, outfile) という関数を作成してください。

なお、半角英文字の小文字を大文字に変換するには upper() というメソッドが使えます。たとえば line という名前の変数に半角文字列が入っている場合、line.upper() とすれば小文字に変換した文字列を返します。

```
[23]: def file_upper(infile, outfile):
      ...
```

上のセルで解答を作成した後、以下のセルを実行し、実行結果が True になることを確認してください。

```
[24]: with open('print-test.txt', 'w') as f:
      print('hello', 'world', file=f)
file_upper('print-test.txt', 'print-test-upper.txt')
with open('print-test-upper.txt', 'r') as f:
    print(f.read() == 'HELLO WORLD\n')
```

```
-----
FileNotFoundError                                Traceback (most recent call last)
Input In [24], in <module>
      2     print('hello', 'world', file=f)
      3 file_upper('print-test.txt', 'print-test-upper.txt')
----> 4 with open('print-test-upper.txt', 'r') as f:
      5     print(f.read() == 'HELLO WORLD\n')

FileNotFoundError: [Errno 2] No such file or directory: 'print-test-upper.txt'
```

12.11 ファイルの読み書きにおける文字コード指定

`open` でファイルを開くと、通常そのファイルをテキストモードで開きます（テキストモード以外にバイナリモードもあります）。

テキストモードでファイルを開くときは、さらに特定の**文字コード**によってそのファイルを開こうとします。文字コードを指定しないと、デフォルトの文字コードでそのファイルを開こうとしますが、この文字コードがファイルを書き込む際に指定したものと異なる場合、エラーが出たり文字化けしてしまいます。

デフォルトの文字コードは、Windows は Shift_JIS、macOS や Linux は UTF-8 になっていることが多いです。UTF-8 で文字を記録されたファイルを Windows で、ただ `open('utf-8.txt', 'w')` のように文字コードを指定せずに開くとエラーが出ます。同じく、Shift_JIS で文字を記録されたファイルを macOS で `open('shift_jis.txt', 'w')` として開くとエラーが出ます。

なお、この教材の冒頭で `open('sample.txt', 'r')` と、文字コードを指定せずにファイルを開きましたがエラーは出ませんでしたね。これは、`sample.txt` では半角英数字しか使われておらず、半角英数字に関しては、Shift_JIS も UTF-8 も共通のルールでエンコードされているためです。

```
[25]: # macOS ならこちらでエラー
with open('shift_jis.txt', 'r') as f:
    print(f.read())

# Windows ならこちらでエラー
with open('utf-8.txt', 'r') as f:
    print(f.read())
```

```
-----
UnicodeDecodeError                                Traceback (most recent call last)
Input In [25], in <module>
      1 # macOS ならこちらでエラー
      2 with open('shift_jis.txt', 'r') as f:
----> 3     print(f.read())
      5 # Windows ならこちらでエラー
      6 with open('utf-8.txt', 'r') as f:

File /usr/lib/python3.8/codecs.py:322, in BufferedIncrementalDecoder.decode(self,
↳ input, final)
    319 def decode(self, input, final=False):
    320     # decode input (taking the buffer into account)
    321     data = self.buffer + input
--> 322     (result, consumed) = self._buffer_decode(data, self.errors, final)
    323     # keep undecoded input until the next call
    324     self.buffer = data[consumed:]

UnicodeDecodeError: 'utf-8' codec can't decode byte 0x82 in position 0: invalid start_
↳ byte
```

特に半角英数以外の文字を記録する際は文字コードを指定すること、またそのようなファイルを開くときは、記録するときに指定した文字コードでファイルを開いてください。

文字コードは、`open` のキーワード引数として `encoding='utf-8'`（文字コードに UTF-8 を指定する場合）のように指定することができます。

なお、日本語の文字コードには UTF-8, Shift_JIS, EUC-JP などがありますが、Python では OS の種類に限らず、UTF-8 という文字コードがよく使われます。本授業でも UTF-8 を推奨します。

```
[26]: # 文字コードを指定しないと macOS ならこちらでエラー
with open('shift_jis.txt', 'r', encoding='shift_jis') as f:
    print(f.read())
```

(continues on next page)

(continued from previous page)

```
# 文字コードを指定しないと Windows ならこちらでエラー
with open('utf-8.txt', 'r', encoding='utf-8') as f:
    print(f.read())

# 文字コードを指定してファイルに書き込む場合
with open('text.txt', 'w', encoding='utf-8') as f:
    f.write(' かきくけこ ')
with open('text.txt', 'r', encoding='utf-8') as f:
    print(f.read())
```

```
あいうえお
あいうえお
かきくけこ
```

12.12 改行文字の削除

ファイルをテキストモードで開いて `read()` や `readline()` を呼び出すと、`str` 型の文字列として読み込まれます。

文字列の末尾にある改行文字の削除には、2-1 で紹介した `rstrip` メソッドが使えます。ただし、無引数で呼び出すと、改行文字以外の空白文字もまとめて削除されます。

```
[27]: 'あいうえお \n'.rstrip()
```

```
[27]: 'あいうえお'
```

削除する空白文字を改行文字 `'\n'` に限定したい場合には、`rstrip` の引数に指定します。

```
[28]: 'あいうえお \n\n'.rstrip('\n')
```

```
[28]: 'あいうえお '
```

上の例から分かるように、末尾の改行文字をただ1つ削除したい場合には、`rstrip('\n')` は適していません。しかし、`readline()` の返す文字列には、末尾に高々1つの `'\n'` しか存在しないので、`rstrip('\n')` で問題ありません。

```
[29]: with open('text/novel.txt', 'r', encoding='utf-8') as f:
        while True:
            line = f.readline()
            if line == '':
                break
            print(line)

print('----- 末尾の改行文字を削除すると以下ようになります-----')
with open('text/novel.txt', 'r', encoding='utf-8') as f:
    while True:
        line = f.readline()
        if line == '':
            break
        print(line.rstrip('\n'))
```

二人の若い紳士が、すっかりイギリスの兵隊のかたちをして、ぴか／＼する鉄砲をかついで、白熊のやうな犬を二疋つれて、だいぶ山奥の、木の葉のかさ／＼したところを、こんなことを云ひながら、あるいてをりました。

「ぜんたい、こゝらの山は怪しからんね。鳥も獣も一疋も居やがらん。なんでも構はないから、早くタンタアーンと、やつて見たいもんだなあ。」

(continues on next page)

(continued from previous page)

「鹿の黄いろな横つ腹なんぞに、二三発お見舞まうしたら、ずるぶん痛快だらうねえ。くる／＼まはつて、それからどたつと倒れるだらうねえ。」

----- 末尾の改行文字を削除すると以下ようになります-----

二人の若い紳士が、すっかりイギリスの兵隊のかたちをして、ぴか／＼する鉄砲をかついで、白熊のやうな犬を二疋つれて、だいぶ山奥の、木の葉のかさ／＼したところを、こんなことを云ひながら、あるいてをりました。

「ぜんたい、こゝらの山は怪しからんね。鳥も獣も一疋も居やがらん。なんでも構はないから、早くタンタアーンと、やつて見たいもんだなあ。」

「鹿の黄いろな横つ腹なんぞに、二三発お見舞まうしたら、ずるぶん痛快だらうねえ。くる／＼まはつて、それからどたつと倒れるだらうねえ。」

12.13 練習の解答

```
[30]: def number_of_characters(name):
        f = open(name, 'r')
        s = f.read()
        f.close()
        return len(s)
```

```
[31]: def last_line(name):
        last = ''
        with open(name, 'r') as f:
            while True:
                line = f.readline()
                if line == '':
                    return last
            last = line
```

```
[32]: def file_upper(infile, outfile):
        with open(infile, 'r') as f:
            with open(outfile, 'w') as g:
                g.write(f.read().upper())
```

以下のように1つの with 文に複数の open を書くこともできます。

```
[33]: def file_upper(infile, outfile):
        with open(infile, 'r') as f, open(outfile, 'w') as g:
            g.write(f.read().upper())
```

```
[ ]:
```

CHAPTER 13

4-2. イテレータ

イテレータについて簡単に説明します。

参考

- <https://docs.python.org/ja/3/tutorial/classes.html#iterators>

13.1 next

ファイルオブジェクトは、**イテレータ**と呼ばれるオブジェクトの一種です。（1-3 で説明があったように、Python における値はオブジェクトと総称されます。）`iterate` は繰り返すという意味ですよね。イテレータは、その要素を 1 つずつ取り出す処理が可能なオブジェクトで、`next` という関数でその処理を 1 回分行うことができます。

変数 `f` にファイルオブジェクトが入っているとすると、`next(f)` は、ファイルから新たに 1 行を読んで文字列として返します。

```
[1]: f = open('sample.txt', 'r')
```

```
[2]: next(f)
```

```
[2]: 'Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor_
↳incidunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud_
↳exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.\n'
```

```
[3]: next(f)
```

```
[3]: 'Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu_
↳fugiat nulla pariatur.\n'
```

```
[4]: next(f)
```

```
[4]: 'Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt_
↳mollit anim id est laborum.\n'
```

```
[5]: next(f)
```

```
-----
StopIteration                                Traceback (most recent call last)
Input In [5], in <module>
----> 1 next(f)

StopIteration:
```

`f` をファイルオブジェクトとしたとき、`f.readline()` と `next(f)` は、ほぼ同じで、ファイルから新たに 1 行を読んで文字列として返します。

ただし、`f.readline()` と `next(f)` では、ファイルの終わりに来たときの挙動が異なります。`f.readline()` は `''` という空文字列を返すのですが、`next(f)` は `StopIteration` というエラーを発生します。以下で説明する `for` 文は、このエラーを検知しています。つまり、`next(f)` が `StopIteration` を発生したら `for` ループから抜け出します。

このように、関数 `next` が適用できて、`next` が何らかの値を返すか、`StopIteration` を発生するようなオブジェクトを、イテレータと呼びます。

```
[6]: f.close()
```

13.2 for 文による繰り返しとファイルオブジェクト

一般に、イテレータは、`for` 文の `in` の後に指定することができます。

したがって、以下のように、ファイルオブジェクトを値とする変数 `f` を、`for` 文の `in` の後に指定することができます。

```
for line in f:
    ...
```

繰り返しの各ステップで、`next(f)` が呼び出されて、変数 `line` にその値が設定され、`for` 文の中身が実行されます。

以下の例を見てください。

```
[7]: with open('sample.txt', 'r') as f:
      for line in f:
          print(line)
```

```

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor
↳ incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud
↳ exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.
```

```

Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu
↳ fugiat nulla pariatur.
```

```

Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt
↳ mollit anim id est laborum.
```

ファイルオブジェクトに対して、一度 `for` 文で処理をすると、繰り返し処理がファイルの終わりまで達しているので、もう一度同じファイルオブジェクトを `for` 文に与えても何も実行されません。

```
[8]: with open('sample.txt', 'r') as f:
      print('---- 最初 ----')
      for line in f:
          print(line)
      print('---- もう一度 ----')
```

(continues on next page)

(continued from previous page)

```
for line in f:
    print(line)
```

```
---- 最初 ----
```

```
Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor
↳incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud
↳exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.
```

```
Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu
↳fugiat nulla pariatur.
```

```
Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt
↳mollit anim id est laborum.
```

```
---- もう一度 ----
```

ファイルを for 文によって二度読みたい場合は、もう一度ファイルをオープンして、ファイルのオブジェクトを新たに生成してください。

13.3 練習

文字列 `name` をファイル名とするファイルの最後の行を文字列として返す関数 `last_line(name)` を、ファイルオブジェクトに対する for 文を用いて定義してください。

```
[9]: def last_line(name):
     ...
```

上のセルで解答を作成した後、以下のセルを実行し、実行結果が `True` になることを確認してください。

```
[10]: print(last_line('sample.txt')==
↳"Excepteur sint occaecat cupidatat non proident, sunt
↳in culpa qui officia deserunt mollit anim id est laborum.\n")
False
```

13.4 iter

いうまでもなく、リストに対して for 文を用いることができますが、リストはイテレータでしょうか。

```
[11]: next([1,2,3])
```

```
-----
TypeError                                Traceback (most recent call last)
Input In [11], in <module>
----> 1 next([1,2,3])

TypeError: 'list' object is not an iterator
```

リストに対して `next` を適用するとエラーになってしまいます。したがって、リストはイテレータではありません。

では、なぜリストに対して for 文が適用できるのでしょうか。

実は、for 文の `in` の後に指定されたオブジェクトに対しては、必ず `iter` という組み込み関数が適用される、という仕掛けになっているのです。

実際に、リストに `iter` を適用してみましょう。

```
[12]: it = iter([1,2,3])
```

変数 `it` には、リスト `[1,2,3]` から作ったイテレータが入っています。

```
[13]: it
```

```
[13]: <list_iterator at 0x7f47dc59ad90>
```

```
[14]: next(it)
```

```
[14]: 1
```

```
[15]: next(it)
```

```
[15]: 2
```

```
[16]: next(it)
```

```
[16]: 3
```

```
[17]: next(it)
```

```
-----
StopIteration                                Traceback (most recent call last)
Input In [17], in <module>
----> 1 next(it)

StopIteration:
```

ここで、もう一度イテレータを作り直しましょう。

```
[18]: it = iter([1,2,3])
```

このイテレータに対して以下のように `for` 文を用いると、もとのリストの要素が網羅されます。

```
[19]: for x in it:
      print(x)
```

```
1
2
3
```

ファイルの場合と同様に、もう一回 `for` 文を回しても何も出力されません。

```
[20]: for x in it:
      print(x)
```

もう一度説明すると、`for` 文の `in` の後に指定されたオブジェクトに対しては、必ず `iter` という組み込み関数が適用されて、イテレータが得られます。そして、そのイテレータに対して `next` が次々と呼ばれます。

では、`for` 文の `in` の後にイテレータが指定されるとどうなるのでしょうか。やはり、`iter` が適用されるのですが、Python では、イテレータに対して `iter` が適用されても、そのイテレータ自身が返ります。

```
[21]: with open('sample.txt', 'r') as f:
      print(f is iter(f))
```

```
True
```

`is` は、2-2 に説明がありますが、その両辺が同じオブジェクトかどうかを調べる演算子です。

13.5 練習

リストをもらって、そのイテレータを作り、最初の要素だけ取り出した後、そのイテレータを返す関数 `but_first(ls)` を定義してください。

```
[22]: def but_first(ls):
      ...
```

上のセルで解答を作成した後、以下のセルを実行し、実行結果が `True` になることを確認してください。

```
[23]: it = but_first([0,2,4,6,8])
      print(type(it) == type(iter([])))
      print(list(it) == [2,4,6,8])
```

False

```
-----
TypeError                                Traceback (most recent call last)
Input In [23], in <module>
      1 it = but_first([0,2,4,6,8])
      2 print(type(it) == type(iter([])))
----> 3 print(list(it) == [2,4,6,8])

TypeError: 'NoneType' object is not iterable
```

13.6 イテラブル

一般に、関数 `iter` が適用できるオブジェクトを**イテラブル**と呼びます。イテラブルは、`for` 文の `in` の後に指定することができます。

イテレータに `iter` を適用すると自分自身が返るので、イテレータはイテラブルでもあります。

リストはイテラブルですが、イテレータではありません。

```
[24]: ln = [1,2,3]
      for x in ln:
          print(x)
      for x in ln:
          print(x)
```

1
2
3
1
2
3

上の例では、2つの `for` 文ごとに、`ln` に `iter` が適用されて、別々のイテレータが作られたのです。

関数 `range` が返すオブジェクトもイテラブルですが、イテレータではありません。

```
[25]: r = range(3)
      for x in r:
          print(x)
      for x in r:
          print(x)
```

0
1

(continues on next page)

(continued from previous page)

```
2
0
1
2
```

13.7 イテレータを返す enumerate

3-2 で紹介した組み込み関数の `enumerate()` は、イテレータを返します。

```
[26]: it = enumerate([10,20,30])
```

```
[27]: next(it)
```

```
[27]: (0, 10)
```

```
[28]: for x in it:
      print(x)
```

```
(1, 20)
(2, 30)
```

```
[29]: for i, c in enumerate('ACDB'):
      print(i, ' 番目の文字 =', c)
```

```
0 番目の文字 = A
1 番目の文字 = C
2 番目の文字 = D
3 番目の文字 = B
```

一方、`enumerate` はイテラブルを引数として受け取ります。上の例で、リストも文字列もイテラブルです。イテレータもイテラブルなので、`enumerate` の引数になり得ます。たとえば、ファイルオブジェクトはイテレータなので、以下のように `enumerate` の引数になります。

```
[30]: with open('sample.txt', 'r') as f:
      for i, s in enumerate(f):
          print(i, ' 行目:')
          print(s)
```

```
0 行目:
Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor
↳incidunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud
↳exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.

1 行目:
Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu
↳fugiat nulla pariatur.

2 行目:
Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt
↳mollit anim id est laborum.
```

変数 `i` は 0 から順に増えていきます。変数 `s` には各行の文字列が代入されます。 `i` は 0 から始まりますが、各行の行番号と考えられます。

13.8 練習の解答

```
[31]: def last_line(name):  
      f = open(name, 'r')  
      for line in f:  
          pass  
      f.close()  
      return line
```

```
[32]: def but_first(ls):  
      it = iter(ls)  
      next(it)  
      return it
```

```
[ ]:
```

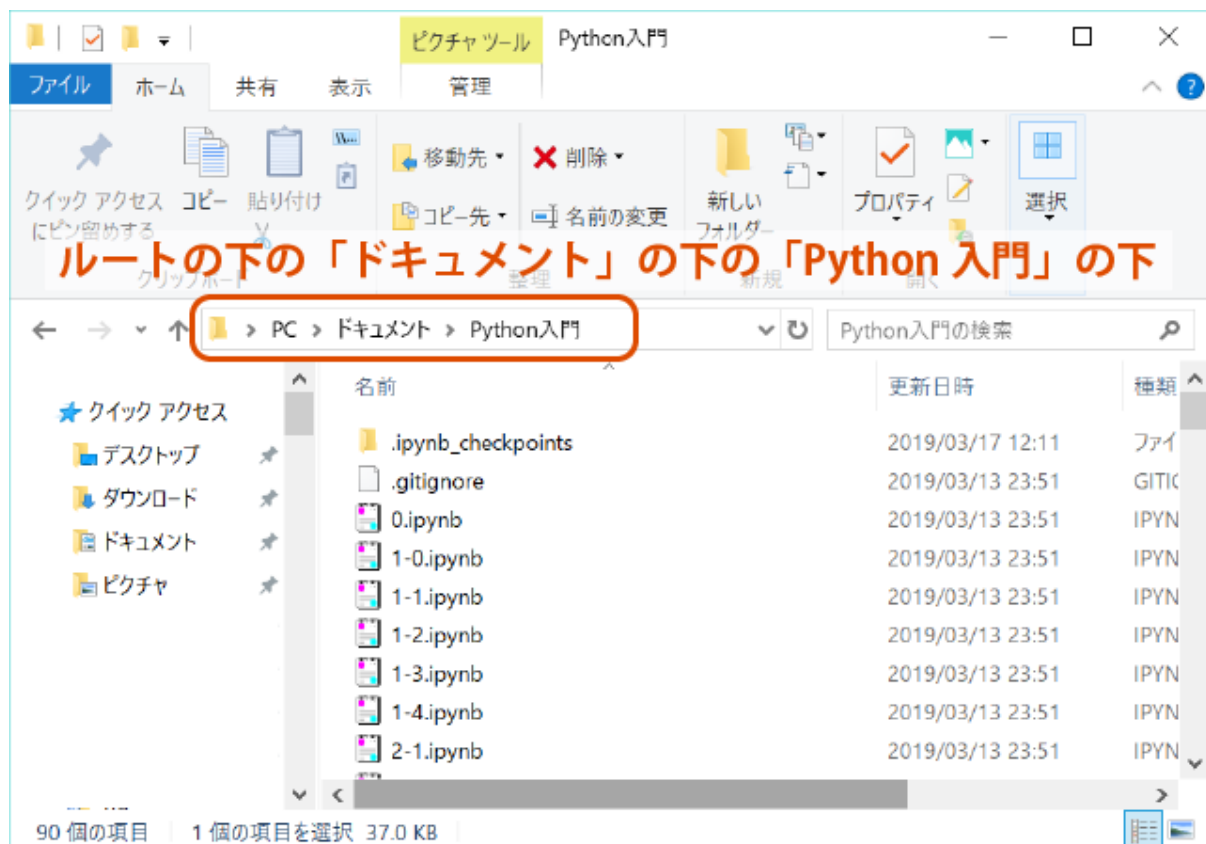

CHAPTER 14

4-3. コンピュータにおけるファイルやディレクトリの配置

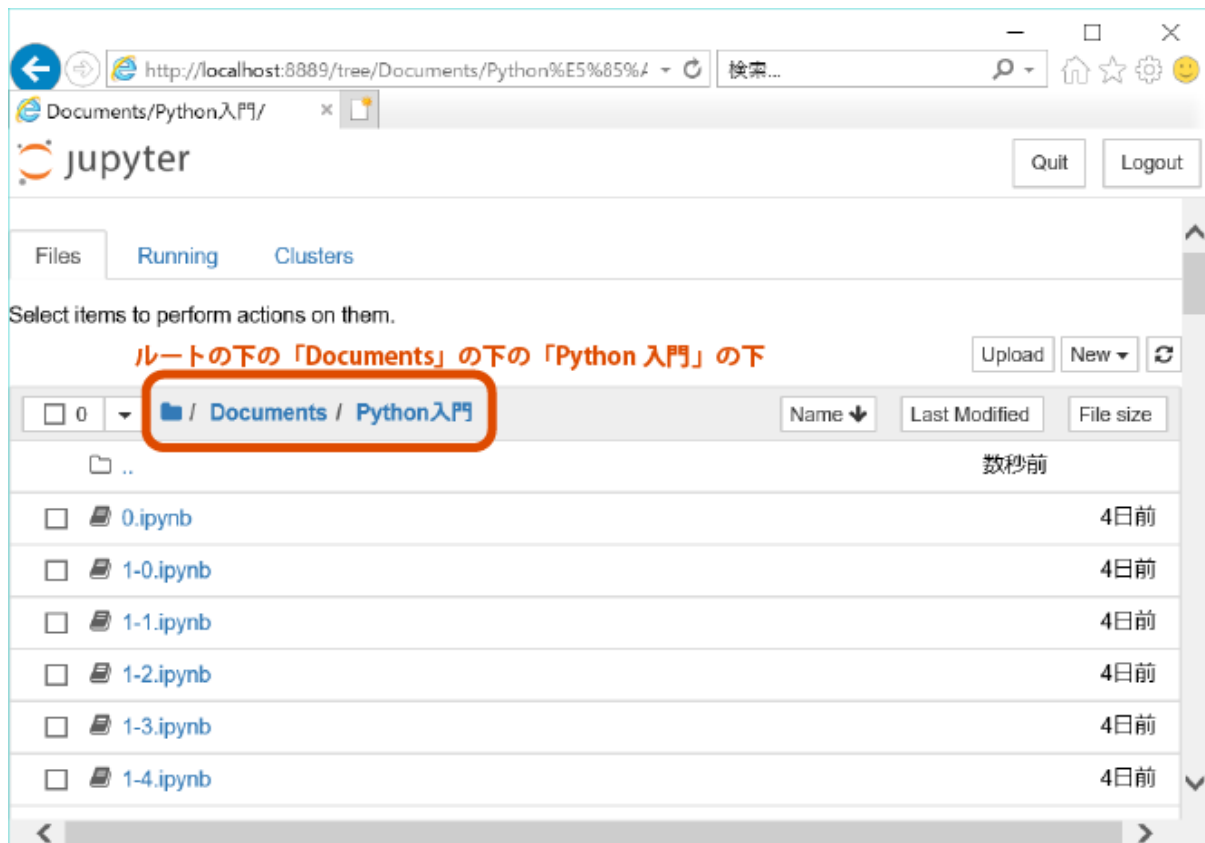
木構造のデータ形式について説明します。この内容は Python 言語に限らず、Windows や Mac、Linux などの一般的な OS において共通する概念です。Colaboratory では、同様の構造が仮想環境上に作られます。

みなさん、Windows ではエクスプローラ、Mac では Finder を使ってファイルを階層的に保存していますよね。

下の例では、Windows で ドキュメント (Documents) という名前のフォルダの中に Python 入門 というフォルダを作り、その下にこの教材を置いた時の、エクスプローラの様子を表しています。

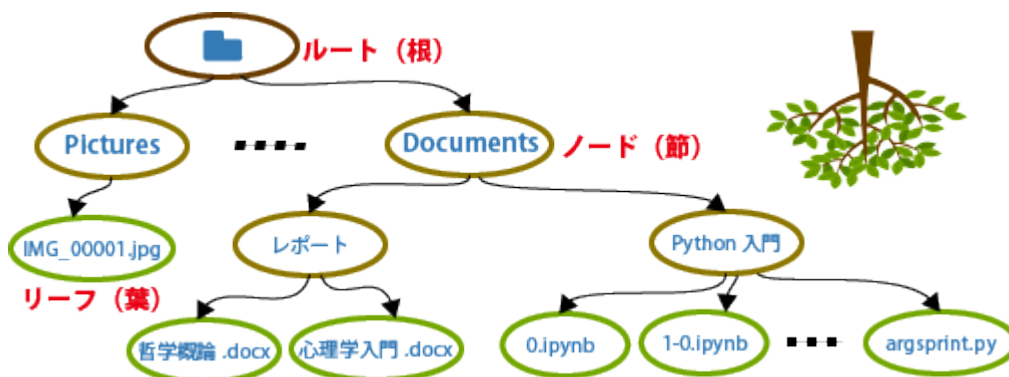


これは Jupyter Notebook では以下のように見えます。



このようなデータ形式は以下のような図で表すこともできます。まるで木を逆さにしたような形に見えますね。ですからこのようなデータの形式を「木構造」と呼びます。

また、一番根っこにあたるデータを「ルート（根）」、先端にあたるデータを「リーフ（葉）」、その間にあたるデータを「ノード（節）」と呼びます。



データの保存においては、ファイルはリーフ（葉）に相当し、フォルダはノード（節）に相当します。ルートはハードディスクやUSBメモリなど記録媒体自体に対応することが多いです。ハードディスクに入っているファイルと、USBメモリに入っているファイルは、それぞれ違う木に属するデータということです。

特にファイルの場所を意味するとき、フォルダのことをディレクトリと呼びます。

14.1 カレントワーキングディレクトリ

プログラムは、必ずどこかのディレクトリで動いています。このプログラムが動作しているディレクトリのことを、**ワーキングディレクトリ**（もしくは**作業ディレクトリ**）と呼びます。通例、特に Windows や macOS では、何らかのファイルをクリックしてアプリケーションが起動したとき、その開いたファイルのある場所がワーキングディレクトリになります。

ワーキングディレクトリは、プログラムの実行中に変更できます。Python 上では `os.chdir` を使うことで変更できます。

プログラム実行中の現在のワーキングディレクトリのことを、**カレントワーキングディレクトリ**、もしくは単に**カレントディレクトリ**と呼びます。カレントディレクトリは頻繁に利用するので、`.`という特別な記号によって表現できます。`##パス`

カレントディレクトリに置かれているファイルは、ファイル名を指定するだけで開くことができます。だから、4-1 で示したように、カレントディレクトリにある `sample.txt` は、ファイル名を指定するだけで開けます。

```
[1]: open('sample.txt', 'r', encoding='utf-8')
[1]: <_io.TextIOWrapper name='sample.txt' mode='r' encoding='utf-8'>
```

一方、それ以外の場所にあるファイルについては、そのファイルのディレクトリまで含めて指定しなければ、開くことができません。

```
[2]: open('novel.txt', 'r')

-----
FileNotFoundError                                Traceback (most recent call last)
Input In [2], in <module>
----> 1 open('novel.txt', 'r')

FileNotFoundError: [Errno 2] No such file or directory: 'novel.txt'
```

カレントディレクトリに存在しない `novel.txt` を開こうとしたので `FileNotFoundError` が生じました。`novel.txt` は、`text` というディレクトリの中にあるので、それ明示するために、`/` で区切って、次のように指定します。

```
[3]: open('text/novel.txt', 'r', encoding='utf-8')
[3]: <_io.TextIOWrapper name='text/novel.txt' mode='r' encoding='utf-8'>
```

実は、カレントディレクトリにあるファイルが、ファイル名の指定だけで開けるのは、自動的に `./` が補われていたからでした。

`open` の第 1 引数に渡す文字列のような、ファイルやディレクトリの場所を指定する表記を、**パス**と呼びます。パス（経路）と呼ぶのは、`/` 区切りで 1 歩ずつ次に進むディレクトリを指定することに由来しています。

パスを記述する際、ルートディレクトリは `/` で表されます。ルートディレクトリから始まるパスを、**絶対パス**と呼びます。一方、カレントディレクトリからのパスを、**相対パス**と呼びます。パス表記において `/` 以外から始まる場合は、自動的に先頭に `./` が補われて、相対パスとして扱われます。

さて、`./text/` というパス表記は、カレントディレクトリにある一段下の `text` ディレクトリに進むことに対応します。これに、一段上のディレクトリを表す `..` を組み合わせることで、より柔軟にパスを指定できます。

たとえば、`./text/../` は `./` と同じディレクトリを指します。

```
[4]: with open('sample.txt', 'r', encoding='utf-8') as f:
      print(f.read(), end='')
```

```

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor
↳incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud
↳exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.
Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu
↳fugiat nulla pariatur.
Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt
↳mollit anim id est laborum.

```

```

[5]: with open('./text/../sample.txt', 'r', encoding='utf-8') as f:
      print(f.read(), end='')

```

```

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor
↳incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud
↳exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.
Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu
↳fugiat nulla pariatur.
Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt
↳mollit anim id est laborum.

```

また、カレントディレクトリを `./text/` に変化させた後に、`sample.txt` を開くときには、`../sample.txt` と指定することができます。

```

[6]: import os
      os.chdir('./text') # 1段下の text に行く
      with open('../sample.txt', 'r', encoding='utf-8') as f:
          print(f.read())
      os.chdir('../') # 元のディレクトリに戻る

```

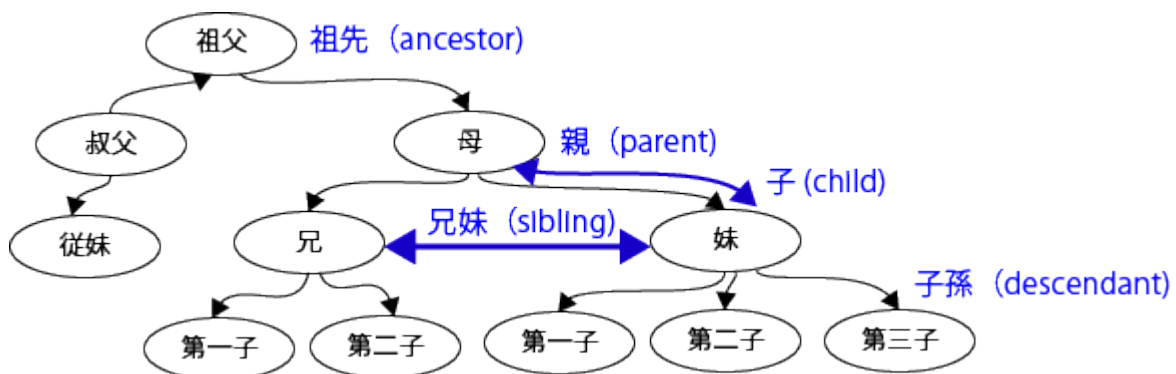
```

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor
↳incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud
↳exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.
Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu
↳fugiat nulla pariatur.
Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt
↳mollit anim id est laborum.

```

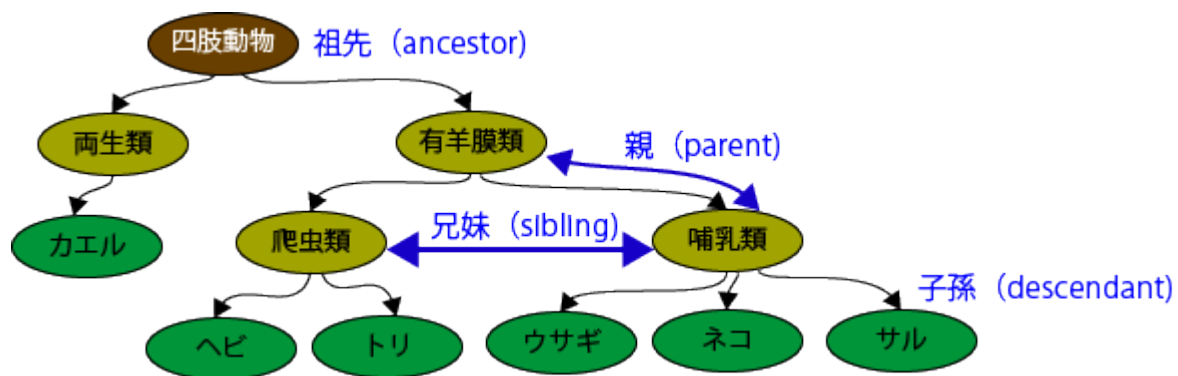
14.2 木構造によるデータ表現

木構造はファイルやディレクトリの保存形式だけでなく、データの表現として幅広く利用されます。たとえば家系図も木構造による表現です。「家系図」は英語で “family tree” ですよね。



このような構造を持つデータでは、まるで家系図のように、上位下位関係にあるデータ同士を「親子 (parent/child)」と呼んだり、同位関係にあるものを「兄妹 (sibling)」と呼んだりします。「祖先 (ancestor)」や「子孫 (descendant)」という表現も使われます。

データのこのような表現は、実際に親子関係にあるかは関係ありません。たとえば下の図は四肢動物の系統樹です。



データ構造的には、「有羊膜類」と「哺乳類」は親子関係にあるというわけです。

[]:

5-1. モジュールの使い方

モジュールの使い方について説明します。

参考

- <https://docs.python.org/ja/3/reference/import.html>
- <https://docs.python.org/ja/3/tutorial/modules.html>
- <https://docs.python.org/ja/3/library/math.html>

15.1 モジュールのインポート

Python では特別な関数や値をまとめたもの（これを**モジュール**といいます）を使うために、`import` という文を使います（第 1 回 (1-1) においても説明しました）。具体的には次のように記述します。

```
import モジュール名
```

たとえば、数学関係の機能をまとめた `math` というモジュールがあります。これらの関数や値を使いたいときは、以下のようにして `math` モジュールを `import` で**インポート**します。そうすると、`math.` 関数名 という形で関数を用いることができます。

```
[1]: import math# import は大抵セルの一番上に記述します
print(math.sqrt(2)) # sqrt は平方根を計算する関数
print(math.pi) # πの値
print(math.sin(math.pi/4)) # sin 関数
print(math.cos(0)) # cos 関数
print(math.log(32,2)) # 2を底とする 32 の対数 (texで記述すると、 $\log_2 32$ )

1.4142135623730951
3.141592653589793
0.7071067811865475
1.0
5.0
```

上の例では、`math` モジュールの中の関数や値を使用しています。

注意しなければならないのは、モジュールの中の関数を使う場合には、

```
モジュール名. モジュールの中の関数名
```

とする必要があるということです。

モジュールの中の値（たとえば `math.pi`）も同様です。

なお、複数の関数名をコンマ , で区切って並べて同時にインポートすることもできます。

15.2 from

モジュール内で定義されている関数を「モジュールの中の関数名」のようにして、「モジュール名」を付けずにそのままの名前で、モジュールの読み込み元のプログラムで使いたい場合には、`from` を以下のように書くことで利用することができます。

```
from モジュール名 import モジュールの中の関数名
```

たとえば、次のようになります。

```
[2]: from math import sqrt
print(sqrt(2)) # sqrt は平方根を計算する関数
from math import pi
print(pi) # πの値
from math import sin
print(sin(math.pi/4)) # sin 関数
from math import cos
print(cos(0)) # cos 関数
from math import log
print(log(32,2)) # 2を底とする 32 の対数 (texで記述すると、$\log_2 32$)

1.4142135623730951
3.141592653589793
0.7071067811865475
1.0
5.0
```

この方法では、関数ごとに `from` を用いてインポートする必要があります。

なお、関数だけではなく、グローバル変数や後に学習するクラスも、このようにしてインポートすることができます。

別の方法として、**ワイルドカード *** を利用する方法もあります。

```
from math import *
```

この方法ではアンダースコア `_` で始まるものを除いた全ての名前が読み込まれるため、明示的に名前を指定する必要はありません。

```
[3]: from math import *
print(factorial(5)) # 5 の階乗 # import mathを使う場合、math.factorial(5)
print(floor(2.31)) # 2.31 以下の最大の整数 # import mathを使う場合、math.floor(2.31)
print(e) # ネイピア数 # import mathを使う場合、math.e

120
2
2.718281828459045
```

ただしこの方法は推奨されていません。理由は読み込んだモジュール内の未知の名前とプログラム内の名前が衝突する可能性があるためです。

```
[4]: pi = 'パイ' # pi という変数に文字列「パイ」を代入する
      print(pi)
      from math import *
      print(pi) # math モジュールの pi の値で上書きされる (衝突)
```

```
パイ
3.141592653589793
```

15.3 as

モジュール名が長すぎるなどの理由から別の名前としたい場合は、`as` を利用する方法もあります。たとえば、5-3 において学習する NumPy というライブラリは `numpy` モジュールとして提供されていますが、次のように、`numpy` を `np` という略称で使うことがあります。

```
[5]: import numpy
      print(numpy.ones((3, 5))) # 3 × 5 の行列を表示
      import numpy as np
      print(np.ones((3, 5))) # np という短い名称で同じ関数を利用する
```

```
[[1.  1.  1.  1.  1.]
 [1.  1.  1.  1.  1.]
 [1.  1.  1.  1.  1.]]
[[1.  1.  1.  1.  1.]
 [1.  1.  1.  1.  1.]
 [1.  1.  1.  1.  1.]]
```

個々の関数ごとに別の名前を付けることもできます。

```
[6]: import math
      print(math.factorial(5)) # 階乗を求める関数 factorial # 5 の階乗
      from math import factorial as fact # fact という名前で math.factorial を使用したい
      print(fact(5))
```

```
120
120
```

15.4 練習

第 1 回では、数学関数を以下のようにインポートし、`math.sqrt()` のようにして、数学関数や数学関係の変数を利用していました。

```
import math
print(math.sqrt(2))
print(math.sin(math.pi))
```

以下のセルを、モジュール名を付けずにこれらの関数や変数を参照できるように変更してください。

```
[7]: import ...
      ...

      print(sqrt(2))
      print(sin(pi))
```

```
Input In [7]
import ...
```

(continues on next page)

(continued from previous page)

```
^  
SyntaxError: invalid syntax
```

15.5 練習の解答

from を使ってモジュールを指定、参照する関数を import でインポートしてください。

```
[8]: from math import sqrt, sin, pi  
print(sqrt(2))  
print(sin(pi))
```

```
1.4142135623730951  
1.2246467991473532e-16
```

5-2. モジュールの作り方

モジュールの作り方について説明します。

参考

- <https://docs.python.org/ja/3/tutorial/modules.html>
- <https://docs.python.org/ja/3/reference/import.html>

Python ではプログラムを**モジュール**という単位で、複数のファイルに分割することができます。通例、一度定義した便利な関数などを別のプログラムで再利用するときには、再利用される部分をモジュールとして切り出します。プログラムが大きくなると、このように複数のファイルに分割した方が開発や保守が簡単になります。

16.1 モジュールファイル

本授業で扱ってきたノートブックファイル（拡張子 `.ipynb`）は、コードセル（Code セル）に Python ソースコード、Markdown セルに文書を持ち、内部的に出力結果も保存しています。一方、モジュールファイル（拡張子 `.py`）は、Python ソースコードのみを含んだファイルです。

モジュールファイルを作るときには、Jupyter Notebook におけるコードセルの内容のみをファイルに記述することになります。

モジュールファイルの文字コードは UTF-8 であることが公式に推奨されています。原則として UTF-8 でエンコードして保存してください。

16.1.1 ノートブックファイルをモジュールファイルに変換する

本授業で利用しているノートブックファイルを `.py` としてセーブするには、「ファイル」メニューの「`.py` をダウンロード」項目を選択します。

そうすると、コードセルだけがプログラム行として有効になり、その他の行はコメントアウトされたモジュールファイルがダウンロードできます。ダウンロード先はブラウザによって定まりますが、ダウンロードフォルダになることが一般的でしょう。

この方法では、全てのコードセルの内容を一度に実行するプログラムとして保存されます。ノートブックのようにセル単位の実行するわけではないことに注意する必要があります。

16.1.2 モジュールファイルをアップロードする

次に、ダウンロードしたモジュールファイルを Colaboratory で使うには、Colaboratory の実行環境の中のファイルシステムのカレントディレクトリにモジュールファイルをアップロードする必要があります。このためには、以下のセルを実行してください。

```
[1]: import sys
if 'google.colab' in sys.modules:
    from google.colab import files
    uploaded = files.upload() # Upload to the current directory
```

16.2 自作モジュールの使い方

モジュールで定義されている関数を利用するには、`import` を用いて `import モジュール名` と書きます。モジュール名は、モジュールファイル名から拡張子 `.py` を除いたものです。

すると、モジュールで定義されている関数は `モジュール名.関数名` によって参照できます。

次の関数が記述された `factorial.py` というモジュールを読み込む場合を説明します。ただし、読み込み元と同じディレクトリに `factorial.py` が存在すると仮定します。

`factorial.py`:

```
# 階乗 n! を返す
def fact(n):
    prod = 1
    for i in range(1, n + 1):
        prod *= i
    return prod
```

```
[2]: import factorial
```

```
factorial.fact(6)
```

```
[2]: 720
```

`from` や `as` の使い方も既存のモジュールと全く同じです。

モジュール内で定義されている名前を読み込み元のプログラムでそのまま使いたい場合は、`from` を用いて以下のように書くことができます。

```
[3]: from factorial import fact
```

```
fact(6)
```

```
[3]: 720
```

ワイルドカード `*` を利用する方法もありますが、推奨されていません。読み込まれるモジュール内の未知の名前と、読み込み元のプログラム中の名前が衝突する可能性があるためです。

```
[4]: from factorial import *
```

モジュール名が長すぎるなどの理由から別の名前としたい場合は、`as` を利用する方法もあります。

```
[5]: import factorial as f
```

```
f.fact(6)
```

```
[5]: 720
```

5-3. NumPy ライブラリ

NumPy について説明します。

参考

- <https://docs.scipy.org/doc/numpy/user/quickstart.html>
- <https://docs.scipy.org/doc/numpy/user/basics.html>

NumPy とは、多次元配列を効率的に扱うライブラリです。Python の標準ライブラリではありませんが、科学技術計算や機械学習など、ベクトルや行列の演算が多用される分野では、事実上の標準ライブラリとしての地位を確立しています。

NumPy を用いるには、まず、`numpy` モジュールをインポートする必要があります。慣習として、`np` と別名をつけて利用されます。

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

NumPy では、Python 標準の数値やリストの代わりに、特別な数値や配列を用いることで、格段に効率的な配列演算を実現します。以下では、配列の基本的な操作や機能を説明します。

17.1 配列の構築

配列とは、特定の型の値の並びです。`numpy.array()` 関数で構築できます。このとき、配列の要素は Python 標準のリストやタプルで指定します。どちらを用いて作成しても全く同じ配列を作成できます。

```
[2]: a = np.array([1,2,3]) # リストから配列作成
print(a)
b = np.array((1,2,3)) # タプルからの配列作成
print(b)
```

```
[1 2 3]
[1 2 3]
```

`print` の結果はリストと似ていますが、要素が、ではなく空白で区切られているに注意してください。`print` ではなく、式の評価結果の場合、より違いが明示されます。

```
[3]: a
```

```
[3]: array([1, 2, 3])
```

配列は `numpy.ndarray` というデータ型によって実現されています。組み込み関数 `type()` を使うと、データ型を調べられます。

```
[4]: type(np.array([1,2,3,4,5])) # 配列の型
```

```
[4]: numpy.ndarray
```

```
[5]: type([1,2,3,4,5])
```

```
[5]: list
```

`array()` が、リストではなく `ndarray` を返していることがわかります。

17.1.1 要素型

配列の要素を構成する値には幾つかの型がありますが、次の4つの型を知っていればとりあえずは十分です。

型名	説明
<code>numpy.int32</code>	整数 (32-bit) を表す型
<code>numpy.float64</code>	実数 (64-bit) を表す型
<code>numpy.complex128</code>	複素数 (64-bit 実数の組) を表す型
<code>numpy.bool_</code>	真理値を表す型

配列は、リストと異なり、型の異なる要素を混在させることはできません。

`array()` の `dtype` 引数に、要素型を表すオブジェクトや文字列値を与えることで、指定された要素型の配列を構築できます。

```
[6]: print(np.array([-1,0,1], dtype=np.int32)) # np.int32 の代わりに'int32'でも同じ
```

```
[-1  0  1]
```

実数には、小数点が付与されて印字されます。

```
[7]: print(np.array([-1,0,1], dtype=np.float64)) # np.float64 の代わりに'float64'でも同じ
```

```
[-1.  0.  1.]
```

複素数は実部と虚部を表す実数の組であり、虚部には `j` が付与されて印字されます。

```
[8]: print(np.array([-1,0,1], dtype=np.complex128)) # np.complex128 の代わりに'complex128'でも同じ
```

```
[-1.+0.j  0.+0.j  1.+0.j]
```

数値から真理値への変換では、`0` が `False` で、`0` 以外が `True` になります。

```
[9]: print(np.array([-1,0,1], dtype=np.bool_)) # np.bool_ の代わりに'bool'でも同じ
```

```
[ True False  True]
```

17.1.2 多次元配列

多次元配列は、配列の中に配列がある入れ子の配列です。入れ子のリストやタプルを `numpy.array()` に渡すことで構築できます。

```
[10]: print(np.array([[1,2],[3,4]])) # 2次元配列の構築
```

```
[[1 2]
 [3 4]]
```

```
[11]: print(np.array([[[1,2],[3,4]],[[5,6],[7,8]]])) # 3次元配列の構築
```

```
[[[1 2]
   [3 4]]
 [[5 6]
   [7 8]]]
```

上の例からわかるように、2次元配列は行列のように、3次元配列は行列の配列のように印字されます。

多次元配列は、要素となる配列の長さが等しいことが想定されます。つまり、2次元配列は、行列のように各行の長さが等しくなければなりません。

```
[12]: print(np.array([[1,2],[3]])) # 行の長さが異なる場合
```

```
[list([1, 2]) list([3])]
```

このように行の長さが異なる場合は、多次元配列とは見做されません。

多次元配列の各次元の長さの組を、多次元配列の**形** (shape) と呼びます。特に2次元配列の場合、行列と同様に、行数（内側にある配列の数）と列数（内側にある配列の要素数）の組を使って、行数×列数で形を表記します。

1次元配列に対して `reshape()` メソッドを使うと、引数で指定された形の多次元配列に変換することができます。

```
[13]: a1 = np.array([0, 1, 2, 3, 4, 5]) # 1次元配列
      a2 = a1.reshape(2,3)           # 2×3の2次元配列
      a2
```

```
[13]: array([[0, 1, 2],
            [3, 4, 5]])
```

ここで、`reshape()` を適用する前後の配列（ここでは `a1` と `a2`）は、内部的にデータを共有していることに注意してください。つまり、`a1` の要素を更新すると、`a2` にも影響を及ぼします。

```
[14]: a1[1] = 6
      print(a1)
      print(a2)
```

```
[0 6 2 3 4 5]
[[0 6 2]
 [3 4 5]]
```

`ravel()` メソッドを使うと、多次元配列を1次元配列に戻すことができます。

```
[15]: a = np.array([0, 1, 2, 3, 4, 5]).reshape(2,3)
      print(a)
      print(a.ravel())
```

```
[[0 1 2]
 [3 4 5]]
[0 1 2 3 4 5]
```

`ravel()` の結果も、`reshape()` と同様に、元の配列と要素を共有します。

```
[16]: elems = np.array([0, 1, 2, 3, 4, 5])
a = elems.reshape(2,3).ravel() # ravel() は要素を elems と共有
elems[1] = 6
print(a)

[0 6 2 3 4 5]
```

なお、要素をコピーして変換する `flatten()` メソッドもありますが、コピーしない `ravel()` の方が効率的です。

17.1.3 ▲配列のデータ属性

配列はオブジェクトであり、その配列に関する様々な情報を属性として保持します。（オブジェクトの属性については 6-3 に簡単な説明があります。）配列が持つ代表的なデータ属性（メソッド以外の属性）を次の表にまとめます。

属性	意味
<code>a.dtype</code>	配列 <code>a</code> の要素型
<code>a.shape</code>	配列 <code>a</code> の形（各次元の長さのタプル）
<code>a.ndim</code>	配列 <code>a</code> の次元数（ <code>len(a.shape)</code> と等しい）
<code>a.size</code>	配列 <code>a</code> の要素数（ <code>a.shape</code> の総乗と等しい）
<code>a.flat</code>	配列 <code>a</code> の 1 次元表現（ <code>a.ravel()</code> と等しい）
<code>a.T</code>	配列 <code>a</code> を転置した配列（ <code>a</code> と要素を共有）

17.2 配列要素を生成する構築関数

要素を生成して配列を構築する代表的な関数を紹介します。特に断りが無い場合、ここで紹介する関数は、`array()` と同様に `dtype` 引数で要素型を指定可能です。

17.2.1 arange

`numpy.arange()` は、組み込み関数 `range()` の配列版です（`arange` は `array range` の略）。開始値・終了値・刻み幅を引数にとります。デフォルトの開始値は 0、刻み幅は 1 です。`range()` と違って、引数の値は整数に限定されません。

```
[17]: print(np.arange(3)) # range(3) に対応する配列
print(np.arange(0, 1, 0.2)) # 0 を開始値として 0.2 刻みで 1 未満の要素を生成

[0 1 2]
[0. 0.2 0.4 0.6 0.8]
```

17.2.2 linspace

`numpy.linspace()` 関数は、範囲を等分割した値からなる配列を生成します。第 1 引数と第 2 引数には、それぞれ範囲の開始値と終了値、第 3 引数には分割数を指定します。

```
[18]: print(np.linspace(0, 1, 4)) # 0 から 1 の値を 4 分割した値を要素に持つ配列

[0.          0.33333333 0.66666667 1.          ]
```

17.2.3 zeros と ones

`numpy.zeros()` 関数は、0 からなる配列を生成します。同様に、`numpy.ones()` 関数は、1 からなる配列を生成します。どちらも、生成される形を第 1 引数に取ります。デフォルトの要素型は、実数です。

```
[19]: print(np.zeros(4))      # 長さ 4 の 1 次元配列
      print(np.zeros((2,3))) # 2 × 3 の 2 次元配列を生成
      print(np.ones(4))     # 長さ 4 の 1 次元配列
      print(np.ones((2,3))) # 2 × 3 の 2 次元配列を生成
```

```
[0. 0. 0. 0.]
[[0. 0. 0.]
 [0. 0. 0.]]
[1. 1. 1. 1.]
[[1. 1. 1.]
 [1. 1. 1.]]
```

17.2.4 random.rand

`numpy.random.rand()` 関数は、0 以上 1 未満の乱数からなる配列を生成します。引数には生成される配列の形を指定します。要素型は実数に限定されます。

```
[20]: print(np.random.rand(4))    # 長さ 4 の 1 次元配列
      print(np.random.rand(2,3)) # 2 × 3 の 2 次元配列を生成
```

```
[0.78999303 0.90482231 0.67346348 0.16217231]
[[0.61202115 0.34825497 0.25168207]
 [0.02817442 0.5133705  0.24220049]]
```

この他にも、`numpy.random.randn()`・`numpy.random.binomial()`・`numpy.random.poisson()` は、それぞれ、正規分布・二項分布・ポアソン分布の乱数からなる配列を生成します。

17.3 練習

引数に整数 n を取り、 i から始まる連番の整数からなる配列を i 番目 ($i \geq 0$) の行として持つ $n \times n$ の 2 次元配列を返す関数 `range_square_matrix()` を、`arange()` を用いて定義してください。

たとえば、`range_square_matrix(3)` は、

```
[[0 1 2]
 [1 2 3]
 [2 3 4]]
```

と印字されるような 2 次元配列を返します。

```
[21]: def arange_square_matrix(n):
      ...
```

以下のセルを実行して、True が表示されることを確認してください。

```
[22]: print(all(map(all, (arange_square_matrix(3) == np.array([[0,1,2],[1,2,3],[2,3,4]]))))))
False
```


17.4 配列要素の操作

17.4.1 インデックスアクセス

配列の要素には、リストの場合と同様に、`0` から始まるインデックスを使って参照できます。リストと同じく、配列の先頭要素のインデックスは `0`、最後の要素のインデックスは `-1` となります。

```
[23]: a = np.arange(3)
      print(a)
      [0 1 2]
```

```
[24]: a[0]
```

```
[24]: 0
```

```
[25]: a[-1]
```

```
[25]: 2
```

```
[26]: a[-1] = 3 # 要素への代入もできる
      print(a)
      [0 1 3]
```

多次元配列では、高次元（入れ子の外側）から順にインデックスを指定します。特に 2 次元配列、すなわち行列の場合は、行インデックスと列インデックスを順に指定します。

```
[27]: a = np.arange(6).reshape(2,3)
      print(a)
      [[0 1 2]
       [3 4 5]]
```

```
[28]: a[1,2] # 行と列のインデックスをまとめて指定
```

```
[28]: 5
```

```
[29]: a[1,2] = 6 # 要素への代入もできる
      print(a)
      [[0 1 2]
       [3 4 6]]
```

17.4.2 スライス

リストと同様に、配列の**スライス**を構築できます。

```
[30]: a = np.arange(5)
      print(a)
      print(a[1:4])
      print(a[1:])
      print(a[:-2])
      print(a[:2])
      print(a[::-1])
      [0 1 2 3 4]
      [1 2 3]
```

(continues on next page)

(continued from previous page)

```
[1 2 3 4]
[0 1 2]
[0 2 4]
[4 3 2 1 0]
```

配列のスライスに対して代入すると、右辺の値がコピーされて、スライス元の配列にまとめて代入されます。

```
[31]: a = np.arange(5)
      print(a)
      a[1:4] = 6
      print(a)
      a = np.arange(5)
      a[:,2] = 6
      print(a)
```

```
[0 1 2 3 4]
[0 6 6 6 4]
[6 1 6 3 6]
```

一方、リストに対しては、以下はエラーになります。

```
[32]: a = [0,1,2,3,4]
      print(a)
      a[1:4] = 6
```

```
[0, 1, 2, 3, 4]
```

```
-----
TypeError                                Traceback (most recent call last)
Input In [32], in <module>
      1 a = [0,1,2,3,4]
      2 print(a)
----> 3 a[1:4] = 6

TypeError: can only assign an iterable
```

```
[33]: a = [0,1,2,3,4]
      print(a)
      a[1:4] = [6]
      print(a)
```

```
[0, 1, 2, 3, 4]
[0, 6, 4]
```

このように、配列のスライスに対する代入の振舞いは、リストの場合と異なることに注意してください。多次元配列に対しては、インデックスの参照と同様に、高い次元のスライスから順に並べて指定します。

```
[34]: a = np.arange(9).reshape(3,3)
      print(a)
      print(a[:,2])
      print(a[1:,1:])
```

```
[[0 1 2]
 [3 4 5]
 [6 7 8]]
[[0 1]
 [3 4]]
[[4 5]
 [7 8]]
```

多次元配列に対するスライス、入れ子リストに対するスライスとは意味が異なることに注意してください。

17.4.3 for 文

リストと同様に、for 文を用いて、配列要素への反復処理を記述できます。

```
[35]: for v in np.arange(3):  
      print(v)
```

```
0  
1  
2
```

多次元配列の場合は、最外の配列に対して反復します。つまり、2次元配列の場合、行の配列に対する反復処理となります。

```
[36]: for row in np.arange(6).reshape(2,3):  
      print(row)
```

```
[0 1 2]  
[3 4 5]
```

for 文と併用される `enumerate()` の多次元配列版として、`numpy.ndenumerate()` 関数が提供されています。`numpy.ndenumerate()` は、(多次元) インデックスと要素の組を列挙します。

```
[37]: for idx, e in np.ndenumerate(np.arange(6).reshape(2,3)):  
      print(idx, e)
```

```
(0, 0) 0  
(0, 1) 1  
(0, 2) 2  
(1, 0) 3  
(1, 1) 4  
(1, 2) 5
```

```
[38]: for idx, e in np.ndenumerate(np.arange(3)):  
      print(idx, e)
```

```
(0,) 0  
(1,) 1  
(2,) 2
```

17.5 要素毎の演算

配列に対する要素毎の演算は、簡潔に記述できます。しかも、for 文で記述するより、効率がよいです。要素毎の演算を上手く使えるかどうか、NumPy プログラミングの肝と言っても過言ではないでしょう。

17.5.1 配列のスカラ演算

配列とスカラとの算術演算を記述すると、要素毎のスカラ演算となります。演算結果として、新しい配列が返ります。

```
[39]: a = np.arange(4)
print(a)

print(a + 1) # 各要素に 1 を加算
print(a - 1) # 各要素に 1 を減算
print(a * 2) # 各要素に 2 を乗算
print(a / 2) # 各要素を 2 で除算
print(a // 2) # 各要素を 2 で整数除算
print(a % 2) # 各要素に 2 の剰余演算
print(a ** 2) # 各要素を 2 乗

print(1 + a) # 左側がスカラでもよい
print(1 - a) # 左側がスカラでもよい
print(2 * a) # 左側がスカラでもよい
b = a + 1
print(1 / b) # 左側がスカラでもよい
print(9 // b) # 左側がスカラでもよい

[0 1 2 3]
[1 2 3 4]
[-1  0  1  2]
[0 2 4 6]
[0.  0.5 1.  1.5]
[0 0 1 1]
[0 1 0 1]
[0 1 4 9]
[1 2 3 4]
[ 1  0 -1 -2]
[0 2 4 6]
[1.          0.5          0.33333333 0.25          ]
[9 4 3 2]
```

17.5.2 配列同士の演算

形が同じ配列同士の算術演算は、同じ位置の要素同士の演算となります。演算結果として、新しい配列が返ります。

```
[40]: a = np.arange(4).reshape(2,2)
b = np.arange(1,5).reshape(2,2)
print(a)
print(b)
print(a + b)
print(a - b)
print(a * b)
print(a / b)
c = 3 * a
print(c // b)
print(a % b)
print(a ** b)

[[0 1]
 [2 3]]
[[1 2]
 [3 4]]
[[1 3]
 [5 7]]
[[0 0]
 [2 1]]
[[0 1]
 [2 3]]
[[0 3]
 [6 9]]
[[0 1]
 [2 3]]
[[0 1]
 [2 3]]
[[0 1]
 [2 3]]
[[0 1]
 [2 3]]
```

(continues on next page)

(continued from previous page)

```
[3 4]]
[[1 3]
 [5 7]]
[[-1 -1]
 [-1 -1]]
[[ 0  2]
 [ 6 12]]
[[0.          0.5          ]
 [0.66666667  0.75          ]]
[[0 1]
 [2 2]]
[[0 1]
 [2 3]]
[[ 0  1]
 [ 8 81]]
```

実は、形が同じでない配列同士の算術演算も可能ですが、振舞いが複雑なので間違いやすいです。配列同士の算術演算は、形が同じ配列に限定する方が賢明です。

17.5.3 ユニバーサル関数

NumPy には**ユニバーサル関数**と呼ばれる、任意の形の配列を取り、各要素に所定の演算を与えた結果を返す関数があります。その代表例は、`numpy.sqrt()` 関数です。

```
[41]: a = np.zeros(3) + 2
      print(a)
      print(np.sqrt(a)) # 各要素は sqrt(2)
      b = np.zeros((2,2)) + 2
      print(np.sqrt(b)) # 各要素は sqrt(2)
      print(np.sqrt(2)) # スカラ (0次元配列) も扱える

[2. 2. 2.]
[1.41421356 1.41421356 1.41421356]
[[1.41421356 1.41421356]
 [1.41421356 1.41421356]]
1.4142135623730951
```

この他にも、多数のユニバーサル関数が提供されています。詳しくは、[ユニバーサル関数の一覧](#)を参照してください。

17.6 よく使われる配列操作

17.6.1 dot

`numpy.dot()` は、2つの配列を引数に取り、そのドット積を返します。両者が1次元配列のときは、ベクトル内積と等しいです。

```
[42]: np.dot(np.arange(4), np.arange(1,5)) # 0*1 + 1*2 + 2*3 + 3*4
[42]: 20
```

2次元配列同士だと、行列乗算と等しいです。

```
[43]: # [[0 1]    [[1 2]
      # [2 3]] と [3 4]] の行列積
      print(np.dot(np.arange(4).reshape(2,2), np.arange(1,5).reshape(2,2)))
```

```
[[ 3  4]
 [11 16]]
```

17.6.2 sort

`numpy.sort()` 関数は、昇順でソートされた新しい配列を返します。これは、組み込み関数 `sorted()` の配列版です。

```
[44]: a = np.array([3, 4, -1, 0, 2])
      print(a)
      print(np.sort(a))
```

```
[ 3  4 -1  0  2]
[-1  0  2  3  4]
```

一方、配列の `sort()` メソッドは、配列を破壊的に（インプレースで）ソートします。これは、リストの `sort()` メソッドの配列版です。

```
[45]: a = np.array([3, 4, -1, 0, 2])
      print(a)
      a.sort()
      print(a)
```

```
[ 3  4 -1  0  2]
[-1  0  2  3  4]
```

17.6.3 sum, max, min, mean

配列のメソッド `sum()`・`max()`・`min()`・`mean()` は、それぞれ総和・最大値・最小値・算術平均を返します。これらのメソッドは、引数が与えられない場合、全要素を集計した結果を返します。多次元配列の場合、集計する次元を指定できます。具体的には、2次元配列の場合、`0` を指定すると各列に、`1` を指定すると各行に、対応するメソッドを適用した結果が返されます。

```
[46]: a = np.arange(6).reshape(2,3)
      print(a)
      print(a.sum())
      print(a.sum(0))
      print(a.sum(1))
```

```
[[0 1 2]
 [3 4 5]]
15
[3 5 7]
[ 3 12]
```

この他にも、多数の数学・統計関連のメソッドや関数が提供されています。詳しくは、[数学関数](#)や[統計関数](#)を参照してください。

17.7 配列の保存と復元

配列は、ファイルに保存したり、ファイルから読み出したりすることが、簡単にできます。

`numpy.savetxt()` 関数は、与えられた配列を指定されたファイル名をつけてテキスト形式で保存します。

```
[47]: np.savetxt('arange3.txt', np.arange(3))
```

この `arange3.txt` は、次のような内容になっているはずです。

```
0.0000000000000000e+00
1.0000000000000000e+00
2.0000000000000000e+00
```

2次元配列は、列が空白区切りで保存されます

```
[48]: np.savetxt('arange2x3.txt', np.arange(6).reshape(2,3))
```

この `arange2x3.txt` は、次のような内容になっているはずです。

```
0.0000000000000000e+00 1.0000000000000000e+00 2.0000000000000000e+00
3.0000000000000000e+00 4.0000000000000000e+00 5.0000000000000000e+00
```

一方、`numpy.loadtxt()` 関数は、与えられた名前のファイルに保存された配列を復元します。

```
[49]: a = np.loadtxt('arange2x3.txt')
print(a)
```

```
[[0. 1. 2.]
 [3. 4. 5.]]
```

保存するときに、列の区切り文字をデフォルトの `' '` 以外にしたい場合、`savetxt()` の `delimiter` 引数に区切り文字 (列) を指定します。これを復元するときには、`loadtxt()` の `delimiter` 引数に同じ値を指定する必要があります。ただし、区切り文字列は ASCII (正確には Latin-1) で解釈可能でなければなりません。

大規模な配列をテキスト形式で保存すると、ファイルサイズがとて大きくなります。そういう場合、圧縮保存が有効です。

保存するファイル名の拡張子を `.gz` とすることで、`savetxt()` は自動的に GZip 形式で圧縮して保存します。復元するファイル名の拡張子が `.gz` であれば、`loadtxt()` は GZip 形式だと判断して、自動的に解凍して復元します。

17.8 ▲真理値配列によるインデックスアクセス

配列に対して、比較演算を適用すると、算術演算と同様に要素毎に演算されて、真理値の配列が返ります。

```
[50]: a = np.arange(6)
print(a)
print(a < 3)
```

```
[0 1 2 3 4 5]
[ True  True  True False False False]
```

このように作られた真理値配列は、インデックスとして利用することができます。これによって、条件を満たす範囲を取り出すような記述が可能になります。次の具体例を見てみましょう。

```
[51]: a = np.array([0,1,2,-3,-4,5,-6,-7])
print(a)
print(a[a < 0]) # 負の要素を取り出し
print(a[(a < 0) & (a % 2 == 0)]) # 負で偶数の要素を取り出し
a[a < 0] = 8 # 負の要素を 8 に上書き
print(a)

[ 0  1  2 -3 -4  5 -6 -7]
[-3 -4 -6 -7]
[-4 -6]
[0 1 2 8 8 5 8 8]
```

一見すると単なる条件式のように見えますが、インデックスとなるのは真理値ではなく真理値の配列です。したがって、真理値を返す `and`・`or`・`not` の代わりに、要素毎の演算を行う `&`・`|`・`~` を用いる必要があります。

同様の記法は、7-1 で扱う `pandas` ライブラリでも利用されます。

17.9 ▲線形代数の演算

`numpy.dot()` は、2 次元配列を与えたときには、行列積となりました。それだけでなく、行列積専用の `numpy.matmul()` も提供されています。

また、単位行列は `numpy.identity()` 関数で作成することができます。引数に行列のサイズを指定します。

```
[52]: I = np.identity(3)
print(I)
a = np.arange(9).reshape(3,3)
print(a)
print(np.matmul(a, I))

[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
[[0 1 2]
 [3 4 5]
 [6 7 8]]
[[0. 1. 2.]
 [3. 4. 5.]
 [6. 7. 8.]]
```

`numpy.linalg.norm()` 関数は、与えられたベクトル（1 次元配列）もしくは行列（2 次元配列）のノルムを返します。

```
[53]: np.linalg.norm(np.ones(3)) # ユークリッドノルムを計算するので sqrt(3) と等しい
[53]: 1.7320508075688772
```

NumPy では、行列の分解、転置、行列式などの計算を含む線形代数の演算は、`numpy.linalg` モジュールで提供されています。詳しくは、[線形代数関連関数を参照してください](#)。

17.10 練習の解答

```
[54]: def arange_square_matrix(n):  
      return np.array([np.arange(i, n+i) for i in range(n)])
```

6-1. 内包表記

内包表記について説明します。

参考:

- <https://docs.python.org/ja/3/tutorial/datastructures.html#list-comprehensions>
- <https://docs.python.org/ja/3/tutorial/datastructures.html#nested-list-comprehensions>

18.1 リスト内包表記

Python では各種の内包表記 (comprehension) が利用できます。

以下のような整数の自乗を要素に持つリストを作るプログラムでは、

```
[1]: squares1 = []
    for x in range(6):
        squares1.append(x**2)
    squares1
```

```
[1]: [0, 1, 4, 9, 16, 25]
```

squares1 として [0, 1, 4, 9, 16, 25] が得られます。これを内包表記を用いて書き換えると、以下のように行で書け、プログラムが読みやすくなります。

```
[2]: squares2 = [x**2 for x in range(6)]
    squares2
```

```
[2]: [0, 1, 4, 9, 16, 25]
```

関数 `sum` は与えられた数のリストの総和を求めます。(2-2 の練習にあった `sum_list` と同じ機能を持つ組み込みの関数です。) 内包表記に対して `sum` を適用すると以下ようになります。

```
[3]: sum([x**2 for x in range(6)])
```

```
[3]: 55
```

以下の内包表記は 3-2 で用いられていました。

```
[4]: [chr(i + ord('a')) for i in range(26)]
```

```
[4]: ['a',
      'b',
      'c',
      'd',
      'e',
      'f',
      'g',
      'h',
      'i',
      'j',
      'k',
      'l',
      'm',
      'n',
      'o',
      'p',
      'q',
      'r',
      's',
      't',
      'u',
      'v',
      'w',
      'x',
      'y',
      'z']
```

18.2 練習

文字列のリストが変数 `strings` に与えられたとき、それぞれの文字列の長さからなるリストを返す内包表記を記述してください。

`strings = ['The', 'quick', 'brown']` のとき、結果は `[3, 5, 5]` となります。

```
[5]: strings = ['The', 'quick', 'brown']
      [ここに内包表記を書く]
```

```
-----
NameError                                Traceback (most recent call last)
Input In [5], in <module>
      1 strings = ['The', 'quick', 'brown']
----> 2 [ここに内包表記を書く]

NameError: name 'ここに内包表記を書く' is not defined
```

18.3 練習

コンマで区切られた 10 進数からなる文字列が変数 `str1` に与えられたとき、それぞれの 10 進数を数に変換して得られるリストを返す内包表記を記述してください。

`str1 = '123,45,-3'` のとき、結果は `[123, 45, -3]` となります。

なお、コンマで区切られた 10 進数からなる文字列を、10 進数の文字列のリストに変換するには、メソッド `split` を用いることができます。また、10 進数の文字列を数に変換するには、`int` を関数として用いることができます。

```
[6]: str1 = '123,45,-3'
     [ここに内包表記を書く]
```

```
-----
NameError                                Traceback (most recent call last)
Input In [6], in <module>
      1 str1 = '123,45,-3'
----> 2 [ここに内包表記を書く]

NameError: name 'ここに内包表記を書く' is not defined
```

18.4 練習

数のリストが与えられたとき、リストの要素の分散を求める関数 `var` を内包表記と関数 `sum` を用いて定義してください。以下のセルの ... のところを書き換えて `var` を作成してください。

```
[7]: def var(lst):
     ...
```

上のセルで解答を作成した後、以下のセルを実行し、実行結果が `True` になることを確認してください。

```
[8]: print(var([3,4,1,2]) == 1.25)

False
```

18.5 内包表記の入れ子

また内包表記を入れ子（ネスト）にすることも可能です：

```
[9]: [[x*y for y in range(x+1)] for x in range(4)]
[9]: [[0], [0, 1], [0, 2, 4], [0, 3, 6, 9]]
```

ネストした内包表記は、外側から読むとわかりやすいです。`x` を 0 から 3 まで動かしてリストが作られます。そのリストの要素 1 つ 1 つは内包表記によるリストになっていて、それぞれのリストは `y` を 0 から `x` まで動かして得られます。

以下のリストは、上の 2 重のリストをフラットにしたものです。この内包表記では、`for` が 2 重になっていますが、自然に左から読んでください。`x` を 0 から 3 まで動かし、そのそれぞれに対して `y` を 0 から `x` まで動かします。その各ステップで得られた `x*y` の値をリストにします。

```
[10]: [x*y for x in range(4) for y in range(x+1)]
[10]: [0, 0, 1, 0, 2, 4, 0, 3, 6, 9]
```

以下の関数は、与えられた文字列の全ての空でない部分文字列からなるリストを返します。

```
[11]: def allsubstrings(s):
      return [s[i:j] for i in range(len(s)) for j in range(i+1,len(s)+1)]

allsubstrings('abc')

[11]: ['a', 'ab', 'abc', 'b', 'bc', 'c']
```

18.6 練習

次のような関数 `sum_lists` を作成してください。

- `sum_lists` はリスト `list1` を引数とします。
- `list1` の各要素はリストであり、そのリストの要素は数です。
- `sum_lists` は、`list1` の各要素であるリストの総和を求め、それらの総和を足し合せて返します。

ここでは、内包表記と関数 `sum` を用いて `sum_lists` を定義してください。以下のセルの ... のところを書き換えて `sum_lists` を作成してください。

```
[12]: def sum_lists(list1):
      ...
```

上のセルで解答を作成した後、以下のセルを実行し、実行結果が `True` になることを確認してください。

```
[13]: print(sum_lists([[20, 5], [6, 16, 14, 5], [16, 8, 16, 17, 14], [1], [5, 3, 5, 7]])) == 158)
      ↪ 158)

False
```

18.7 練習

リスト `list1` と `list2` が引数として与えられたとき、次のようなリスト `list3` を返す関数 `sum_matrix` を作成してください。

- `list1, list2, list3` は、3つの要素を持ちます。
- 各要素は大きさ3のリストになっており、そのリストの要素は全て数です。
- `list3[i][j]` (ただし、`i` と `j` は共に、0 以上 2 以下の整数) は `list1[i][j]` と `list2[i][j]` の値の和になっています。

ここでは、内包表記を用いて `sum_matrix` を定義してください。以下のセルの ... のところを書き換えて `sum_matrix` を作成してください。

```
[14]: def sum_matrix(list1, list2):
      ...
```

上のセルで解答を作成した後、以下のセルを実行し、実行結果が `True` になることを確認してください。

```
[15]: print(sum_matrix([[1,5,3],[4,5,6],[7,8,9]], [[1,4,7],[2,5,8],[3,6,9]])) == [[2, 9, 10],
      ↪ [6, 10, 14], [10, 14, 18]])

False
```

18.8 ▲条件付き内包表記

内包表記は for に加えて if を使うこともできます:

```
[16]: words = ['cat', 'dog', 'elephant', None, 'giraffe']
length = [len(w) for w in words if w != None]
print(length)

[3, 3, 8, 7]
```

この場合、length として要素が None の場合を除いた [3, 3, 8, 7] が得られます。

18.9 ▲セット内包表記

内包表記はセット（集合）に対しても使うことができます:

```
[17]: words = ['cat', 'dog', 'elephant', 'giraffe']
length_set = {len(w) for w in words}
print(length_set)

{8, 3, 7}
```

length_set として {3, 7, 8} が得られます。セット型なので、リストと異なり重複する要素は除かれます。

18.10 ▲辞書内包表記

さらに、内包表記は辞書型でも使うことができます。

```
[18]: words = ['cat', 'dog', 'elephant', 'giraffe']
length_dic = {w:len(w) for w in words}
print(length_dic)

{'cat': 3, 'dog': 3, 'elephant': 8, 'giraffe': 7}
```

length_dic として {'cat': 3, 'dog': 3, 'elephant': 8, 'giraffe': 7} が得られます。

長さと文字列を逆にするとどうなるでしょうか。

```
[19]: length_rdic = {len(w): w for w in words}
print(length_rdic)

{3: 'dog', 8: 'elephant', 7: 'giraffe'}
```

18.11 ▲ジェネレータ式

内包表記と似たものとして、ジェネレータ式というものがあります。リスト内包表記の [] を () に置き換えれば、ジェネレータ式になります。ジェネレータ式は、4-2 で説明したイテレータを構築します。4-2 で説明したように、イテレータは、for 文で走査（全要素を訪問）できます。

```
[20]: it = (x * 3 for x in 'abc')
for x in it:
    print(x)
```

```
aaa
bbb
ccc
```

イテレータを組み込み関数 `list()` や `tuple()` に渡すと、対応するリストやタプルが構築されます。なお、ジェネレータ式を直接引数とするときには、ジェネレータ式の外側の `()` は省略可能です。

```
[21]: list(x ** 2 for x in range(5))
```

```
[21]: [0, 1, 4, 9, 16]
```

```
[22]: tuple(x ** 2 for x in range(5))
```

```
[22]: (0, 1, 4, 9, 16)
```

総和を計算する組み込み関数 `sum()` など、リストやタプルを引数に取れる大抵の関数には、イテレータも渡せます。

```
[23]: sum(x ** 2 for x in range(5))
```

```
[23]: 30
```

上の例において、ジェネレータ式の代わりにリスト内包表記を用いても同じ結果を得ますが、計算の途中で実際にリストを構築するので、メモリ消費が大きいです。ジェネレータ式では、リストのように走査できるイテレータを構築するだけなので、リスト内包表記よりメモリ効率がよいです。したがって、関数に渡すだけの一時オブジェクトには、リスト内包表記ではなくジェネレータ式を用いるのが有効です。

18.12 練習の解答

```
[24]: strings = ['The', 'quick', 'brown']
      [len(x) for x in strings]
```

```
[24]: [3, 5, 5]
```

```
[25]: str1 = '123,45,-3'
      [int(x) for x in str1.split(',')]

```

```
[25]: [123, 45, -3]
```

```
[26]: def var(lst):
      n = len(lst)
      av = sum(lst)/n
      return sum([(x - av)*(x - av) for x in lst])/n
```

```
[27]: def var(lst):
      n = len(lst)
      av = sum(lst)/n
      return sum([x*x for x in lst])/n - av*av
```

```
[28]: def sum_lists(list1):
      return sum([sum(lst) for lst in list1])
```

```
[29]: def sum_matrix(list1,list2):
      return [[list1[i][j] + list2[i][j] for j in range(3)] for i in range(3)]
```

[]:

6-2. 高階関数

Python における高階関数について説明します。

参考

- <https://docs.python.org/ja/3/howto/functional.html>

19.1 max

例として、関数 `max` について考察します。`max` は与えられたリストの要素のうち、最大のものを返します。

```
[1]: ls = [3,-8,1,0,7,-5]
      max(ls)
```

```
[1]: 7
```

`max` に `key` というキーワード引数として、たとえば関数 `abs` を与えることができます。（キーワード引数について詳しくは、3-3 を参照してください。）

```
[2]: max(ls, key=abs)
```

```
[2]: -8
```

この場合、各要素に関数 `abs` が適用されて、その結果が最も大きい要素が返ります。（各要素に `abs` 適用した結果の中の最大値が返るわけではないことに注意してください。）なお、`abs(x)` は `x` の絶対値を返します。

この場合、`max` という関数は、関数を引数として受け取っています。

一般に、関数を引数として受け取ったり返値として返したりする関数を **高階関数** といいます。

19.2 sorted

`sorted` も高階関数で、`max` と同様に `key` というキーワード引数を取ります。

```
[3]: sorted(ls, key=abs)
```

```
[3]: [0, 1, 3, -5, 7, -8]
```

このように、各要素に関数 `abs` を適用した結果によって、各要素をソートします。

リストを降順にソートするには、次のような関数を用いればよいです。

```
[4]: def invert(x):
      return -x
```

```
[5]: sorted(ls, key=invert)
```

```
[5]: [7, 3, 1, 0, -5, -8]
```

なお、リストを降順にソートするには、`reverse` というキーワード引数に `True` を指定するという方法もあります。

```
[6]: sorted(ls, reverse=True)
```

```
[6]: [7, 3, 1, 0, -5, -8]
```

19.3 ラムダ式

上の `invert` のような簡単な関数の場合、いちいち `def` で定義するのは面倒と思いませんか。

そのようなときは、`lambda` を使った**ラムダ式**（または**無名関数**）を用いることができます。上の例は、以下のように書くことができます。

```
[7]: sorted(ls, key=lambda x: -x)
```

```
[7]: [7, 3, 1, 0, -5, -8]
```

`lambda x: -x` という式は、`x` をもらって `-x` を返す関数を表します。`return` は書かないことに注意してください。

さて、ここまで関数と呼んでいたものは、Python では、オブジェクトの一種に他なりません。実際に、`abs` や `lambda x: -x` という式の値を調べてみてください。

```
[8]: abs
```

```
[8]: <function abs(x, /)>
```

```
[9]: lambda x: -x
```

```
[9]: <function __main__.<lambda>(x)>
```

したがって、Python では、関数を他の種類のデータ（数やリストや文字列など）と同様に、関数の引数にしたり、リストの要素にしたり、することができます。

19.4 リストからイテラブルへ

以上の例では、`max` や `sorted` はリストを受け取っていましたが、リストではなく、タプルでもいいですし、文字列でも構いません。

```
[10]: max((3,-8,1,0,7,-5))
```

```
[10]: 7
```

```
[11]: sorted((3,-8,1,0,7,-5))
```

```
[11]: [-8, -5, 0, 1, 3, 7]
```

```
[12]: max('hello world')
```

```
[12]: 'w'
```

```
[13]: sorted('hello world')
```

```
[13]: [' ', 'd', 'e', 'h', 'l', 'l', 'l', 'o', 'o', 'r', 'w']
```

すなわち、`max` や `sorted` は、一般にイテラブルを引数に取ることができます。

イテラブルについては4-2に説明がありましたが、簡単に言うと、**イテラブル**とは `for` 文の `in` の後に來ることができるものです。`max` や `sorted` は、イテラブルの各要素を次々と求めて、その中の最大値を求めたり、整列した結果をリストとして返したりします。

以下の例では、`max` にファイルオブジェクトが渡されます。ファイルオブジェクトはイテレータですので、イテラブルでもあります。ファイルオブジェクトを `for` 文の `in` の後に指定すると、ファイルの各行が文字列として得られます。以下の例では、`key` として関数 `len` が指定されていますので、ファイルの中で最も長い行が表示されます。

```
[14]: with open('jugemu.txt', 'r', encoding='utf-8') as f:
      print(max(f, key=len))
```

```
グーリンダイのポンポコピーのポンポコナーの、
```

辞書もイテラブルです。`max` に辞書与えると、最大のキーが返ります。

```
[15]: max({3:10, 5:2, 9:1})
```

```
[15]: 9
```

19.5 練習

辞書 `d` が与えられたとき、最大の値を持つキー（複数個ならばそのいずれか）を返す関数 `max_value_key(d)` を、`max` を使って定義してください。

ヒント：辞書 `d` のキー `k` に対して、`k` に対応する値を返す関数は `lambda k: d[k]` という式で表すことができます。

```
[16]: def max_value_key(d):
```

```
    ...
```

上のセルで解答を作成した後、以下のセルを実行し、実行結果が `True` になることを確認してください。

```
[17]: print(max_value_key({3:10, 5:2, 9:1}) == 3)
```

```
False
```

19.6 map

以下は内包表記の例です。

```
[18]: [abs(x) for x in [3,-8,1,0,7,-5]]
```

```
[18]: [3, 8, 1, 0, 7, 5]
```

リストの各要素に関数 `abs` が適用された結果がリストになります。

同様のことを、高階関数 `map` を用いて行うことができます。関数 `map` は、2 番目の引数としてイテラブルを取ります。1 番目の引数は関数です。例を見ましょう。

```
[19]: map(abs, [3,-8,1,0,7,-5])
```

```
[19]: <map at 0x7fde8c3e8730>
```

何が返ったか、よくわからないと思います。以下のように、`map` の結果を `for` 文の `in` の後に指定してみましょう。

```
[20]: for x in map(abs, [3,-8,1,0,7,-5]):
      print(x)
```

```
3
8
1
0
7
5
```

すなわち、`map` が返すものは**イテレータ**です。このイテレータは、2 番目の引数のイテラブル（この例ではリスト）の各要素に関数 `abs` を適用したものを、次々と返すようなイテレータです。

内包表記を使えば、以下のようにリストにまとめることができます。

```
[21]: [x for x in map(abs, [3,-8,1,0,7,-5])]
```

```
[21]: [3, 8, 1, 0, 7, 5]
```

関数 `list` をイテレータに適用してもよいです。

```
[22]: list(map(abs, [3,-8,1,0,7,-5]))
```

```
[22]: [3, 8, 1, 0, 7, 5]
```

しかし、これでは、イテレータがどのように動くのか、よくわからないと思います。

以下では、呼ばれるたびにメッセージを出力する関数 `abs1` を用います。

```
[23]: def abs1(x):
      print('abs called on', x)
      return abs(x)
```

```
[24]: it = map(abs1, [3,-8,1,0,7,-5])
```

このように、`map` がイテレータを返した時点では、各要素に対する計算は何も行われません。

このイテレータに `next` を適用するとどうなるか、見てください。

```
[25]: next(it)
abs called on 3
```

```
[25]: 3
```

```
[26]: next(it)
abs called on -8
```

```
[26]: 8
```

関数 `next` が呼ばれるたびに、次の要素を求める計算が行われていることがわかります。

イテレータはイテラブルですから、`map` の結果にさらに `map` を適用することができます。

```
[27]: list(map(lambda x: x+1, map(abs, [3,-8,1,0,7,-5])))
```

```
[27]: [4, 9, 2, 1, 8, 6]
```

`lambda x: x+1` は、`x` をもらって `x+1` を返す関数です。すなわち、引数に 1 を足した結果を返します。

関数 `sum` は、`max` と同様に、イテラブルを受け取って、その要素の総和を返します。したがって、`map` が返したイテレータに対しても適用できます。（イテレータをリストに変換する必要はありません。）

```
[28]: sum(map(lambda x: x+1, map(abs, [3,-8,1,0,7,-5])))
```

```
[28]: 30
```

19.7 練習

数のリストが与えられたとき、その要素の絶対値の最大値を返す関数 `max_abs` を、`map` と `max` を使って定義してください。

```
[29]: def max_abs(ln):
    ...
```

上のセルで解答を作成した後、以下のセルを実行し、実行結果が `True` になることを確認してください。

```
[30]: print(max_abs([3,-8,1,0,7,-5]) == 8)
```

```
False
```

19.8 filter

関数 `filter` もイテラブルをもらってイテレータを返します。最初の引数としては、真理値を返す関数を指定します。

```
[31]: def pos(x):
    if x>0:
        return True
    else:
        return False
```

この関数 `pos` は、引数が正ならば `True`、そうでなければ `False` を返します。

すると、以下のように、`filter` は `pos` を適用すると `True` が返る要素のみからなるイテレータを返します。

```
[32]: list(filter(pos, [3,-8,1,0,7,-5]))
```

```
[32]: [3, 1, 7]
```

`filter` は、条件付き内包表記に対応しています。同じ計算を以下のようにして行うことができます。

```
[33]: [x for x in [3,-8,1,0,7,-5] if pos(x)]
```

```
[33]: [3, 1, 7]
```

19.9 練習

数のリスト `ln` と数 `n` を受け取って、`ln` の要素のうち、`n` より大きい個数を返す関数 `number_of_big_numbers(ln, n)` を、`for` 文や `while` 文を用いずに、`filter` を用いて定義してください。

```
[34]: def number_of_big_numbers(ln, n):
```

```
    ...
```

上のセルで解答を作成した後、以下のセルを実行し、実行結果が `True` になることを確認してください。

```
[35]: print(number_of_big_numbers([10, 0, 7, 1, 5, 2, 9], 5) == 3)
```

```
False
```

19.10 練習

ファイル名 `file` と整数 `n` を受け取って、そのファイルをオープンし、（改行文字も含めて）長さが `n` より長い行の数を返す関数 `number_of_long_lines(file,n)` を定義してください。（ファイルは `encoding='utf-8'` でオープンしてください。）

```
[36]: def number_of_long_lines(file, n):
```

```
    ...
```

上のセルで解答を作成した後、以下のセルを実行し、実行結果が `True` になることを確認してください。

```
[37]: print(number_of_long_lines('jugemu.txt', 10) == 6)
```

```
False
```

19.11 練習の解答

```
[38]: def max_value_key(d):
       return max(d, key=lambda k: d[k])
```

```
[39]: def max_abs(ln):
       return max(map(abs, ln))
```

```
[40]: def number_of_big_numbers(ln, n):
       return sum(map(lambda x: 1, filter(lambda x: x>n, ln)))
```

```
[41]: def number_of_long_lines(file, n):  
      with open(file, 'r', encoding='utf-8') as f:  
          return sum(map(lambda x: 1, filter(lambda x: len(x)>n, f)))
```

```
[ ]:
```

6-3. クラス

Python におけるオブジェクト指向プログラミングのうち、クラスを定義する方法について簡単に説明します。

参考

- <https://docs.python.org/ja/3/tutorial/classes.html>
- <https://docs.python.org/ja/3/reference/datamodel.html>

20.1 クラス定義

Python では全てのデータはオブジェクトなのですが、以下では特に、クラス定義によって作成されたクラスを型とするデータを扱います。このようなデータは、**オブジェクト指向プログラミング**における典型的な**オブジェクト**です。そこで以下では、オブジェクトという用語をもっぱら使います。

4-1 で見たように、ファイルオブジェクトに対して `readline()` というメソッドを呼び出すと、ファイルの行が文字列として次々と返されます。ここでは、ファイルオブジェクトのようなオブジェクトで、`readline()` というメソッドが呼び出されると、常に `'Hello.\n'` という文字列を返すようなものを作ってみましょう。

そのためには、新しいクラスを定義します。**クラス**とは、オブジェクトの種類を意味します。新しいクラスを定義すると、そのクラスに属するオブジェクトを作ることができるようになります。それらのオブジェクトの型は、その新しいクラスになります。

ここでは、ずっと `'Hello.\n'` を返し続けるので、`HelloForEver` という名前を持つクラスを定義しましょう。そして、`HelloForEver` というクラスを型とするオブジェクトを作ります。

```
[1]: class HelloForEver:
      def readline(self):
          return 'Hello.\n'
```

一般にクラス定義は、以下のような形をしています。

```
class クラス名:
    def メソッド名(self, 引数, ...):
        実行文
    def メソッド名(self, 引数, ...):
        実行文
    ...
```


メソッド定義は関数定義と同じ形をしています、クラス定義の中に入っています。メソッド定義において、その最初の引数には慣例として `self` という名前を付けます。この引数には、メソッドが呼び出されたオブジェクト自身が渡されます。

上の例では、`readline` というメソッドが1つ定義されています。

以下のようにして、このクラスのオブジェクトを作ることができます。

```
[2]: f = HelloForEver()
```

`HelloForEver` を型とする新しいオブジェクトが作られて変数 `f` の値となります。

一般に、オブジェクトの生成は、

```
クラス名 (式, ...)
```

という式で行います。このようにオブジェクトを生成する式は**コンストラクタ**と呼ばれます。なお、上の例では、括弧の中に式は1つありません。

このようにして作ったオブジェクトの型を確認してください。

```
[3]: type(f)
```

```
[3]: __main__.HelloForEver
```

`__main__.HelloForEver` と表示されたでしょう。`__main__` は、ノートブックの式が評価されているモジュールを指すので、このオブジェクトの型が、上で定義した `HelloForEver` クラスであることがわかります。クラスのコンストラクタによって生成されたオブジェクトを、そのクラスの**インスタンス**と言います。上のオブジェクトは `HelloForEver` クラスのインスタンスです。

オブジェクトそのものは以下のように表示されます。

```
[4]: f
```

```
[4]: <__main__.HelloForEver at 0x7f705c56e910>
```

このオブジェクトに対して、`readline` というメソッドを呼び出すことができます。

```
[5]: f.readline()
```

```
[5]: 'Hello.\n'
```

この例では、`f` という変数に入っているオブジェクトが `self` という引数に渡されて、`readline` の本体である以下の文が実行されました。

```
return 'Hello.\n'
```

(この例では `self` は参照されていません。)

何回やっても同じです。

```
[6]: f.readline()
```

```
[6]: 'Hello.\n'
```

```
[7]: f.readline()
```

```
[7]: 'Hello.\n'
```

20.2 初期化と属性

以下の例では、初期化のメソッドが定義され、オブジェクトに属性が与えられます。

初期化のメソッドは `__init__` という名前を持ち、オブジェクトが作られたときに自動的に呼び出されます。`__init__` の引数は、オブジェクト自身と、クラス名の後に与えられる式の値です。

```
[8]: class HelloFile:
      def __init__(self, n):
          self.n = n
      def readline(self):
          if self.n == 0:
              return ''
          self.n = self.n - 1
          return 'Hello.\n'
```

この例では、以下のようにしてオブジェクトが作られます。

```
[9]: f = HelloFile(3)
```

すると、`HelloFile` を型とする新しいオブジェクトが作られて、そのオブジェクト自身が `self` に、3 が `n` に渡されて、`self.n = n` という文が実行されます。

`self.n` という式は、このオブジェクトの `n` という名前の属性を表します。

一般に、`class` の構文によって定義されたクラスを型とするオブジェクトは、属性を持つことができます。属性とは、個々のオブジェクトごとに記録される値であり、オブジェクト内の変数と考えられます。オブジェクトの属性は、オブジェクトに対してその属性名を指定して、参照したり設定したりできます。オブジェクトの属性は、`self.属性名` という式で参照されます。`self.属性名` を代入文の左辺に書けば、属性を設定することができます。

`self.n = n` のうち、`self.` の次の `n` は属性を表し、右辺の `n` は、`__init__` メソッドの引数を表していますので、混同しないようにしてください。

この例では、新しく作られたオブジェクトの `n` という属性が、引数 `n` の値である 3 に設定されます。

`readline` メソッドは以下のように定義されています。

```
def readline(self):
    if self.n == 0:
        return ''
    self.n = self.n - 1
    return 'Hello.\n'
```

オブジェクトの属性 `n` を参照して、それが 0 ならば空文字列を返します。そうでなければ、属性 `n` を 1 減らしてから文字列 `'Hello.\n'` を返します。

```
[10]: f.readline()
```

```
[10]: 'Hello.\n'
```

```
[11]: f.readline()
```

```
[11]: 'Hello.\n'
```

```
[12]: f.readline()
```

```
[12]: 'Hello.\n'
```

```
[13]: f.readline()
```

```
[13]: ''
```

変数 `f` の値であるオブジェクトの属性 `n` は、`f.n` という式によって参照できます。

```
[14]: f.n
```

```
[14]: 0
```

ここでは詳しく説明しませんが、オブジェクトのメソッドも属性の一種です。

20.3 継承

継承は、既存のクラスをもとにして、変更部分だけを与えることにより、新たなクラスを定義する機能です。

以下の例では、`HelloForEver` をもとにして `HelloFile` を定義しています。一般に、新しく定義されるクラスを**子クラス**、そのもとになるクラスを**親クラス**と言います。

```
[15]: class HelloFile(HelloForEver):
      def __init__(self, n):
          self.n = n
      def readline(self):
          if self.n == 0:
              return ''
          self.n = self.n - 1
          return super().readline()
```

ここでは、`__init__` と `readline` を新たに定義しています。

`HelloForEver` にも `readline` があります。こちらの `readline` は、`super().readline()` という式で呼び出すことができます。`super()` は、子クラスのオブジェクトに対して親クラスのメソッドを呼び出すための構文です。実際に、`HelloFile` の `readline` の中で、`HelloForEver` の `readline` を呼び出しています。

```
[16]: f = HelloFile(3)
```

```
[17]: f.readline()
```

```
[17]: 'Hello.\n'
```

20.4 特殊メソッド

Python では、**特殊メソッド**と呼ばれるメソッドが多数あります。これらのメソッドの名前は `__` で始まり `__` で終わります。

クラス定義の中で特殊メソッドを定義すると、そのクラスのオブジェクトに対して、その特殊メソッドに対応する機能が付与されます。初期化メソッド `__init__` も特殊メソッドですが、以下のクラス `HelloFileIterator` では、`__iter__` と `__next__` という特殊メソッドが定義されています。このクラスは、`HelloFile` を継承して定義されています。

`__iter__` メソッドは、オブジェクトに対して関数 `iter` が適用されたときに呼び出されます。`__iter__` メソッドの値が関数 `iter` の値となります。以下の例では、`__iter__` はオブジェクト自身を返しています。したがって、オブジェクトに `iter` が適用されると、オブジェクト自身が返ります。

```
[18]: class HelloFileIterator(HelloFile):
      def __iter__(self):
          return self
```

(continues on next page)

(continued from previous page)

```
def __next__(self):
    line = self.readline()
    if line == '':
        raise StopIteration
    return line
```

```
[19]: f = HelloFileIterator(3)
```

```
[20]: print(f is iter(f))
```

```
True
```

上の例で、`iter(f)` として関数 `iter` を呼び出すと、`f.__iter__()` としてメソッド `__iter__` が `f` に対して呼び出され、その結果が `iter(f)` の値となります。したがって、`iter(f)` は `f` と同じ値を返します。

`__next__` メソッドも、オブジェクトに対して関数 `next` が適用されたときに呼び出されます。`__next__` メソッドの値が `next` の値となります。

上の例では、`self.readline()` として、オブジェクト自身に対してメソッド `readline` を呼び出しています。その値が空文字列ならば、

```
raise StopIteration
```

という文を実行して、`StopIteration` というエラーを投げます。実は、このエラーは、`for` 文が捕まえて繰り返しを止める効果を持ちます。なお、`raise` は強制的にエラーを発生させる構文です。

```
[21]: for line in f:
        print(line)
```

```
Hello.
```

```
Hello.
```

```
Hello.
```

4-2 で説明したように、上の `for` 文では、まず `f` のオブジェクトに対して関数 `iter` が適用されます。すると `f` のオブジェクト自身が返ります。そして、このオブジェクトに対して関数 `next` が繰り返し適用されて、その結果が変数 `line` の値となります。`StopIteration` のエラーが検知されると、`for` 文が終了します。

20.5 継承による振舞いの改変

上で示された、`HelloFileIterator` の `__next__` メソッドでは、`self.readline()` というメソッド呼び出しがありました。上の例の振舞いから、そのメソッド呼び出しは、`HelloFileIterator` には `readline` メソッドが定義されていないので、親の `HelloFile` を見に行き、そこで定義された `readline` メソッドが使われたように見えます。しかし、それは正確ではありません。

`self.readline()` では、その呼び出し場所がどこであるかに関わらず、常にオブジェクト `self` 中のメソッドを探索します。そして、継承があるために、`__next__(self)` における `self` が、`HelloFileIterator` のインスタンスであるとも限りません。次を見てみましょう。

```
[22]: class EmptyFile(HelloFileIterator):
        def readline(self):
            return ''
```

```
f = EmptyFile(3)
next(f)
```

```

-----
StopIteration                                Traceback (most recent call last)
Input In [22], in <module>
      3         return ''
      5 f = EmptyFile(3)
----> 6 next(f)

Input In [18], in HelloFileIterator.__next__(self)
      5 line = self.readline()
      6 if line == '':
----> 7     raise StopIteration
      8 return line

StopIteration:

```

コンストラクタに3を与えているので、`HelloFileIterator`と同様に`next`を3回適用できてもよさそうですが、即座に`StopIteration`が生じました。これは、`__next__(self)`における`self`が、`EmptyFile`のインスタンスであり、`self.readline()`が常に`''`を返すからです。

このように、継承は、メソッドの部分的な再定義を通じて、再定義されたメソッドを呼び出しているメソッドの振舞いを、間接的に改変することを可能にします。

20.6 練習

'Hello.\n'ではなくて、初期時に指定された文字列を繰り返し返すように、新たなクラス`StringFileIterator`を定義してください。

`StringFileIterator`は`HelloFileIterator`を継承し、初期化メソッドには、文字列と回数を指定します。

```

[23]: class StringFileIterator(HelloFileIterator):
      def __init__(self, s, n):
          ...
          ...

```

上のセルで解答を作成した後、以下のセルを実行し、実行結果が`True`になることを確認してください。

```

[24]: f = StringFileIterator('abc', 3)
      print(list(f) == ['abc', 'abc', 'abc'])

```

```

-----
AttributeError                                Traceback (most recent call last)
Input In [24], in <module>
      1 f = StringFileIterator('abc', 3)
----> 2 print(list(f) == ['abc', 'abc', 'abc'])

Input In [18], in HelloFileIterator.__next__(self)
      4 def __next__(self):
----> 5     line = self.readline()
      6     if line == '':
      7         raise StopIteration

Input In [15], in HelloFile.readline(self)
      4 def readline(self):
----> 5     if self.n == 0:
      6         return ''
      7     self.n = self.n - 1

```

(continues on next page)

(continued from previous page)

```
AttributeError: 'StringFileIterator' object has no attribute 'n'
```

20.7 ▲ with 文への対応

ここでは詳しく説明しませんが、さらに特殊メソッドである `__enter__` と `__exit__` を定義すると、**with** 文にも対応できます。

```
[25]: class HelloFileIterator(HelloFile):
        def __enter__(self):
            return self
        def __exit__(self, exception_type, exception_value, traceback):
            pass
        def __next__(self):
            line = self.readline()
            if line == '':
                raise StopIteration
            return line
        def __iter__(self):
            return self
```

```
[26]: with HelloFileIterator(3) as f:
        for line in f:
            print(line)
```

```
Hello.
```

```
Hello.
```

```
Hello.
```

20.8 練習の解答

```
[27]: class StringFileIterator(HelloFileIterator):
        def __init__(self, s, n):
            self.s = s
            self.n = n
        def readline(self):
            if self.n == 0:
                return ''
            self.n = self.n - 1
            return self.s
```

```
[ ]:
```

7-1. pandas ライブラリ

pandas ライブラリについて説明します。

参考

- http://pandas.pydata.org/pandas-docs/stable/getting_started/index.html
- <http://pandas.pydata.org/pandas-docs/stable/>

pandas ライブラリにはデータ分析作業を支援するためのモジュールが含まれています。以下では、pandas ライブラリのモジュールの基本的な使い方について説明します。

pandas ライブラリを使用するには、まず **pandas** モジュールをインポートします。慣例として、同モジュールを **pd** と別名をつけてコードの中で使用します。データの生成に用いるため、ここでは **numpy** モジュールも併せてインポートします。

```
[1]: import pandas as pd
import numpy as np
```

21.1 シリーズとデータフレーム

pandas モジュールは、リスト、配列や辞書などのデータを**シリーズ (Series)** あるいは**データフレーム (DataFrame)** のオブジェクトとして保持します。シリーズは列、データフレームは複数の列で構成されます。シリーズやデータフレームの行は**インデックス index** で管理され、インデックスには **0** から始まる番号、または任意のラベルが付けられています。インデックスが番号の場合は、シリーズやデータフレームはそれぞれ NumPy の配列、2 次元配列とみなすことができます。また、インデックスがラベルの場合は、ラベルをキー、各行を値とした辞書としてシリーズやデータフレームをみなすことができます。

21.2 シリーズ (Series) の作成

シリーズのオブジェクトは、以下のように、リスト、配列や辞書から作成することができます。

```
[2]: # リストからシリーズの作成
s1 = pd.Series([0,1,2])
print(s1)

# 配列からシリーズの作成
s2 = pd.Series(np.random.rand(3))
print(s2)

# 辞書からシリーズの作成
s3 = pd.Series({0:'boo',1:'foo',2:'woo'})
print(s3)
```

```
0    0
1    1
2    2
dtype: int64
0    0.155919
1    0.040590
2    0.842579
dtype: float64
0    boo
1    foo
2    woo
dtype: object
```

以下では、シリーズ（列）より一般的なデータフレームの操作と機能について説明していきますが、データフレームオブジェクトの多くの操作や機能はシリーズオブジェクトにも適用できます。

21.3 データフレーム (DataFrame) の作成

データフレームのオブジェクトは、以下のように、リスト、配列や辞書から作成することができます。行のラベルは、`DataFrame` の `index` 引数で指定できますが、以下のデータフレーム作成の例、`d2`、`d3`、では同インデックスを省略しているため、`0` から始まるインデックス番号がラベルとして行に自動的に付けられます。列のラベルは `columns` 引数で指定します。辞書からデータフレームを作成する際は、`columns` 引数で列の順番を指定することになります。

```
[3]: # 多次元リストからデータフレームの作成
d1 = pd.DataFrame([[0,1,2],[3,4,5],[6,7,8],[9,10,11]], index=[10,11,12,13], columns=[
    ↪ 'c1', 'c2', 'c3'])
print(d1)

# 多次元配列からデータフレームの作成
d2 = pd.DataFrame(np.random.rand(12).reshape(4,3), columns=['c1', 'c2', 'c3'])
print(d2)

# 辞書からデータフレームの作成
d3 = pd.DataFrame({'Initial':['B','F','W'], 'Name':['boo', 'foo', 'woo']}, columns=[
    ↪ 'Name', 'Initial'])
print(d3)
```

```
   c1  c2  c3
10  0   1   2
11  3   4   5
```

(continues on next page)

(continued from previous page)

```

12  6  7  8
13  9 10 11
      c1      c2      c3
0  0.399834  0.712746  0.428730
1  0.918278  0.579665  0.437314
2  0.686534  0.361645  0.692042
3  0.284798  0.081365  0.630376
Name Initial
0  boo      B
1  foo      F
2  woo      W

```

21.4 CSV ファイルからのデータフレームの作成

pandas モジュールの `read_csv()` 関数を用いて、以下のように CSV ファイルを読み込んで、データフレームのオブジェクトを作成することができます。`read_csv()` 関数の `encoding` 引数にはファイルの文字コードを指定します。CSV ファイル `iris.csv` には、以下のようにアヤメの種類 (species) と花弁 (petal) ・ がく片 (sepal) の長さ (length) と幅 (width) のデータが含まれています。

```

sepal_length, sepal_width, petal_length, petal_width, species
5.1, 3.5, 1.4, 0.2, setosa
4.9, 3.0, 1.4, 0.2, setosa
4.7, 3.2, 1.3, 0.2, setosa
...

```

`head()` メソッドを使うとデータフレームの先頭の複数行を表示させることができます。引数には表示させたい行数を指定し、行数を指定しない場合は、5 行分のデータが表示されます。

```
[4]: # CSV ファイルの読み込み
iris_d = pd.read_csv('iris.csv')
```

```
# 先頭 10 行のデータを表示
iris_d.head(10)
```

```
[4]:   sepal_length  sepal_width  petal_length  petal_width  species
0           5.1           3.5           1.4           0.2   setosa
1           4.9           3.0           1.4           0.2   setosa
2           4.7           3.2           1.3           0.2   setosa
3           4.6           3.1           1.5           0.2   setosa
4           5.0           3.6           1.4           0.2   setosa
5           5.4           3.9           1.7           0.4   setosa
6           4.6           3.4           1.4           0.3   setosa
7           5.0           3.4           1.5           0.2   setosa
8           4.4           2.9           1.4           0.2   setosa
9           4.9           3.1           1.5           0.1   setosa
```

データフレームオブジェクトの `index` 属性により、データフレームのインデックスの情報が確認できます。`len()` 関数を用いると、データフレームの行数が取得できます。

```
[5]: print(iris_d.index) #インデックスの情報
len(iris_d.index) #インデックスの長さ
```

```
RangeIndex(start=0, stop=150, step=1)
```

```
[5]: 150
```

21.5 データの参照

シリーズやデータフレームでは、行の位置（行は 0 から始まります）をスライスとして指定することで任意の行を抽出することができます。

```
[6]: # データフレームの先頭 5 行のデータ
iris_d[:5]
```

```
[6]:      sepal_length  sepal_width  petal_length  petal_width  species
0           5.1           3.5           1.4           0.2   setosa
1           4.9           3.0           1.4           0.2   setosa
2           4.7           3.2           1.3           0.2   setosa
3           4.6           3.1           1.5           0.2   setosa
4           5.0           3.6           1.4           0.2   setosa
```

```
[7]: # データフレームの終端 5 行のデータ
iris_d[-5:]
```

```
[7]:      sepal_length  sepal_width  petal_length  petal_width  species
145           6.7           3.0           5.2           2.3  virginica
146           6.3           2.5           5.0           1.9  virginica
147           6.5           3.0           5.2           2.0  virginica
148           6.2           3.4           5.4           2.3  virginica
149           5.9           3.0           5.1           1.8  virginica
```

データフレームから任意の列を抽出するには、`DataFrame`. 列名 のように、データフレームオブジェクトに . で列名をつなげることで、その列を指定してシリーズオブジェクトとして抽出することができます。なお、列名を文字列として、`DataFrame[' 列名']` のように添字指定しても同様です。

```
[8]: # データフレームの'species'の列の先頭 10 行のデータ
iris_d['species'].head(10)
```

```
[8]: 0    setosa
1    setosa
2    setosa
3    setosa
4    setosa
5    setosa
6    setosa
7    setosa
8    setosa
9    setosa
Name: species, dtype: object
```

データフレームの添字として、列名のリストを指定すると複数の列をデータフレームオブジェクトとして抽出することができます。

```
[9]: # データフレームの'sepal_length'と'species'の列の先頭 10 行のデータ
iris_d[['sepal_length', 'species']].head(10)
```

```
[9]:      sepal_length  species
0           5.1   setosa
1           4.9   setosa
2           4.7   setosa
3           4.6   setosa
4           5.0   setosa
5           5.4   setosa
6           4.6   setosa
7           5.0   setosa
```

(continues on next page)

(continued from previous page)

8	4.4	setosa
9	4.9	setosa

21.5.1 iloc と loc

データフレームオブジェクトの `iloc` 属性を用いると、NumPy の多次元配列のスライスと同様に、行と列の位置を指定して任意の行と列を抽出することができます。

```
[10]: # データフレームの 2 行のデータ
iris_d.iloc[1]
```

```
[10]: sepal_length    4.9
      sepal_width      3
      petal_length    1.4
      petal_width     0.2
      species        setosa
      Name: 1, dtype: object
```

```
[11]: # データフレームの 2 行, 2 列目のデータ
iris_d.iloc[1, 1]
```

```
[11]: 3.0
```

```
[12]: # データフレームの 1 から 5 行目と 1 から 2 列目のデータ
iris_d.iloc[0:5, 0:2]
```

```
[12]:   sepal_length  sepal_width
0         5.1           3.5
1         4.9           3.0
2         4.7           3.2
3         4.6           3.1
4         5.0           3.6
```

データフレームオブジェクトの `loc` 属性を用いると、抽出したい行のインデックス・ラベルや列のラベルを指定して任意の行と列を抽出することができます。複数のラベルはリストで指定します。行のインデックスは各行に割り当てられた番号で、`iloc` で指定する行の位置とは必ずしも一致しないことに注意してください。

```
[13]: # データフレームの行インデックス 5 のデータ
iris_d.loc[5]
```

```
[13]: sepal_length    5.4
      sepal_width    3.9
      petal_length    1.7
      petal_width     0.4
      species        setosa
      Name: 5, dtype: object
```

```
[14]: # データフレームの行インデックス 5 と 'sepal_length' と列のデータ
iris_d.loc[5, 'sepal_length']
```

```
[14]: 5.4
```

```
[15]: # データフレームの行インデックス 1 から 5 と 'sepal_length' と species' の列のデータ
iris_d.loc[1:5, ['sepal_length', 'species']]
```

```
[15]:      sepal_length species
1          4.9    setosa
2          4.7    setosa
3          4.6    setosa
4          5.0    setosa
5          5.4    setosa
```

21.6 データの条件取り出し

データフレームの列の指定と併せて条件を指定することで、条件にあった行からなるデータフレームを抽出することができます。NumPy の多次元配列の真理値配列によるインデックスアクセスと同様に、条件式のブール演算では、`and`, `or`, `not` の代わりに `&`, `|`, `~` を用います。

```
[16]: # データフレームの'sepal_length'列の値が7より大きく、'species'列の値が3より小さいデータ
iris_d[(iris_d['sepal_length'] > 7.0) & (iris_d['sepal_width'] < 3.0)]
```

```
[16]:      sepal_length  sepal_width  petal_length  petal_width  species
107          7.3          2.9          6.3          1.8  virginica
118          7.7          2.6          6.9          2.3  virginica
122          7.7          2.8          6.7          2.0  virginica
130          7.4          2.8          6.1          1.9  virginica
```

21.7 列の追加と削除

データフレームに列を追加する場合は、以下のように、追加したい新たな列名を指定し、値を代入すると新たな列を追加できます。

```
[17]: # データフレームに'mycolumn'という列を追加
iris_d['mycolumn']=np.random.rand(len(iris_d.index))
iris_d.head(10)
```

```
[17]:      sepal_length  sepal_width  petal_length  petal_width  species  mycolumn
0          5.1          3.5          1.4          0.2    setosa  0.344553
1          4.9          3.0          1.4          0.2    setosa  0.414366
2          4.7          3.2          1.3          0.2    setosa  0.548023
3          4.6          3.1          1.5          0.2    setosa  0.908320
4          5.0          3.6          1.4          0.2    setosa  0.963215
5          5.4          3.9          1.7          0.4    setosa  0.899296
6          4.6          3.4          1.4          0.3    setosa  0.908529
7          5.0          3.4          1.5          0.2    setosa  0.133910
8          4.4          2.9          1.4          0.2    setosa  0.300779
9          4.9          3.1          1.5          0.1    setosa  0.041093
```

`del` 文を用いると、以下のようにデータフレームから任意の列を削除できます。

```
[18]: # データフレームから'mycolumn'という列を削除
del iris_d['mycolumn']
iris_d.head(10)
```

```
[18]:      sepal_length  sepal_width  petal_length  petal_width  species
0          5.1          3.5          1.4          0.2    setosa
1          4.9          3.0          1.4          0.2    setosa
2          4.7          3.2          1.3          0.2    setosa
3          4.6          3.1          1.5          0.2    setosa
4          5.0          3.6          1.4          0.2    setosa
```

(continues on next page)

(continued from previous page)

5	5.4	3.9	1.7	0.4	setosa
6	4.6	3.4	1.4	0.3	setosa
7	5.0	3.4	1.5	0.2	setosa
8	4.4	2.9	1.4	0.2	setosa
9	4.9	3.1	1.5	0.1	setosa

`assign()` メソッドを用いると、追加したい列名とその値を指定することで、以下のように新たな列を追加したデータフレームを新たに作成することができます。この際、元のデータフレームは変更されないことに注意してください。

```
[19]: # データフレームに'mycolumn'という列を追加し新しいデータフレームを作成
myiris1 = iris_d.assign(mycolumn=np.random.rand(len(iris_d.index)))
myiris1.head(5)
```

```
[19]:   sepal_length  sepal_width  petal_length  petal_width  species  mycolumn
0           5.1           3.5           1.4           0.2   setosa   0.236054
1           4.9           3.0           1.4           0.2   setosa   0.262237
2           4.7           3.2           1.3           0.2   setosa   0.337256
3           4.6           3.1           1.5           0.2   setosa   0.475727
4           5.0           3.6           1.4           0.2   setosa   0.204825
```

`drop()` メソッドを用いると、削除したい列名を指定することで、以下のように任意の列を削除したデータフレームを新たに作成することができます。列を削除する場合は、`axis` 引数に 1 を指定します。この際、元のデータフレームは変更されないことに注意してください。

```
[20]: # データフレームから'mycolumn'という列を削除し、新しいデータフレームを作成
myiris2 = myiris1.drop('mycolumn',axis=1)
myiris2.head(5)
```

```
[20]:   sepal_length  sepal_width  petal_length  petal_width  species
0           5.1           3.5           1.4           0.2   setosa
1           4.9           3.0           1.4           0.2   setosa
2           4.7           3.2           1.3           0.2   setosa
3           4.6           3.1           1.5           0.2   setosa
4           5.0           3.6           1.4           0.2   setosa
```

21.8 行の追加と削除

`pandas` モジュールの `append()` メソッドを用いると、データフレームに新たな行を追加することができます。以下では、`iris_d` データフレームの最終行に新たな行を追加しています。`ignore_index` 引数を `True` にすると追加した行に新たなインデックス番号がつけられます。

```
[21]: # 追加する行のデータフレーム
row = pd.DataFrame([[1,1,1,1, 'setosa']], columns=iris_d.columns)

# データフレームに行を追加し新しいデータフレームを作成
myiris4 = iris_d.append(row, ignore_index=True)
myiris4[-2:]
```

```
[21]:   sepal_length  sepal_width  petal_length  petal_width  species
149           5.9           3.0           5.1           1.8  virginica
150           1.0           1.0           1.0           1.0    setosa
```

`drop()` メソッドを用いると、行のインデックスまたはラベルを指定することで行を削除することもできます。このときに、`axis` 引数は省略することができます。

```
[22]: # データフレームから行インデックス 150 の行を削除し、新しいデータフレームを作成
myiris4 = myiris4.drop(150)
myiris4[-2:]
```

```
[22]:      sepal_length  sepal_width  petal_length  petal_width  species
148           6.2           3.4           5.4           2.3  virginica
149           5.9           3.0           5.1           1.8  virginica
```

21.9 データの並び替え

データフレームオブジェクトの `sort_index()` メソッドで、データフレームのインデックスに基づくソートができます。また、`sort_values()` メソッドで、任意の列の値によるソートができます。列は複数指定することもできます。いずれのメソッドでも、`inplace` 引数により、ソートにより新しいデータフレームを作成する (False) か、元のデータフレームを更新する (True) を指定できます。デフォルトは `inplace` は False になっており、これらのメソッドは新しいデータフレームを作成します。

```
[23]: # iris_d データフレームの 4 つ列の値に基づいて昇順にソート
sorted_iris = iris_d.sort_values(['sepal_length', 'sepal_width', 'petal_length',
↪ 'petal_width'])
sorted_iris.head(10)
```

```
[23]:      sepal_length  sepal_width  petal_length  petal_width  species
13           4.3           3.0           1.1           0.1  setosa
8            4.4           2.9           1.4           0.2  setosa
38           4.4           3.0           1.3           0.2  setosa
42           4.4           3.2           1.3           0.2  setosa
41           4.5           2.3           1.3           0.3  setosa
3            4.6           3.1           1.5           0.2  setosa
47           4.6           3.2           1.4           0.2  setosa
6            4.6           3.4           1.4           0.3  setosa
22           4.6           3.6           1.0           0.2  setosa
2            4.7           3.2           1.3           0.2  setosa
```

列の値で降順にソートする場合は、`sort_values()` メソッドの `ascending` 引数を False にしてください。

```
[24]: # iris_d データフレームの 4 つ列の値に基づいて降順にソート
sorted_iris = iris_d.sort_values(['sepal_length', 'sepal_width', 'petal_length',
↪ 'petal_width'], ascending=False)
sorted_iris.head(10)
```

```
[24]:      sepal_length  sepal_width  petal_length  petal_width  species
131           7.9           3.8           6.4           2.0  virginica
117           7.7           3.8           6.7           2.2  virginica
135           7.7           3.0           6.1           2.3  virginica
122           7.7           2.8           6.7           2.0  virginica
118           7.7           2.6           6.9           2.3  virginica
105           7.6           3.0           6.6           2.1  virginica
130           7.4           2.8           6.1           1.9  virginica
107           7.3           2.9           6.3           1.8  virginica
109           7.2           3.6           6.1           2.5  virginica
125           7.2           3.2           6.0           1.8  virginica
```

21.10 データの統計量

データフレームオブジェクトの `describe()` メソッドで、データフレームの各列の要約統計量を求めることができます。要約統計量には平均、標準偏差、最大値、最小値などが含まれます。その他の統計量を求める `pandas` モジュールのメソッドは以下を参照してください。

`pandas` での統計量計算

```
[25]: # iris_d データフレームの各数値列の要約統計量を表示
iris_d.describe()
```

```
[25]:
```

	sepal_length	sepal_width	petal_length	petal_width
count	150.000000	150.000000	150.000000	150.000000
mean	5.843333	3.054000	3.758667	1.198667
std	0.828066	0.433594	1.764420	0.763161
min	4.300000	2.000000	1.000000	0.100000
25%	5.100000	2.800000	1.600000	0.300000
50%	5.800000	3.000000	4.350000	1.300000
75%	6.400000	3.300000	5.100000	1.800000
max	7.900000	4.400000	6.900000	2.500000

21.11 ▲データの連結

`pandas` モジュールの `concat()` 関数を用いると、データフレームを連結して新たなデータフレームを作成することができます。以下では、`iris_d` データフレームの先頭 5 行と最終 5 行を連結して、新しいデータフレームを作成しています。

```
[26]: # iris_d データフレームの先頭 5 行と最終 5 行を連結
concat_iris = pd.concat([iris_d[:5], iris_d[-5:]])
concat_iris
```

```
[26]:
```

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa
145	6.7	3.0	5.2	2.3	virginica
146	6.3	2.5	5.0	1.9	virginica
147	6.5	3.0	5.2	2.0	virginica
148	6.2	3.4	5.4	2.3	virginica
149	5.9	3.0	5.1	1.8	virginica

`concat()` 関数の `axis` 引数に 1 を指定すると、以下のように、データフレームを列方向に連結することができます。

```
[27]: # iris_d データフレームの 'sepal_length' 列と 'species' 列を連結
sepal_len = pd.concat([iris_d.loc[:, ['sepal_length']], iris_d.loc[:, ['species']]],
    ↪ axis=1)
sepal_len.head(10)
```

```
[27]:
```

	sepal_length	species
0	5.1	setosa
1	4.9	setosa
2	4.7	setosa
3	4.6	setosa
4	5.0	setosa

(continues on next page)

(continued from previous page)

```

5          5.4  setosa
6          4.6  setosa
7          5.0  setosa
8          4.4  setosa
9          4.9  setosa

```

21.12 ▲データの結合

pandas モジュールの `merge()` 関数を用いると、任意の列の値をキーとして異なるデータフレームを結合することができます。結合のキーとする列名は `on` 引数で指定します。以下では、`species` の列の値をキーに、2つのデータフレーム、`sepal_len`, `sepal_wid`、を結合して新しいデータフレーム `sepal` を作成しています。

```

[28]: # 'sepal_length' と 'species' 列からなる 3 行のデータ
sepal_len = pd.concat([iris_d.loc[[0,51,101],['sepal_length']],iris_d.loc[[0,51,101],
↪['species']]], axis=1)
# 'sepal_width' と 'species' 列からなる 3 行のデータ
sepal_wid = pd.concat([iris_d.loc[[0,51,101],['sepal_width']],iris_d.loc[[0,51,101],
↪['species']]], axis=1)

# sepal_len と sepal_wid を 'species' をキーにして結合
sepal = pd.merge(sepal_len, sepal_wid, on='species')
sepal

```

```

[28]:   sepal_length  species  sepal_width
0          5.1    setosa          3.5
1          6.4  versicolor          3.2
2          5.8   virginica          2.7

```

21.13 ▲データのグループ化

データフレームオブジェクトの `groupby()` メソッドを使うと、データフレームの任意の列の値に基づいて、同じ値を持つ行をグループにまとめることができます。列は複数指定することもできます。`groupby()` メソッドを適用するとグループ化オブジェクト (`DataFrameGroupBy`) が作成されますが、データフレームと同様の操作を多く適用することができます。

```

[29]: # iris_d データフレームの 'species' の値で行をグループ化
iris_d.groupby('species')

```

```

[29]: <pandas.core.groupby.generic.DataFrameGroupBy object at 0x7fad82476130>

```

```

[30]: # グループごとの先頭 5 行を表示
iris_d.groupby('species').head(5)

```

```

[30]:   sepal_length  sepal_width  petal_length  petal_width  species
0          5.1          3.5          1.4          0.2    setosa
1          4.9          3.0          1.4          0.2    setosa
2          4.7          3.2          1.3          0.2    setosa
3          4.6          3.1          1.5          0.2    setosa
4          5.0          3.6          1.4          0.2    setosa
50         7.0          3.2          4.7          1.4  versicolor
51         6.4          3.2          4.5          1.5  versicolor
52         6.9          3.1          4.9          1.5  versicolor

```

(continues on next page)

(continued from previous page)

53	5.5	2.3	4.0	1.3	versicolor
54	6.5	2.8	4.6	1.5	versicolor
100	6.3	3.3	6.0	2.5	virginica
101	5.8	2.7	5.1	1.9	virginica
102	7.1	3.0	5.9	2.1	virginica
103	6.3	2.9	5.6	1.8	virginica
104	6.5	3.0	5.8	2.2	virginica

```
[31]: # グループごとの'sepal_length'列,'sepal_width'列の値の平均を表示
iris_d.groupby('species')[['sepal_length','sepal_width']].mean()
```

```
[31]:
```

	sepal_length	sepal_width
species		
setosa	5.006	3.418
versicolor	5.936	2.770
virginica	6.588	2.974

21.14 ▲欠損値、時系列データの処理

pandas では、データ分析における欠損値、時系列データの処理を支援するための便利な機能が提供されています。詳細は以下を参照してください。

欠損値の処理

時系列データの処理

7-2. scikit-learn ライブラリ

scikit-learn ライブラリについて説明します。

参考

- <https://scikit-learn.org/stable/tutorial/index.html>
- https://scikit-learn.org/stable/getting_started.html

機械学習の各手法の詳細については以下を参考にしてください

- <https://elf-c.he.u-tokyo.ac.jp/courses/364> （線形回帰）
- <https://elf-c.he.u-tokyo.ac.jp/courses/365> （ロジスティック回帰）
- <https://elf-c.he.u-tokyo.ac.jp/courses/360> （クラスタリング）
- <https://elf-c.he.u-tokyo.ac.jp/courses/363> （次元削減（主成分分析））

scikit-learn ライブラリには分類、回帰、クラスタリング、次元削減、前処理、モデル選択などの機械学習の処理を行うためのモジュールが含まれています。以下では、scikit-learn ライブラリのモジュールの基本的な使い方について説明します。

以下の説明では **scikit-learn** ライブラリのバージョン **0.22** 以降を想定しています。Anaconda（Individual Edition 2020.02）では同 0.22 がインストールされています。colaboratory でも同 0.22 以降が使用可能です。

22.1 機械学習について

機械学習では、観察されたデータをよく表すようにモデルのパラメータの調整を行います。パラメータを調整することでモデルをデータに適合させるので、「学習」と呼ばれます。学習されたモデルを使って、新たに観測されたデータに対して予測を行うことが可能になります。

22.2 教師あり学習

機械学習において、観測されたデータの特徴（特徴量）に対して、そのデータに関するラベルが存在する時、**教師あり学習**と呼びます。教師あり学習では、ラベルを教師として、データからそのラベルを予測するようなモデルを学習することになります。この時、ラベルが連続値であれば回帰、ラベルが離散値であれば分類の問題となります。

22.3 教師なし学習

ラベルが存在せず、観測されたデータの特徴のみからそのデータセットの構造やパターンをよく表すようなモデルを学習することを**教師なし学習**と呼びます。クラスタリングや次元削減は教師なし学習です。クラスタリングでは、観測されたデータをクラスタと呼ばれる集合にグループ分けします。次元削減では、データの特徴をより簡潔に（低い次元で）表現します。

22.4 データ

機械学習に用いるデータセットは、データフレームあるいは2次元の配列として表すことができます。行はデータセットの個々のデータを表し、列はデータが持つ特徴を表します。以下では、例として `pandas` モジュールの説明で用いたアイリスデータセットを表示しています。

```
[1]: import pandas as pd
iris = pd.read_csv('iris.csv')
iris.head(5)
```

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa

データセットの各行は1つの花のデータに対応しており、行数はデータセットの花データの総数を表します。また、1列目から4列目までの各列は花の特徴（特徴量）に対応しています。scikit-learnでは、このデータと特徴量からなる2次元配列（行列）をNumPy配列またはpandasのデータフレームに格納し、入力データとして処理します。5列目は、教師あり学習におけるデータのラベルに対応しており、ここでは各花データの花の種類（全部で3種類）を表しています。ラベルは通常1次元でデータの数だけの長さを持ち、NumPy配列またはpandasのシリーズに格納します。先に述べた通り、ラベルが連続値であれば回帰、ラベルが離散値であれば分類の問題となります。機械学習では、特徴量からこのラベルを予測することになります。

アイリスデータセットはscikit-learnが持つデータセットにも含まれており、`load_iris`関数によりアイリスデータセットの特徴量データとラベルデータを以下のようにNumPyの配列として取得することもできます。この時、ラベルは数値(0, 1, 2)に置き換えられています。

```
[2]: from sklearn.datasets import load_iris
iris = load_iris()
X_iris = iris.data
y_iris = iris.target
```

22.5 モデル学習の基礎

scikit-learn では、以下の手順でデータからモデルの学習を行います。

- 使用するモデルのクラスの選択
- モデルのハイパーパラメータの選択とインスタンス化
- データの準備
 - 教師あり学習では、特徴量データとラベルデータを準備
 - 教師あり学習では、特徴量・ラベルデータをモデル学習用の訓練データとモデル評価用のテストデータに分ける
 - 教師なし学習では、特徴量データを準備
- モデルをデータに適合 (fit() メソッド)
- モデルの評価
 - 教師あり学習では、predict() メソッドを用いてテストデータの特徴量データからラベルデータを予測しその精度の評価を行う
 - 教師なし学習では、transform() または predict() メソッドを用いて特徴量データのクラスタリングや次元削減などを行う

22.6 教師あり学習・分類の例

以下では、アイリスデータセットを用いて花の4つの特徴から3つの花の種類を分類する手続きを示しています。scikit-learn では、全てのモデルは Python クラスとして実装されており、ここでは分類を行うモデルの1つであるロジスティック回帰 (LogisticRegression) クラスをインポートしています。

LogisticRegression クラス

train_test_split() はデータセットを訓練データとテストデータに分割するための関数、accuracy_score() はモデルの予測精度を評価するための関数です。

特徴量データ (X_iris) とラベルデータ (y_iris) からなるデータセットを訓練データ (X_train, y_train) とテストデータ (X_test, y_test) に分割しています。ここでは、train_test_split() 関数の test_size 引数にデータセットの 30% をテストデータとすることを指定しています。また、stratify 引数にラベルデータを指定することで、訓練データとテストデータ、それぞれでラベルの分布が同じになるようにしています。

ロジスティック回帰クラスのインスタンスを作成し、fit() メソッドによりモデルを訓練データに適合させています。そして、predict() メソッドを用いてテストデータの特徴量データ (X_test) のラベルを予測し、accuracy_score() 関数で実際のラベルデータ (y_test) と比較して予測精度の評価を行なっています。97%の精度で花の4つの特徴から3つの花の種類を分類できていることがわかります。

```
[3]: from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.datasets import load_iris

iris = load_iris()
X_iris = iris.data # 特徴量データ
y_iris = iris.target # ラベルデータ

# 訓練データとテストデータに分割
X_train, X_test, y_train, y_test = train_test_split(X_iris, y_iris, test_size=0.3,
↪random_state=1, stratify=y_iris)
```

(continues on next page)

(continued from previous page)

```
# ロジスティック回帰モデル: solver 引数には最適化手法、multi_class には多クラス分類の方法を指定
# ここではそれぞれのデフォルト値、lbfgs と auto を指定
model=LogisticRegression(solver='lbfgs', multi_class='auto')

model.fit(X_train, y_train) # モデルを訓練データに適合
y_predicted=model.predict(X_test) # テストデータでラベルを予測
accuracy_score(y_test, y_predicted) # 予測精度 (accuracy) の評価
```

```
[3]: 0.9777777777777777
```

22.7 練習

アイリスデータセットの2つの特徴量、petal_length と petal_width、から2つの花の種類、versicolor か virginica、を予測するモデルをロジスティック回帰を用いて学習し、その予測精度を評価してください。以下では pandas データフレームの values 属性を用いて NumPy 配列を取得しています。

```
[4]: from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

iris = pd.read_csv('iris.csv')
iris2=iris[(iris['species']=='versicolor')|(iris['species']=='virginica')]
X_iris=iris2[['petal_length','petal_width']].values
y_iris=iris2['species'].values

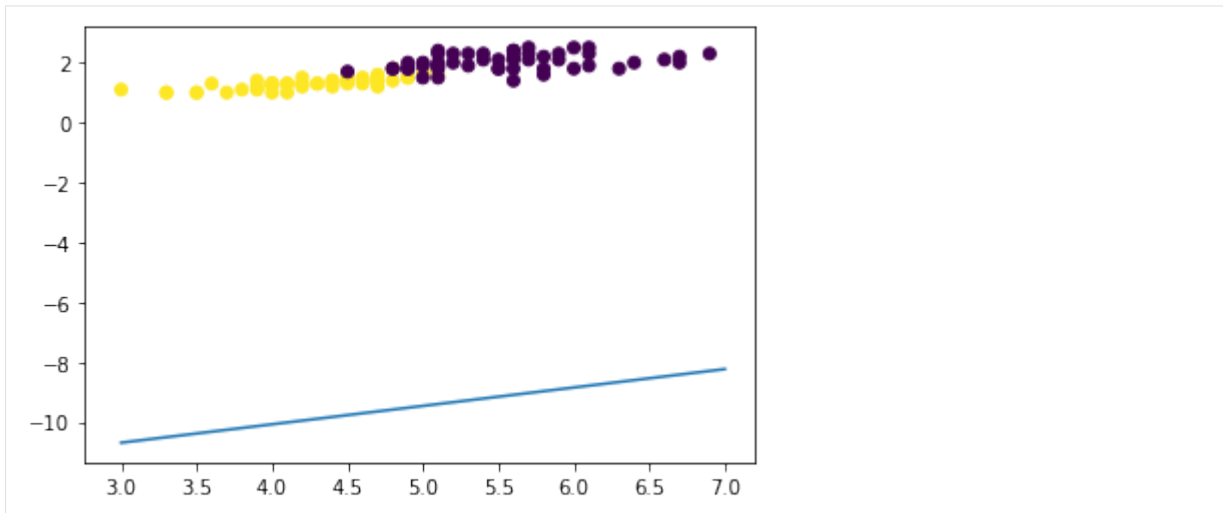
### your code here
```

上記のコードが完成したら、以下のコードを実行して、2つの特徴量、petal_length と petal_width、から2つの花の種類、versicolor か virginica、を分類するための決定境界を可視化してみてください。model は上記の練習で学習されたモデルとします。決定境界は、学習の結果得られた、特徴量の空間においてラベル（クラス）間を分離する境界を表しています。

```
[5]: import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

w2 = model.coef_[0,1]
w1 = model.coef_[0,0]
w0 = model.intercept_[0]

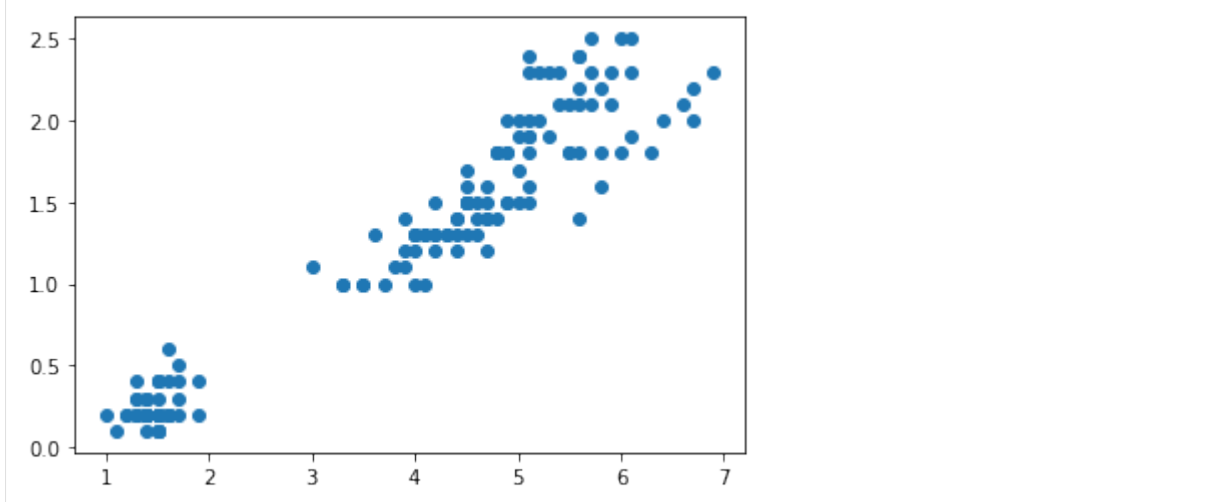
line=np.linspace(3,7)
plt.plot(line, -(w1*line+w0)/w2)
y_c = (y_iris=='versicolor').astype(int)
plt.scatter(iris2['petal_length'],iris2['petal_width'],c=y_c);
```



22.8 教師あり学習・回帰の例

以下では、アイリスデータセットを用いて花の特徴の 1 つ、`petal_length`、からもう 1 つの特徴、`petal_width`、を回帰する手続きを示しています。この時、`petal_length` は特徴量、`petal_width` は連続値のラベルとなっています。まず、`matplotlib` の散布図を用いて `petal_length` と `petal_width` の関係を可視化してみましょう。関係があるといえそうでしょうか。

```
[6]: iris = pd.read_csv('iris.csv')
X=iris[['petal_length']].values
y=iris['petal_width'].values
plt.scatter(X,y);
```



次に、回帰を行うモデルの 1 つである線形回帰 (LinearRegression) クラスをインポートしています。

`LinearRegression` クラス

`mean_squared_error()` は平均二乗誤差によりモデルの予測精度を評価するための関数です。

データセットを訓練データ (`X_train, y_train`) とテストデータ (`X_test, y_test`) に分割し、線形回帰クラスのインスタンスの `fit()` メソッドによりモデルを訓練データに適合させています。そして、`predict()` メソッドを用いてテストデータの `petal_length` の値から `petal_width` の値を予測し、`mean_squared_error()` 関数で実際の `petal_width` の値 (`y_test`) と比較して予測精度の評価を行なっています。

```
[7]: from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error

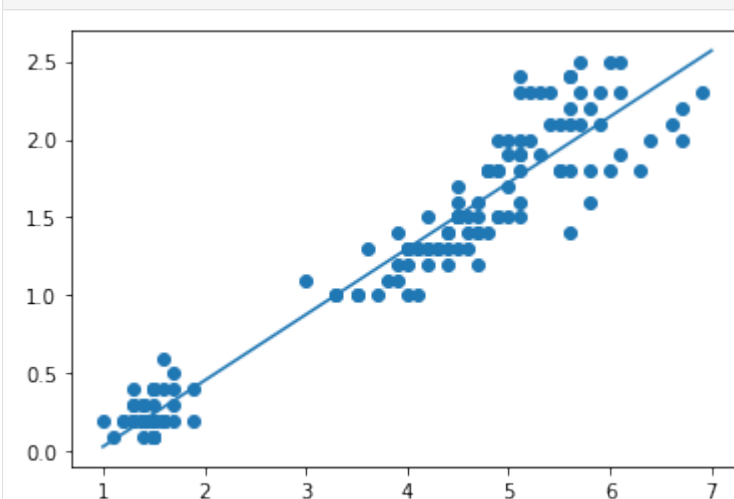
# 訓練データとテストデータに分割
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_
↪state=1)

model=LinearRegression() # 線形回帰モデル
model.fit(X_train,y_train) # モデルを訓練データに適合
y_predicted=model.predict(X_test) # テストデータで予測
mean_squared_error(y_test,y_predicted) # 予測精度（平均二乗誤差）の評価
```

[7]: 0.0397444576090427

以下では、線形回帰モデルにより学習された `petal_length` と `petal_width` の関係を表す回帰式を可視化しています。学習された回帰式が実際のデータに適合していることがわかります。

```
[8]: x_plot=np.linspace(1,7)
X_plot=x_plot[:,np.newaxis]
y_plot=model.predict(X_plot)
plt.scatter(X,y)
plt.plot(x_plot,y_plot);
```



22.9 教師なし学習・クラスタリングの例

以下では、アイリスデータセットを用いて花の2つの特微量、`petal_length` と `petal_width`、を元に花のデータをクラスタリングする手続きを示しています。ここではクラスタリングを行うモデルの1つである `KMeans` クラスをインポートしています。

KMeans クラス

特微量データ (`X_iris`) を用意し、引数 `n_clusters` にハイパーパラメータとしてクラスタ数、ここでは3、を指定して `KMeans` クラスのインスタンスを作成しています。そして、`fit()` メソッドによりモデルをデータに適合させ、`predict()` メソッドを用いて各データが所属するクラスタの情報 (`y_km`) を取得しています。

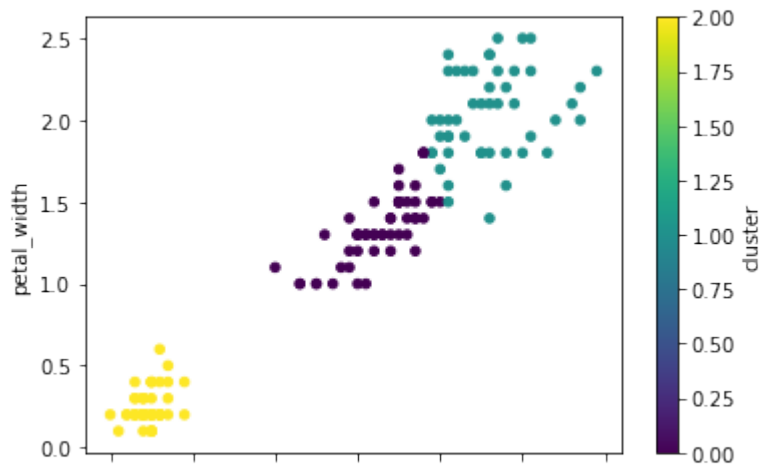
学習された各花データのクラスタ情報を元のデータセットのデータフレームに列として追加し、クラスタごとに異なる色でデータセットを可視化しています。2つの特微量、`petal_length` と `petal_width`、に基づき、3つのクラスタが得られていることがわかります。

```
[9]: from sklearn.cluster import KMeans

iris = pd.read_csv('iris.csv')
X_iris=iris[['petal_length', 'petal_width']].values

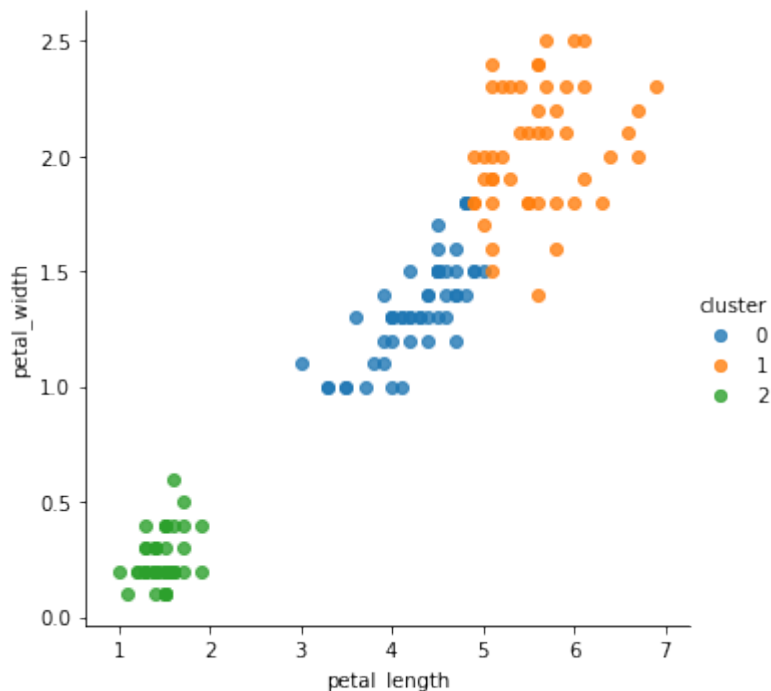
model = KMeans(n_clusters=3) # k-means モデル
model.fit(X_iris) # モデルをデータに適合
y_km=model.predict(X_iris) # クラスタを予測

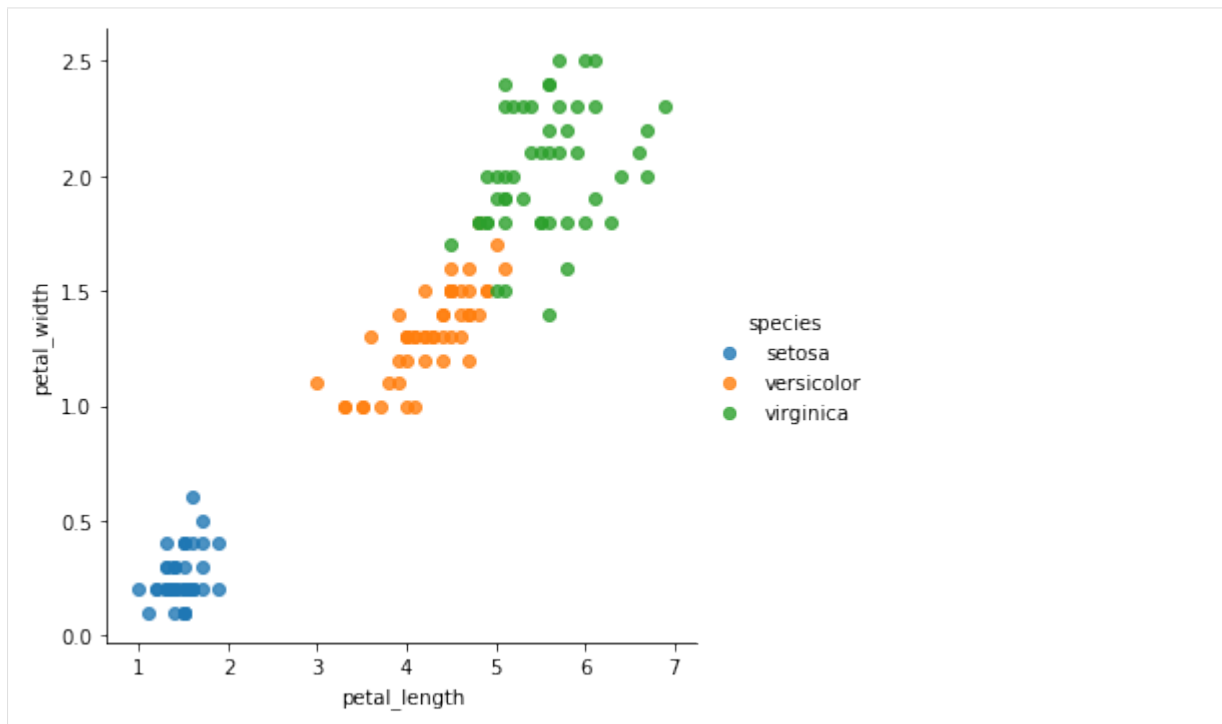
iris['cluster']=y_km
iris.plot.scatter(x='petal_length', y='petal_width', c='cluster', colormap='viridis');
```



3つのクラスタと3つの花の種類の分布を2つの特徴量、petal_length と petal_width、の空間で比較してみると、クラスタと花の種類には対応があり、2つの特徴量から花の種類をクラスタとしてグループ分けできていることがわかります。以下では可視化に seaborn モジュールを用いています。

```
[10]: import seaborn as sns
sns.lmplot(x='petal_length', y='petal_width', hue='cluster', data=iris, fit_reg=False);
sns.lmplot(x='petal_length', y='petal_width', hue='species', data=iris, fit_reg=False);
```





22.10 練習

アイリスデータセットの2つの特徴量、`sepal_length`と`sepal_width`、を元に、KMeans モデルを用いて花のデータをクラスタリングしてください。クラスタの数は任意に設定してください。

```
[11]: from sklearn.cluster import KMeans
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline

iris = pd.read_csv('iris.csv')
X_iris=iris[['sepal_length', 'sepal_width']].values

### your code here
```

22.11 教師なし学習・次元削減の例

以下では、アイリスデータセットを用いて花の4つの特徴量を元に花のデータを次元削減する手続きを示しています。ここでは次元削減を行うモデルの1つであるPCAクラスをインポートしています。

PCA クラス

特徴量データ(`X_iris`)を用意し、引数`n_components`にハイパーパラメータとして削減後の次元数、ここでは2、を指定してPCAクラスのインスタンスを作成しています。そして、`fit()`メソッドによりモデルをデータに適合させ、`transform()`メソッドを用いて4つの特徴量を2次元に削減した特徴量データ(`X_2d`)を取得しています。

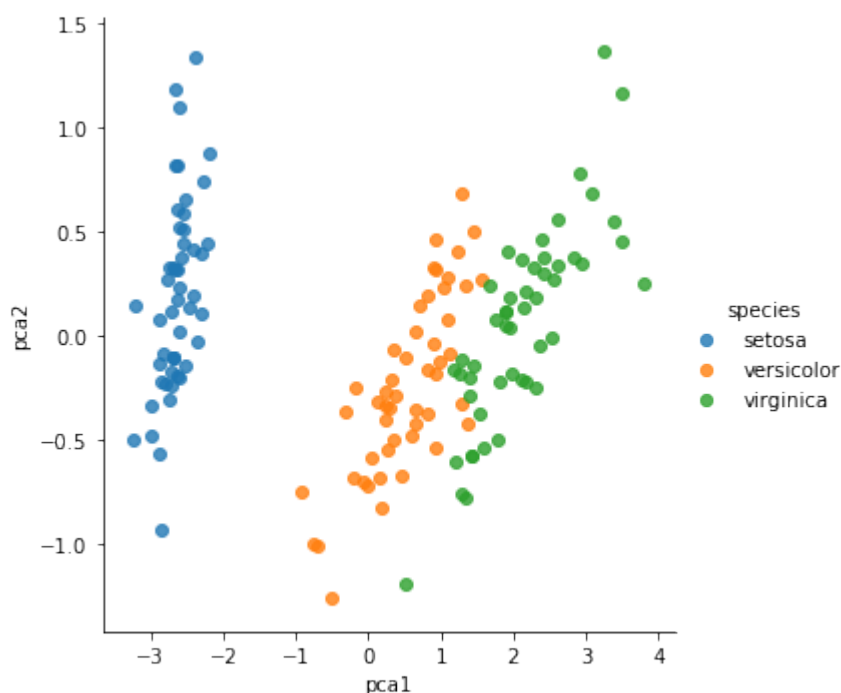
学習された各次元の値を元のデータセットのデータフレームに列として追加し、データセットを削減して得られた次元の空間において、データセットを花の種類ごとに異なる色で可視化しています。削減された次元の空間において、花の種類をグループ分けできていることがわかります。

```
[12]: from sklearn.decomposition import PCA

iris = pd.read_csv('iris.csv')
X_iris=iris[['sepal_length', 'sepal_width', 'petal_length', 'petal_width']].values

model = PCA(n_components=2) # PCA モデル
model.fit(X_iris) # モデルをデータに適合
X_2d=model.transform(X_iris) # 次元削減
```

```
[13]: import seaborn as sns
iris['pca1']=X_2d[:,0]
iris['pca2']=X_2d[:,1]
sns.lmplot(x='pca1',y='pca2',hue='species',data=iris,fit_reg=False);
```



22.12 練習の解答例

```
[14]: from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

iris = pd.read_csv('iris.csv')
iris2=iris[(iris['species']=='versicolor')|(iris['species']=='virginica')]
X_iris=iris2[['petal_length', 'petal_width']].values
y_iris=iris2['species'].values

X_train, X_test, y_train, y_test = train_test_split(X_iris, y_iris, test_size=0.3,
↳ random_state=1, stratify=y_iris)

model=LogisticRegression(solver='lbfgs', multi_class='auto')
model.fit(X_train, y_train)
y_model=model.predict(X_test)
accuracy_score(y_test, y_model)
```

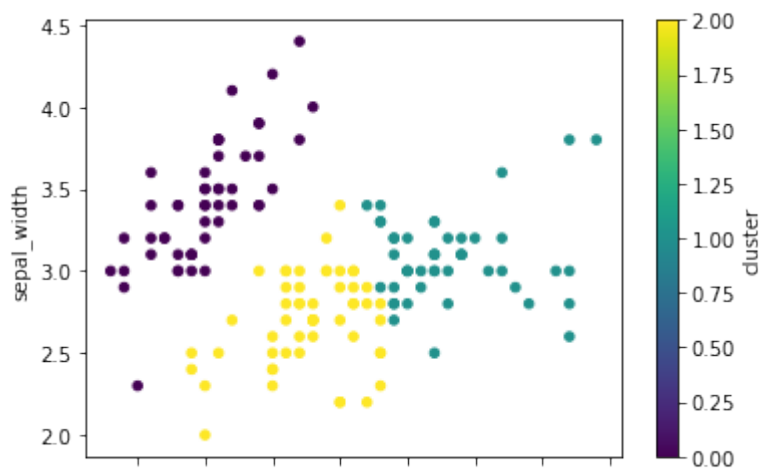
```
[14]: 0.9666666666666667
```

```
[15]: from sklearn.cluster import KMeans
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline

iris = pd.read_csv('iris.csv')
X_iris=iris[['sepal_length', 'sepal_width']].values

model = KMeans(n_clusters=3)
model.fit(X_iris)
y_km=model.predict(X_iris)

iris['cluster']=y_km
iris.plot.scatter(x='sepal_length', y='sepal_width', c='cluster', colormap='viridis');
```



▲ Jupyter Notebook の使い方

Jupyter Notebook について説明します。

参考

- <https://jupyter.readthedocs.io/en/latest/>

教材等の既存のノートブックは、ディレクトリのページで選択することによって開くことができます。ノートブックには `ipynb` という拡張子（エクステンション）が付きます。

ノートブックを新たに作成するには、ディレクトリが表示されているページで、**New** のメニューで **Python3** を選択してください。 **Untitled** (1 などが付くことあり) というノートブックが作られます。タイトルをクリックして変更することができます。

ノートブックの上方には、**File** や **Edit** などのメニュー、**↓** や **↑** や **■** などのアイコンが表示されています。

右上に **Python 3** と表示されていることに注意してください。

Ctrl+s (Mac の場合は **Cmd+s**) を入力することによって、ノートブックをファイルにセーブできます。オートセーブもされますが、適当なタイミングでセーブしましょう。

ノートブックはセルから成り立っています。

23.1 セル

主に次の二種類のセルを使います。

- **Code** : Python のコードが書かれたセルです。Code セルの横には **In []:** と表示されています。コードを実行するには、**Shift+Enter** (または **Return**) を押します。このセルの次のセルは Code セルです。**Shift+Enter** を押してみてください。
- **Markdown** : 説明が書かれたセルです。このセル自身は Markdown セルです。

セルの種類はノートブックの上のメニューで変更できます。

[]:

23.2 コマンドモード

セルを選択するとコマンドモードになります。ただし、Code セルを選択したとき、マウスカーソルが入力フィールドに入っていると、編集モードになってしまいます。

コマンドモードでは、セルの左の線が青色になります。

コマンドモードで Enter を入力すると、編集モードになります。Markdown のセルでは、ダブルクリックでも編集モードになります。

コマンドモードでは、一文字コマンドが有効なので注意してください。

- a: 上にセルを挿入 (above)
- b: 下にセルを挿入 (below)
- x: セルを削除（そのセルが削除されてしまいますので注意！）
- l: セルの行に番号を振るか振らないかをスイッチ
- s または Ctrl+s: ノートブックをセーブ (checkpoint)
- Enter: 編集モードに移行
- Shift+Enter: セルを実行して次のセルに

23.3 編集モード

編集モードでは文字カーソルが表示されて、セルの編集が可能です。Ctrl の付かない文字はそのまま挿入されます。

編集モードでは、セルの左の線が緑色になります。

編集モードでは、以下のような編集コマンドが使えます。

- Ctrl+c: copy
- Ctrl+x: cut
- Ctrl+v: paste
- Ctrl+z: undo
- ...

Code セルでは、編集モードでも Shift+Enter を入力すると、セルの中のコードが実行されて、次のセルに移動します。Markdown セルはフォーマットされて、次のセルに移動します。次のセルではコマンドモードになっています。

Esc でコマンドモードになります。

Ctrl+s でノートブックをセーブ (checkpoint)。これはコマンドモードの場合と同じです。

23.4 練習


次のセルを編集モードにして 10/3 と入力して実行してください。

[]:

23.5 （注意）Shift-Enter に反応がなくなったとき

Code セルで Shift-Enter をしても反応がないとき、特にセルの左の部分が

```
In [*]:
```

となったままで、* が数に置き換わらないとき、 のアイコンを押して、kernel（Python のインタプリタ）を停止させてください。

それでも反応がないときは、右回りの矢印のアイコンを押して、kernel（Python のインタプリタ）を起動し直してください。

たとえば、次のような例です。 のアイコンを押してください。

```
[ ]: while True:  
      pass
```

```
[ ]:
```

CHAPTER 24

▲セット (set)

セットについて説明します。

参考

- <https://docs.python.org/ja/3/tutorial/datastructures.html#sets>

セット（集合）は、リストと同様に複数の要素から構成されるデータです。セットでは、リストと異なり要素の重複がありません、また要素の順番也没有ありません。

セットを作成するには、次のように波括弧で要素を囲みます。辞書と似ていますが、辞書では：でキーと値を対応させる必要がありました。

```
[1]: set1= {2, 1, 2, 3, 2, 3, 1, 3, 3, 1}
      set1
```

```
[1]: {1, 2, 3}
```

```
[2]: type(set1)
```

```
[2]: set
```

セットのデータ型は `set` であり、`set` は組み込み関数でもあります。

組み込み関数 `set` を用いてもセットを作成することができます。

```
[3]: set([2, 1, 2, 3, 2, 3, 1, 3, 3, 1])
```

```
[3]: {1, 2, 3}
```

空のセットを作成する場合、次のようにします。（`{}` では空の辞書が作成されます。）

```
[4]: set2 = set() # 空のセット
      set2
```

```
[4]: set()
```

```
[5]: set2 = {} # 空の辞書
      set2
```

```
[5]: {}
```

`set` を用いて、文字列、リストやタプルなどからセットを作成することができます。

```
[6]: set([1,1,2,2,2,3])
```

```
[6]: {1, 2, 3}
```

```
[7]: set((1,1,2,2,2,3))
```

```
[7]: {1, 2, 3}
```

```
[8]: set('aabdceabdae')
```

```
[8]: {'a', 'b', 'c', 'd', 'e'}
```

```
[9]: set({'apple' : 3, 'pen' : 5})
```

```
[9]: {'apple', 'pen'}
```

24.1 セットの組み込み関数

リストなどと同様に、次の関数などはセットにも適用可能です。

```
[10]: len(set1) # 集合を構成する要素数
```

```
[10]: 3
```

```
[11]: x,y,z = set1 # 多重代入
      x
```

```
[11]: 1
```

```
[12]: 2 in set1 # 指定した要素を集合が含むかどうかの判定
```

```
[12]: True
```

```
[13]: 10 in set1 # 指定した要素を集合が含むかどうかの判定
```

```
[13]: False
```

```
[14]: 10 not in set1
```

```
[14]: True
```

セットの要素は、順序付けられていないのでインデックスを指定して取り出すことはできません。

```
[15]: set1[0]
```

```
-----
TypeError                                Traceback (most recent call last)
Input In [15], in <module>
----> 1 set1[0]

TypeError: 'set' object is not subscriptable
```


24.2 練習

文字列 `str1` が引数として与えられたとき、`str1` に含まれる要素（文字、すなわち長さ 1 の文字列）の種類を返す関数 `check_characters` を作成してください（大文字と小文字は区別し、スペースや句読点も 1 つと数えます）。たとえば、`'aabccc'` には `'a'` と `'b'` と `'c'` が含まれるので、`check_characters('aabccc')` は 3 を返します。

以下のセルの ... のところを書き換えて `check_characters(str1)` を作成してください。

```
[16]: def check_characters(str1):
      ...
```

上のセルで解答を作成した後、以下のセルを実行し、実行結果が `True` になることを確認してください。

```
[17]: print(check_characters('Onde a terra acaba e o mar come^^c3^^a7a') == 13)

False
```

24.3 練習

辞書 `dic1` が引数として与えられたとき、`dic1` に登録されているキーの数を返す関数 `check_dicsize` を作成してください。

以下のセルの ... のところを書き換えて `check_dicsize(dic1)` を作成してください。

```
[18]: def check_dicsize(dic1):
      ...
```

上のセルで解答を作成した後、以下のセルを実行し、実行結果が `True` になることを確認してください。

```
[19]: print(check_dicsize({'apple': 0, 'orange': 2, 'pen': 1}) == 3)

False
```

24.4 集合演算

複数のセットから、**和集合・積集合・差集合・対称差**を求める集合演算が存在します。

```
[20]: set1 = {1, 2, 3, 4}
      set2 = {3, 4, 5, 6}
```

```
[21]: set1 | set2 # 和集合
```

```
[21]: {1, 2, 3, 4, 5, 6}
```

```
[22]: set1 & set2 # 積集合
```

```
[22]: {3, 4}
```

```
[23]: set1 - set2 # 差集合
```

```
[23]: {1, 2}
```

```
[24]: set1 ^ set2 # 対称差
```

```
[24]: {1, 2, 5, 6}
```

24.5 比較演算

数値などを比較するのに用いた比較演算子を用いて、2つのセットを比較することもできます。

```
[25]: print({1, 2, 3} == {1, 2, 3})
      print({1, 2} == {1, 2, 3})
```

```
True
False
```

```
[26]: print({1, 2, 3} != {1, 2, 3})
      print({1, 2} != {1, 2, 3})
```

```
False
True
```

`<=` や `<` は、集合の間の包含関係を判定します。

```
[27]: print({1, 2, 3} <= {1, 2, 3})
      print({1, 2, 3} < {1, 2, 3})
      print({1, 2} < {1, 2, 3})
```

```
True
False
True
```

```
[28]: print({1, 2} <= {2, 3, 4})
```

```
False
```

24.6 セットのメソッド

セットにも様々なメソッドが存在します。なお、以下のメソッドは全て破壊的です。

24.6.1 add

指定した要素を新たにセットに追加します。

```
[29]: set1 = {1, 2, 3}
      set1.add(4)
      set1
```

```
[29]: {1, 2, 3, 4}
```

24.6.2 remove

指定した要素をセットから削除します。その要素がセットに含まれていない場合、エラーになります。

```
[30]: set1.remove(1)
      set1
```

```
[30]: {2, 3, 4}
```

```
[31]: set1.remove(10)
```

```

-----
KeyError                                Traceback (most recent call last)
Input In [31], in <module>
----> 1 set1.remove(10)

KeyError: 10

```

24.6.3 discard

指定した要素をセットから削除します。その要素がセットに含まれていなくともエラーになりません。

```
[32]: set1 = {1, 2, 3, 4}
      set1.discard(1)
      set1
```

```
[32]: {2, 3, 4}
```

```
[33]: set1.discard(5)
```

24.6.4 clear

全ての要素を削除して対象のセットを空にします。

```
[34]: set1 = {1, 2, 3, 4}
      set1.clear()
      set1
```

```
[34]: set()
```

24.6.5 pop

セットからランダムに1つの要素を取り出します。

```
[35]: set1 = {1, 2, 3, 4}
      print(set1.pop())
      print(set1)
```

```
1
{2, 3, 4}
```

24.6.6 union, intersection, difference

和集合・積集合・差集合・対称差を求めるメソッドも存在します。

```
[36]: set1 = {1, 2, 3, 4}
      set2 = {3, 4, 5, 6}
      set1.union(set2) # 和集合
```

```
[36]: {1, 2, 3, 4, 5, 6}
```

```
[37]: set1.intersection(set2) # 積集合
```

```
[37]: {3, 4}
```

```
[38]: set1.difference(set2) # 差集合
```

```
[38]: {1, 2}
```

```
[39]: set1.symmetric_difference(set2) # 対称差
```

```
[39]: {1, 2, 5, 6}
```

24.7 練習

英語の文章からなる文字列 `str_engsentences` が引数として与えられたとき、`str_engsentences` 中に含まれる単語の種類数を返す関数 `count_words2` を作成してください。

以下のセルの ... のところを書き換えて `count_words2(str_engsentences)` を作成してください。

```
[40]: def count_words2(str_engsentences):
      ...
```

上のセルで解答を作成した後、以下のセルを実行し、実行結果が `True` になることを確認してください。

```
[41]: print(count_words2('From Stettin in the Baltic to Trieste in the Adriatic an iron
↪curtain has descended across the Continent.')== 15)
```

```
False
```

24.8 練習の解答

```
[42]: def check_characters(str1):
      set1 = set(str1)
      return len(set1)
      #check_characters('Onde a terra acaba e o mar come^^c3^^a7a')
```

```
[43]: def check_dicsize(dic1):
      return len(set(dic1))
      #check_dicsize({'apple': 0, 'orange': 2, 'pen': 1})
```

実は `len` は辞書に対してキーの数を返すので、セットを使う必要はありません。

```
[44]: def check_dicsize(dic1):
      return len(dic1)
```

```
[45]: def count_words2(str_engsentences):
      str1 = str_engsentences.replace('.', ' ') # 句読点を削除する
      str1 = str1.replace(',', ' ')
      str1 = str1.replace(':', ' ')
      str1 = str1.replace('; ', ' ')
      str1 = str1.replace('!', ' ')
      str1 = str1.replace('?', ' ')
      list1 = str1.split(' ') # 句読点を削除した文字列を、単語ごとにリストに格納する
      set1 = set(list1) # リストを集合に変換して同じ要素を1つにまとめる
      return len(set1)
      #count_words2('From Stettin in the Baltic to Trieste in the Adriatic an iron curtain
↪has descended across the Continent.')
```

[]:

▲再帰

再帰について説明します。

関数の**再帰呼び出し**とは、定義しようとしている関数を、その定義の中で呼び出すことです。定義の中で直接呼び出す場合に限らず、他の関数を経由して間接的に呼び出す場合も、再帰呼び出しに含まれます。再帰呼び出しを行う関数を、**再帰関数**といいます。

再帰関数は、**分割統治**アルゴリズムの記述に適しています。分割統治とは、問題を容易に解ける小さな粒度まで分割していき、個々の小さな問題を解いて、その部分解を合成することで問題全体を解くような方法を指します。分割統治の考え方は、関数型プログラミングにおいてもよく用いられます。再帰関数による分割統治の典型的な形は、次の通りです。

```
def recursive_function(...):
    if 問題粒度の判定:
        再帰呼び出しを含まない基本処理
    else:
        再帰呼び出しを含む処理（問題の分割や部分解の合成を行う）
```

以下で、再帰関数を使った処理の例をいくつか見ていきましょう。

25.1 再帰関数の例：接頭辞リストと接尾辞リスト

[1]: # 入力の文字列の接頭辞リストを返す関数 *prefixes*

```
def prefixes(s):
    if s == '':
        return []
    else:
        return [s] + prefixes(s[:-1])
```

```
prefixes('aabcc')
```

[1]: ['aabcc', 'aabcc', 'aab', 'aa', 'a']

[2]: # 入力の文字列の接尾辞リストを返す関数 *suffixes*

```
def suffixes(s):
    if s == '':
```

(continues on next page)

(continued from previous page)

```

    return []
else:
    return [s] + suffixes(s[1:])

suffixes('aabcc')

```

```
[2]: ['aabcc', 'abcc', 'bcc', 'cc', 'c']
```

25.2 再帰関数の例：べき乗の計算

```

[3]: # 入力の底 base と冪指数 expt からべき乗を計算する関数 power
def power(base, expt):
    if expt == 0:
        # expt が 0 ならば 1 を返す
        return 1
    else:
        # expt を 1 つずつ減らしながら power に渡し、再帰的にべき乗を計算
        # (2*(2*(2*...*1)))
        return base * power(base, expt - 1)

power(2,10)

```

```
[3]: 1024
```

一般に、再帰処理は、繰り返し処理としても書くことができます。

```

[4]: # べき乗の計算を繰り返し処理で行った例
def power(base, expt):
    e = 1
    for i in range(expt):
        e *= base
    return e

power(2,10)

```

```
[4]: 1024
```

単純な処理においては、繰り返しのほうが効率的に計算できることが多いですが、特に複雑な処理になると、再帰的に定義した方が読みやすいコードで効率的なアルゴリズムを記述できることもあります。たとえば、次に示すべき乗計算は、上記よりも高速なアルゴリズムですが、計算の見通しは明快です。

```

[5]: # べき乗を計算する高速なアルゴリズム
def power(base, expt):
    if expt == 0:
        return 1
    elif expt % 2 == 0:
        return power(base * base, expt // 2) # x**(2m) == (x*x)**m
    else:
        return base * power(base, expt - 1)

power(2,10)

```

```
[5]: 1024
```

25.3 再帰関数の例：マージソート

マージソートは、典型的な分割統治アルゴリズムで、以下のように再帰関数で実装することができます。

```
[6]: # マージソートを行い、比較回数 n を返す
def merge_sort_rec(data, l, r, work):
    n = 0
    if r - l <= 1:
        return n
    m = l + (r - l) // 2
    n1 = merge_sort_rec(data, l, m, work)
    n2 = merge_sort_rec(data, m, r, work)
    i1 = l
    i2 = m
    for i in range(l, r):
        from1 = False
        if i2 >= r:
            from1 = True
        elif i1 < m:
            n = n + 1
            if data[i1] <= data[i2]:
                from1 = True
        if from1:
            work[i] = data[i1]
            i1 = i1 + 1
        else:
            work[i] = data[i2]
            i2 = i2 + 1
    for i in range(l, r):
        data[i] = work[i]
    return n1 + n2 + n

def merge_sort(data):
    return merge_sort_rec(data, 0, len(data), [0]*len(data))
```

`merge_sort` は、与えられた配列をインプレースでソートするとともに、比較の回数を返します。`merge_sort` は、再帰関数 `merge_sort_rec` を呼び出します。

`merge_sort_rec(data, l, r, work)` は、配列 `data` のインデックスが `l` 以上で `r` より小さいところをソートします。

- 要素が 1 つかないときは何もしません。
- そうでなければ、`l` から `r` までの要素を半分にしてそれぞれを再帰的にソートします。
- その結果を作業用の配列 `work` に順序を保ちながらコピーします。この操作はマージ（併合）と呼ばれます。
- 最後に、`work` から `data` に要素を戻します。

`merge_sort_rec` は自分自身を 2 回呼び出していますので、繰り返しては容易には実装できません。

```
[7]: import random
a = [random.randint(1,10000) for i in range(100)]
merge_sort(a)
```

```
[7]: 546
```

```
[8]: a
```



```
[8]: [29,  
      388,  
      480,  
      504,  
      608,  
      630,  
      687,  
      736,  
      760,  
      898,  
      1002,  
      1040,  
      1102,  
      1181,  
      1205,  
      1311,  
      1337,  
      1396,  
      1888,  
      1948,  
      2244,  
      2269,  
      2320,  
      2387,  
      2425,  
      2437,  
      2626,  
      2863,  
      2913,  
      2947,  
      2959,  
      3223,  
      3325,  
      3408,  
      3479,  
      3559,  
      3577,  
      3620,  
      3787,  
      3846,  
      3873,  
      3887,  
      3918,  
      4217,  
      4222,  
      4279,  
      4298,  
      4528,  
      4535,  
      4577,  
      4596,  
      4605,  
      4701,  
      4701,  
      4757,  
      4875,  
      5031,
```

(continues on next page)

(continued from previous page)

```
5483,  
5589,  
5729,  
5868,  
6011,  
6129,  
6147,  
6325,  
6342,  
6697,  
6773,  
6800,  
6839,  
6925,  
7120,  
7166,  
7195,  
7224,  
7406,  
7547,  
7633,  
7817,  
7963,  
7976,  
8387,  
8461,  
8624,  
8707,  
8748,  
9154,  
9393,  
9480,  
9483,  
9542,  
9608,  
9650,  
9688,  
9747,  
9799,  
9808,  
9830,  
9836,  
9879]
```

[]:

▲簡単なデータの可視化

第3回までに学んだ各種のデータに対する簡単な可視化について触れます。

参考

- <https://matplotlib.org/tutorials/introductory/pyplot.html#sphx-glr-tutorials-introductory-pyplot-py> (English Only)

matplotlib については、5-matplotlib に詳しい説明があります。

26.1 matplotlib

Python では可視化のための様々な仕組みが用意されています。ここでは最も広く利用され、ノートブック上で容易に動作を確認できる matplotlib について触れます。matplotlib を利用するには第5回で取り上げるモジュールについても知る必要がありますが、第2回と第3回でデータについてだけ学ぶのでは、みなさんのモチベーションの維持が難しいと思われるので、この段階でリスト・辞書だけで2次元グラフを表示させてみます。したがって、ここではモジュールの使い方については説明しません。

matplotlib の出力をノートブックで表示させるには、以下を Code セルで1回だけ実行します。`%matplotlib` のように % で始まる文をマジックコマンドと呼びます。

```
[1]: %matplotlib inline
```

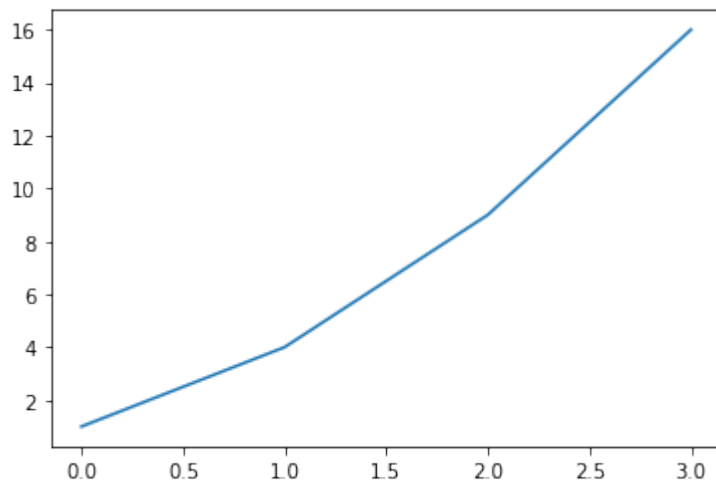
さらに matplotlib モジュールを読み込む次の処理もプログラムの冒頭で行う必要があります。

```
import matplotlib.pyplot as plt
```

26.2 折れ線グラフ

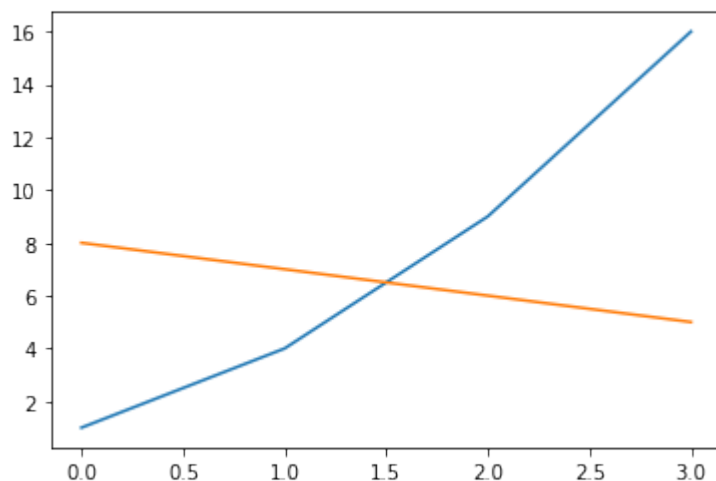
`ls1 = [1, 4, 9, 16]` といった数を要素とするリストを折れ線グラフで表示するには、次のように行います。

```
[2]: import matplotlib.pyplot as plt
ls1 = [1, 4, 9, 16]
plt.plot(ls1)
plt.show()
```



折れ線グラフを複数表示させるには、`plt.plot` を繰り返します。

```
[3]: import matplotlib.pyplot as plt
ls1 = [1, 4, 9, 16]
ls2 = [8, 7, 6, 5]
plt.plot(ls1, label='1st plot')
plt.plot(ls2, label='2nd plot')
plt.show()
```

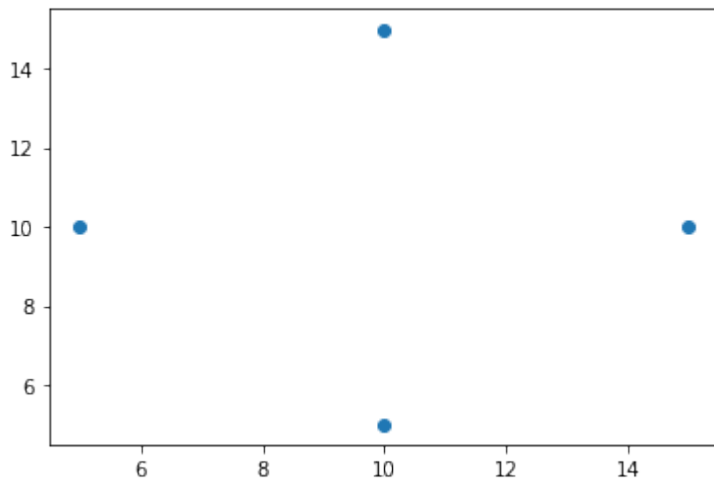


26.3 散布図

散布図を表示させるには、`plt.scatter` にそれぞれの点に対応する水平、垂直座標をリストで与えます。この2つのリストの要素数は同じでなければなりません。

```
[4]: import matplotlib.pyplot as plt
x = [5, 10, 15, 10]
y = [10, 5, 10, 15]
plt.scatter(x,y)
```

```
[4]: <matplotlib.collections.PathCollection at 0x7f1bb4f91c10>
```

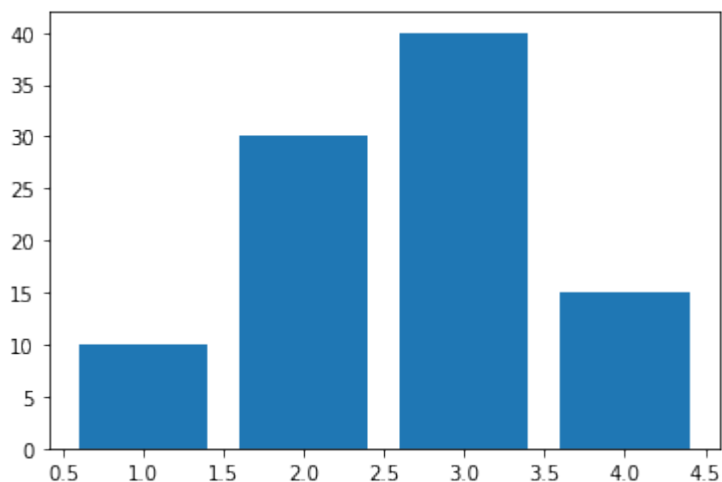


26.4 棒グラフ

棒グラフを表示させるには、`plt.bar` に水平座標、高さをリストで与えます。この2つのリストの要素数は同じでなければなりません。以下の例では、等間隔でグラフを表示させるため水平軸に整数列を使っています。

```
[5]: import matplotlib.pyplot as plt
x = [1, 2, 3, 4]
y = [10, 30, 40, 15]
plt.bar(x,y)
```

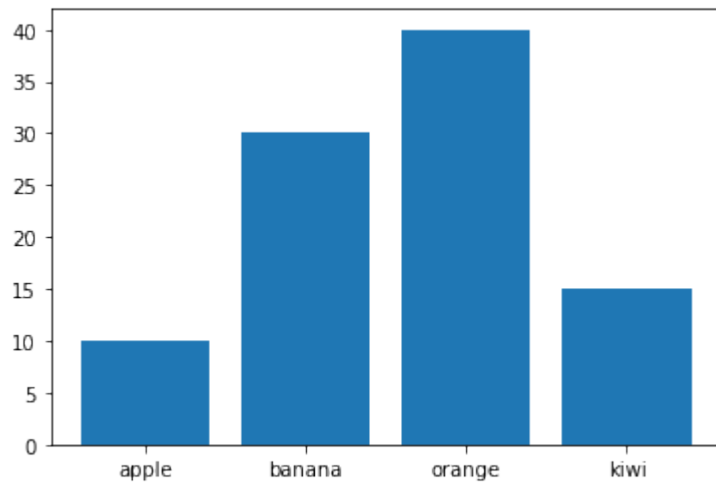
```
[5]: <BarContainer object of 4 artists>
```



第2回と第3回では文字列、辞書について学びました。文字列をキー、整数を値とする辞書を棒グラフで可視化します。さらに、水平軸にはキーをラベルとして表示されます。

```
[6]: import matplotlib.pyplot as plt
d = {'apple':10, 'banana':30, 'orange': 40, 'kiwi': 15}
x = [1,2,3,4]
plt.bar(x, d.values(), tick_label=list(d.keys()))
```

```
[6]: <BarContainer object of 4 artists>
```



```
[ ]:
```

▲ CSV ファイルの入出力

CSV ファイルの入出力について説明します。

参考

- <https://docs.python.org/ja/3/library/csv.html>

27.1 CSV 形式とは

CSV 形式とは “comma-separated values” の略で、複数の値をコンマで区切って記録するファイル形式です。

みなさん Excel を使ったことがあると思いますが、Excel では 1 つのセルに 1 つの値（数値や文字など）が入っていて、その他のセルの値とは独立に扱えますよね。

それと同じように、CSV 形式では、,（コンマ）で区切られた要素はそれぞれ独立の値として扱われます。

たとえばサークルのメンバーデータを作りたいを考えましょう。メンバーは「鈴木一郎」と「山田花子」の 2 名で、それぞれ『氏名』『ニックネーム』『出身地』を記録しておきたいと思います。

表で表すとこんなデータです。

ID	氏名	ニックネーム	出身地
user1	鈴木一郎	イチロー	広島
user2	山田花子	はなこ	名古屋

これを CSV 形式で表すと次のようになります。

'user1','鈴木一郎','イチロー','広島' 'user2','山田花子','はなこ','名古屋'

27.2 CSV ファイルの読み込み

CSV ファイルを読み書きするには、ファイルをオープンして、そのファイルオブジェクトから、CSV リーダを作ります。

CSV リーダとは、CSV ファイルからデータを読み込むためのオブジェクトで、このオブジェクトのメソッドを呼び出すことにより、CSV ファイルからデータを読み込むことができます。

CSV リーダを作るには、`csv` というモジュールの `csv.reader` という関数にファイルオブジェクトを渡します。

たとえば、次のような表で表される CSV ファイル `small.csv` を読み込んでみましょう。

0 列目 | 1 列目 | 2 列目 | 3 列目 | 4 列目

11 | 12 | 13 | 14 | 15 21 | 22 | 23 | 24 | 25 31 | 32 | 33 | 34 | 35

```
[1]: import csv
f = open('small.csv', 'r')
dataReader = csv.reader(f)
```

このオブジェクトもイテレータで、`next` という関数を呼び出すことができます。

```
[2]: next(dataReader)
```

```
[2]: ['11', '12', '13', '14', '15']
```

このようにして CSV ファイルを読むと、CSV ファイルの各行のデータが文字列のリストとなって返されます。

```
[3]: next(dataReader)
```

```
[3]: ['21', '22', '23', '24', '25']
```

```
[4]: row = next(dataReader)
```

```
[5]: row
```

```
[5]: ['31', '32', '33', '34', '35']
```

```
[6]: row[2]
```

```
[6]: '33'
```

数値が `' '` で囲われている場合、数値ではなく文字列として扱われているので、そのまま計算に使用することができません。

文字列が整数を表す場合、`int` 関数によって文字列を整数に変換することができます。文字列が小数を含む場合は `float` 関数で浮動小数点数型に変換、文字列が複素数を表す場合は `complex` 関数で複素数に変換します。

```
[7]: int(row[2])
```

```
[7]: 33
```

ファイルの終わりまで達した後に `next` 関数を実行すると、下のようにエラーが返ってきます。

```
[8]: next(dataReader)
```

```
-----
StopIteration                                Traceback (most recent call last)
Input In [8], in <module>
----> 1 next(dataReader)
```

(continues on next page)

(continued from previous page)

StopIteration:

ファイルを使い終わったらクローズすることを忘れないようにしましょう。

```
[9]: f.close()
```

27.3 CSV ファイルに対する for 文

CSV リーダもイテレータですので、for 文の in の後に書くことができます。

```
for row in dataReader:  
    ...
```

繰り返しの各ステップで、`next(dataReader)` が呼び出されて、`row` にその値が設定され、for 文の中身が実行されます。

```
[10]: f = open('small.csv', 'r')  
dataReader = csv.reader(f)  
for row in dataReader:  
    print(row)  
f.close()  
  
['11', '12', '13', '14', '15']  
['21', '22', '23', '24', '25']  
['31', '32', '33', '34', '35']
```

27.4 CSV ファイルに対する with 文

以下は with 文を使った例です。

```
[11]: with open('small.csv', 'r') as f:  
    dataReader = csv.reader(f)  
    for row in dataReader:  
        print(row)  
  
['11', '12', '13', '14', '15']  
['21', '22', '23', '24', '25']  
['31', '32', '33', '34', '35']
```

27.5 CSV ファイルの書き込み

CSV ファイルを作成して書き込むには、CSV ライターを作ります。

CSV ライターとは、CSV ファイルを作ってデータを書き込むためのオブジェクトで、このオブジェクトのメソッドを呼び出すことにより、データが CSV 形式でファイルに書き込まれます。

CSV ライターを作るには、`csv` というモジュールの `csv.writer` という関数にファイルオブジェクトを渡します。ここで、半角英数文字以外の文字（たとえば日本語文字や全角英数文字）を書き込み・書き出しする際には、文字コード（たとえば `encoding='utf-8'`）を指定し、また書き出しの際にはさらに改行コードとして `newline=''` を指定しないと文字化けが生じる可能性があります。

```
[12]: f = open('out.csv', 'w', encoding='utf-8', newline='')
```

```
[13]: dataWriter = csv.writer(f)
```

```
[14]: dir(dataWriter)
```

```
[14]: ['__class__',
      '__delattr__',
      '__dir__',
      '__doc__',
      '__eq__',
      '__format__',
      '__ge__',
      '__getattribute__',
      '__gt__',
      '__hash__',
      '__init__',
      '__init_subclass__',
      '__le__',
      '__lt__',
      '__ne__',
      '__new__',
      '__reduce__',
      '__reduce_ex__',
      '__repr__',
      '__setattr__',
      '__sizeof__',
      '__str__',
      '__subclasshook__',
      'dialect',
      'writerow',
      'writerows']
```

```
[15]: dataWriter.writerow([1,2,3])
```

```
[15]: 7
```

```
[16]: dataWriter.writerow([21,22,23])
```

```
[16]: 10
```

書き込みモードの場合も、ファイルを使い終わったらクローズすることを忘れないようにしましょう。

```
[17]: f.close()
```

読み込みのときと同様、with 文を使うこともできます。

```
[18]: with open('out.csv', 'w', encoding='utf-8', newline='') as f:
      dataWriter = csv.writer(f)
      dataWriter.writerow([1,2,3])
      dataWriter.writerow([21,22,23])
```

27.5.1 東京の7月の気温

tokyo-temps.csv には、気象庁のオープンデータからダウンロードした、東京の7月の平均気温のデータが入っています。

<http://www.data.jma.go.jp/gmd/risk/obsdl/>

48 行目の第2列に 1875 年 7 月の平均気温が入っており、以下、2016 年まで、12 行ごとに 7 月の平均気温が入っています。

以下は、これを取り出す Python の簡単なコードです。

```
[19]: import csv

with open('tokyo-temps.csv', 'r', encoding='shift_jis') as f:
    dataReader = csv.reader(f) # csv リーダを作成
    n=0
    year = 1875
    years = []
    july_temps = []
    for row in dataReader: # CSV ファイルの中身を 1 行ずつ読み込み
        n = n+1
        if n>=48 and (n-48)%12 == 0: # 48 行目からはじめて 12 か月ごとに if 内を実行
            years.append(year)
            july_temps.append(float(row[1]))
            year = year + 1
```

ファイルをオープンするときに、キーワード引数の `encoding` が指定されています。このファイルは Shift_JIS という文字コードで書かれているため、この引数で、ファイルの符号（文字コード）を指定します。'shift_jis' は Shift_JIS を意味します。この他に、'utf-8'（UTF-8、すなわちビットの Unicode）があります。

変数 `years` に年の配列、変数 `july_temps` に対応する年の 7 月の平均気温の配列が設定されます。

```
[20]: years
```

```
[20]: [1875,
1876,
1877,
1878,
1879,
1880,
1881,
1882,
1883,
1884,
1885,
1886,
1887,
1888,
1889,
1890,
1891,
1892,
1893,
1894,
1895,
1896,
1897,
1898,
1899,
```

(continues on next page)

(continued from previous page)

```
1900,  
1901,  
1902,  
1903,  
1904,  
1905,  
1906,  
1907,  
1908,  
1909,  
1910,  
1911,  
1912,  
1913,  
1914,  
1915,  
1916,  
1917,  
1918,  
1919,  
1920,  
1921,  
1922,  
1923,  
1924,  
1925,  
1926,  
1927,  
1928,  
1929,  
1930,  
1931,  
1932,  
1933,  
1934,  
1935,  
1936,  
1937,  
1938,  
1939,  
1940,  
1941,  
1942,  
1943,  
1944,  
1945,  
1946,  
1947,  
1948,  
1949,  
1950,  
1951,  
1952,  
1953,  
1954,  
1955,
```

(continues on next page)

(continued from previous page)

1956,
1957,
1958,
1959,
1960,
1961,
1962,
1963,
1964,
1965,
1966,
1967,
1968,
1969,
1970,
1971,
1972,
1973,
1974,
1975,
1976,
1977,
1978,
1979,
1980,
1981,
1982,
1983,
1984,
1985,
1986,
1987,
1988,
1989,
1990,
1991,
1992,
1993,
1994,
1995,
1996,
1997,
1998,
1999,
2000,
2001,
2002,
2003,
2004,
2005,
2006,
2007,
2008,
2009,
2010,
2011,

(continues on next page)

(continued from previous page)

```
2012,  
2013,  
2014,  
2015,  
2016]
```

```
[21]: july_temps
```

```
[21]: [26.0,  
24.3,  
26.5,  
26.0,  
26.1,  
24.2,  
24.0,  
24.2,  
23.7,  
23.4,  
23.1,  
25.0,  
23.6,  
24.5,  
23.4,  
23.5,  
24.9,  
25.7,  
25.3,  
26.8,  
22.1,  
24.1,  
22.9,  
25.9,  
23.2,  
22.8,  
22.1,  
21.8,  
23.2,  
24.8,  
23.3,  
23.5,  
22.7,  
22.1,  
24.3,  
23.0,  
24.5,  
24.3,  
23.3,  
25.5,  
24.2,  
23.9,  
25.7,  
26.0,  
23.6,  
26.1,  
24.3,  
25.0,  
24.0,
```

(continues on next page)

(continued from previous page)

26.1,
23.2,
24.6,
26.0,
23.4,
25.9,
26.3,
21.8,
25.7,
26.6,
23.9,
24.3,
24.9,
26.3,
25.0,
26.5,
26.9,
23.7,
27.5,
25.1,
25.6,
22.0,
26.2,
25.7,
26.0,
25.3,
26.5,
24.3,
24.3,
24.7,
22.3,
27.6,
24.2,
24.4,
24.9,
26.1,
25.8,
27.4,
25.1,
25.7,
25.5,
24.2,
24.4,
26.3,
24.7,
25.0,
25.4,
25.8,
25.2,
26.1,
23.4,
25.6,
23.9,
25.8,
27.8,
25.2,

(continues on next page)

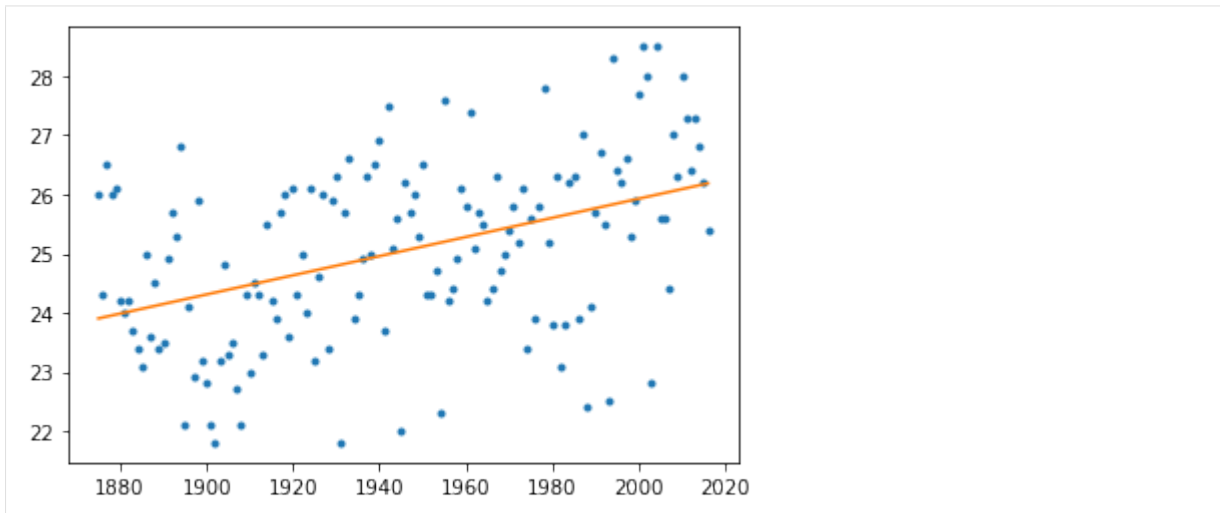
(continued from previous page)

```
23.8,  
26.3,  
23.1,  
23.8,  
26.2,  
26.3,  
23.9,  
27.0,  
22.4,  
24.1,  
25.7,  
26.7,  
25.5,  
22.5,  
28.3,  
26.4,  
26.2,  
26.6,  
25.3,  
25.9,  
27.7,  
28.5,  
28.0,  
22.8,  
28.5,  
25.6,  
25.6,  
24.4,  
27.0,  
26.3,  
28.0,  
27.3,  
26.4,  
27.3,  
26.8,  
26.2,  
25.4]
```

ここでは詳しく説明しませんが、線形回帰によるフィッティングを行ってみましょう。

```
[22]: import numpy  
import matplotlib.pyplot as plt  
%matplotlib inline  
  
fitp = numpy.poly1d(numpy.polyfit(years, july_temps, 1))  
ma = max(years)  
mi = min(years)  
xp = numpy.linspace(mi, ma, (ma - mi))
```

```
[23]: plt.plot(years, july_temps, '.', xp, fitp(xp), '-')  
plt.show()
```

27.6 練習

1. `tokyo-temps.csv` を読み込んで、各行が西暦年と7月の気温のみからなる '`tokyo-july-temps.csv`' という名前の CSV ファイルを作成してください。西暦年は 1875 から 2016 までとします。
2. 作成した CSV ファイルを Excel で読み込むとどうなるか確認してください。

[]:

以下のセルによってテストしてください。(years と july_temps の値がそのままと仮定しています。)

```
[24]: with open('tokyo-july-temps.csv', 'r', encoding='shift_jis') as f:
        i = 0
        dataReader = csv.reader(f)
        for row in dataReader:
            if int(row[0]) != years[i] or abs(float(row[1]) - july_temps[i]) > 0.000001:
                print('error', int(row[0]), float(row[1]))
            i += 1
        print(i == 142) # 1875 年から 2016 年まで 142 年間分のデータがあるはず
```

True

27.7 練習

整数データのみからなる CSV ファイルの名前を受け取ると、その CSV ファイルの各行を読み込んで整数のリストを作り、ファイル全体の内容を、そのようなリストのリストとして返す関数 `csv_matrix(name)` を定義してください。

たとえば上で用いた `small.csv` には次のようなデータが入っています。

0 列目 | 1 列目 | 2 列目 | 3 列目 | 4 列目

11 | 12 | 13 | 14 | 15 21 | 22 | 23 | 24 | 25 31 | 32 | 33 | 34 | 35

この `small.csv` の名前が引数として与えられた場合、

```
[[11, 12, 13, 14, 15], [21, 22, 23, 24, 25], [31, 32, 33, 34, 35]]
```

というリストを返します。

```
[25]: def csv_matrix(name):  
      ...
```

以下のセルによってテストしてください。

```
[26]: print(csv_matrix('small.csv') == [[11, 12, 13, 14, 15], [21, 22, 23, 24, 25], [31, 32,  
      33, 34, 35]])  
  
False
```

27.8 練習の解答

```
[27]: with open('tokyo-july-temps.csv', 'w', encoding='utf-8', newline='') as f:  
      i = 0  
      dataWriter = csv.writer(f)  
      for i in range(len(years)):  
          dataWriter.writerow([years[i], july_temps[i]])
```

```
[28]: def csv_matrix(name):  
      rows = []  
      with open(name, 'r') as f:  
          dataReader = csv.reader(f)  
          for row in dataReader:  
              rows.append([int(x) for x in row])  
      return rows
```

```
[ ]:
```

▲ Bokeh ライブラリ

Bokeh ライブラリについて説明します。

参考

- <https://bokeh.pydata.org/>

Bokeh は、データを可視化するためのライブラリです。 `bokeh` モジュールを使った、基本的なグラフの描画について説明します。

28.1 線グラフ

Bokeh ライブラリを使用してグラフを描画するには、`bokeh.plotting` のモジュールをインポートします。基本的なグラフの描画をノートブック上で行うには、図形を生成する `bokeh.plotting.figure()`、図形を表示する `bokeh.plotting.show()`、出力先をノートブック上に設定する `bokeh.plotting.output_notebook()` があれば充分です。通例、`output_notebook()` は最初に呼び出されます。

グラフで可視化するデータは配列を用いることが多いため、`numpy` モジュールも併せてインポートします。

```
[1]: import numpy as np
from bokeh.plotting import figure, output_notebook, show
output_notebook()
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_load.v0+json

次は、`figure()` が返す `Figure` クラスの `line()` メソッドを使って、リストの要素の数値を `y` 軸の値としてグラフを描画しています。 `y` 軸の値に対応する `x` 軸の値は、リストの各要素のインデックスとしています。

```
[2]: # プロットするデータ
d = [0, 1, 4, 9, 16]
p = figure()
p.line(range(len(d)), d) # 第 1 引数が x 軸、第 2 引数が y 軸
show(p)
```

(continued from previous page)

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_exec.v0+json

`line()` メソッド（及び他の描画用メソッド）では、キーワード引数も使えます。

```
[3]: p = figure()
p.line(y=d, x=range(len(d)))
show(p)
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_exec.v0+json

次に示すように、複数のグラフをまとめてプロットして表示することもできます。プロットするメソッドではグラフの線の色や線の種類を、`line_color` 引数や `line_dash` 引数で指定できます。また、`legend_label` 引数に値を設定すると、プロットしたグラフが凡例に現れます。引数の詳細は [Figure.line](#) のページ（英語）を参照してください。

```
[4]: data = [0, 1, 4, 9, 16]
x = range(len(data))
p = figure()
p.line(x, x, line_color='blue', legend_label='linear', line_dash='dashed')
p.line(x, data, line_color='green', legend_label='quad', line_dash='dotted')
show(p)
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_exec.v0+json

`figure()` 関数の引数に、軸のラベルや、グラフのタイトルを設定できます。プロット点を線グラフ上に重ねたいときには、`circle()` メソッドや `cross()` メソッドで同色の円や十字を追加で描けばよいです。

```
[5]: p = figure(x_axis_label='x', y_axis_label='y', title='Linear vs. Quadratic')
p.line(x, x, line_color='blue', legend_label='linear', line_dash='dashed')
p.circle(x, x, color='blue', line_width=5)
p.line(x, data, line_color='green', legend_label='quad', line_dash='dotted')
p.cross(x, data, color='green', size=16)
show(p)
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_exec.v0+json

色の使い分けを全て自分で決めるのは面倒です。良く使われる色のリストがパレットとして、提供されています。次は、`d3` の `Category10` という種類の 3 色パレットを用いています。詳細は、[palette](#) のページを参照してください。

```
[6]: from bokeh.palettes import d3
c = d3['Category10'][3]
p = figure(x_axis_label='x', y_axis_label='y', title='Linear vs. Quadratic')
p.line(x, x, line_color=c[0], legend_label='linear', line_dash='dashed')
p.circle(x, x, color=c[0], line_width=5)
p.line(x, data, line_color=c[1], legend_label='quad', line_dash='dotted')
p.cross(x, data, color=c[1], size=16)
show(p)
```

(continued from previous page)

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_exec.v0+json

グラフを描画するときのプロット数を増やすことで任意の曲線のグラフを作成することもできます。次の例では、`numpy` モジュールの `arange()` 関数を用いて、 $-\pi$ から π の範囲を `0.1` 刻みで x 軸の値を配列として準備しています。その x 軸の値に対して、`numpy` モジュールの `cos()` 関数と `sin()` 関数を用いて、 y 軸の値をそれぞれ準備し、`cos` カーブと `sin` カーブを描画しています。

```
[7]: # グラフの x 軸の値となる配列
x = np.arange(-np.pi, np.pi, 0.1)

# 上記配列を cos, sin 関数に渡し、y 軸の値として描画
p = figure(title='cos and sin Curves', x_axis_label='x', y_axis_label='y')
p.line(x, np.cos(x), line_color=c[0])
p.line(x, np.sin(x), line_color=c[1])
show(p)
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_exec.v0+json

プロットの数进行少なくすると、曲線は直線をつなぎ合わせることで描画されていることがわかります。

```
[8]: x = np.arange(-np.pi, np.pi, 0.5)
p = figure(title='cos and sin Curves', x_axis_label='x', y_axis_label='y')
p.line(x, np.cos(x), line_color=c[0])
p.line(x, np.sin(x), line_color=c[1])
show(p)
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_exec.v0+json

28.1.1 グラフの例：ソートアルゴリズムにおける比較回数

```
[9]: import random

def bubble_sort(lst):
    n = 0
    for j in range(len(lst) - 1):
        for i in range(len(lst) - 1 - j):
            n = n + 1
            if lst[i] > lst[i+1]:
                lst[i + 1], lst[i] = lst[i], lst[i+1]
    return n

def merge_sort_rec(data, l, r, work):
    if l+1 >= r:
        return 0
    m = l+(r-l)//2
    n1 = merge_sort_rec(data, l, m, work)
    n2 = merge_sort_rec(data, m, r, work)
    n = 0
    i1 = l
    i2 = m
```

(continues on next page)

(continued from previous page)

```

for i in range(1, r):
    from1 = False
    if i2 >= r:
        from1 = True
    elif i1 < m:
        n = n + 1
        if data[i1] <= data[i2]:
            from1 = True
    if from1:
        work[i] = data[i1]
        i1 = i1 + 1
    else:
        work[i] = data[i2]
        i2 = i2 + 1
for i in range(1, r):
    data[i] = work[i]
return n1+n2+n

def merge_sort(data):
    return merge_sort_rec(data, 0, len(data), [0]*len(data))

```

```

[10]: x = np.arange(100, 1100, 100)
      bdata = np.array([bubble_sort([random.randint(1,10000) for i in range(k)]) for k in
      ↪x])
      mdata = np.array([merge_sort([random.randint(1,10000) for i in range(k)]) for k in x])

```

```

[11]: p = figure(title='bubble sort vs. merge sort', x_axis_label='number of items', y_axis_
      ↪label='number of comparisons')
      p.line(x, bdata, line_color=c[0])
      p.circle(x, bdata, color=c[0], line_width=5)
      p.line(x, mdata, line_color=c[1])
      p.circle(x, mdata, color=c[1], line_width=5)
      show(p)

```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_exec.v0+json

28.2 散布図

散布図の描画には、点のプロットを `marker` 引数で指定できる `scatter()` メソッドが便利です。以下では、ランダムに生成した 20 個の要素からなる配列 `x`、`y` の各要素の値の組みを点としてプロットした散布図を表示します。プロットする点のマーカは円とし、`size` 引数で大きさを、`alpha` 引数で透明度を設定しています。

```

[12]: # グラフの x 軸の値となる配列
      x = np.random.rand(20)
      # グラフの y 軸の値となる配列
      y = np.random.rand(20)

      p = figure()
      p.scatter(x, y, marker='circle', size=16, alpha=0.5)
      show(p)

```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_exec.v0+json

これと同じグラフは、単に `circle()` メソッドでプロットをすることでも描画できます。

```
[13]: p = figure()
      p.circle(x, y, size=16, alpha=0.5)
      show(p)
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_exec.v0+json

28.3 棒グラフ

棒グラフは、`vbar()` メソッドを用いて描画できます。次の例では、ランダムに生成した 10 個の要素からなる配列 `y` の各要素の値を縦の棒グラフで表示しています。`x` は、`x` 軸上で棒グラフのバーの並ぶ位置を示しています。ここでは、`numpy` モジュールの `arange()` 関数を用いて、1 から 10 の範囲を 1 刻みで `x` 軸上のバーの並ぶ位置として配列を準備しています。

```
[14]: # x 軸上で棒の並ぶ位置となる配列
      x = np.arange(1, 11, 1)
      # グラフの y 軸の値となる配列
      y = np.random.rand(10)

      p = figure()
      p.vbar(x, 0.5, y) # 第 2 引数は幅
      show(p)
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_exec.v0+json

28.4 ヒストグラム

ヒストグラムの描画には、`quad()` メソッドが便利です。次の例では、`numpy.random.randn()` 関数を用いて、正規分布に基づく 1000 個の数値の要素からなる配列を用意し、`numpy.histogram()` 関数を使って 20 個のビンに分類したヒストグラムを計算しています。その計算結果を、`quad()` メソッドを使って、描画しています。ビンの境界を見やすくするように、`line_color` と `fill_color` (デフォルト色) を別の色にしています。

```
[15]: # 正規分布に基づく 1000 個の数値の要素からなる配列
      d = np.random.randn(1000)
      # numpy.histogram で 20 のビンに分割
      hist, bin_edges = np.histogram(d, 20)
      p = figure()
      p.quad(top=hist, bottom=0, left=bin_edges[:-1], right=bin_edges[1:], line_color='white',
            ↪ alpha=0.5)
      show(p)
```

(continued from previous page)

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_exec.v0+json

28.5 ヒートマップ

最後に、複雑な応用例として、ヒートマップの描画方法を示します。次の例は、10x10 のマスに 0.0 以上 1.0 未満の乱数の温度を割り当て、その値に応じた色で塗ったヒートマップです。ここでは、これまでと違って、x 軸、y 軸、温度の 3 つの値が必要になります。そこで、`bokeh.models.ColumnDataSource` 型を用いて、その 3 つ組を、'x'・'y'・'T' の属性を持った表データを構築しています。この表データの構築には、7-1 で説明する pandas も使えます。

ヒートマップでは、温度に応じた階調のある色選択が必要です。そこで、色階調と値を対応付ける `bokeh.models.LinearColorMapper` 型の mapper を準備します。`rect()` メソッドでは、表データの属性を参照して描画しています。色は、表データの値を mapper に適用して色に変化させたものを用いることで、温度に応じた色選択を実現しています。最後に、目盛り付きのカラーバーを生成して、右に配置しています。

```
[16]: from bokeh.models import LinearColorMapper, BasicTicker, PrintfTickFormatter,
      ↪ ColorBar, ColumnDataSource
      from bokeh.transform import transform

      # 10 行 10 列のランダム要素からなる行列
      n = 10
      data = np.random.rand(n*n)
      src = ColumnDataSource({'x': [yx % n for yx in range(n*n)], 'y': [yx // n for yx in
      ↪ range(n*n)], 'T': data})

      colors = ['#75968f', '#a5bab7', '#c9d9d3', '#e2e2e2', '#dfccce', '#ddb7b1', '#cc7878',
      ↪ '#933b41', '#550b1d']
      mapper = LinearColorMapper(palette=colors, low=data.min(), high=data.max())
      p = figure()
      p.rect('x', 'y', 1, 1, source=src, line_color=None, fill_color=transform('T', mapper))
      color_bar = ColorBar(color_mapper=mapper, location=(0, 0),
      ↪ ticker=BasicTicker(desired_num_ticks=len(colors)),
      ↪ formatter=PrintfTickFormatter(format='%2.1f'))
      p.add_layout(color_bar, 'right')
      show(p)
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_exec.v0+json

28.6 グラフのファイル出力

これまで表示されてきたグラフには画像保存ボタンがあるので、それをクリックすれば PNG 形式の画像を保存できます。

`bokeh.plotting.output_file()` を用いると、グラフ単独を HTML ファイルとして保存できるようになります。ただし、既に `output_notebook()` を読んでいる場合、`bokeh.plotting.reset_output()` で状態をリセットする必要があります。

```
[17]: from bokeh.plotting import save, output_file, reset_output
      x = np.arange(-2*np.pi, 2*np.pi, 0.1)
      p = figure(title='sin Curves', x_axis_label='x', y_axis_label='y')
```

(continues on next page)

(continued from previous page)

```
p.line(x, np.sin(x))

reset_output() # output_notebook() の効果を消す
output_file('sin.html') # 出力先の設定
save(p) # グラフを保存するだけ
show(p) # 保存した上でブラウザを開く
```

注意：output_notebook() を呼んだ状態と output_file() を呼んだ状態が重なると、show() でエラーが起きます。

```
[ ]:
```

▲ Python スクリプトとコマンドライン実行

Python スクリプトとコマンドライン実行について説明します。

参考

- <https://docs.python.org/ja/3/tutorial/interpreter.html>
- <https://docs.python.org/ja/3/tutorial/modules.html>
- <https://docs.python.org/ja/3/tutorial/appendix.html>
- <https://docs.python.org/ja/3/library/sys.html>
- <https://docs.python.org/ja/3/reference/import.html>

実は、第 4 回で紹介したモジュールファイル（拡張子 `.py`）は、それ単独で直接実行可能な自己完結したプログラムです。直接実行される Python プログラムコードのことを指して特に、**Python スクリプト**と呼びます。モジュールかスクリプトかを区別しないときには、Python ソースファイルや `.py` ファイル等と呼ばれます。

たとえば、次のコードセルを実行してみてください。

```
[1]: a1 = 10
      print('a1 contains the value of', a1)

a1 contains the value of 10
```

この内容と全く同じコードを記述した Python スクリプトファイル `sample.py` を教材として用意しました。オペレーティングシステム（実際にはシェル）から `sample.py` を実行するには、以下のようになります。

```
>>> python sample.py
```

あるいは

```
>>> python3 sample.py
```

ここで、`>>>` は、シェルのプロンプト（コマンド入力を促す記号）を意味します。後に示す具体例を見るとわかるように、環境によっては `>` であったり、`$` であったりします。

このようにスクリプトをシェルから実行することを、**コマンドライン実行**と呼びます。

29.1 コマンドライン実行の具体例

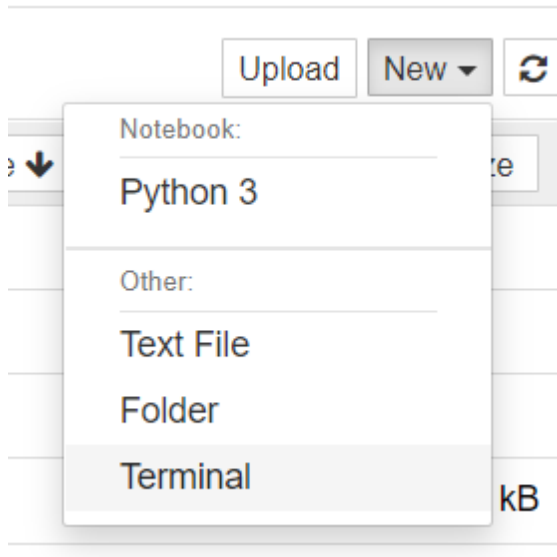
sample.py をコマンドライン実行する具体例を、実行環境毎に説明します。

29.1.1 Jupyter Notebook での実行方法

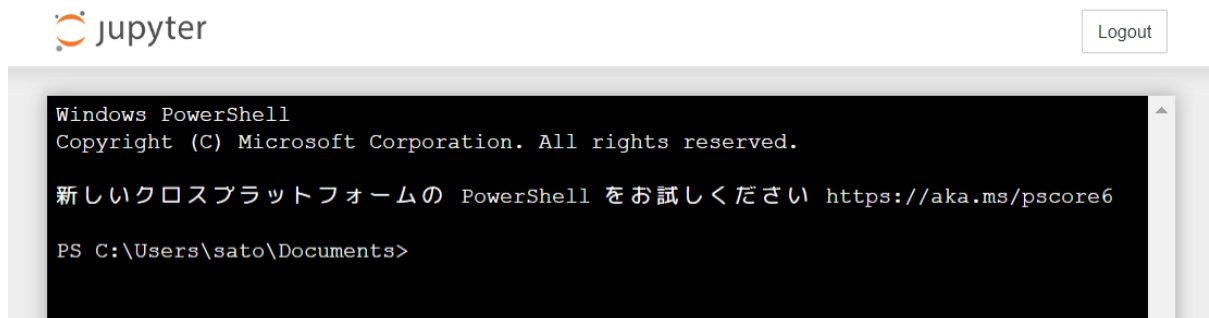
ファイルマネージャ画面で、

New ⇒ Terminal

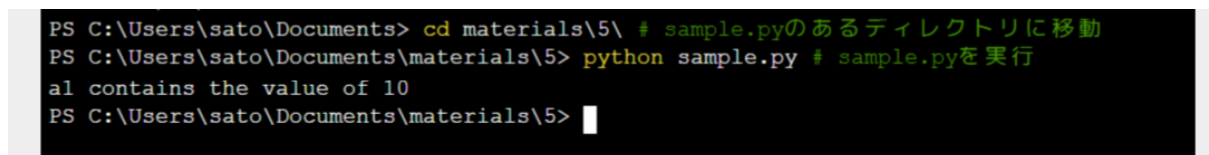
を選択すると、ターミナルのタブが生成されます。



Windows 10 の環境（ユーザーアカウント名 sato）では、次のように表示されます。



次の例では、cd コマンドで sample.py が存在するディレクトリ materials/5/ に移動し、その上で sample.py を実行しています。

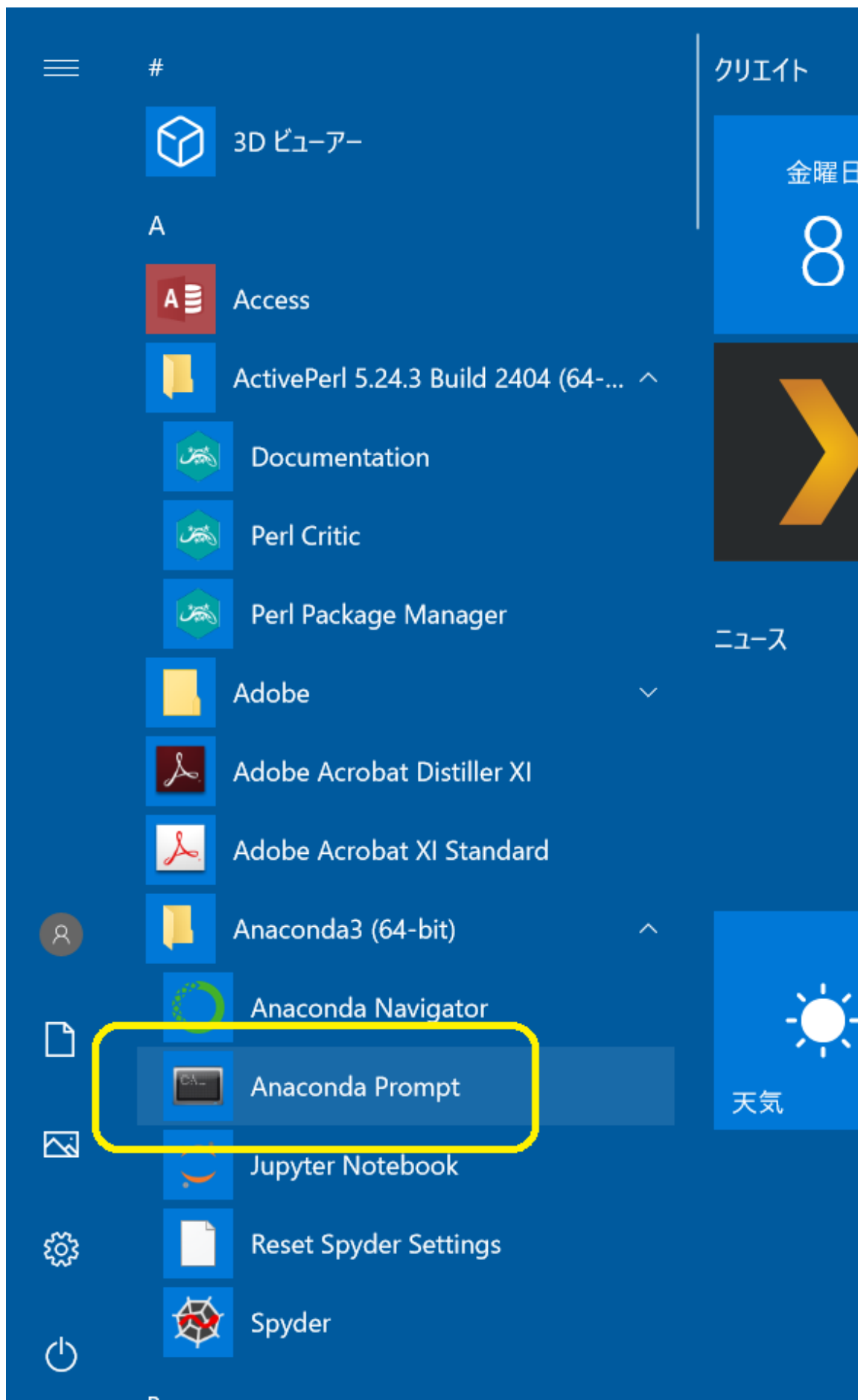


Jupyter Notebook 上で開かれるターミナルは、環境によって違います。デフォルトでは、Windows 10 では PowerShell が起動し、macOS ならば bash が起動するでしょう。

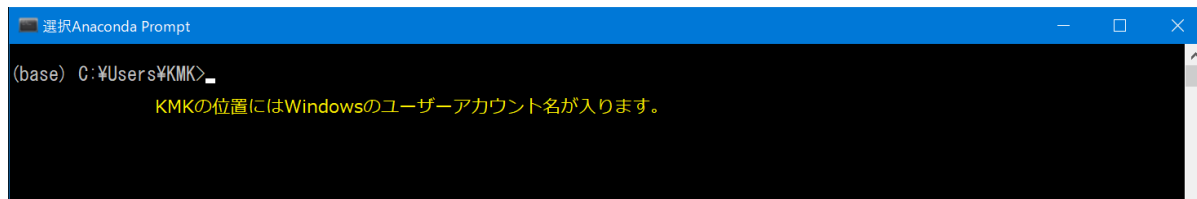
29.1.2 Windows での実行方法

以下をクリックすれば、ターミナルが開いて `python` をコマンドとして実行できます。

Start メニュー ⇒ Anaconda3(64-bit) ⇒ Anaconda Prompt

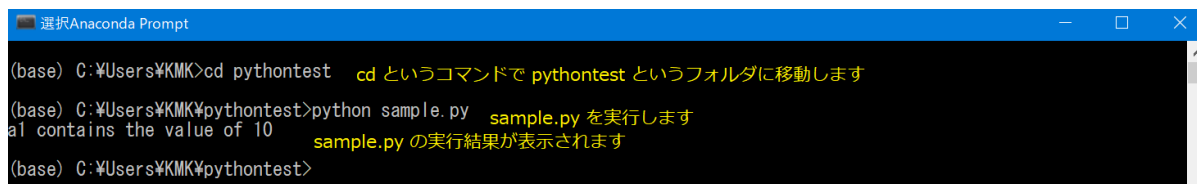


下記のようなウィンドウが表示されます。



Windows のユーザーアカウント名のついたフォルダ（画像では、KMK）の中に `pythontest` というフォルダを作成し、その中に `sample.py` を格納した場合の実行例を示します。

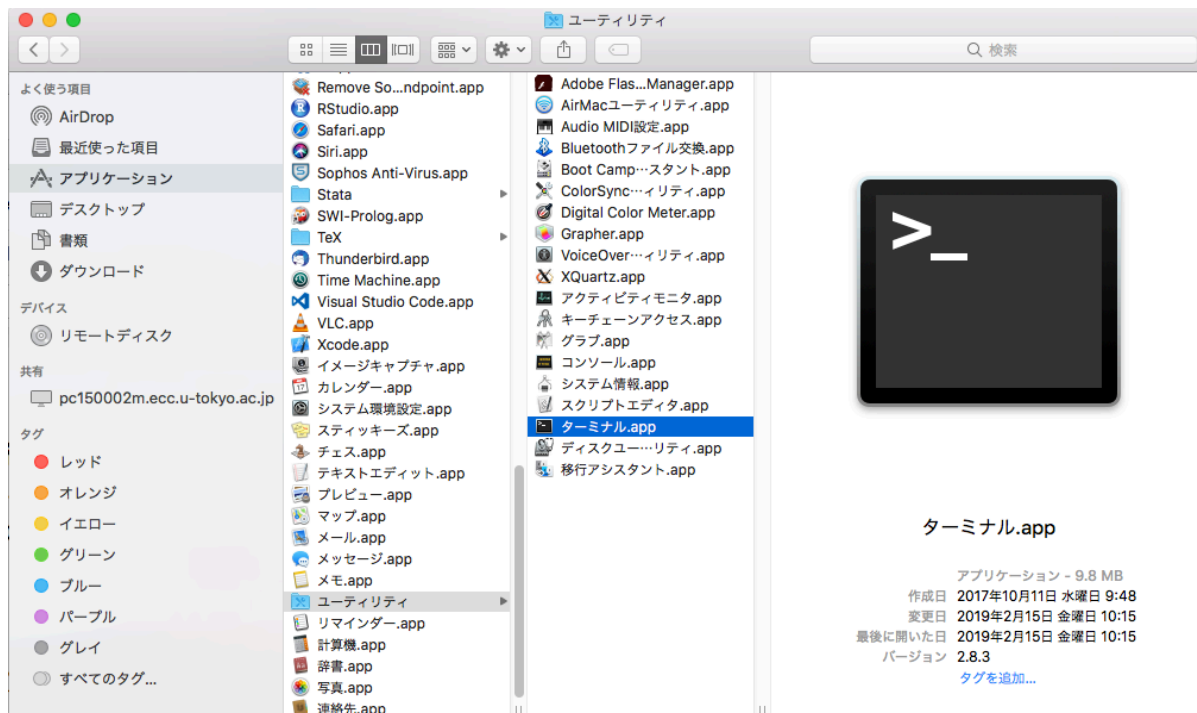
例では、`cd` というコマンドで `sample.py` を格納したフォルダ `pythontest` に移動し、その上で `sample.py` を実行しています。



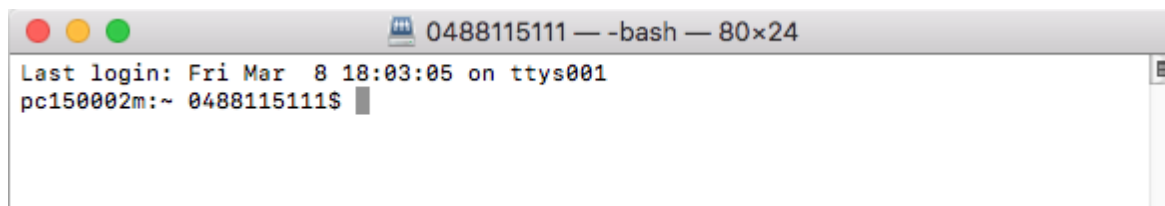
29.1.3 macOS での実行方法

Application ⇒ Utilities ⇒ Terminal.app を起動します。

アプリケーション ⇒ ユーティリティ ⇒ ターミナル.app を起動します。（日本語の場合）

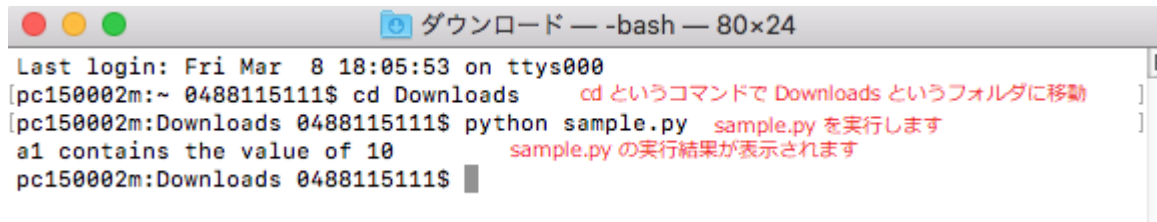


下記のようなウィンドウが表示されます。



ダウンロードフォルダ (Downloads) に `sample.py` を格納した場合の実行例を示します。

例では、`cd` というコマンドで `sample.py` を格納した `Downloads` フォルダに移動し、その上で `sample.py` を実行しています。



```

Last login: Fri Mar  8 18:05:53 on ttys000
[pc150002m:~ 0488115111$ cd Downloads      cd というコマンドで Downloads というフォルダに移動
[pc150002m:Downloads 0488115111$ python sample.py  sample.py を実行します
a1 contains the value of 10                sample.py の実行結果が表示されます
[pc150002m:Downloads 0488115111$

```

29.2 コマンドライン引数

コマンドライン実行時には、実行スクリプト名の後に、文字列を書き込むことにより、実行スクリプトへ引数を与えることができます。この引数のことを、**コマンドライン引数**と呼びます。

たとえば、`argsprint.py` というスクリプトファイルをコマンドライン実行することを考えます。

```
>>> python argsprint.py
```

ここで、`argsprint.py` の後ろに、適当な文字列を付け加えます。たとえば、以下のように3つの文字列 `firstvalue secondvalue thirdvalue` をスペースで区切って付け加えてみます。

```
>>> python argsprint.py firstvalue secondvalue thirdvalue
```

このとき、この3つの文字列が `argsprint.py` にコマンドライン引数として与えられることになります。

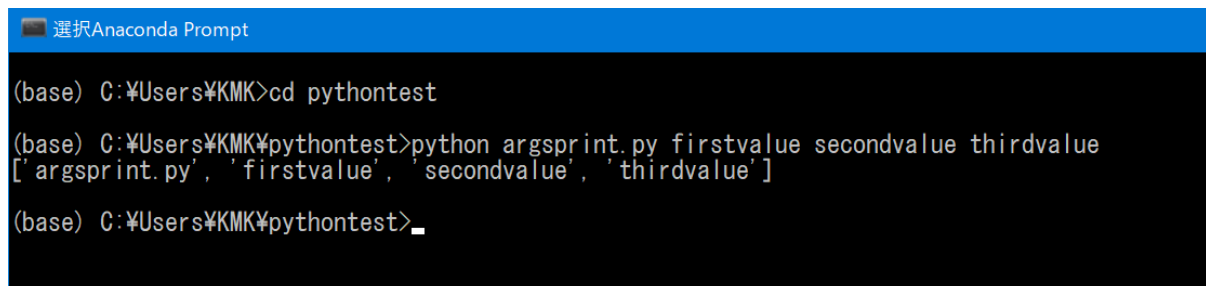
この引数は、`sys` モジュールの `argv` という変数 (`sys.argv`) にリストとして格納されます。

`argsprint.py` を次のようなコードからなるファイルとしましょう。

`argsprint.py`:

```
import sys
print(sys.argv) # リスト sys.argv の中身を印字
```

このような `argsprint.py` を先の例のように実行すると、以下の画像のような結果が得られます。リスト `sys.argv` に2番目の要素として文字列 `firstvalue` が、3番目の要素として文字列 `secondvalue` が、4番目の要素として文字列 `thirdvalue` が格納されていることを確認してください。また、リストの最初の要素には、実行したスクリプト名（ここでは `argsprint.py`）が格納されることに注意してください。



```

選択Anaconda Prompt

(base) C:\Users\KMK>cd pythontest

(base) C:\Users\KMK\pythontest>python argsprint.py firstvalue secondvalue thirdvalue
['argsprint.py', 'firstvalue', 'secondvalue', 'thirdvalue']

(base) C:\Users\KMK\pythontest>_

```

29.2.1 練習

上記に従って `argsprint.py` ファイルを作成して、引数を変更したり、引数の数を増やしたり減らしたりして、表示がどう変わるか調べてください。

29.2.2 練習

コマンドライン実行時に、コマンドライン引数の 1 番目を印字 (`print`) する `arg1.py` を作成せよ。

29.2.3 練習

コマンドライン実行時に、スクリプト名を印字する `scriptname.py` を作成してください。

29.2.4 練習

コマンドライン実行時に、コマンドライン引数の数を印字する `numargs.py` を作成してください。

29.2.5 練習

コマンドライン引数として与えられた任意個の整数の和を印字する `sum.py` を作成してください。

たとえば、次のように実行すると、

```
>>> python sum.py 1 2 3
```

6 と印字されます。

なお、コマンドライン引数は文字列型であることに注意してください。

```
[2]: v1 = '100'
      v2 = '200'
      int(v1) + int(v2) # 整数加算
```

```
[2]: 300
```

```
[3]: v1 + v2 # 文字列結合
```

```
[3]: '100200'
```

29.3 モジュールのコマンドライン実行

さて、モジュールファイルは、それ自体が単独で実行可能であると述べました。つまり、Python ソースファイルは、モジュールとしてインポートされる場合と、スクリプトとしてコマンドライン実行される場合の 2 通りが考えられるわけです。

あるモジュールが、インポートされたのか、スクリプトとしてコマンドライン実行されたのかは、プログラム中の `__name__` という組み込み変数を参照することで区別できます。

具体的には、モジュール `mod.py` がコマンドライン実行されたとき、`__name__` の値は `'__main__'` になります。一方、`import mod` されたとき、`__name__` の値は `'mod'` になります。

これを利用することで、インポートされた場合とコマンドライン実行された場合で、モジュールの振舞いを変えることができます。たとえば、次に示す `factorial.py` モジュールを考えます。

`factorial.py`:


```
import sys

# 階乗 n! を返す
def fact(n):
    prod = 1
    for i in range(1, n + 1):
        prod *= i
    return prod

if __name__ == '__main__':
    n = int(sys.argv[1]) # 整数 n が 1 番目のコマンドライン引数で与えられる
    print(fact(n))      # n! を印字
```

これに対して、`import factorial` すると、階乗を計算する関数 `factorial.fact()` が利用できるようになります。一方、`python factorial.py 6` とコマンドライン実行すると、6 の階乗である 720 が印字されます。つまり、このモジュールは、階乗を計算するライブラリとしても、階乗を計算するスクリプトとしても利用できるわけです。

もし `if __name__ == '__main__':` の条件分岐が無かったら、モジュールとしてインポートしたときに、インポート元のスクリプトのために与えられたコマンドライン引数を使って、階乗を計算・印字しようとします。これは一般に、望ましい振舞いではありません。

このように、`if __name__ == '__main__':` の分岐中には、自己完結したスクリプトとしての振舞いが記述されます。

ライブラリモジュールとして使うことのみが想定されている場合、テストコードが記述されることもあります。たとえば、次のように記述すると、

`factorial.py`:

```
import sys

# 階乗 n! を返す
def fact(n):
    prod = 1
    for i in range(1, n + 1):
        prod *= i
    return prod

if __name__ == '__main__':
    print('test n = 6:', fact(6) == 720)
    print('test n = 0:', fact(0) == 1)
```

コマンドライン実行したときには、`fact()` が正しく計算されているかテストした結果が印字されます。このテストコードは、ライブラリモジュールとして `import` して利用するときには実行されません。このようにすると、1 つの Python ソースファイルの中で、ライブラリ実装とテストをひとまとめにできて、保守しやすくなります。

29.4 ソースファイル先頭部分にある宣言

29.4.1 文字コード宣言

Python ソースコードは UTF-8 で記述することが公式に推奨されています。

しかし、特に Windows 環境では、歴史的事情から Shift_JIS (`shift_jis`) が使われることがあります。このとき、Python ソースファイルの先頭部分には、次のような文字コード宣言が必要です。

```
# -*- coding: shift_jis -*-
```

これがないと、Python インタプリタがエラーを出して止まります。

なお、UTF-8 で記述されている場合には、文字コード宣言を記述しないことが公式に推奨されています。

29.4.2 shebang

Unix 環境（macOS を含む）では、スクリプトファイルの先頭行には、そのスクリプトを実行するコマンドを指定できるようになっています。この先頭行のことは、**shebang** と呼ばれます。

Unix 環境で Python スクリプトに用いられる標準的な shebang は次です。

```
#!/usr/bin/env python3
```

#! に続く部分で、コマンドを絶対パスで指定します。env コマンドは、その引数（ここでは python3）の名前のコマンドを、環境の中から探して実行します。したがって、上のように記述すると、Python インタプリタがインストールされている場所を気にせずに、Unix 環境における Python3 系列の標準コマンド名である python3 を使って実行できるようになります。

shebang と文字コード宣言の両方を含む場合は、たとえば、次のようになります。

```
#!/usr/bin/env python3
# -*- coding: shift_jis -*-
```

29.5 練習の解答

各セルのコードを保存した .py ファイルが解答です。

```
[4]: #arg1.py
import sys
print(sys.argv[1])

-f
```

```
[5]: #scriptname.py
import sys
print(sys.argv[0])

/home/sato/.local/lib/python3.8/site-packages/ipykernel_launcher.py
```

```
[6]: #numargs.py
import sys
num = len(sys.argv) - 1 # sys.argv[0] はコマンドライン引数ではないので 1 減らす
print(num)

3
```

```
[7]: #sum.py
import sys
s = 0
for v in sys.argv[1:]:
    s += int(v)
print(s)

-----
ValueError                                Traceback (most recent call last)
Input In [7], in <module>
      3 s = 0
      4 for v in sys.argv[1:]:
```

(continues on next page)

(continued from previous page)

```
----> 5      s += int(v)
      6 print(s)
```

ValueError: invalid literal for int() with base 10: '-f'

▲ Matplotlib ライブラリ

Matplotlib ライブラリについて説明します。

参考

- <https://matplotlib.org/>

Matplotlib ライブラリにはグラフを可視化するためのモジュールが含まれています。以下では、Matplotlib ライブラリのモジュールを使った、グラフの基本的な描画について説明します。

Matplotlib ライブラリを使用するには、まず `matplotlib` のモジュールをインポートします。ここでは、基本的なグラフを描画するための `matplotlib.pyplot` モジュールをインポートします。慣例として、同モジュールを `plt` と別名をつけてコードの中で使用します。また、グラフで可視化するデータはリストや配列を用いることが多いため、5-3 で使用した `numpy` モジュールも併せてインポートします。なお、`%matplotlib inline` はノートブック内でグラフを表示するために必要です。

`matplotlib` では、通常 `show()` 関数を呼ぶと描画を行いますが、`inline` 表示指定の場合、`show()` 関数を省略できます。

```
[1]: import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

30.1 線グラフ

`pyplot` モジュールの `plot()` 関数を用いて、リストの要素の数値を y 軸の値としてグラフを描画します。 y 軸の値に対応する x 軸の値は、リストの各要素のインデックスとなっています。

具体的には、次のようにすることでリスト `A` のインデックス `i` に対して、`(i, リスト A[i])` の位置に点を打ち、各点を線でつなぎます。

```
plt.plot(リスト A)
```

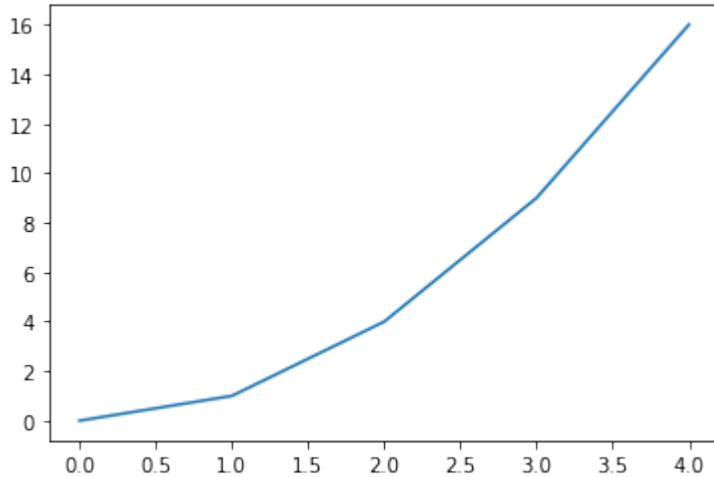
たとえば、次のようになります。

```
[2]: # plot するデータ
d =[0, 1, 4, 9, 16]
```

(continues on next page)

(continued from previous page)

```
# plot 関数で描画
plt.plot(d);
# セルの最後に評価されたオブジェクトの出力表示を抑制するために、以下ではセルの最後の行にセミコロン (;) をつけています。
# 試しにセミコロンを消した場合も試してみてください。
```



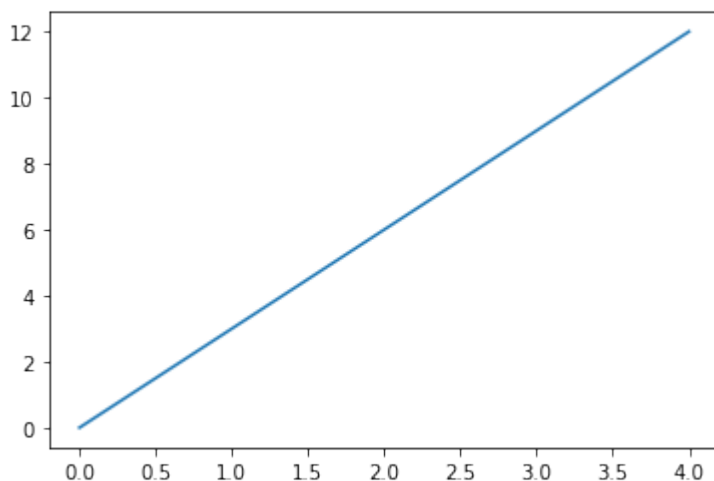
plot() 関数では、x, y の両方の軸の値を引数に渡すこともできます。

具体的には、次のように リスト X と リスト Y を引数として与えると、各 i に対して、(リスト X[i], リスト Y[i]) の位置に点を打ち、各点を線でつなぎます。

```
plt.plot(リスト X, リスト Y)
```

```
[3]: # plot するデータ
x =[0, 1, 2, 3, 4]
y =[0, 3, 6, 9, 12]
```

```
# plot 関数で描画
plt.plot(x,y);
```



リストの代わりに NumPy ライブラリの配列を与えても同じ結果が得られます。

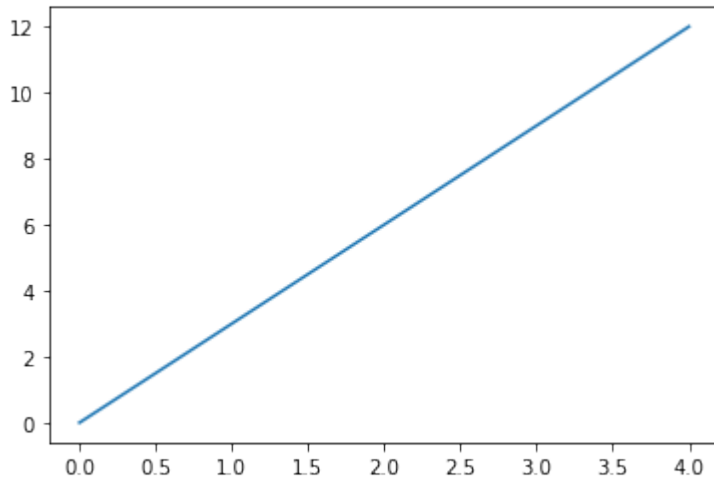
```
[4]: # plot するデータ
x =[0, 1, 2, 3, 4]
aryx = np.array(x) # リストから配列を作成
```

(continues on next page)

(continued from previous page)

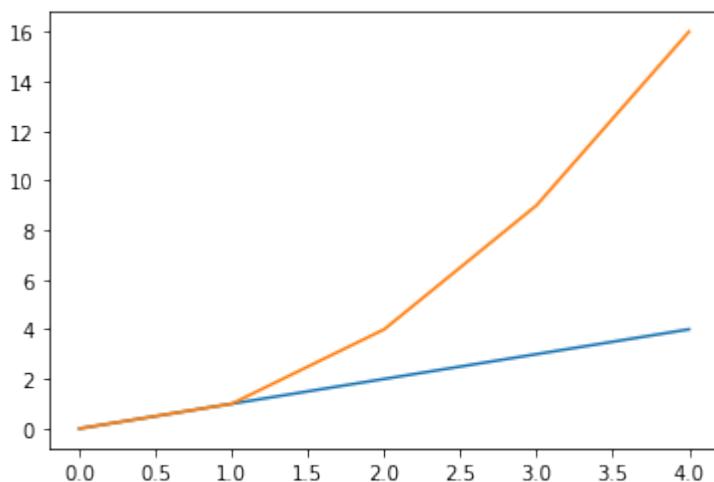
```
y = [0, 3, 6, 9, 12]
arry = np.array(y) # リストから配列を作成

# plot 関数で描画
plt.plot(aryx, arry);
```



以下のようにグラフを複数まとめて表示することもできます。複数のグラフを表示すると、線ごとに異なる色が自動で割り当てられます。

```
[5]: # plot するデータ
data = [0, 1, 4, 9, 16]
x = [0, 1, 2, 3, 4]
y = [0, 1, 2, 3, 4]
# plot 関数で描画。
plt.plot(x, y)
plt.plot(data);
```

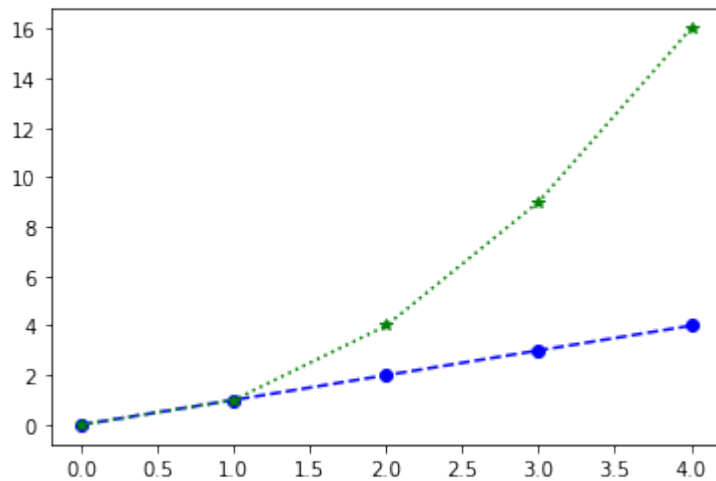


plot() 関数ではグラフの線の色、形状、データポイントのマーカの種類を、それぞれ以下のように `linestyle`, `color`, `marker` 引数で指定して変更することができます。それぞれの引数で指定可能な値は以下を参照してください。

- `linestyle`
- `color`
- `marker`

```
[6]: # plot するデータ
data =[0, 1, 4, 9, 16]
x =[0, 1, 2, 3, 4]
y =[0, 1, 2, 3, 4]

# plot 関数で描画。線の形状、色、データポイントのマーカ指定
plt.plot(x,y, linestyle='--', color='blue', marker='o')
plt.plot(data, linestyle=':', color='green', marker='*');
```

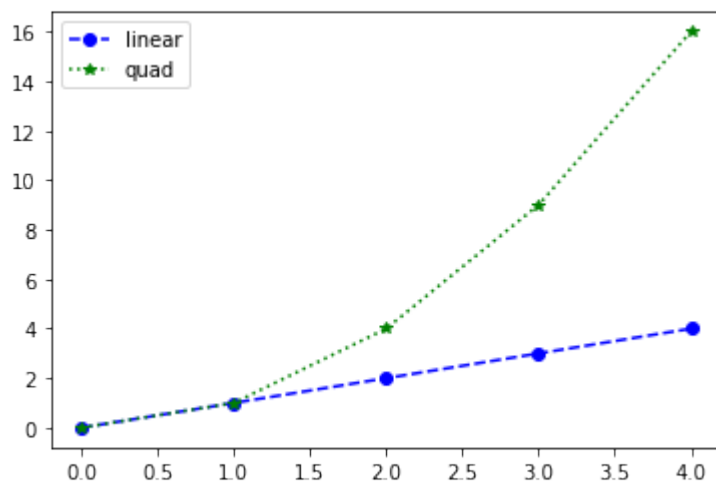


plot() 関数の label 引数にグラフの各線の凡例を文字列として渡し、legend() 関数を呼ぶことで、グラフ内に凡例を表示できます。legend() 関数の loc 引数で凡例を表示する位置を指定することができます。引数で指定可能な値は以下を参照してください。

- legend() 関数

```
[7]: # plot するデータ
data =[0, 1, 4, 9, 16]
x =[0, 1, 2, 3, 4]
y =[0, 1, 2, 3, 4]

# plot 関数で描画。線の形状、色、データポイントのマーカ指定
plt.plot(x,y, linestyle='--', color='blue', marker='o', label='linear')
plt.plot(data, linestyle=':', color='green', marker='*', label='quad')
#凡例を表示
plt.legend();
```



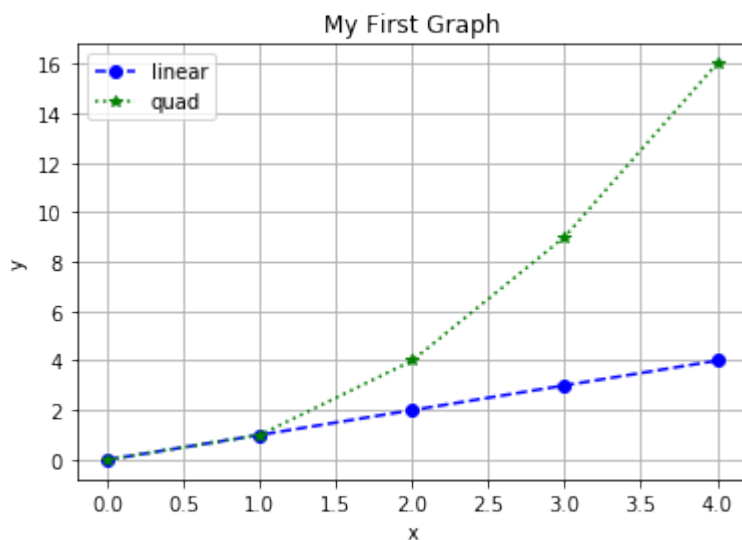
pyplot モジュールでは、以下のようにグラフのタイトルと各軸のラベルを指定して表示することができます

す。タイトル、x 軸のラベル、y 軸のラベル、はそれぞれ `title()` 関数、`xlabel()` 関数、`ylabel()` 関数に文字列を渡して指定します。また、`grid()` 関数を用いるとグリッドを併せて表示することもできます。グリッドを表示させたい場合は、`grid()` 関数に `True` を渡してください。

```
[8]: # plot するデータ
data = [0, 1, 4, 9, 16]
x = [0, 1, 2, 3, 4]
y = [0, 1, 2, 3, 4]

# plot 関数で描画。線の形状、色、データポイントのマーカ、凡例を指定
plt.plot(x,y, linestyle='--', color='blue', marker='o', label='linear')
plt.plot(data, linestyle=':', color='green', marker='*', label='quad')
plt.legend()

plt.title('My First Graph') # グラフのタイトル
plt.xlabel('x') # x 軸のラベル
plt.ylabel('y') # y 軸のラベル
plt.grid(True); # グリッドの表示
```

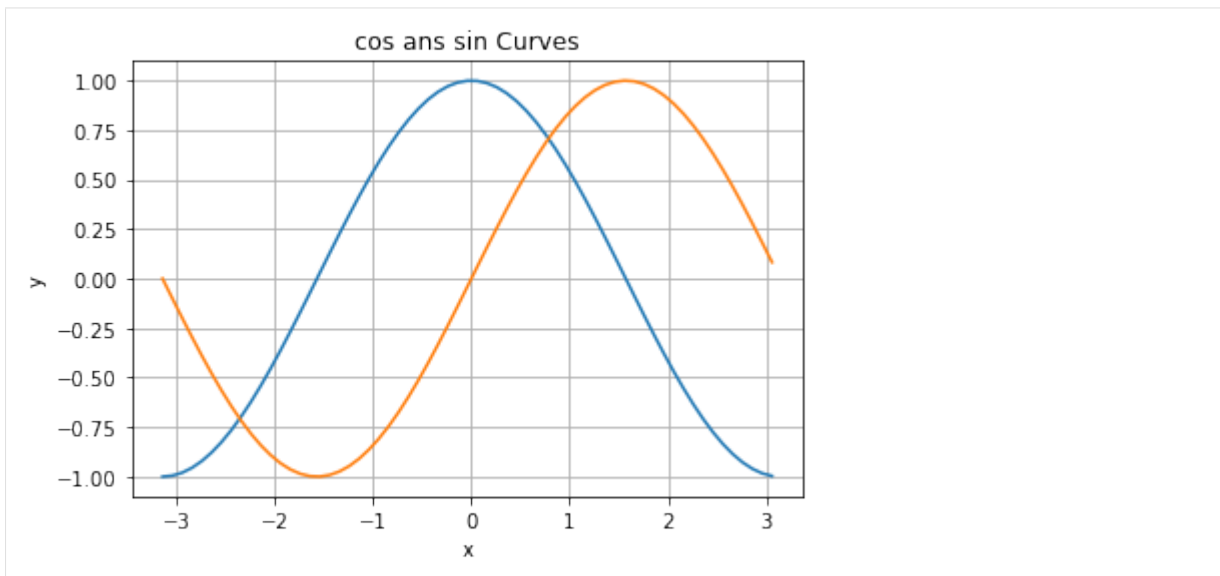


グラフを描画するときのプロット数を増やすことで任意の曲線のグラフを作成することもできます。以下では、`numpy` モジュールの `arange()` 関数を用いて、 $-\pi$ から π の範囲を 0.1 刻みで x 軸の値を配列として準備しています。その x 軸の値に対して、`numpy` モジュールの `cos()` 関数と `sin()` 関数を用いて、y 軸の値をそれぞれ準備し、`cos` カーブと `sin` カーブを描画しています。

```
[9]: # グラフの x 軸の値となる配列
x = np.arange(-np.pi, np.pi, 0.1)

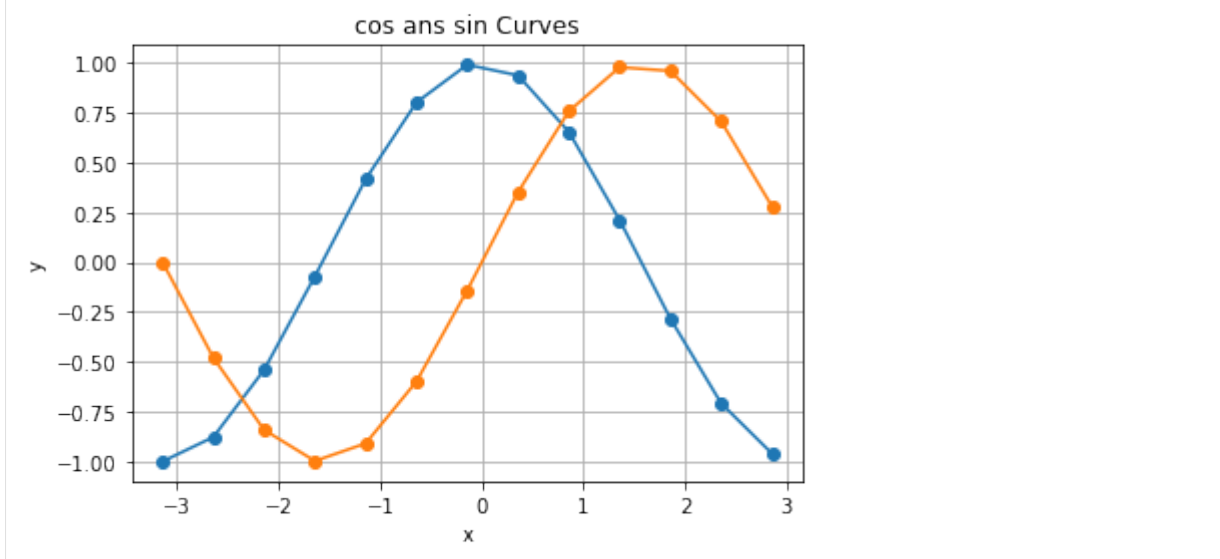
# 上記配列を cos, sin 関数に渡し、y 軸の値として描画
plt.plot(x,np.cos(x))
plt.plot(x,np.sin(x))

plt.title('cos ans sin Curves') # グラフのタイトル
plt.xlabel('x') # x 軸のラベル
plt.ylabel('y') # y 軸のラベル
plt.grid(True); # グリッドの表示
```

プロットの数进行少なくすると、曲線は直線をつなぎ合わせることで描画されていることがわかります。

```
[10]: x = np.arange(-np.pi, np.pi, 0.5)
plt.plot(x, np.cos(x), marker='o')
plt.plot(x, np.sin(x), marker='o')
plt.title('cos ans sin Curves')
plt.xlabel('x')
plt.ylabel('y')
plt.grid(True);
```



30.1.1 グラフの例：ソートアルゴリズムにおける比較回数

```
[11]: import random

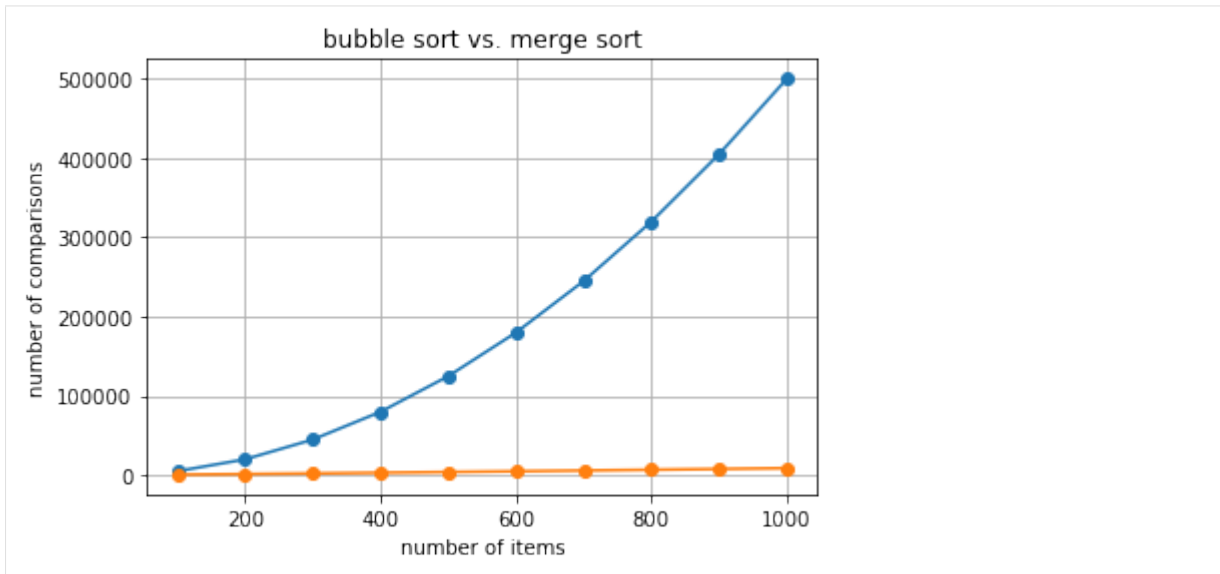
def bubble_sort(lst):
    n = 0
    for j in range(len(lst) - 1):
        for i in range(len(lst) - 1 - j):
            n = n + 1
            if lst[i] > lst[i+1]:
                lst[i + 1], lst[i] = lst[i], lst[i+1]
    return n

def merge_sort_rec(data, l, r, work):
    if l+1 >= r:
        return 0
    m = l+(r-l)//2
    n1 = merge_sort_rec(data, l, m, work)
    n2 = merge_sort_rec(data, m, r, work)
    n = 0
    i1 = l
    i2 = m
    for i in range(l, r):
        from1 = False
        if i2 >= r:
            from1 = True
        elif i1 < m:
            n = n + 1
            if data[i1] <= data[i2]:
                from1 = True
        if from1:
            work[i] = data[i1]
            i1 = i1 + 1
        else:
            work[i] = data[i2]
            i2 = i2 + 1
    for i in range(l, r):
        data[i] = work[i]
    return n1+n2+n

def merge_sort(data):
    return merge_sort_rec(data, 0, len(data), [0]*len(data))

[12]: x = np.arange(100, 1100, 100)
bdata = np.array([bubble_sort([random.randint(1,10000) for i in range(k)]) for k in x])
mdata = np.array([merge_sort([random.randint(1,10000) for i in range(k)]) for k in x])

[13]: plt.plot(x, bdata, marker='o')
plt.plot(x, mdata, marker='o')
plt.title('bubble sort vs. merge sort')
plt.xlabel('number of items')
plt.ylabel('number of comparisons')
plt.grid(True);
```



30.2 練習

-2 から 2 の範囲を 0.1 刻みで x 軸の値を配列として作成し、その x 軸の値に対して `numpy` モジュールの `exp()` 関数を用いて y 軸の値を作成し、 $y = e^x$ のグラフを描画する関数 `plot_exp` を作成してください。ただし、そのグラフに任意のタイトル、x 軸、y 軸の任意のラベル、任意の凡例、グリッドを表示させてください。

```
[14]: import ...
      ...
      def plot_exp():
      ...

Input In [14]
import ...
      ^
SyntaxError: invalid syntax
```

上のセルで解答を作成した後、以下のセルを実行し、実行結果が全て `True` になることを確認してください。

```
[15]: res_x = plot_exp()
      print(len(res_x) == 41, int(res_x[0]) == -2, int(res_x[9]) == -1)

-----
NameError                                Traceback (most recent call last)
Input In [15], in <module>
----> 1 res_x = plot_exp()
      2 print(len(res_x) == 41, int(res_x[0]) == -2, int(res_x[9]) == -1)

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

30.3 練習

4-csv で説明したように、tokyo-temps.csv には、気象庁のオープンデータからダウンロードした、東京の平均気温のデータが入っています。具体的には、各行の第 2 列に気温の値が格納されており、47 行目に 1875 年 6 月の、48 行目に 1875 年 7 月の、…、53 行目に 1875 年 12 月の、54 行目に 1876 年 1 月の、…という風に 2017 年 1 月のデータまでが格納されています。

そこで、2つの整数 year と month を引数として取り、year 年以降の month 月の平均気温の値を y 軸に、年を x 軸に描画した線グラフを表示するとともに、描画した x 軸と y 軸の値をタプルに格納して返す関数 plot_tokyotemps を作成してください。

以下のセルの ... のところを書き換えて解答してください。

```
[16]: import ...
      ...
      def plot_tokyotemps(year, month):
          ...
```

```
Input In [16]
import ...
      ^
SyntaxError: invalid syntax
```

上のセルで解答を作成した後、以下のセルを実行し、実行結果が全て True になることを確認してください。

```
[17]: res_years, res_temps = plot_tokyotemps(1875, 7)
print(len(res_years) == 142, len(res_temps) == 142, res_years[0] == 1875, res_
      ↪temps[0] == 26.0)
res_years, res_temps = plot_tokyotemps(1875, 6)
print(len(res_years) == 142, len(res_temps) == 142, res_years[0] == 1875, res_
      ↪temps[0] == 22.3)
res_years, res_temps = plot_tokyotemps(1875, 12)
print(len(res_years) == 142, len(res_temps) == 142, res_years[0] == 1875, res_
      ↪temps[0] == 4.6)
res_years, res_temps = plot_tokyotemps(1876, 1)
print(len(res_years) == 142, len(res_temps) == 142, res_years[0] == 1876, res_
      ↪temps[0] == 1.6)
res_years, res_temps = plot_tokyotemps(1876, 6)
print(len(res_years) == 141, len(res_temps) == 141, res_years[0] == 1876, res_
      ↪temps[0] == 18.5)
res_years, res_temps = plot_tokyotemps(1900, 6)
print(len(res_years) == 117, len(res_temps) == 117, res_years[0] == 1900, res_
      ↪temps[0] == 19.3)
```

```
-----
NameError                                Traceback (most recent call last)
Input In [17], in <module>
----> 1 res_years, res_temps = plot_tokyotemps(1875, 7)
      2 print(len(res_years) == 142, len(res_temps) == 142, res_years[0] == 1875, res_
      ↪temps[0] == 26.0)
      3 res_years, res_temps = plot_tokyotemps(1875, 6)

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

30.4 散布図

散布図は、`pyplot` モジュールの `scatter()` 関数を用いて描画できます。

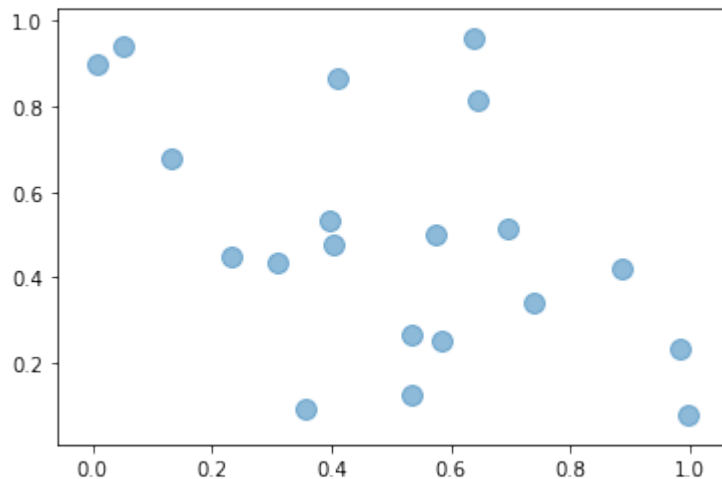
具体的には、次のように リスト `X` と リスト `Y`（もしくは、配列 `X` と 配列 `Y`）を引数として与えると、各 `i` に対して、(リスト `X[i]`, リスト `Y[i]`) の位置に点を打ちます。

```
plt.scatter(リスト X, リスト Y)
```

以下では、ランダムに生成した 20 個の要素からなる配列 `x, y` の各要素の値の組を点としてプロットした散布図を表示しています。プロットする点のマーカの色や形状は、線グラフの時と同様に、`color, marker` 引数で指定して変更することができます。加えて、`s, alpha` 引数で、それぞれマーカの大きさと透明度を指定することができます。

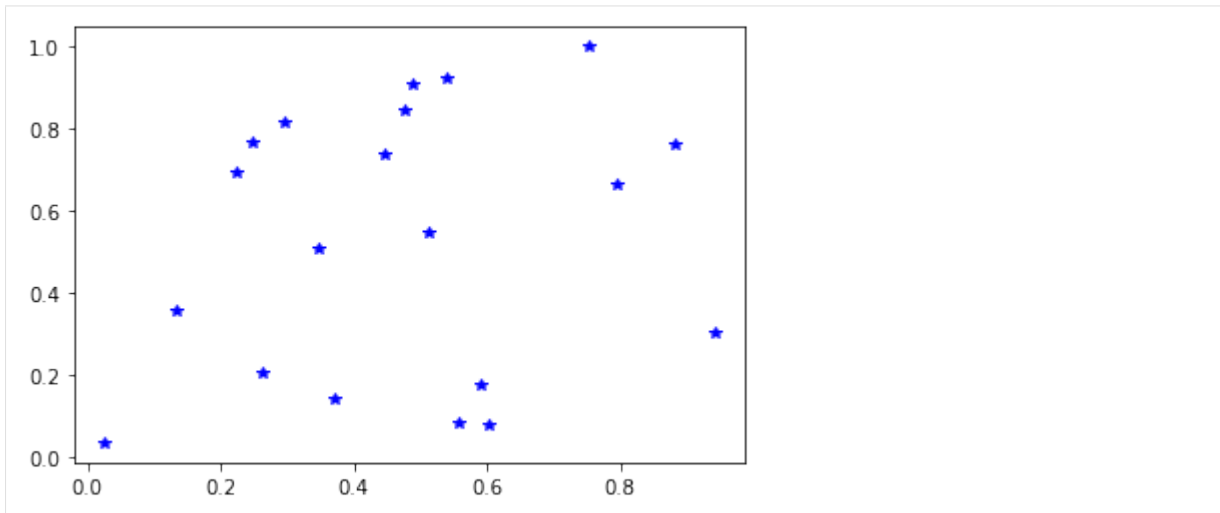
```
[18]: # グラフの x 軸の値となる配列
x = np.random.rand(20)
# グラフの y 軸の値となる配列
y = np.random.rand(20)

# scatter 関数で散布図を描画
plt.scatter(x, y, s=100, alpha=0.5);
```



以下のように、`plot()` 関数を用いても同様の散布図を表示することができます。具体的には、3 番目の引数にプロットする点のマーカの形状を指定することにより実現します。

```
[19]: x = np.random.rand(20)
y = np.random.rand(20)
plt.plot(x, y, '*', color='blue');
```



30.5 練習

tokyo-temps.csv には、気象庁のオープンデータからダウンロードした、東京の平均気温のデータが入っています。具体的には、各行の第 2 列に気温の値が格納されており、47 行目に 1875 年 6 月の、48 行目に 1875 年 7 月の、…、53 行目に 1875 年 12 月の、54 行目に 1876 年 1 月の、…という風に 2017 年 1 月のデータまでが格納されています。

そこで、1875 年以降の平均気温の値を y 軸に、月の値を x 軸に描画した散布図を表示するとともに、描画した x 軸と y 軸の値をタプルに格納して返す関数 `scatter_tokyotemps()` を作成してください。

以下のセルの ... のところを書き換えて解答してください。

```
[20]: import ...
      ...
      def scatter_tokyotemps():
          ...

Input In [20]
import ...
      ^
SyntaxError: invalid syntax
```

上のセルで解答を作成した後、以下のセルを実行し、実行結果が全て True になることを確認してください。

```
[21]: res_months, res_temps = scatter_tokyotemps()
print(len(res_months) == 1700, len(res_temps) == 1700, res_months[0] == 6, res_
      ↪months[1] == 7, res_months[12] == 6, res_months[13] == 7)
print(res_temps[0] == 22.3, res_temps[1] == 26.0, res_temps[12] == 18.5, res_
      ↪temps[13] == 24.3)

-----
NameError                                Traceback (most recent call last)
Input In [21], in <module>
----> 1 res_months, res_temps = scatter_tokyotemps()
      2 print(len(res_months) == 1700, len(res_temps) == 1700, res_months[0] == 6,
      ↪res_months[1] == 7, res_months[12] == 6, res_months[13] == 7)
      3 print(res_temps[0] == 22.3, res_temps[1] == 26.0, res_temps[12] == 18.5, res_
      ↪temps[13] == 24.3)

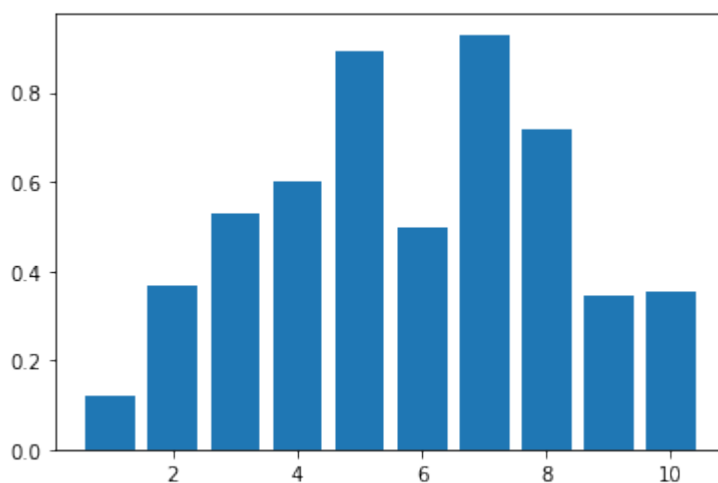
NameError: name 'scatter_tokyotemps' is not defined
```

30.6 棒グラフ

棒グラフは、`pyplot` モジュールの `bar()` 関数を用いて描画できます。以下では、ランダムに生成した 10 個の要素からなる配列 `y` の各要素の値を縦の棒グラフで表示しています。`x` は、`x` 軸上で棒グラフのバーの並ぶ位置を示しています。ここでは、`numpy` モジュールの `arange()` 関数を用いて、1 から 10 の範囲を 1 刻みで `x` 軸上のバーの並ぶ位置として配列を準備しています。

```
[22]: # x 軸上で棒の並ぶ位置となる配列
x = np.arange(1, 11, 1)
# グラフの y 軸の値となる配列
y = np.random.rand(10)

# bar 関数で棒グラフを描画
#print(x, y)
plt.bar(x,y);
```



30.7 練習

`tokyo-temps.csv` には、気象庁のオープンデータからダウンロードした、東京の平均気温のデータが入っています。具体的には、各行の第 2 列に気温の値が格納されており、47 行目に 1875 年 6 月の、48 行目に 1875 年 7 月の、…、53 行目に 1875 年 12 月の、54 行目に 1876 年 1 月の、…という風に 2017 年 1 月のデータまでが格納されています。

そこで、4 つの引数 `year1, month1, year2, month2` を引数に取り、`year1` 年 `month1` 月から `year2` 年 `month2` 月までの各月の平均気温の値を `y` 軸に、年月の値 (`tokyo-temps.csv` の 1 列目の値) を `x` 軸に描画した棒グラフを表示するとともに、描画した `x` 軸と `y` 軸の値をタプルに格納して返す関数 `bar_tokyotemps` を作成してください。

以下のセルの ... のところを書き換えて解答してください。

```
[23]: import ...
...
def bar_tokyotemps(year1, month1, year2, month2):
    ...
```

Input In [23]

```
import ...
```

^

SyntaxError: invalid syntax

上のセルで解答を作成した後、以下のセルを実行し、実行結果が全て True になることを確認してください。

```
[24]: res_months, res_temps = bar_tokyotemps(2000, 6, 2001, 6)
print(len(res_months) == 13, res_months[0] == '2000/6', res_temps[0] == 22.5, res_
      ↪ months[12] == '2001/6', res_temps[12] == 23.1)
```

```
-----
NameError                                Traceback (most recent call last)
Input In [24], in <module>
----> 1 res_months, res_temps = bar_tokyotemps(2000, 6, 2001, 6)
      2 print(len(res_months) == 13, res_months[0] == '2000/6', res_temps[0] == 22.5, ↪
      ↪ res_months[12] == '2001/6', res_temps[12] == 23.1)

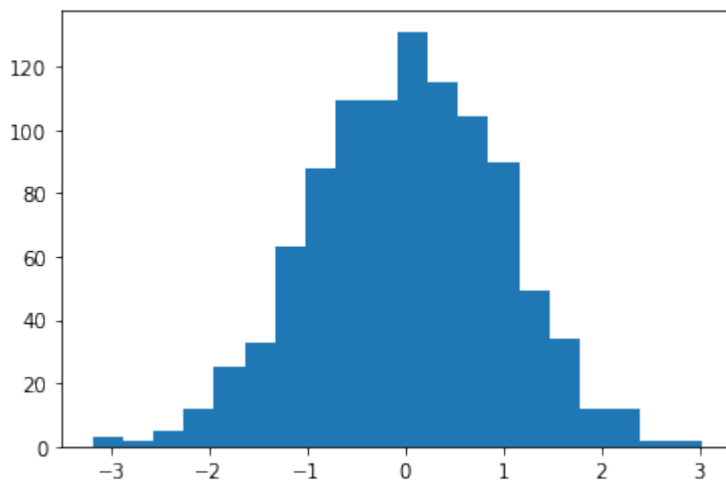
NameError: name 'bar_tokyotemps' is not defined
```

30.8 ヒストグラム

ヒストグラムは、pyplot モジュールの `hist()` 関数を用いて描画できます。以下では、`numpy` モジュールの `random.randn()` 関数を用いて、正規分布に基づく 1000 個の数値の要素からなる配列を用意し、ヒストグラムとして表示しています。`hist()` 関数の `bins` 引数でヒストグラムの箱（ビン）の数を指定します。

```
[25]: # 正規分布に基づく 1000 個の数値の要素からなる配列
d = np.random.randn(1000)

# hist 関数でヒストグラムを描画
plt.hist(d, bins=20);
```



30.9 練習

`tokyo-temps.csv` には、気象庁のオープンデータからダウンロードした、東京の平均気温のデータが入っています。具体的には、各行の第 2 列に気温の値が格納されており、47 行目に 1875 年 6 月の、48 行目に 1875 年 7 月の、…、53 行目に 1875 年 12 月の、54 行目に 1876 年 1 月の、…という風に 2017 年 1 月のデータまでが格納されています。

そこで、5 つの引数 `year1, month1, year2, month2, mybin` を引数に取り、`year1` 年 `month1` 月から `year2` 年 `month2` 月までの各月の平均気温の値を格納したリスト `temps` から `mybin` 個のヒストグラムを表示するとともに、`temps` を返す関数 `hist_tokyotemps` を作成してください。

以下のセルの … のところを書き換えて解答してください。


```
[26]: import ...
...
def hist_tokyotemps(year1, month1, year2, month2, mybin):
    ...
```

Input In [26]

```
import ...
```

^

SyntaxError: invalid syntax

上のセルで解答を作成した後、以下のセルを実行し、実行結果が全て True になることを確認してください。

```
[27]: res_temps = hist_tokyotemps(1875, 6, 2000, 6, 50)
print(len(res_temps) == 1501, res_temps[0] == 22.3, res_temps[1500] == 22.5)
```

NameError Traceback (most recent call last)

Input In [27], in <module>

```
----> 1 res_temps = hist_tokyotemps(1875, 6, 2000, 6, 50)
      2 print(len(res_temps) == 1501, res_temps[0] == 22.3, res_temps[1500] == 22.5)
```

NameError: name 'hist_tokyotemps' is not defined

30.10 ヒートマップ

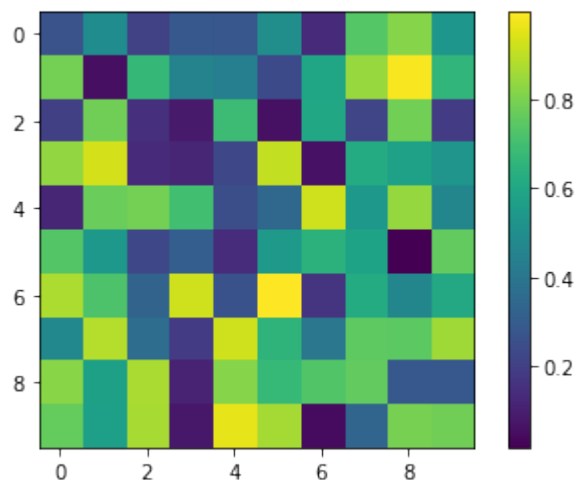
imshow() 関数を用いると、以下のように行列の要素の値に応じて色の濃淡を変えることで、行列をヒートマップとして可視化することができます。colorbar() 関数は行列の値と色の濃淡の対応を表示します。

```
[28]: # 10 行 10 列のランダム要素からなる行列
ary1 = np.random.rand(100)
ary2 = ary1.reshape(10,10)
#ary2 = np.random.rand(100).reshape(10,10) #と同じ
```

imshow 関数でヒートマップを描画

```
im=plt.imshow(ary2)
```

```
plt.colorbar(im);
```



30.11 練習

tokyo-temps.csv には、気象庁のオープンデータからダウンロードした、東京の平均気温のデータが入っています。具体的には、各行の第 2 列に気温の値が格納されており、47 行目に 1875 年 6 月の、48 行目に 1875 年 7 月の、…、53 行目に 1875 年 12 月の、54 行目に 1876 年 1 月の、…という風に 2017 年 1 月のデータまでが格納されています。

そこで、 30×12 の NumPy の配列 `ary1` を作成し、各月の平均気温を整数に丸めた値を求めて、月ごとにその値の数を数えて配列 `ary1` に格納して、`ary1` からなるヒートマップを表示しつつ、`ary1` を返す関数 `heat_tokyotemps` を作成してください。ただし、厳密には `x` が 0 以上 11 以下の任意の整数とし、`y` を 0 以上 29 以下の整数とするとき、`ary1[y][x]` には、`y` °C 以上、`y+1` °C より小さい平均気温を持つ `x+1` 月の数が格納されているものとします。

以下のセルの ... のところを書き換えて解答してください。

```
[29]: import ...
      ...
      def heat_tokyotemps():
          ...

Input In [29]
import ...
      ^
SyntaxError: invalid syntax
```

上のセルで解答を作成した後、以下のセルを実行し、実行結果が全て True になることを確認してください。

```
[30]: ary1 = heat_tokyotemps()
      print(ary1[0][0] == 2, ary1[1][1] == 2, ary1[2][0] == 28)
      #画像の向きが気になる人は、以下の 2 行を同時に実行してみてください
      #ary1 = np.flip(ary1, axis=0)
      #im=plt.imshow(ary1)

-----
NameError                                Traceback (most recent call last)
Input In [30], in <module>
----> 1 ary1 = heat_tokyotemps()
      2 print(ary1[0][0] == 2, ary1[1][1] == 2, ary1[2][0] == 28)

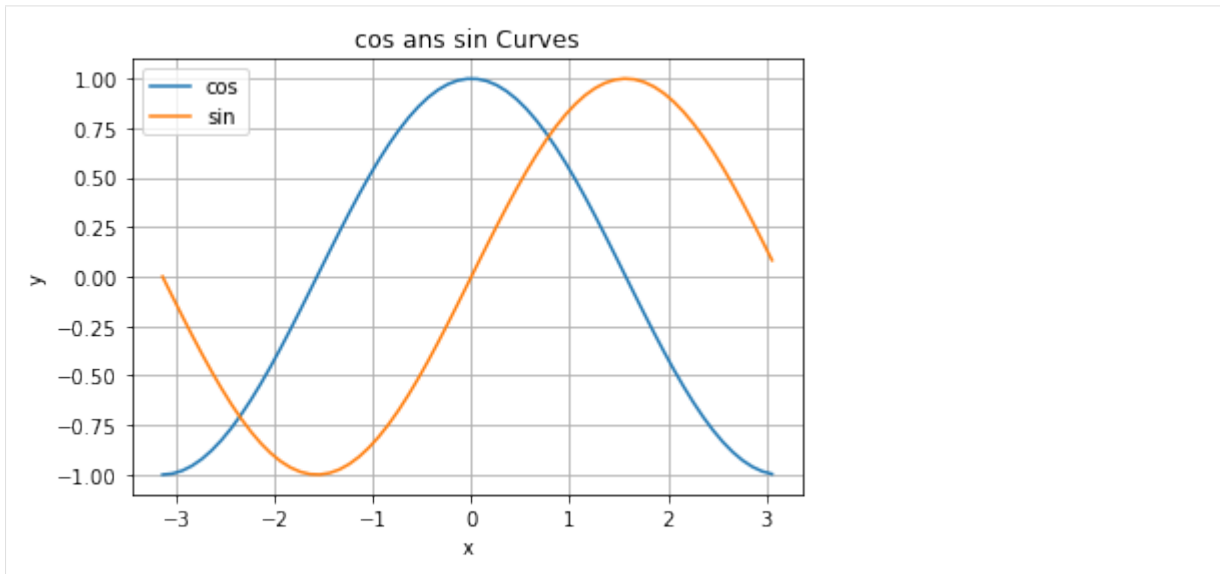
NameError: name 'heat_tokyotemps' is not defined
```

30.12 グラフの画像ファイル出力

`savefig()` 関数を用いると、以下のように作成したグラフを画像としてファイルに保存することができます。

```
[31]: x = np.arange(-np.pi, np.pi, 0.1)
      plt.plot(x,np.cos(x), label='cos')
      plt.plot(x,np.sin(x), label='sin')
      plt.legend()
      plt.title('cos ans sin Curves')
      plt.xlabel('x')
      plt.ylabel('y')
      plt.grid(True)

      # savefig 関数でグラフを画像保存
      plt.savefig('cos_sin.png');
```



30.13 練習の解答

```
[32]: import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

def plot_exp():
    x = np.arange(-2, 2.1, 0.1)
    y = np.exp(x)
    plt.plot(x, y, linestyle='--', color='blue', marker='x', label='exp(x)')
    plt.title('y = exp(x)') # タイトル
    plt.xlabel('x') # x 軸のラベル
    plt.ylabel('exp(x)') # y 軸のラベル
    plt.grid(True); # グリッドを表示
    plt.legend() # 盆例を表示
    return x
```

```
[33]: import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
import csv

def plot_tokyotemps(year, month):
    with open('tokyo-temps.csv', 'r', encoding='sjis') as f:
        dataReader = csv.reader(f) # csv リーダを作成
        n=0
        # 1875 年 6 月が 47 行目なので、指定された year 年 6 月のデータの行番号をまず求める
        init_row = (year - 1875) * 12 + 47
        # その上で、year 年 month 月のデータの行番号を求める
        init_row = init_row + month - 6
        years = [] # 年
        temps = [] # 平均気温
        for row in dataReader: # CSV ファイルの中身を 1 行ずつ読み込み
            n = n+1
            if n >= init_row and (n - init_row) % 12 == 0: # init_row 行目からはじめて
                12 か月ごとに if 内を実行
```

(continues on next page)

(continued from previous page)

```

        years.append(year)
        temp = float(row[1]) # float 関数で実数のデータ型に変換する
        temps.append(temp)
        year = year + 1
    #print(years)
    #print(temps)
    plt.plot(years, temps)
    return years, temps

```

```

[34]: import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
import csv

def scatter_tokyotemps():
    with open('tokyo-temps.csv', 'r', encoding='sjis') as f:
        dataReader = csv.reader(f) # csv リーダを作成
        n=0
        months = [] # 月
        temps = [] # 平均気温
        month = 6 # 47行目は6月
        for row in dataReader: # CSV ファイルの中身を1行ずつ読み込み
            n = n+1
            if n >= 47: # 47行目から if 内を実行
                months.append(month)
                temp = float(row[1]) # float 関数で実数のデータ型に変換する
                temps.append(temp)
                month = month + 1
                if month > 12:
                    month = 1
        #print(months)
        #print(temps)
        plt.scatter(months, temps, alpha=0.5)
        return months, temps

```

```

[35]: import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
import csv

def bar_tokyotemps(year1, month1, year2, month2):
    with open('tokyo-temps.csv', 'r', encoding='sjis') as f:
        dataReader = csv.reader(f) # csv リーダを作成
        n=0
        months = [] #
        temps = [] # 平均気温
        init_row = (year1 - 1875) * 12 - 6 + month1 + 47
        end_row = (year2 - 1875) * 12 - 6 + month2 + 47
        for row in dataReader: # CSV ファイルの中身を1行ずつ読み込み
            n = n+1
            if n >= init_row and n <= end_row: # init_row 行目から、end_row 行まで if 内を
                months.append(row[0])
                temp = float(row[1]) # float 関数で実数のデータ型に変換する
                temps.append(temp)
        #print(months)

```

(continues on next page)

(continued from previous page)

```
#print(temps)
plt.bar(months, temps)
return months, temps
```

```
[36]: import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
import csv

def hist_tokyotemps(year1, month1, year2, month2, mybin):
    with open('tokyo-temps.csv', 'r', encoding='sjis') as f:
        dataReader = csv.reader(f) # csv リーダを作成
        n=0
        months = [] #
        temps = [] # 平均気温
        init_row = (year1 - 1875) * 12 - 6 + month1 + 47
        end_row = (year2 - 1875) * 12 - 6 + month2 + 47
        for row in dataReader: # CSV ファイルの中身を 1 行ずつ読み込み
            n = n+1
            if n >= init_row and n <= end_row: # init_row 行目から、end_row 行まで if 内を
                temp = float(row[1]) # float 関数で実数のデータ型に変換する
                temps.append(temp)

        #print(months)
        #print(temps)
        plt.hist(temps, bins=mybin)
        return temps
```

実行

```
[37]: import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
import csv

def heat_tokyotemps():
    ary1 = np.zeros(30*12, dtype=int) # 30 × 12 の配列を作成
    ary1 = ary1.reshape(30, 12)
    with open('tokyo-temps.csv', 'r', encoding='sjis') as f:
        dataReader = csv.reader(f) # csv リーダを作成
        n=0
        month = 6 # 一番最初の月（47 行目）は 6 月
        for row in dataReader: # CSV ファイルの中身を 1 行ずつ読み込み
            n = n+1
            if n >= 47: # 47 行目から if 内を実行
                temp = int(float(row[1])) # まず float 関数で実数型に変換してから、int 関数
                # 整数のデータ型に変換する
                ary1[temp][month-1] += 1 # month 月の値は month-1 行目に格納する
                month += 1
                if month == 13:
                    month = 1
        im=plt.imshow(ary1)
        plt.colorbar(im);
        #print(ary1)
        return ary1
```

[]:

▲正規表現

正規表現について説明します。

参考

- <https://docs.python.jp/3/library/re.html>

正規表現 (regular expression) を扱う場合、`re` というモジュールをインポートする必要があります。

```
[1]: import re
```

31.1 正規表現の基本

正規表現とは、文字列の**パターン**を表す式です。文字列が正規表現に**マッチする**とは、文字列が正規表現の表すパターンに適合していることを意味します。また、正規表現が文字列にマッチするという言い方もします。

たとえば、正規表現 `abc` は文字列 `abcde`（の部分文字列 `abc`）にマッチします。

正規表現に文字列がマッチしているかどうかを調べることができる関数に `match` があります。

`match` は、指定した 正規表現 `A` が 文字列 `B`（の先頭部分）にマッチするかどうか調べます。

```
re.match(正規表現 A, 文字列 B)
```

```
[2]: match1 = re.match('abc', 'abcde') #マッチする
      print(match1)
      match1 = re.match('abc', 'ababc') #マッチしない
      print(match1)
```

```
<re.Match object; span=(0, 3), match='abc'>
None
```

`match` では、マッチが成立している場合、**match オブジェクト**と呼ばれる特殊なデータを返します。マッチが成立しない場合、`None` を返します。

つまり、マッチする部分文字列を含む場合、返値は `None` ではないので、`if` 文などの条件で真とみなされます。したがって以下のようにして条件分岐することができます。

```
if re.match(正規表現, 文字列):
    ...
```

```
[3]: if re.match('abc', 'abcde'): #マッチする
      print(' 正規表現 abc が文字列 abcde にマッチする')
      else:
      print(' 正規表現 abc が文字列 abcde にマッチしない')
      if re.match('abc', 'ababc'): #マッチしない
      print(' 正規表現 abc が文字列 ababc にマッチする')
      else:
      print(' 正規表現 abc が文字列 ababc にマッチしない')
```

```
正規表現 abc が文字列 abcde にマッチする
正規表現 abc が文字列 ababc にマッチしない
```

さて、上で紹介した match オブジェクトには、マッチした文字列の情報が格納されています。上のセルの1つ目の実行結果を印字したのを見てください。

<_sre.SRE_Match object; span=(0, 3), match='abc'> と表示されていると思います。このオブジェクト内の match という値は、マッチした文字列を、span という値はマッチしたパターンが存在する、文字列のインデックスの範囲を表します。

正規表現では大文字と小文字は区別されます。たとえば、正規表現 abc は文字列 ABCdef にはマッチしません。勿論、正規表現 ABC も文字列 abcdef にはマッチしません。

```
[4]: match1 = re.match('abc', 'ABCdef')
      print(match1)
      match1 = re.match('ABC', 'abcdef')
      print(match1)
```

```
None
None
```

そこで match の3番目の引数として re.IGNORECASE もしくは re.I を指定すると、大文字と小文字を区別せずにマッチするかどうかを調べることができます。

```
[5]: match1 = re.match('abc', 'ABCdef', re.IGNORECASE)
      print(match1)
      match1 = re.match('ABC', 'abcdef', re.IGNORECASE)
      print(match1)
      match1 = re.match('ABC', 'ABCdef', re.IGNORECASE)
      print(match1)
      match1 = re.match('abc', 'ABCdef', re.I)
      print(match1)
      match1 = re.match('ABC', 'abcdef', re.I)
      print(match1)
      match1 = re.match('ABC', 'ABCdef', re.I)
      print(match1)
      match1 = re.match('AbC', 'aBcdef', re.I)
      print(match1)
```

```
<re.Match object; span=(0, 3), match='ABC'>
<re.Match object; span=(0, 3), match='abc'>
<re.Match object; span=(0, 3), match='ABC'>
<re.Match object; span=(0, 3), match='ABC'>
<re.Match object; span=(0, 3), match='abc'>
<re.Match object; span=(0, 3), match='ABC'>
<re.Match object; span=(0, 3), match='aBc'>
```

match は文字列の先頭がマッチするかどうか調べますので、次のような場合、match オブジェクトを返さずに None が返されます。

```
[6]: match1 = re.match('def', 'abcdef')
      print(match1)
      match1 = re.match('xyz', 'abcdef')
      print(match1)
```

```
None
None
```

文字列の先頭しか調べられないのでは、いかにも不便です。

そこで、関数 `search` は、指定した正規表現 A が文字列 B に（文字列の先頭以外でも）マッチするかどうか調べることができます。

```
re.search(正規表現 A, 文字列 B)
```

```
[7]: match1 = re.search('abc', 'abcdef')
      print(match1)
      match1 = re.search('abc', 'ababcd')
      print(match1)
      match1 = re.search('def', 'abcdef')
      print(match1)
```

```
<re.Match object; span=(0, 3), match='abc'>
<re.Match object; span=(2, 5), match='abc'>
<re.Match object; span=(3, 6), match='def'>
```

`search` の場合も 3 番目の引数として `re.IGNORECASE`、もしくは `re.I` を指定することで、大文字と小文字を区別せずにマッチするかどうかを調べることができます。

```
[8]: match1 = re.search('abc', 'ABCdef')
      print(match1)
      match1 = re.search('DEF', 'abcdef')
      print(match1)
      match1 = re.search('abc', 'ABCdef', re.IGNORECASE)
      print(match1)
      match1 = re.search('DEF', 'abcdef', re.I)
      print(match1)
      match1 = re.search('not', 'It is NOT me.', re.I)
      print(match1)
      match1 = re.search('NOT', 'It is not mine.', re.I)
      print(match1)
```

```
None
None
<re.Match object; span=(0, 3), match='ABC'>
<re.Match object; span=(3, 6), match='def'>
<re.Match object; span=(6, 9), match='NOT'>
<re.Match object; span=(6, 9), match='not'>
```

`match` 関数と同様に、`search` 関数においても、`if` 文を使った条件分岐が可能であることは覚えておいてください。

```
if re.search(正規表現, 文字列):
    ...
```

```
[9]: if re.search('not', 'It is NOT me.'):
      print('正規表現 not が文字列 It is NOT me. にマッチする')
      else:
          print('正規表現 not が文字列 It is NOT me. にマッチしない')
```

(continues on next page)

(continued from previous page)

```

if re.search('not', 'It is NOT me.', re.I):
    print(' 正規表現 not が文字列 It is NOT me. にマッチする')
else:
    print(' 正規表現 not が文字列 It is NOT me. にマッチしない')

```

```

正規表現 not が文字列 It is NOT me. にマッチしない
正規表現 not が文字列 It is NOT me. にマッチする

```

文字列の先頭からのマッチを調べたいときには、正規表現の先頭にキャレット (^) をつけてください。また、文字列の最後からマッチさせたいときは、正規表現の最後にドル記号 (\$) をつけてください。

```

[10]: match1 = re.search('abc', 'ababcd') #キャレットなしだとマッチする
      print(match1)
      match1 = re.search('^abc', 'ababcd') #キャレットありだとマッチしない
      print(match1)
      match1 = re.search('def', 'abcdefg') #ドル記号なしだとマッチする
      print(match1)
      match1 = re.search('def$', 'abcdefg') #ドル記号ありだとマッチしない
      print(match1)
      match1 = re.search('def$', 'abcdefxyzdef') # 2つあるうちの 2 番目（最後）の def にマッチする
      print(match1)

```

```

<re.Match object; span=(2, 5), match='abc'>
None
<re.Match object; span=(3, 6), match='def'>
None
<re.Match object; span=(9, 12), match='def'>

```

ただ、ここまでの内容だと、正規表現を用いずに文字列のメソッド（find など）によっても実現が可能です。これでは正規表現を使うメリットはほとんどありません。

というのも、ここまで見てきた 1 つの正規表現によって、1 つの文字列を表していたからです。しかし、最初に言った通り、正規表現は文字列の「パターン」を表します。すなわち、1 つの正規表現で複数の文字列を表すことが可能なのです。

たとえば、正規表現 ab と正規表現 de という 2 つの正規表現を | という記号で繋げた ab|de も正規表現を表します。この正規表現では、ab と de という 2 つの文字列を表しており、これらのいずれかを含む文字列にマッチします。この | の記号（演算）を和、もしくは選択といいます。

```

[11]: match1 = re.search('ab|de', 'bcdef')
      print(match1)
      match1 = re.search('ab|de', 'abcdef')
      print(match1)
      match1 = re.search('ab|de', 'fgdeab')
      print(match1)
      match1 = re.search('ab|de', 'acdf')
      print(match1)
      match1 = re.search('a|an|the', 'I slipped on a piece of the banana.')
      print(match1)
      match1 = re.search('a|an|the', 'I slipped on the banana.')
      print(match1)
      match1 = re.search('Good (Morning|Evening)', 'Good Evening, Vietnam.') #正規表現内の ( )
      #については下で述べます
      print(match1)
      match1 = re.search('colo(u|l)r', 'That color matches your suit.') #正規表現内の ( ) につい
      #ては下で述べます
      print(match1)
      match1 = re.search('colo(u|l)r', 'That colour matches your suit.') #正規表現内の ( ) につ
      #いては下で述べます

```

(continues on next page)

(continued from previous page)

```
print(match1)

<re.Match object; span=(2, 4), match='de'>
<re.Match object; span=(0, 2), match='ab'>
<re.Match object; span=(2, 4), match='de'>
None
<re.Match object; span=(13, 14), match='a'>
<re.Match object; span=(13, 16), match='the'>
<re.Match object; span=(0, 12), match='Good Evening'>
<re.Match object; span=(5, 10), match='color'>
<re.Match object; span=(5, 11), match='colour'>
```

上記の3番目の例に注意してください。正規表現 `ab | de` では `ab` が `de` よりも先に記述されていますが、マッチする文字列は文字列上で先に出てきた方 (`ab` ではなく、`de`) であることに注意してください。

細かい話ですが、正規表現 `abc` は、正規表現 `a` と正規表現 `b` と正規表現 `c` という3つの正規表現を繋げて構成された正規表現であり、このように正規表現を繋げて新しい正規表現を作る演算を**連接**といいます。

`a*` は、正規表現 `a` を0回以上繰り返した文字列とマッチします。この `*` の演算を**閉包**といいます。

```
[12]: match1 = re.search('a*', 'abcdef')
print(match1)
match1 = re.search('a*', 'aabbcc')
print(match1)
match1 = re.search('a*', 'cde')
print(match1)
match1 = re.search('bo*', 'boooo!')
print(match1)

<re.Match object; span=(0, 1), match='a'>
<re.Match object; span=(0, 2), match='aa'>
<re.Match object; span=(0, 0), match=''>
<re.Match object; span=(0, 5), match='boooo'>
```

上記の3番目の例において (文字 `a` が含まれていないにも関わらず) `None` が返らずに、マッチしているのを不思議に思うかもしれません。しかし、`a*` は `a` を0回反復した文字列にもマッチします。この0回反復した文字列とは、長さが0の文字列であり、**空列**とか**空文字列**と呼ばれます。文字列 `cde` の先頭には、空列があると見なせるので、`a*` が先頭部分にマッチしているのです。

たとえば、正規表現 `abb*` は、`ab`, `abb`, `abbb`, ... という文字列にマッチします。

```
[13]: match1 = re.search('abb*', 'abcdef')
print(match1)
match1 = re.search('abb*', 'aabbcc')
print(match1)
match1 = re.search('abb*', 'cde')
print(match1)
match1 = re.search('hello*', 'Hi, hellooooo!')
print(match1)
match1 = re.search('hello*', 'Hi, good morning!')
print(match1)

<re.Match object; span=(0, 2), match='ab'>
<re.Match object; span=(1, 4), match='abb'>
None
<re.Match object; span=(4, 13), match='hellooooo'>
None
```

これまでに紹介した連接、和、閉包という3つの演算を組み合わせることで様々な正規表現を記述することができますが、これらの演算には結合の強さが存在します。たとえば、先に見た `ab | cd` という正規表

現は、ab もしくは cd という文字列にマッチします (ab)|(cd) と同じ意味です)。つまり、接続の方が和よりも強く結合しているのです。そこで、丸括弧を使って a(b|c)d とすると、この正規表現は、abd |acd と同じ意味になります。

```
[14]: match1 = re.search('ab|de', 'fgdeab')
print(match1)
match1 = re.search('(ab)|(de)', 'fgdeab')
print(match1)
match1 = re.search('a(b|d)e', 'fgdeab')
print(match1)
match1 = re.search('a(b|d)e', 'fgadeab')
print(match1)
match1 = re.search('abe|ade', 'fgadeab')
print(match1)
match1 = re.search("(I|i)t('s| is| was)", "It was rainy yesterday, but it's fine.
↪today.")
print(match1)
match1 = re.search("(I|i)t('s| is| was)", "It rained yesterday, but it's fine today.")
print(match1)

<re.Match object; span=(2, 4), match='de'>
<re.Match object; span=(2, 4), match='de'>
None
<re.Match object; span=(2, 5), match='ade'>
<re.Match object; span=(2, 5), match='ade'>
<re.Match object; span=(0, 6), match='It was'>
<re.Match object; span=(25, 29), match='it's'>
```

演算の結合の強さは、「和<接続<閉包」という順序になっています。これは数学の、「和 (+) < 積 (×) < べき」と同じですので、直感的にもわかりやすいと思います。これまでに紹介した接続、和、閉包という3つの演算と結合の順序を明記する丸括弧 () とを組み合わせることで様々な正規表現を記述することができます。

```
[15]: match1 = re.search('a(bc|b)*', 'defabcxyz')
print(match1)
match1 = re.search('a(bc|b)*', 'bbacbabbbbc')
print(match1)
match1 = re.search('ca(r|t(egory|tle|))', 'What category is this cat in?')
print(match1)
match1 = re.search('ca(r|t(egory|tle|))', 'No, this is not a carpet.')
print(match1)
match1 = re.search('ca(r|t(egory|tle|))', 'We saw a cattle car almost hit the cat.')
print(match1)
match1 = re.search('ca(r|t(egory|tle|))', 'Please locate him.')
print(match1)
match1 = re.search('ca(r|t(egory|tle|))', "Don't play castanets.")
print(match1)

<re.Match object; span=(3, 6), match='abc'>
<re.Match object; span=(2, 3), match='a'>
<re.Match object; span=(5, 13), match='category'>
<re.Match object; span=(18, 21), match='car'>
<re.Match object; span=(9, 15), match='cattle'>
<re.Match object; span=(9, 12), match='cat'>
None
```

Python では正規表現は文字列によって表していることに注意してください。たとえば、match 関数の第一引数を文字列の変数で置き換えられるということです。

```
[16]: match1 = re.match('abc', 'abcde')
print(match1)
reg1 = 'abc' # 正規表現を文字列で記述する
match2 = re.match(reg1, 'abcde') #match1 と同じ結果になる
print(match2)

<re.Match object; span=(0, 3), match='abc'>
<re.Match object; span=(0, 3), match='abc'>
```

このことを覚えておくと複雑な正規表現を書くときに、少しずつ分解して記述することができて便利です。

```
[17]: match1 = re.search("(I|i)t('s| is| was)", "It was rainy yesterday, but it's fine.
↪today.")
print(match1)
reg1 = '(I|i)t' # 正規表現の前半部分
reg2 = "('s| is| was)" # 正規表現の後半部分
reg3 = reg1 + reg2 # 正規表現を表す 2つの文字列を結合する
print(reg3)
match2 = re.search(reg3, "It was rainy yesterday, but it's fine today.")
print(match2)

<re.Match object; span=(0, 6), match='It was'>
(I|i)t('s| is| was)
<re.Match object; span=(0, 6), match='It was'>
```

31.2 練習

文字列 `str1` を引数として取り、`str1` の中に「月を表す文字列」が含まれているかどうか調べて、含まれていればマッチしたときの `match` オブジェクトを、含まれいなければ `None` を返す関数 `check_monthstr` を作成してください。ただし、「月を表す文字列」は次のような文字列とします。

- 長さ 2 の `mm` という文字列
- `mm` は、`00`, `01`, ..., `12` のいずれかの文字列

以下のセルの ... のところを書き換えて解答してください。

```
[18]: import ...
def check_monthstr(str1):
    ...

Input In [18]
import ...
^
SyntaxError: invalid syntax
```

上のセルで解答を作成した後、以下のセルを実行し、実行結果が全て `True` になることを確認してください。

```
[19]: print(check_monthstr('10').group() == '10') # group() については後半に説明があります (オプション)
print(check_monthstr('mon1521vb') == None)
print(check_monthstr('00an23') == None)
print(check_monthstr('13302').group() == '02')
```

```
-----
NameError                                Traceback (most recent call last)
Input In [19], in <module>
----> 1 print(check_monthstr('10').group() == '10') # group() については後半に説明があります (オプション)
```

(continues on next page)

(continued from previous page)

```
2 print(check_monthstr('mon1521vb') == None)
3 print(check_monthstr('00an23') == None)
```

NameError: name 'check_monthstr' is not defined

31.3 練習

文字列 `str1` を引数として取り、`str1` を構成する文字列が A, C, G, T の 4 種類の文字以外の文字を含むかどうか調べて、これら以外を含む場合は `False` を、そうでない場合は `True` を返す関数 `check_ACGTstr` を作成してください。ただし、大文字と小文字は区別しません。また、空列の場合は `False` を返してください。

以下のセルの ... のところを書き換えて解答してください。

```
[20]: import ...
def check_ACGTstr(str1):
    ...

Input In [20]
import ...
^
SyntaxError: invalid syntax
```

上のセルで解答を作成した後、以下のセルを実行し、実行結果が全て `True` になることを確認してください。

```
[21]: print(check_ACGTstr('AcCGTAGCacATcGgAaaTtGCacT') == True)
print(check_ACGTstr(':ACaacgta24FgtGH') == False)
print(check_ACGTstr('') == False)
```

NameError Traceback (most recent call last)

Input In [21], in <module>

```
----> 1 print(check_ACGTstr('AcCGTAGCacATcGgAaaTtGCacT') == True)
      2 print(check_ACGTstr(':ACaacgta24FgtGH') == False)
      3 print(check_ACGTstr('') == False)
```

NameError: name 'check_ACGTstr' is not defined

31.4 練習

文字列 `str1` を引数として取り、`str1` の中に「時刻を表す文字列」が含まれているかどうか調べて、含まれていればマッチしたときの `match` オブジェクトを、含まれない場合は `None` を返す関数 `check_timestr` を作成してください。ただし、「時刻を表す文字列」は次のような文字列とします。

1. 長さ 5 の `hh:mm` という文字列であり、12 時間表示で時間を表す。
2. 前半の 2 文字 `hh` は、`00, 01, ..., 11` のいずれかの文字列
3. 後半の 2 文字 `mm` は、`00, 01, ..., 59` のいずれかの文字列

以下のセルの ... のところを書き換えて解答してください。

```
[22]: import ...
def check_timestr(str1):
    ...
```

```
Input In [22]
import ...
      ^
SyntaxError: invalid syntax
```

上のセルで解答を作成した後、以下のセルを実行し、実行結果が全て True になることを確認してください。

```
[23]: print(check_timestr('10:23').group() == '10:23') # group() については後半に説明があります
      (オプション)
      print(check_timestr('time?1023') == None)
      print(check_timestr('time?11:23').group() == '11:23')
      print(check_timestr('12:3xx1;23ah23:23') == None)
```

```
-----
NameError                                Traceback (most recent call last)
Input In [23], in <module>
----> 1 print(check_timestr('10:23').group() == '10:23') # group() については後半に説明
      2 print(check_timestr('time?1023') == None)
      3 print(check_timestr('time?11:23').group() == '11:23')

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

31.5 練習

文字列 `str1` を引数として取り、`str1` の中に「IPv4 を表す文字列」が含まれているかどうか調べて、含まれていればマッチしたときの `match` オブジェクトを、含まれていなければ `None` を返す関数 `check_ipv4str` を作成してください。ただし、「IPv4 を表す文字列」は次のような文字列とします。

1. `aaa:bbb:ccc:ddd` という形式の長さ 15 の文字列
2. `aaa, bbb, ccc, ddd` はいずれも、`000, 001, …, 254, 255` のいずれかの文字列

以下のセルの … のところを書き換えて解答してください。

```
[24]: import ...
      def check_ipv4str(str1):
          ...
```

```
Input In [24]
import ...
      ^
SyntaxError: invalid syntax
```

上のセルで解答を作成した後、以下のセルを実行し、実行結果が全て True になることを確認してください。

```
[25]: print(check_ipv4str('IP=255:255:255:255').group() == '255:255:255:255')
      print(check_ipv4str('notIP=2x5:a5b:2c:255:14:444') == None)
      print(check_ipv4str('IP?=25:25:55:155') == None)
      print(check_ipv4str('IP?=255:255:255') == None)
```

```
-----
NameError                                Traceback (most recent call last)
Input In [25], in <module>
----> 1 print(check_ipv4str('IP=255:255:255:255').group() == '255:255:255:255')
      2 print(check_ipv4str('notIP=2x5:a5b:2c:255:14:444') == None)
```

(continues on next page)

(continued from previous page)

```
3 print(check_ipv4str('IP?=25:25:55:155') == None)
```

```
NameError: name 'check_ipv4str' is not defined
```

31.6 練習

文字列 `str1` を引数として取り、`str1` の中に「月と日を表す文字列」が含まれているかどうか調べて、含まれていればマッチしたときの `match` オブジェクトを、含まれていなければ `None` を返す関数 `check_monthdaystr` を作成してください。ただし、「月と日を表す文字列」は次のような文字列とします。

1. `mm/dd` という長さ 5 の文字列
2. `mm` は、`01`, `02`, ..., `12` のいずれかの文字列
3. `dd` は、`mm` が `01`, `03`, `05`, `07`, `08`, `10`, `12` ならば、`01`, `02`, ..., `31` のいずれかの文字列
4. `dd` は、`mm` が `04`, `06`, `09`, `11` ならば、`01`, `02`, ..., `30` のいずれかの文字列
5. `dd` は、`mm` が `02` ならば、`01`, `02`, ..., `29` のいずれかの文字列

以下のセルの ... のところを書き換えて解答してください。

```
[26]: import ...
def check_monthdaystr(str1):
    ...
```

```
Input In [26]
```

```
import ...
```

```
^
```

```
SyntaxError: invalid syntax
```

上のセルで解答を作成した後、以下のセルを実行し、実行結果が全て `True` になることを確認してください。

```
[27]: print(check_monthdaystr('year11/31month11/30day15hour/27minute/sec').group() == '11/30
↪')
print(check_monthdaystr('11/31') == None)
print(check_monthdaystr('x02f/2d5ax') == None)
print(check_monthdaystr('03/24').group() == '03/24')
```

```
-----
NameError                                Traceback (most recent call last)
Input In [27], in <module>
----> 1 print(check_monthdaystr('year11/31month11/30day15hour/27minute/sec').group())
↪== '11/30')
      2 print(check_monthdaystr('11/31') == None)
      3 print(check_monthdaystr('x02f/2d5ax') == None)

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


31.7 文字クラス

[abc] は a|b|c と同じ意味の正規表現です。この角括弧用いた表記は、**文字クラス**と呼ばれます。

```
[28]: match1 = re.search('[abc]', 'defabcxyz')
      print(match1)
      match1 = re.search('[3456]', '1234567890')
      print(match1)
      match1 = re.search('ha[sd]', 'He has an apple and they have pineapples.')
      print(match1)
```

```
<re.Match object; span=(3, 4), match='a'>
<re.Match object; span=(2, 3), match='3'>
<re.Match object; span=(3, 6), match='has'>
```

勿論、これまでの和や閉包と組み合わせて用いることができます。

```
[29]: match1 = re.search('[def][abc]', 'defabcxyz')
      print(match1)
      match1 = re.search('4[3456][3456]([3456]|[7890])', '1234567890')
      print(match1)
      match1 = re.search('6[789]*', '1234567890')
      print(match1)
      match1 = re.search('she ha[sd]|they ha(ve|d)', 'He has an apple and they have_
      ↪pineapples.', re.I)
      print(match1)
```

```
<re.Match object; span=(2, 4), match='fa'>
<re.Match object; span=(3, 7), match='4567'>
<re.Match object; span=(5, 9), match='6789'>
<re.Match object; span=(20, 29), match='they have'>
```

ただし、文字クラスの中で接続、和、閉包は無効化されます。たとえば、[a*] という正規表現は、a もしくは、* にマッチします。

```
[30]: match1 = re.search('[a*]', 'aaaaaa') # a 一文字にマッチ
      print(match1)
      match1 = re.search('[a*]', '*') # * 一文字にマッチ
      print(match1)
      match1 = re.search('a*', 'aaaaaa')
      print(match1)
      match1 = re.search('a*', '*') # 文字クラスでない場合、*にはマッチしない
      print(match1)
      match1 = re.search('[a|b]', 'defabcxyz') # a 一文字にマッチ
      print(match1)
      match1 = re.search('[a|b]', '|') # | 一文字にマッチ
      print(match1)
      match1 = re.search('a|b', '|') # 文字クラスでない場合、|にはマッチしない
      print(match1)
```

```
<re.Match object; span=(0, 1), match='a'>
<re.Match object; span=(0, 1), match='*'>
<re.Match object; span=(0, 6), match='aaaaaa'>
<re.Match object; span=(0, 0), match=''>
<re.Match object; span=(3, 4), match='a'>
<re.Match object; span=(0, 1), match='|'>
None
```

文字クラスでは一文字分の連続する和演算を表すことができますが、長さ 2 以上の文字列を表すことはで

きません。すなわち、`ab | cd` という正規表現を（1つの）文字クラスで表すことはできません。

また、`[abcdefg]` や `[gcdbeaf]` などは `[a-g]`、`[1234567]` や `[4271635]` などは `[1-7]` などとハイフン（-）を用いることで簡潔に表すことができます。たとえば、全てのアルファベットと数字を表す場合は、`[a-zA-Z0-9]` で表されます。

```
[31]: match1 = re.search('[a-c]', 'defabcxyz')
      print(match1)
      match1 = re.search('3[4-8]', '1234567890')
      print(match1)
      match1 = re.search(':[a-zA-Z0-9]*:', 'a1b2c3:d4e5f:6g7A8B:9C0D')
      print(match1)
```

```
<re.Match object; span=(3, 4), match='a'>
<re.Match object; span=(2, 4), match='34'>
<re.Match object; span=(6, 13), match=':d4e5f:'>
```

文字クラスの内側をキャレット (^) で始めると、**否定文字クラス** となり、キャレットの後ろに指定した文字以外の文字とマッチする正規表現となります。たとえば、`[^abc]` は a, b, c 以外の 1 文字とマッチする正規表現です。

```
[32]: match1 = re.search('[^abc]', 'abcdefxyz')
      print(match1)
      match1 = re.search('[^def]', 'defabcxyz')
      print(match1)
      match1 = re.search('[^1-7]', '1234567890')
      print(match1)
      match1 = re.search('ha[^sd]e', 'He has an apple and they have pineapples.')
      print(match1)
```

```
<re.Match object; span=(3, 4), match='d'>
<re.Match object; span=(3, 4), match='a'>
<re.Match object; span=(7, 8), match='8'>
<re.Match object; span=(25, 29), match='have'>
```

キャレットを先頭以外につけた場合は、単なる文字クラスになります。すなわち、キャレットにマッチするかどうかは判定されません。たとえば、`[d^ef]` は、d, ^, e, f のいずれかにマッチします。

```
[33]: match1 = re.search('[d^ef]', 'defabcxyz')
      print(match1)
      match1 = re.search('[d^ef]', 'a^bcdef') # キャレットにマッチする
      print(match1)
```

```
<re.Match object; span=(0, 1), match='d'>
<re.Match object; span=(1, 2), match='^'>
```

31.8 正規表現に関する基本的な関数

上で紹介した正規表現を利用してマッチする文字列が存在するかどうかを調べるだけでなく、マッチした文字列に対して色々な処理を加えることができます。以下では2つの基本的な関数を紹介します。

31.8.1 sub

sub は、正規表現 R にマッチする 文字列 A 中の全ての文字列を、指定した 文字列 B で置き換えることができます。

具体的には次のようにすると、

```
re.sub(正規表現 R, 置換する文字列 B, 元になる文字列 A)
```

R とマッチする A 中の全ての文字列を B と置き換えることができます。置き換えられた結果の文字列（新たに作られて）が返値となります。（もちろん、もとの文字列 A は変化しません。）

```
[34]: str1 = re.sub('[346]', 'x', '03-5454-68284') #3,4,6 を x に置き換える
print(str1)
str1 = re.sub('[.,;!?]', '', "He has three pets: a cat, a dog and a giraffe, doesn't he?") #句読点を削除する（空文字列に置き換える）
print(str1)
str1 = re.sub('\(a\)|あっとまーく|@', '@', 'accountname あっとまーく test.ecc.u-tokyo.ac.jp') # スпам回避の文字列を@に置き換える
print(str1) # \(&\) の意味については、下記の「正規表現のエスケープシーケンス」の節を参照してください
```

```
0x-5x5x-x828x
He has three pets a cat a dog and a giraffe doesn't he
accountname@test.ecc.u-tokyo.ac.jp
```

```
re.sub(r'[\t\n][\t\n]*', ' ', str1)
```

とすると、文字列 str1 の空白文字の並びがスペース 1 個に置き換わります。

ここで、`r'[\t\n][\t\n]*'` という正規表現は、空白かタブか改行の **1 回以上** の繰り返しのパターンを表します。つまり、`aa*` という形をした「1 回以上の a という文字列とマッチする正規表現」は `a+` という + を使った正規表現で置き換えることが可能です。この + は後で正式に紹介します。

```
[35]: re.sub(r'[\t\n][\t\n]*', ' ', 'Hello,\n    World!\tHow are you?')
```

```
[35]: 'Hello, World! How are you?'
```

以下では、HTML や XML のタグを消しています（空文字列に置き換えています）。

```
[36]: re.sub(r'<[^>]*>', '', '<body>\nClick <a href="a.href">this</a>\n</body>\n')
```

```
[36]: '\nClick this\n'
```

`r'<[^>]*>'` という正規表現は、< の後に > 以外の文字の繰り返しがあって最後に > が来るというパターンを表します。

31.8.2 re.split

`split` は、正規表現 `R` にマッチする文字列を区切り文字（デリミタ）として、文字列 `A` を分割します。分割された文字列がリストに格納されて返値となります。

具体的には次のように用います。

```
re.split(正規表現 R, 元になる文字列 A)
```

以下が典型例です。

```
re.split(r'^a-zA-Z[^a-zA-Z]*', 'Hello, World! How are you?')
```

`^[a-zA-Z][a-zA-Z]*` という正規表現は、英文字以外の文字が 1 回以上繰り返されている、というパターンを表します。この正規表現を Python の式の中で用いるときは、`r'^a-zA-Z[^a-zA-Z]*'` という構文を用います。先頭の `r` については、以下の説明を参照してください。

```
[37]: list1 = re.split(' ', "He has three pets a cat a dog and a giraffe doesn't he")
      print(list1)
      list2 = re.split(r'^a-zA-Z[^a-zA-Z]*', 'Hello, World! How are you?')
      print(list2)

['He', 'has', 'three', 'pets', 'a', 'cat', 'a', 'dog', 'and', 'a', 'giraffe', "doesn't", 'he']
['Hello', 'World', 'How', 'are', 'you', '']
```

この例のように、返されたリストに空文字列が含まれる場合がありますので、注意してください。

31.8.3 r を付ける理由

さて、以上のような正規表現は、`'hello*'` のように Python の文字列として `re.split` や `re.sub` などの関数に与えればよいのですが、以下のように文字列の前に `r` を付けることが推奨されます。

```
[38]: r'hello*'
```

```
[38]: 'hello*'
```

`r'hello*'` の場合は `r` を付けても付けなくても同じなのですが、以下のように `r` を付けるとエスケープすべき文字がエスケープシーケンスになった文字列が得られます。

```
[39]: r'[\t\n]+'
```

```
[39]: '[\t\n]+'
```

`\t` が `\\t` に変わったことでしょう。`\\` はバックスラッシュを表すエスケープシーケンスです。`\t` はタブという文字を表しますが、`\\t` はバックスラッシュと `t` という 2 文字から成る文字列です。この場合、正規表現を解釈する段階でバックスラッシュが処理されます。

特に `\` という文字そのものを正規表現に含めたいときは `\\` と書いた上で `r` を付けてください。

```
[40]: r'\\t t/'
```

```
[40]: '\\t t/'
```

この場合、文字列の中に `\` が 2 個含まれており、正規表現を解釈する段階で正しく処理されます。すなわち、`\` という文字そのものを表します。

31.9 練習

英語の文書が保存された `text-sample.txt` というファイルから読み込み、出現する単語のリストを返す関数 `get_engsentences` を作成してください。ただし、**重複する単語を削除してはいけません**が、空文字列は除きます。また、リストは返す前に中身を昇順に並べ替えてください。

以下のセルの ... のところを書き換えて解答してください。

```
[41]: import ...
def get_engsentences():
    ...

Input In [41]
import ...
      ^
SyntaxError: invalid syntax
```

上のセルで解答を作成した後、以下のセルを実行し、実行結果が `True` になることを確認してください。

```
[42]: list1 = get_engsentences()
print(len(list1) == 289)
print(list1[0] == 'a')
print(list1[100] == 'in')
print(list1[288] == 'would')
```

```
-----
NameError                                Traceback (most recent call last)
Input In [42], in <module>
----> 1 list1 = get_engsentences()
      2 print(len(list1) == 289)
      3 print(list1[0] == 'a')

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

31.10 練習

英語の文書が保存された `text-sample.txt` というファイルから読み込み、出現する単語のリストを返す関数 `get_engsentences2` を作成してください。ただし、空文字列は除きます。また、リストは返す前に中身を昇順に並べ替えてください。

以下のセルの ... のところを書き換えて解答してください。

```
[43]: import ...
def get_engsentences2():
    ...

Input In [43]
import ...
      ^
SyntaxError: invalid syntax
```

上のセルで解答を作成した後、以下のセルを実行し、実行結果が `True` になることを確認してください。

```
[44]: list1 = get_engsentences2()
print(len(list1) == 149)
print(list1[0] == 'a')
```

(continues on next page)

(continued from previous page)

```
print(list1[100] == 'proclaim')
print(list1[148] == 'would')
```

```
-----
NameError                                Traceback (most recent call last)
Input In [44], in <module>
----> 1 list1 = get_engsentences2()
      2 print(len(list1) == 149)
      3 print(list1[0] == 'a')

NameError: name 'get_engsentences2' is not defined
```

31.11 その他の反復演算

閉包以外の反復演算を説明します。

`a?` は、正規表現 `a` を高々1 回反復する文字列とマッチします。たとえば、`a(bc)?` は `a|abc` と同じ意味の正規表現です。

```
[45]: match1 = re.search('colou?r', 'colour')
      print(match1)
      match1 = re.search('colou?r', 'color')
      print(match1)

<re.Match object; span=(0, 6), match='colour'>
<re.Match object; span=(0, 5), match='color'>
```

`a+` は、正規表現 `a` を 1 回以上反復する文字列とマッチします。つまり、`a+` は `aa*` と同じ意味の正規表現です。

```
[46]: match1 = re.search('boo+', 'boooo!')
      print(match1)
      match1 = re.search('boo+', 'bo!')
      print(match1)
      match1 = re.search('a+', 'abcdef')
      print(match1)
      match1 = re.search('a+', 'aabbcc')
      print(match1)
      match1 = re.search('a+', 'cde')
      print(match1)
      match1 = re.search('[^a-zA-Z]+', 'abc12345efg67hi89j0k')
      print(match1)
      match1 = re.search('[a-zA-Z]+', 'abc12345efg67hi89j0k')
      print(match1)

<re.Match object; span=(0, 5), match='boooo'>
None
<re.Match object; span=(0, 1), match='a'>
<re.Match object; span=(0, 2), match='aa'>
None
<re.Match object; span=(3, 8), match='12345'>
<re.Match object; span=(0, 3), match='abc'>
```

上記の例を `*` を使う形に書き換えてみてください。

`a{x,y}` は正規表現 `a` を `x` 回以上かつ `y` 回以下繰り返す文字列とマッチします。

```
[47]: match1 = re.search('bo{3,5}', 'boooooo!')
print(match1)
match1 = re.search('bo{3,5}', 'boo!')
print(match1)
match1 = re.search('a{2,5}', 'bacaad')
print(match1)
match1 = re.search('[0-9]{1,3},[0-9]{3,3}', '1,298 円')
print(match1)
match1 = re.search('[0-9]{1,3},[0-9]{3,3}', '298 円')
print(match1)
```

```
<re.Match object; span=(0, 6), match='boooooo'>
None
<re.Match object; span=(3, 5), match='aa'>
<re.Match object; span=(0, 5), match='1,298'>
None
```

31.12 メタ文字

以下では、良く使うメタ文字（特殊文字）を紹介します。

。（ピリオド）は、あらゆる文字にマッチします。

```
[48]: match1 = re.search('.', 'Hello')
print(match1)
match1 = re.search('3.*9', '1234567890')
print(match1)
match1 = re.search('ha(.|..)', 'He has an apple and they have pineapples.')
print(match1)
```

```
<re.Match object; span=(0, 1), match='H'>
<re.Match object; span=(2, 9), match='3456789'>
<re.Match object; span=(3, 6), match='has'>
```

ただし、文字クラスの中で . を用いても、あらゆる文字とはマッチせず、* の場合と同様に、ピリオドとマッチします。

```
[49]: match1 = re.search('[.]', 'Hello')
print(match1)
match1 = re.search('[.]', '3.141592')
print(match1)
```

```
None
<re.Match object; span=(1, 2), match='.'>
```

\t はタブを表します。

```
[50]: match1 = re.search('b\t', 'a      b      c      d')
print(match1)
```

```
<re.Match object; span=(2, 4), match='b\t'>
```

\s は空白文字（スペース、タブ、改行など）を表します。

```
[51]: match1 = re.search('b\s', 'a      b      c      d')
print(match1)
match1 = re.search('a\s\s\s', 'a      b      c      d')
print(match1)
```

(continues on next page)

(continued from previous page)

```
match1 = re.search('b\s*', 'a      b      c      d')
print(match1)
```

```
<re.Match object; span=(2, 4), match='b\t'>
<re.Match object; span=(0, 4), match='a\t\u3000 '>
<re.Match object; span=(4, 7), match='b\t\u3000'>
```

\S は \s 以外の全ての文字を表します。

```
[52]: match1 = re.search('b\S', 'a      b      bc      d')
print(match1)
```

```
<re.Match object; span=(4, 6), match='bc'>
```

\w は [a-zA-Z0-9_] と同じ意味です。

```
[53]: match1 = re.search('\w\w', 'abcde')
print(match1)
match1 = re.search('b\w*g', 'abcdefgh')
```

```
<re.Match object; span=(0, 2), match='ab'>
```

\W は \w 以外の全ての文字を表します。すなわち、[^a-zA-Z0-9_] と同じ意味です。

```
[54]: match1 = re.search('g\W*', 'ab defg hi jklm no p')
print(match1)
match1 = re.search('\W\w*\W', 'ab defg hi jklm no p')
print(match1)
```

```
<re.Match object; span=(6, 9), match='g  ' >
<re.Match object; span=(2, 8), match=' defg ' >
```

\d は [0-9] と同じ意味です。

```
[55]: match1 = re.search('\d\d\d-\d\d\d\d', '153-8902')
print(match1)
match1 = re.search('\d*-\d*', '153-8902')
print(match1)
match1 = re.search('\d\d-\d\d\d\d-\d\d\d\d', '03-5454-6828')
print(match1)
match1 = re.search('\d*-\d*-\d*', '03-5454-6828')
print(match1)
```

```
<re.Match object; span=(0, 8), match='153-8902'>
<re.Match object; span=(0, 8), match='153-8902'>
<re.Match object; span=(0, 12), match='03-5454-6828'>
<re.Match object; span=(0, 12), match='03-5454-6828'>
```

\D は \d 以外の全ての文字を表します。すなわち、[^0-9] と同じ意味です。

```
[56]: match1 = re.search('\D*', 'He has 10 apples.')
print(match1)
```

```
<re.Match object; span=(0, 7), match='He has ' >
```

31.13 練習

文字列から数字列を切り出して、それを整数とみなして足し合せた結果を整数として返す関数 `sumnumbers` を定義してください。

```
[57]: import ...
      def sumnumbers(s):
          ...

Input In [57]
import ...
      ^
SyntaxError: invalid syntax
```

上のセルで解答を作成した後、以下のセルを実行し、実行結果が `True` になることを確認してください。

```
[58]: print(sumnumbers(' 2  33 45, 67.9') == 156)

-----
NameError                                Traceback (most recent call last)
Input In [58], in <module>
----> 1 print(sumnumbers(' 2  33 45, 67.9') == 156)

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

31.14 練習

文字列 `str1` を引数として取り、`str1` を構成する文字列が `A, C, G, T` の 4 種類の文字以外の文字を含むかどうか調べて、これら以外を含む場合は `False`、そうでない場合は `True` を返す関数 `check_ACGTstr` を作成してください。ただし、大文字と小文字は区別しません。また、空列の場合は `False` を返してください。

以下のセルの ... のところを書き換えて解答してください。

```
[59]: import ...
      def check_ACGTstr(str1):
          ...

Input In [59]
import ...
      ^
SyntaxError: invalid syntax
```

上のセルで解答を作成した後、以下のセルを実行し、実行結果が全て `True` になることを確認してください。

```
[60]: print(check_ACGTstr('AcCGTAGCacATcGgAaaTtGCacT') == True)
      print(check_ACGTstr(':ACaacgta24FgtGH') == False)
      print(check_ACGTstr('') == False)

-----
NameError                                Traceback (most recent call last)
Input In [60], in <module>
----> 1 print(check_ACGTstr('AcCGTAGCacATcGgAaaTtGCacT') == True)
      2 print(check_ACGTstr(':ACaacgta24FgtGH') == False)
      3 print(check_ACGTstr('') == False)

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


31.15 練習

文字列 `str1` を引数として取り、`str1` を構成する文字列が「日本の郵便番号」を表す文字列になっている場合は、`True` を返し、そうでない場合は `False` を返す関数 `check_postalcode` を作成してください。ただし、「日本の郵便番号」は `abc-defg` という形になっており、`a, b, c, d, e, d, f, g` はそれぞれ `0` から `9` までの値になっています。

以下のセルの ... のところを書き換えて解答してください。

```
[61]: import ...
def check_postalcode(str1):
    ...
```

```
Input In [61]
import ...
      ^
SyntaxError: invalid syntax
```

上のセルで解答を作成した後、以下のセルを実行し、実行結果が全て `True` になることを確認してください。

```
[62]: print(check_postalcode('113-8654') == True)
print(check_postalcode('119-110') == False)
print(check_postalcode('abc-defg') == False)
print(check_postalcode('〒153-0041') == False)
print(check_postalcode('113-86547') == False)
```

```
-----
NameError                                Traceback (most recent call last)
Input In [62], in <module>
----> 1 print(check_postalcode('113-8654') == True)
      2 print(check_postalcode('119-110') == False)
      3 print(check_postalcode('abc-defg') == False)

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

31.16 練習

文字列 `str1` を引数として取り、`str1` を構成する文字列が「本郷の内線番号」を表す文字列になっている場合は、`True` を返し、そうでない場合は `False` を返す関数 `check_extension` を作成してください。ただし、「本郷の内線番号」は `2abcd` という形になっており、`a, b, c, d` はそれぞれ `0` から `9` までの値になっています。

以下のセルの ... のところを書き換えて解答してください。

```
[63]: import ...
def check_extension(str1):
    ...
```

```
Input In [63]
import ...
      ^
SyntaxError: invalid syntax
```

上のセルで解答を作成した後、以下のセルを実行し、実行結果が全て `True` になることを確認してください。

```
[64]: print(check_extension('24115') == True)
print(check_extension('46858') == False)
print(check_extension('^^e2^^98^^8e46666') == False)
print(check_extension('467890') == False)
```

```
-----
NameError                                Traceback (most recent call last)
Input In [64], in <module>
----> 1 print(check_extension('24115') == True)
      2 print(check_extension('46858') == False)
      3 print(check_extension('^^e2^^98^^8e46666') == False)

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

31.17 正規表現のエスケープシーケンス

丸括弧 () や演算子 (|, *) など正規表現の中で特殊な役割を果たす記号のマッチを行いたい場合、文字列のエスケープシーケンスのように \ を前につけてやる必要があります。

```
[65]: match1 = re.search('03(5454)6666', '03(5454)6666') #電話番号。つけないと丸括弧として扱わ
れないのでマッチしない
print(match1)
match1 = re.search('03(5454)6666', '0354546666') # 括弧が含まれない文字列にマッチ
print(match1)
match1 = re.search('03\ (5454\ )6666', '03(5454)6666') # \ (と \ ) で左右の丸括弧として扱われ
るのでマッチする
print(match1)
match1 = re.search('3*4+5=17', '3*4+5=17') #計算式。*と+が演算子扱いされているのでマッチし
ない
print(match1)
match1 = re.search('3*4+5=17', '33345=17') #\ がないと、たとえば、このような文字列とマッチ
する
print(match1)
match1 = re.search('3\ *4\ +5=17', '3*4+5=17') #意図した文字列にマッチ
print(match1)
match1 = re.search('|ω・` `) チラ ', '|ω・` `) チラ ') #顔文字。 空列にマッチしてしまう
print(match1)
match1 = re.search('\|ω・` `) チラ ', '|ω・` `) チラ ') #意図した文字列にマッチ
print(match1)
```

```
None
<re.Match object; span=(0, 10), match='0354546666'>
<re.Match object; span=(0, 12), match='03(5454)6666'>
None
<re.Match object; span=(0, 8), match='33345=17'>
<re.Match object; span=(0, 8), match='3*4+5=17'>
<re.Match object; span=(0, 0), match=''>
<re.Match object; span=(0, 8), match='|ω・` `) チラ '>
```

特殊な意味を持つ記号は次の 14 個です。

. ^ \$ * + ? { } [] \ | ()

これらの特殊記号が含まれる場合（かつ意図したマッチの結果が得られない場合）には、エスケープシーケンスを使う（エスケープする）べき（可能性がある）ことも考慮に入れておいてください。

31.18 正規表現に関する関数とメソッド

以下では更に幾つかの関数とメソッドを紹介します。

31.18.1 findall

`findall` は、正規表現 `R` にマッチする 文字列 `A` 中の全ての文字列を、リストに格納して返します。

具体的には次のように実行します。

```
re.findall(正規表現 R, 文字列 A)
```

```
[66]: list1 = re.findall('had', 'James while John had had had had had had had had had had had
↳had a better effect on the teacher.')
#James, while John had had 'had', had had 'had had'; 'had had' had had a better effect on
↳the teacher.
print(list1) #全ての had を抜き出す
list1 = re.findall('p[^\.]*', 'Peter Piper picked a peck of pickled peppers.', re.I)
print(list1) # p で始まる全ての単語を取得する, 大文字小文字を区別しない

['had', 'had', 'had', 'had', 'had', 'had', 'had', 'had', 'had', 'had', 'had', 'had']
['Peter', 'Piper', 'picked', 'peck', 'pickled', 'peppers']
```

31.18.2 finditer

`finditer` は、正規表現 `R` にマッチする 文字列 `A` 中の全ての `match` オブジェクトを、特殊なリスト（のようなもの）に格納して返します。

具体的には次のように実行します。

```
re.finditer(正規表現 R, 文字列 A)
```

返値は特殊なリスト（のようなもの）であり、`for` 文の `in` の後ろに置いて使ってください。

```
[67]: print('1:正規表現 had の結果:')
iter1 = re.finditer('had', 'James while John had had had had had had had had had had had
↳had a better effect on the teacher.')
#James, while John had had 'had', had had 'had had'; 'had had' had had a better effect on
↳the teacher.
for match in iter1:
    print(match) #全ての had を抜き出す
print('2:正規表現 p[^\.]* の結果:')
iter1 = re.finditer('p[^\.]*', 'Peter Piper picked a peck of pickled peppers.', re.I)
for match in iter1:
    print(match) # p で始まる全ての単語を取得する, 大文字小文字を区別しない

1:正規表現 had の結果:
<re.Match object; span=(17, 20), match='had'>
<re.Match object; span=(21, 24), match='had'>
<re.Match object; span=(25, 28), match='had'>
<re.Match object; span=(29, 32), match='had'>
<re.Match object; span=(33, 36), match='had'>
<re.Match object; span=(37, 40), match='had'>
<re.Match object; span=(41, 44), match='had'>
<re.Match object; span=(45, 48), match='had'>
<re.Match object; span=(49, 52), match='had'>
<re.Match object; span=(53, 56), match='had'>
```

(continues on next page)

(continued from previous page)

```
<re.Match object; span=(57, 60), match='had'>
2:正規表現 p[^\.]* の結果:
<re.Match object; span=(0, 5), match='Peter'>
<re.Match object; span=(6, 11), match='Piper'>
<re.Match object; span=(12, 18), match='picked'>
<re.Match object; span=(21, 25), match='peck'>
<re.Match object; span=(29, 36), match='pickled'>
<re.Match object; span=(37, 44), match='peppers'>
```

31.18.3 group

match オブジェクトのメソッド `group` は、正規表現にマッチした文字列を（部分的に）取り出します。正規表現内に丸括弧を用いると、括弧内の正規表現とマッチした文字列を取得できるようになっています。なお、`group` によるこの操作を、括弧内の文字列を**キャプチャ**するといいます。

`i` 番目のキャプチャした値を取得するには次のようにします。`i = 0` の場合は、マッチした文字列全体を取得できます。

```
match オブジェクト.group(i)
```

```
[68]: import re
match1 = re.search('03-5454-(\d\d\d\d)', '03-5454-6666')
print(' マッチした文字列=', match1.group(0), ' キャプチャした文字列=', match1.group(1)) #_
↳内線番号の取得
match1 = re.search('([^\@]*)@([^\.]*)(\.[^\.]*)?\.u-tokyo\.ac\.jp', 'accountname@test.ecc.
↳u-tokyo.ac.jp') # \. はピリオドを表します
print(' マッチした文字列=', match1.group(0), ' キャプチャした文字列=', match1.group(1)) #_
↳アカウント名の取得
match1 = re.search('([^\@]*)@([^\.]*)(\.[^\.]*)?\.u-tokyo\.ac\.jp', 'accountname@test.u-
↳tokyo.ac.jp') # \. はピリオドを表します
print(' マッチした文字列=', match1.group(0), ' キャプチャした文字列=', match1.group(1)) #_
↳アカウント名の取得
match1 = re.search('href=\"([^\"]*)\"', '<a href=\"http://www.u-tokyo.ac.jp\" target=\"_
↳blank\">U-Tokyo</a>')# \"はダブルクォートを表します
print(' マッチした文字列=', match1.group(0), ' キャプチャした文字列=', match1.group(1)) #_
↳リンク先 URL の取得
```

```
マッチした文字列= 03-5454-6666   キャプチャした文字列= 6666
マッチした文字列= accountname@test.ecc.u-tokyo.ac.jp   キャプチャした文字列= accountname
マッチした文字列= accountname@test.u-tokyo.ac.jp   キャプチャした文字列= accountname
マッチした文字列= href="http://www.u-tokyo.ac.jp"   キャプチャした文字列= http://www.u-
↳tokyo.ac.jp
```

マッチに失敗した場合は、match オブジェクトが返らずに `None` が返るので、それを確かめずに `group` を使おうとするとエラーが出ますので注意してください。

```
[69]: match1 = re.search('03-5454-(\d\d\d\d)', '03-5454-666') #マッチしない文字列
print(' マッチした文字列=', match1.group(0), ' キャプチャした文字列=', match1.group(1))

-----
AttributeError                                Traceback (most recent call last)
Input In [69], in <module>
      1 match1 = re.search('03-5454-(\d\d\d\d)', '03-5454-666') #マッチしない文字列
----> 2 print(' マッチした文字列=', match1.group(0), ' キャプチャした文字列=', match1.
↳group(1))
```

(continues on next page)

(continued from previous page)

AttributeError: 'NoneType' object has no attribute 'group'

たとえば、if 文でエラーを回避します。

```
[70]: match1 = re.search('03-5454-(\d\d\d\d)', '03-5454-666')
      if match1 != None:
          print(' マッチした文字列=', match1.group(0), ' キャプチャした文字列=', match1.
            ↳group(1))
      else:
          print(' マッチしていません')
```

マッチしていません

31.19 練習

文字列 `str1` を引数として取り、`str1` を構成する文字列が A, C, G, T の 4 種類の文字以外の文字を含むかどうか調べて、これら以外を含む場合は `False` を、そうでない場合は `True` を返す関数 `check_ACGTstr` を作成してください。ただし、大文字と小文字は区別しません。また、空列の場合は `False` を返してください。

以下のセルの ... のところを書き換えて解答してください。

```
[71]: import ...
      def check_ACGTstr(str1):
          ...

      Input In [71]
            import ...
                  ^
      SyntaxError: invalid syntax
```

上のセルで解答を作成した後、以下のセルを実行し、実行結果が全て `True` になることを確認してください。

```
[72]: print(check_ACGTstr('AcCGTAGCacATcGgAaaTtGCacT') == True)
      print(check_ACGTstr(':ACaacgta24FgtGH') == False)
      print(check_ACGTstr('') == False)
```

```
-----
NameError                                Traceback (most recent call last)
Input In [72], in <module>
----> 1 print(check_ACGTstr('AcCGTAGCacATcGgAaaTtGCacT') == True)
      2 print(check_ACGTstr(':ACaacgta24FgtGH') == False)
      3 print(check_ACGTstr('') == False)

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

31.20 練習

xml ファイル B1S.xml は <http://www.natcorp.ox.ac.uk> から入手できるイギリス英語のコーパスのファイルです。

B1S.xml に含まれる w タグで囲まれる英単語をキー **key** に、その w タグの属性 **pos** の値を **key** の値とする辞書を返す関数 `get_pos` を作成してください。ただし、一般に w タグは、次のような形式で記述されます。

```
<w pos="PRON" ( > 記号以外の何らかの文字列) > (英単語) </w>
```

たとえば、以下のような具合です。

```
<w pos="VERB" hw="have" c5="VHI">have </w>
```

以下のセルの ... のところを書き換えて解答してください。

```
[73]: import ...
def get_pos():
    ...

Input In [73]
import ...
^
SyntaxError: invalid syntax
```

上のセルで解答を作成した後、以下のセルを実行し、実行結果が **True** になることを確認してください。

```
[74]: print(get_pos()['They '] == 'PRON')
print(get_pos()['know '] == 'VERB')
```

```
-----
NameError                                Traceback (most recent call last)
Input In [74], in <module>
----> 1 print(get_pos()['They '] == 'PRON')
      2 print(get_pos()['know '] == 'VERB')

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

31.21 練習の解答

```
[75]: import re
def check_monthstr(str1):
    reg_month = '((0(1|2|3|4|5|6|7|8|9))|10|11|12)' #
    #reg_month = '01/02/03/04/05/06/07/08/09/10/11/12' # としてもよい
    match1 = re.search(reg_month, str1) # 文字列を「含む」なので、(matchではなく) search
    を使う
    return match1
```

```
[76]: import re
def check_timestr(str1):
    reg_hour = '((0(0|1|2|3|4|5|6|7|8|9))|10|11)' # 「時」部分の正規表現
    #reg_hour = '((0[0-9]|10|11)' # 文字クラスを使ってと表すこともできます (文字クラスは後で
    学習します
    reg_min = '((0|2|3|4|5)(0|1|2|3|4|5|6|7|8|9))' # 「分」部分の正規表現
    #reg_min = '([0-5][0-9])' # 文字クラスを使ってと表すこともできます (文字クラスは後で学習
    します
    (continues on next page)
```

(continued from previous page)

```

reg_time = reg_hour + ':' + reg_min # 時部分と分部分を、「:」を挟んで結合した新しい正規表現
#print(reg_time)
match1 = re.search(reg3, str1) # 文字列を「含む」なので、(matchではなく) searchを使う
return match1

```

```

[77]: import re
def check_ipv4str(str1):
    reg_0to9 = '(0|1|2|3|4|5|6|7|8|9)' # 0から9の数を表す正規表現
    #reg_0to9 = '[0-9]' # 文字クラスを使ってと表すこともできます (文字クラスは後で学習します)
    reg_0_1 = '(0|1)' + reg_0to9 + reg_0to9 # 先頭の文字が0もしくは1だったときの正規表現 (000から199まで)
    reg_20_24 = '2(0|1|2|3|4)' + reg_0to9 # 先頭が20,21,22,23,24だったときの正規表現 (200から249まで)
    #reg_20_24 = '2[0-4]' + reg_0to9 # 文字クラスを使ってと表すこともできます
    reg_25 = '25(0|1|2|3|4|5)' # 先頭が25だったときの正規表現 (250から255まで)
    #reg_25 = '25[0-5]' # 文字クラスを使ってと表すこともできます
    reg_000_255 = '(' + reg_0_1 + '|' + reg_20_24 + '|' + reg_25 + ')' # aaa (000から255)を表す正規表現
    #print(reg_000_255)
    reg_ip = reg_000_255 + ':' + reg_000_255 + ':' + reg_000_255 + ':' + reg_000_255 # aaa:bbb:ccc:dddを表す正規表現
    #print(reg_ip)
    match1 = re.search(reg_ip, str1) # 文字列を「含む」なので、(matchではなく) searchを使う
    return match1

```

```

[78]: import re
def check_monthdaystr(str1):
    reg_month_31 = '(01|03|05|07|08|10|12)' # ddが01から31になるmm
    reg_month_30 = '(04|06|09|11)' # ddが01から30になるmm
    reg_1to9 = '(1|2|3|4|5|6|7|8|9)' # [1-9]でもよい
    reg_0to9 = '(0|1|2|3|4|5|6|7|8|9)' # [0-9]でもよい
    reg_day_01to09 = '(0' + reg_1to9 + ')' # ddが01から09になる場合
    reg_day_10to19 = '(1' + reg_0to9 + ')' # ddが10から19になる場合
    reg_day_20to29 = '(2' + reg_0to9 + ')' # ddが20から29になる場合
    reg_day_01to29 = reg_day_01to09 + '|' + reg_day_10to19 + '|' + reg_day_20to29 # ddが01から29になる場合
    reg_day_01to30 = reg_day_01to29 + '|' + '30' # ddが01から30になる場合
    reg_day_01to31 = reg_day_01to30 + '|' + '31' # ddが01から31になる場合
    reg_monthday_31 = reg_month_31 + '/' + reg_day_01to31 + ')' # mmとddを組み合わせる (01-31の場合)
    reg_monthday_30 = reg_month_30 + '/' + reg_day_01to30 + ')' # mmとddを組み合わせる (01-30の場合)
    reg_monthday_29 = '02/' + reg_day_01to29 + ')' # mmとddを組み合わせる (01-29の場合はmmは02のみ)
    # 文字列を「含む」なので、(matchではなく) searchを使う
    match1 = re.search(reg_monthday_31, str1) # 問題文の条件3を満たす文字列とマッチするかどうか
    if match1 != None:
        return match1
    match1 = re.search(reg_monthday_30, str1) # 問題文の条件4を満たす文字列とマッチするかどうか
    if match1 != None:
        return match1
    match1 = re.search(reg_monthday_29, str1) # 問題文の条件5を満たす文字列とマッチするかどうか

```

(continues on next page)

(continued from previous page)

```
return match1
```

```
[79]: import re
def sumnumbers(s):
    numbers = re.split('[^0-9]+', s)
    numbers.remove('')
    n = 0
    for number in numbers:
        n += int(number)
    return n
```

```
[80]: import re
def get_engsentences():
    word_list = [] # 結果を格納するリスト
    with open('text-sample.txt', 'r') as f:
        file_str = f.read() # ファイルの中身を文字列に格納
    str_list = re.split(r'[a-zA-Z][a-zA-Z]*', file_str) # 文字列を単語に区切る
    for word in str_list: # re.split(r'[a-zA-Z][a-zA-Z]*', f.read()) は、ファイル全体の
        # 文字列を単語に区切ります。
        # for word in re.split(r'[a-zA-Z][a-zA-Z]*', f.read()): # と一行にまとめてもいい
        if word != '': # 空文字列を除く
            word = word.lower() # 単語（文字列）の中の大文字を小文字に変換します
            word_list.append(word) # リストに追加
            # word_list.append(word.lower()) でも大丈夫
    word_list.sort() # sort メソッドは破壊的
    return word_list
```

```
[81]: import re
def get_engsentences2():
    word_dict = {} # 重複する単語を削除する為に辞書を使ってみる
    with open('text-sample.txt', 'r') as f:
        file_str = f.read() # ファイルの中身を文字列に格納
    str_list = re.split(r'[a-zA-Z][a-zA-Z]*', file_str) # 文字列を単語に区切る
    for word in str_list: # re.split(r'[a-zA-Z][a-zA-Z]*', f.read()) は、ファイル全体の
        # 文字列を単語に区切ります。
        if word != '': # 空文字列を除く
            word = word.lower() # 単語（文字列）の中の大文字を小文字に変換します
            word_dict[word] = 'anything good' # word という単語があったことを辞書に記録する
            # (word に対応する値は何でもよい)
            # word_dict[word.lower()] = 'anything good' でも大丈夫
    word_list = [] # 結果を格納するリスト
    for word in word_dict:
        word_list.append(word)
    word_list.sort()
    return word_list
```

```
[82]: import re
def check_ACGTstr(str1):
    reg_ACGT = '(A|C|G|T)+' # A, C, G, T を表す正規表現 # +→* だと空文字列がマッチしてしまう
    # reg_ACGT = '(A|C|G|T)(A|C|G|T)*' # A, C, G, T を表す正規表現
    # reg_ACGT = '[ACGT]+' # A, C, G, T を表す正規表現
    match1 = re.search(reg_ACGT, str1, re.I) # re.I を入れて、大文字と小文字を区別しない
    if match1 != None and str1 == match1.group(): # str1 全体とマッチした文字列が等しいか
        # チェック
        return True
```

(continues on next page)

(continued from previous page)

```

    return False
#別解
def check_ACGTstr(str1):
    # reg_ACGT = '(A|C|G|T)+' # A,C,G,Tを表す正規表現 # +→* だと空文字列がマッチしてしまう
    # #reg_ACGT = '(A|C|G|T)(A|C|G|T)*' # A,C,G,Tを表す正規表現
    # #reg_ACGT = '[ACGT]+' # A,C,G,Tを表す正規表現
    # match1 = re.search(reg_ACGT, str1, re.I) # re.I を入れて、大文字と小文字を区別しない
    # if match1 != None: # str1 全体とマッチした文字列が等しいかチェック
    #     return True
    # return False

```

```

[83]: import re
def check_postalcode(str1):
    reg1 = '[0-9]{3,3}-[0-9]{4,4}'
    #reg1 = '\d{3,3}-\d{4,4}' #でも可
    #reg1 = '[0-9][0-9][0-9]-[0-9][0-9][0-9][0-9]' #でも可
    match1 = re.match(reg1, str1)
    if match1 == None:
        return False
    if match1.group() == str1:
        return True
    return False
#別解
def check_postalcode(str1):
    # reg1 = '[0-9]{3,3}-[0-9]{4,4}$' #ドル記号を使って行末からマッチを調べる
    # #reg1 = '\d{3,3}-\d{4,4}$' #でも可
    # #reg1 = '[0-9][0-9][0-9]-[0-9][0-9][0-9][0-9]$' #でも可
    # match1 = re.match(reg1, str1)
    # if match1 == None:
    #     return False
    # return True

```

```

[84]: import re
def check_extension(str1):
    reg1 = '2[0-9]{4,4}'
    # reg1 = '2\d{4,4}' #でも可
    # reg1 = '2[0-9][0-9][0-9][0-9]' #でも可
    match1 = re.match(reg1, str1)
    if match1 == None:
        return False
    if match1.group() == str1:
        return True
    return False
#別解
def check_extension(str1):
    # reg1 = '2[0-9]{4,4}$'
    # #reg1 = '2\d{4,4}$' #でも可
    # #reg1 = '2[0-9][0-9][0-9][0-9]$' #でも可
    # match1 = re.match(reg1, str1)
    # if match1 == None:
    #     return False
    # return True

```

```

[85]: import re
def get_pos():

```

(continues on next page)

(continued from previous page)

```
str_file = 'B1S.xml'
with open(str_file, 'r', encoding='utf-8') as f:
    str_script = f.read() # ファイルの中身を 1つの文字列に格納する
    #print(str_script)
itr1 = re.finditer("<w[^>]*pos=\"([^\"]*)\"[^\>]*>([^\<]*)</w>", str_script) # 正規
表現を使って w タグ周辺の文字列をマッチ
dic1 = {} # 辞書初期化
for m1 in itr1:
    #print(m1)
    #print(m1.group(1), m1.group(2))
    dic1[m1.group(2)] = m1.group(1) # group を使ってマッチした文字列をキャプチャする
return dic1
```

- != 1/1-3#様々な条件
- # 1/1-1#コメント
- % 1/1-1#簡単な算術計算
- > 1/1-3#様々な条件
- >= 1/1-3#様々な条件
- < 1/1-3#様々な条件
- <= 1/1-3#様々な条件
- * 1/1-1#簡単な算術計算, 2/2-1#文字列の連結, 2/2-2#リストと演算子, 3/3-3#▲可変長引数, 3/3-3#▲可変長引数, 5/5-1#from, 5/5-2#自作モジュールの使い方
- ** 1/1-1#簡単な算術計算, 3/3-3#▲辞書型の可変長引数
- + 1/1-1#簡単な算術計算, 1/1-1#単項の + と -, 2/2-1#文字列の連結, 2/2-2#リストと演算子
- += 1/1-2#累積代入文
- - 1/1-1#簡単な算術計算, 1/1-1#単項の + と -
- -= 1/1-2#累積代入文
- . appendix/5-re#メタ文字
- / 1/1-1#簡単な算術計算
- // 1/1-1#簡単な算術計算
- 2 項演算子 1/1-1#単項の + と -
- 3 項演算子 2/2-3#▲ 3 項演算子 (条件式)
- = 1/1-2#変数, 1/1-2#代入文
- == 1/1-3#様々な条件
- CSV appendix/4-csv#CSV 形式とは
- CSV ファイル 7/7-1#CSV ファイルからのデータフレームの作成
- CSV ライター appendix/4-csv#CSV ファイルの書き込み
- CSV リーダ appendix/4-csv#CSV ファイルの読み込み

- DataFrame 7/7-1#シリーズとデータフレーム
- False 1/1-3#真理値を返す関数
- KMeans 7/7-2#教師なし学習・クラスタリングの例
- LinearRegression 7/7-2#教師あり学習・回帰の例
- LogisticRegression 7/7-2#教師あり学習・分類の例
- Matplotlib [appendix/5-matplotlib#▲ Matplotlib ライブラリ](#)
- None 1/1-2#関数の定義と返値, 1/1-3#None, 3/3-3#返値
- NumPy 5/5-3#5-3. NumPy ライブラリ, [appendix/5-matplotlib#線グラフ](#)
- PCA 7/7-2#教師なし学習・次元削減の例
- PEP8 1/1-4#コーディングスタイル
- Python スクリプト [appendix/5-command#▲ Python スクリプトとコマンドライン実行](#)
- Series 7/7-1#シリーズとデータフレーム
- StopIteration 4/4-2#next, 6/6-3#特殊メソッド
- True 1/1-3#真理値を返す関数
- \ 2/2-3#▲複数行にまたがる条件式
- \D [appendix/5-re#メタ文字](#)
- \S [appendix/5-re#メタ文字](#)
- \W [appendix/5-re#メタ文字](#)
- \d [appendix/5-re#メタ文字](#)
- \s [appendix/5-re#メタ文字](#)
- \t [appendix/5-re#メタ文字](#)
- \w [appendix/5-re#メタ文字](#)
- __enter__ 6/6-3#▲ with 文への対応
- __exit__ 6/6-3#▲ with 文への対応
- __init__ 6/6-3#初期化と属性
- __iter__ 6/6-3#特殊メソッド
- __name__ [appendix/5-command#モジュールのコマンドライン実行](#)
- __next__ 6/6-3#特殊メソッド
- a* [appendix/5-re#正規表現の基本](#)
- a+ [appendix/5-re#その他の反復演算](#)
- a? [appendix/5-re#その他の反復演算](#)
- add [appendix/2-set#add](#)
- and 1/1-3#様々な条件
- append 2/2-2#リストに要素を追加する, 7/7-1#行の追加と削除
- argument 3/3-3#引数
- as 4/4-1#ファイルに対する with 文, 5/5-1#as, 5/5-2#自作モジュールの使い方
- ascending 7/7-1#データの並び替え
- assert 文 1/1-4#assert 文
- assign 7/7-1#列の追加と削除

- assignment 1/1-2#代入文
- assignment statement 1/1-2#代入文
- augmented assignment statement 1/1-2#累積代入文
- axis 7/7-1#列の追加と削除
- a{x,y} *appendix/5-re*#その他の反復演算
- bar *appendix/5-matplotlib*#棒グラフ
- bokeh *appendix/5-bokeh*#▲ Bokeh ライブラリ
- bokeh.models.ColumnDataSource *appendix/5-bokeh*#ヒートマップ
- bokeh.models.LinearColorMapper *appendix/5-bokeh*#ヒートマップ
- bokeh.plotting *appendix/5-bokeh*#線グラフ
- bokeh.plotting.figure *appendix/5-bokeh*#線グラフ
- bokeh.plotting.output_file *appendix/5-bokeh*#グラフのファイル出力
- bokeh.plotting.output_notebook *appendix/5-bokeh*#線グラフ
- bokeh.plotting.reset_output *appendix/5-bokeh*#グラフのファイル出力
- bokeh.plotting.show *appendix/5-bokeh*#線グラフ
- break 文 3/3-2#break 文, 3/3-2#break 文
- capitalize 2/2-1#大文字・小文字
- chr 3/3-2#for 文による繰り返し
- circle *appendix/5-bokeh*#線グラフ
- clear 3/3-1#▲全てのキーと値の削除
- clear *appendix/2-set*#clear
- close 4/4-1#ファイルのクローズ
- complex *appendix/4-csv*#CSV ファイルの読み込み
- concat 7/7-1#▲データの連結
- continue 文 3/3-2#continue 文
- copy 2/2-2#▲ copy, 3/3-1#▲辞書を複製する
- count 2/2-1#数え上げ, 2/2-2#指定した要素のインデックス取得と数えあげ
- cross *appendix/5-bokeh*#線グラフ
- csv *appendix/4-csv*#CSV ファイルの読み込み, *appendix/4-csv*#CSV ファイルの書き込み
- csv.reader *appendix/4-csv*#CSV ファイルの読み込み
- csv.writer *appendix/4-csv*#CSV ファイルの書き込み
- def 1/1-2#関数の定義と返値, 3/3-3#関数の定義
- del 2/2-2#▲リスト要素を削除する, 3/3-1#3-1. 辞書 (dictionary), 7/7-1#列の追加と削除
- describe 7/7-1#データの統計量
- difference *appendix/2-set*#union, intersection, difference
- discard *appendix/2-set*#discard
- drop 7/7-1#列の追加と削除, 7/7-1#行の追加と削除
- elif 2/2-3#if … elif … else による条件分岐, 2/2-3#if … elif … else による条件分岐, 2/2-3#if … elif … else における条件の評価

- `else` 1/1-3#*if* 文による条件分岐, 2/2-3#*if* … `else` による条件分岐, 2/2-3#*if* … `elif` … `else` による条件分岐, 2/2-3#*if* … `elif` … `else` における条件の評価, 3/3-2#▲*for* 文と *while* 文における `else`
- `encoding` *appendix*/4-*csv*#東京の7月の気温
- `enumerate` 3/3-2#*enumerate*, 4/4-2#イテレータを返す *enumerate*
- `extend` 2/2-2#▲リストにリストの要素を追加する
- `filter` 6/6-2#*filter*
- `find` 2/2-1#検索
- `findall` *appendix*/5-*re*#*findall*
- `finditer` *appendix*/5-*re*#*finditer*
- `fit` 7/7-2#教師あり学習・分類の例
- `flatten` 5/5-3#多次元配列
- `float` 1/1-1#整数と実数の間の変換, 2/2-1#2-1. 文字列 (*string*), *appendix*/4-*csv*#CSV ファイルの読み込み
- `for` 3/3-2#*for* 文による繰り返し
- `for` 文 2/2-2#*for* 文による繰り返しとリスト・タプル, 3/3-2#*for* 文による繰り返し
- `from` 5/5-1#*from*, 5/5-2#自作モジュールの使い方
- `get` 3/3-1#キーを指定して値を得るメソッド
- `global` 3/3-3#▲ *global* 宣言
- `grid` *appendix*/5-*matplotlib*#線グラフ
- `group` *appendix*/5-*re*#*group*
- `groupby` 7/7-1#▲データのグループ化
- `head` 7/7-1#CSV ファイルからのデータフレームの作成
- `hist` *appendix*/5-*matplotlib*#ヒストグラム
- `if` 1/1-3#*if* 文による条件分岐, 2/2-3#2-3. 条件分岐, 2/2-3#*if* … `else` による条件分岐, 2/2-3#*if* … `elif` … `else` による条件分岐, 2/2-3#*if* … `elif` … `else` における条件の評価
- `if` 文 1/1-3#*if* 文による条件分岐
- `iloc` 7/7-1#*iloc* と *loc*
- `import` 1/1-1#数学関数 (モジュールのインポート), 5/5-1#モジュールのインポート, 5/5-2#自作モジュールの使い方
- `in` 2/2-1#文字列の検索, 2/2-2#リストと演算子, 3/3-1#3-1. 辞書 (*dictionary*), 3/3-2#*for* 文による繰り返し, 3/3-2#*in*, 3/3-2#*in*
- `in-place` 2/2-2#破壊的 (インプレース) な操作と非破壊的な生成
- `index` 2/2-1#検索, 2/2-2#指定した要素のインデックス取得と数えあげ, 7/7-1#CSV ファイルからのデータフレームの作成
- `inplace` 7/7-1#データの並び替え
- `insert` 2/2-2#▲リストに要素を挿入する
- `int` 1/1-1#整数と実数の間の変換, 2/2-1#2-1. 文字列 (*string*), 6/6-1#練習, *appendix*/4-*csv*#CSV ファイルの読み込み
- `intersection` *appendix*/2-*set*#*union*, *intersection*, *difference*
- `is` 2/2-2#▲オブジェクトの等価性と同一性, 4/4-2#*iter*
- `is not` 2/2-2#▲オブジェクトの等価性と同一性
- `items` 3/3-1#キーと値の一覧を得る, 3/3-2#*for* 文による繰り返しと辞書

- `iter` 4/4-2#`iter`
- `join` 2/2-2#リストと文字列の相互変換
- `key` 3/3-1#3-1. 辞書 (*dictionary*)
- `key` 6/6-2#`max`, 6/6-2#`sorted`
- `keys` 3/3-1#キーの一覧を得る, 3/3-1#キーの一覧を得る, 3/3-2#`for` 文による繰り返しと辞書
- `lambda` 6/6-2#ラムダ式
- `legend` *appendix/5-matplotlib*#線グラフ
- `len` 2/2-1#2-1. 文字列 (*string*), 2/2-2#リストの要素数, 3/3-1#3-1. 辞書 (*dictionary*)
- `line` *appendix/5-bokeh*#線グラフ
- `list` 2/2-2#2-2. リスト (*list*), 2/2-2#タプル (*tuple*)
- `loc` 7/7-1#`iloc` と `loc`
- `lower` 2/2-1#大文字・小文字
- `map` 6/6-2#`map`
- `match` *appendix/5-re*#正規表現の基本
- `match` オブジェクト *appendix/5-re*#正規表現の基本
- `math` 1/1-1#数学関数 (モジュールのインポート)
- `math.cos` 1/1-1#数学関数 (モジュールのインポート)
- `math.pi` 1/1-1#数学関数 (モジュールのインポート)
- `math.sin` 1/1-1#数学関数 (モジュールのインポート)
- `math.sqrt` 1/1-1#数学関数 (モジュールのインポート)
- `matplotlib` *appendix/3-visualization*#`matplotlib`
- `max` 2/2-2#`max` と `min`, 5/5-3#`sum`, `max`, `min`, `mean`, 6/6-2#`max`
- `mean` 5/5-3#`sum`, `max`, `min`, `mean`
- `merge` 7/7-1#▲データの結合
- `min` 2/2-2#`max` と `min`, 5/5-3#`sum`, `max`, `min`, `mean`
- `next` 4/4-2#`next`
- `not` 1/1-3#様々な条件
- `not in` 2/2-1#文字列の検索, 2/2-2#リストと演算子, 3/3-2#`in`
- `numpy` 5/5-3#5-3. NumPy ライブラリ
- `numpy.arange` 5/5-3#`arange`
- `numpy.array` 5/5-3#配列の構築
- `numpy.bool_` 5/5-3#要素型
- `numpy.complex128` 5/5-3#要素型
- `numpy.dot` 5/5-3#`dot`
- `numpy.float64` 5/5-3#要素型
- `numpy.histogram` *appendix/5-bokeh*#ヒストグラム
- `numpy.identity` 5/5-3#▲線形代数の演算
- `numpy.int32` 5/5-3#要素型
- `numpy.linalg` 5/5-3#▲線形代数の演算

- `numpy.linalg.norm` 5/5-3#▲線形代数の演算
- `numpy.linspace` 5/5-3#`linspace`
- `numpy.loadtxt` 5/5-3#配列の保存と復元
- `numpy.matmul` 5/5-3#▲線形代数の演算
- `numpy.ndarray` 5/5-3#配列の構築
- `numpy.ones` 5/5-3#`zeros` と `ones`
- `numpy.random.binomial` 5/5-3#`random.rand`
- `numpy.random.poisson` 5/5-3#`random.rand`
- `numpy.random.rand` 5/5-3#`random.rand`
- `numpy.random.randn` 5/5-3#`random.rand`
- `numpy.savetxt` 5/5-3#配列の保存と復元
- `numpy.sort` 5/5-3#`sort`
- `numpy.sqrt` 5/5-3#ユニバーサル関数
- `numpy.zeros` 5/5-3#`zeros` と `ones`
- `on` 7/7-1#▲データの結合
- `open` 4/4-1#ファイルのオープン
- `or` 1/1-3#様々な条件
- `ord` 3/3-2#`for` 文による繰り返し
- `os.chdir` 4/4-3#カレントワーキングディレクトリ
- `pandas` 7/7-1#7-1. `pandas` ライブラリ
- `parameter` 3/3-3#引数
- `pass` 文 3/3-2#`pass` 文
- `plot` `appendix/5-matplotlib`#線グラフ
- `pop` 2/2-2#▲リストからインデックスで指定した要素を削除する, 3/3-1#▲キーを指定した削除
- `pop` `appendix/2-set`#`pop`
- `predict` 7/7-2#教師あり学習・分類の例
- `print` 1/1-2#`print`
- `quad` `appendix/5-bokeh`#ヒストグラム
- `raise` 6/6-3#特殊メソッド
- `range` 3/3-2#`range`, 3/3-2#`range` とリスト, 4/4-2#イテラブル
- `ravel` 5/5-3#多次元配列
- `re.I` `appendix/5-re`#正規表現の基本
- `re.IGNORECASE` `appendix/5-re`#正規表現の基本
- `read` 4/4-1#ファイル全体の読み込み
- `read_csv` 7/7-1#CSV ファイルからのデータフレームの作成
- `readline` 4/4-1#行の読み込み
- `remove` 2/2-2#▲リストから要素を削除する
- `remove` `appendix/2-set`#`remove`
- `replace` 2/2-1#置換

- `reshape` 5/5-3#多次元配列
- `return` 1/1-2#関数の定義と返値, 3/3-3#返値
- `return` 文 1/1-2#関数の定義と返値, 3/3-2#制御構造と `return` 文
- `reverse` 2/2-2#▲リストの要素を逆順にする, 6/6-2#`sorted`
- `savefig` `appendix/5-matplotlib`#グラフの画像ファイル出力
- `scatter` `appendix/5-bokeh`#散布図, `appendix/5-matplotlib`#散布図
- `scikit-learn` 7/7-2#7-2. `scikit-learn` ライブラリ
- `search` `appendix/5-re`#正規表現の基本
- `set` `appendix/2-set`#▲セット (`set`)
- `setdefault` 3/3-1#▲キーがない場合に登録を行う
- `shebang` `appendix/5-command`#`shebang`
- `sort` 2/2-2#並び替え (`sort` メソッド) , 5/5-3#`sort`
- `sort_index` 7/7-1#データの並び替え
- `sort_values` 7/7-1#データの並び替え
- `sorted` 2/2-2#並び替え (`sorted` 組み込み関数) , 6/6-2#`sorted`
- `split` 2/2-2#リストと文字列の相互変換, `appendix/5-re`#`re.split`
- `str` 2/2-1#2-1. 文字列 (`string`), 2/2-1#2-1. 文字列 (`string`)
- `sub` `appendix/5-re`#`sub`
- `sum` 2/2-2#`sum`, 5/5-3#`sum`, `max`, `min`, `mean`, 6/6-1#リスト内包表記
- `super` 6/6-3#継承
- `title` `appendix/5-matplotlib`#線グラフ
- `transform` 7/7-2#教師なし学習・次元削減の例
- `tuple` 2/2-2#タプル (`tuple`)
- `type` 2/2-1#2-1. 文字列 (`string`)
- `union` `appendix/2-set`#`union`, `intersection`, `difference`
- `upper` 2/2-1#大文字・小文字
- `value` 3/3-1#3-1. 辞書 (`dictionary`)
- `values` 3/3-1#値の一覧を得る, 3/3-2#`for` 文による繰り返しと辞書
- `vbar` `appendix/5-bokeh`#棒グラフ
- `while` 3/3-2#`while` 文による繰り返し
- `while` 文 3/3-2#`while` 文による繰り返し
- `with` 4/4-1#ファイルに対する `with` 文
- `with` 文 6/6-3#▲`with` 文への対応
- `write` 4/4-1#ファイルへの書き込み
- `xlabel` `appendix/5-matplotlib`#線グラフ
- `ylabel` `appendix/5-matplotlib`#線グラフ
- 値 3/3-1#3-1. 辞書 (`dictionary`)
- 余り 1/1-1#簡単な算術計算
- イテラブル 4/4-2#イテラブル, 6/6-2#リストからイテラブルへ

- イテレータ 4/4-2#next, 6/6-1#▲ジェネレータ式, 6/6-2#map
- 入れ子 1/1-2#関数の定義と返値, 2/2-3#if … else による条件分岐, 3/3-2#for 文の入れ子, 3/3-2#for 文の入れ子, 6/6-1#内包表記の入れ子
- 印字 1/1-2#print
- インスタンス 6/6-3#クラス定義
- インデックス 2/2-1#文字列とインデックス, 7/7-1#シリーズとデータフレーム
- インデント 1/1-2#関数の定義と返値, 2/2-3#インデントによる構文
- インプレース 2/2-2#破壊的（インプレース）な操作と非破壊的な生成
- インポート 1/1-1#数学関数（モジュールのインポート）, 5/5-1#モジュールのインポート
- エスケープシーケンス 2/2-1#▲エスケープシーケンス, 4/4-1#ファイルへの書き込み
- エラー 1/1-1#エラー
- 大文字 2/2-1#大文字・小文字
- オブジェクト 1/1-3#オブジェクト, 6/6-3#クラス定義
- オブジェクト指向プログラミング 6/6-3#クラス定義
- オブジェクトの同一性 2/2-2#▲オブジェクトの等価性と同一性
- オブジェクトの等価性 2/2-2#▲オブジェクトの等価性と同一性
- 親クラス 6/6-3#継承
- オーダー 3/3-2#for 文の計算量
- オープン 4/4-1#ファイルのオープン
- 返値 1/1-2#関数の定義と返値, 3/3-3#返値
- 書き込みモード 4/4-1#ファイルへの書き込み
- 掛け算 1/1-1#簡単な算術計算
- 数え上げ 2/2-1#数え上げ
- 型 2/2-1#2-1. 文字列 (string)
- 形 5/5-3#多次元配列
- 括弧 1/1-1#演算子の優先順位と括弧
- 可変長引数 3/3-3#▲可変長引数
- 仮引数 1/1-2#関数の定義と返値, 3/3-3#引数
- カレントディレクトリ 4/4-3#カレントワーキングディレクトリ
- カレントワーキングディレクトリ 4/4-3#カレントワーキングディレクトリ
- 関数 1/1-2#関数の定義と返値, 3/3-3#関数の定義
- 関数定義 1/1-2#関数の定義と返値, 1/1-2#関数の定義と返値, 3/3-3#関数の定義
- 機械学習 7/7-2#機械学習について
- キャプチャ *appendix/5-re#group*
- 教師あり学習 7/7-2#教師あり学習
- 教師なし学習 7/7-2#教師なし学習
- キー 3/3-1#3-1. 辞書 (dictionary)
- キーワード引数 3/3-3#▲キーワード引数
- 偽 1/1-3#真理値を返す関数
- 行番号 1/1-2#関数の定義と返値

- 空行 1/1-2#コメントと空行
- 空タプル 2/2-2#タプル (tuple)
- 空白 1/1-1#空白
- 空白文字 2/2-1#▲空白文字の削除, appendix/5-re#メタ文字
- 空文字列 2/2-1#空文字列, appendix/5-re#正規表現の基本
- 空リスト 2/2-2#2. リスト (list)
- 空列 2/2-1#空文字列, appendix/5-re#正規表現の基本
- 組み込み関数 1/1-2#print
- 組み込み定数 1/1-3#真理値を返す関数
- クラス 6/6-3#クラス定義
- クラスタリング 7/7-2#教師なし学習・クラスタリングの例
- 繰り返し 3/3-2#3-2. 繰り返し
- クローズ 4/4-1#ファイルのクローズ
- グラフ appendix/5-matplotlib#線グラフ
- グローバル変数 1/1-2#▲グローバル変数, 3/3-3#変数とスコープ
- 計算量 3/3-2#for 文の計算量, 3/3-2#for 文の計算量
- 計算量のオーダー 3/3-2#for 文の計算量
- 継承 6/6-3#継承
- 検索 2/2-1#検索
- 高階関数 6/6-2#max
- 構文エラー 1/1-1#エラー, 1/1-4#構文エラー
- 子クラス 6/6-3#継承
- コマンドライン実行 appendix/5-command#▲ Python スクリプトとコマンドライン実行
- コマンドライン引数 appendix/5-command#コマンドライン引数
- コメント 1/1-1#コメント, 1/1-2#コメントと空行
- 小文字 2/2-1#大文字・小文字
- コンストラクタ 6/6-3#クラス定義
- コーディングスタイル 1/1-4#コーディングスタイル
- 再帰 1/1-3#▲再帰, appendix/3-recursion#▲再帰
- 再帰関数 appendix/3-recursion#▲再帰
- 再帰呼び出し appendix/3-recursion#▲再帰
- 作業ディレクトリ 4/4-3#カレントワーキングディレクトリ
- 差集合 appendix/2-set#集合演算, appendix/2-set#union, intersection, difference
- 散布図 appendix/5-matplotlib#散布図
- 集合 appendix/2-set#▲セット (set)
- 集合演算 appendix/2-set#集合演算
- 商 1/1-1#簡単な算術計算
- 初期化 6/6-3#初期化と属性
- 初期値 3/3-3#▲引数の初期値

- 仕様 1/1-4#仕様・テスト・デバッグ
- シリーズ 7/7-1#シリーズとデータフレーム
- 真 1/1-3#真理値を返す関数
- 真理値 1/1-3#真理値を返す関数
- 真理値配列によるインデックスアクセス 5/5-3#▲真理値配列によるインデックスアクセス, 7/7-1#データの条件取り出し
- ジェネレータ式 6/6-1#▲ジェネレータ式
- 次元削減 7/7-2#教師なし学習・次元削減の例
- 辞書 3/3-1#3-1. 辞書 (dictionary)
- 辞書型の可変長引数 3/3-3#▲辞書型の可変長引数
- 辞書内包表記 6/6-1#▲辞書内包表記
- 実行時エラー 1/1-1#エラー, 1/1-4#実行時エラー
- 実数 1/1-1#整数と実数
- 実引数 3/3-3#引数
- 条件付き内包表記 6/6-1#▲条件付き内包表記
- 条件分岐 1/1-3#if 文による条件分岐, 2/2-3#2-3. 条件分岐
- 剰余 1/1-1#簡単な算術計算
- 除算 1/1-1#簡単な算術計算
- スライス 2/2-1#文字列とスライス, 5/5-3#スライス, 7/7-1#データの参照
- 正規表現 *appendix/5-re#*▲正規表現
- 整数 1/1-1#整数と実数
- 整数除算 1/1-1#簡単な算術計算
- 積集合 *appendix/2-set#集合演算, appendix/2-set#union, intersection, difference*
- セット *appendix/2-set#*▲セット (set)
- セット内包表記 6/6-1#▲セット内包表記
- 線形回帰 7/7-2#教師あり学習・回帰の例, *appendix/4-csv#東京の7月の気温*
- 選択 *appendix/5-re#*正規表現の基本
- 絶対パス 4/4-3#パス
- 相対パス 4/4-3#パス
- 属性 6/6-3#初期化と属性
- 属性名 6/6-3#初期化と属性
- 対称差 *appendix/2-set#集合演算, appendix/2-set#union, intersection, difference*
- 足し算 1/1-1#簡単な算術計算
- 多次元配列 5/5-3#多次元配列
- 多重代入 2/2-2#多重代入
- 多重リスト 2/2-2#多重リスト
- タプル 2/2-2#タプル (tuple)
- 単項演算子 1/1-1#単項の + と -
- 代入 1/1-2#代入文
- 代入演算子 1/1-2#累積代入文

- 代入文 1/1-2#代入文
- 置換 2/2-1#置換
- テスト 1/1-4#仕様・テスト・デバッグ
- テストケース 1/1-4#仕様・テスト・デバッグ
- ディレクトリ 4/4-3#4-3. コンピュータにおけるファイルやディレクトリの配置
- デバッグ 1/1-2#print, 1/1-4#仕様・テスト・デバッグ
- データ型 2/2-1#2-1. 文字列 (*string*)
- データフレーム 7/7-1#シリーズとデータフレーム
- 特殊メソッド 6/6-3#特殊メソッド
- 特徴量 7/7-2#データ
- トップレベル 1/1-2#▲グローバル変数
- 内包表記 2/2-2#for 文によるリスト初期化の短縮記法, 6/6-1#6-1. 内包表記, 6/6-1#リスト内包表記
- ネスト 2/2-3#if ... else による条件分岐, 3/3-2#for 文の入れ子, 6/6-1#内包表記の入れ子
- 配列 2/2-2#2-2. リスト (*list*), 5/5-3#配列の構築
- 配列同士の演算 5/5-3#配列同士の演算
- 配列のスカラー演算 5/5-3#配列のスカラー演算
- 配列のデータ属性 5/5-3#▲配列のデータ属性
- 破壊的 2/2-2#破壊的（インプレース）な操作と非破壊的な生成
- 半角の空白 1/1-1#空白
- バグ 1/1-2#print, 1/1-4#仕様・テスト・デバッグ
- パス 4/4-3#パス
- パターン *appendix/5-re#正規表現の基本*
- 比較演算 *appendix/2-set#比較演算*
- 比較演算子 1/1-3#様々な条件
- 引き算 1/1-1#簡単な算術計算
- 引数 1/1-2#関数の定義と返値, 1/1-2#関数の定義と返値, 3/3-3#関数の定義
- ヒストグラム *appendix/5-matplotlib#ヒストグラム*
- 否定文字クラス *appendix/5-re#文字クラス*
- 非破壊的 2/2-2#破壊的（インプレース）な操作と非破壊的な生成
- ヒートマップ *appendix/5-matplotlib#ヒートマップ*
- ファイル 4/4-1#ファイルのオープン
- ファイルオブジェクト 4/4-1#ファイルのオープン
- 浮動小数点数 1/1-1#整数と実数
- 分割統治 *appendix/3-recursion#▲再帰*
- 閉包 *appendix/5-re#正規表現の基本*
- 変数 1/1-2#変数, 2/2-1#初心者によくある誤解 — 変数と文字列の混乱
- 変数定義 1/1-2#変数
- べき乗 1/1-1#簡単な算術計算
- べき表示 1/1-1#実数のべき表示

- 棒グラフ [appendix/5-matplotlib#棒グラフ](#)
- マジックコマンド [appendix/3-visualization#matplotlib](#)
- マッチする [appendix/5-re#正規表現の基本](#)
- 無名関数 [6/6-2#ラムダ式](#)
- メソッド [2/2-1#文字列とメソッド](#)
- 文字クラス [appendix/5-re#文字クラス](#)
- 文字コード [4/4-1#ファイルの読み書きにおける文字コード指定](#), [appendix/4-csv#東京の7月の気温](#)
- 文字コード宣言 [appendix/5-command#文字コード宣言](#)
- モジュール [1/1-1#数学関数（モジュールのインポート）](#), [5/5-1#モジュールのインポート](#), [5/5-2#5-2.モジュールの作り方](#)
- モジュール名 [5/5-2#自作モジュールの使い方](#)
- モジュールレベル [1/1-2#▲グローバル変数](#)
- 文字列 [2/2-1#2-1. 文字列 \(string\)](#), [2/2-1#初心者によくある誤解 — 変数と文字列の混乱](#)
- 文字列の比較演算 [2/2-1#文字列の比較演算](#)
- 優先順位 [1/1-1#演算子の優先順位と括弧](#)
- ユニバーサル関数 [5/5-3#ユニバーサル関数](#)
- 読み込みモード [4/4-1#ファイルのオープン](#)
- 予約語 [1/1-2#予約語](#)
- ライブラリ [1/1-1#数学関数（モジュールのインポート）](#)
- ラムダ式 [6/6-2#ラムダ式](#)
- リスト [2/2-2#2-2. リスト \(list\)](#)
- 累算代入文 [1/1-2#累積代入文](#)
- ループ [3/3-2#for 文による繰り返し](#)
- 連結 [2/2-1#文字列の連結](#)
- 接続 [appendix/5-re#正規表現の基本](#)
- ロジスティック回帰 [7/7-2#教師あり学習・分類の例](#)
- 論理エラー [1/1-4#論理エラー](#)
- ローカル変数 [1/1-2#ローカル変数](#), [3/3-3#引数](#), [3/3-3#変数とスコープ](#)
- 和 [appendix/5-re#正規表現の基本](#)
- ワイルドカード [5/5-1#from](#)
- 和集合 [appendix/2-set#集合演算](#), [appendix/2-set#union, intersection, difference](#)
- 割り算 [1/1-1#簡単な算術計算](#)
- ワーキングディレクトリ [4/4-3#カレントワーキングディレクトリ](#)