
Python プログラミング入門

(東京大学・数理・情報教育研究センター) 再配布禁止

© Masami Hagiya et al.

Sep 19, 2019

1	1-0. Jupyter Notebook の使い方	1
1.1	セル	1
1.2	コマンドモード	2
1.3	編集モード	2
1.4	練習	2
1.5	(注意) Shift-Enter に反応がなくなったとき	3
2	1-1. 数値演算	4
2.1	簡単な算術計算	4
2.2	コメント	5
2.3	整数と実数	5
2.4	演算子の優先順位と括弧	7
2.5	算術演算子のまとめ	8
2.6	空白	8
2.7	エラー	9
2.8	数学関数 (モジュールの import)	9
2.9	練習	10
3	1-2. 変数と関数の基礎	11
3.1	変数	11
3.2	関数の定義と返値	13
3.3	練習	14
3.4	ローカル変数	15
3.5	print	16
3.6	print と return	16
3.7	コメントと空行	16
3.8	関数の参照の書き方	17
3.9	練習	17
3.10	▲グローバル変数	18
3.11	練習の解答	19
4	1-3. 論理・比較演算と条件分岐の基礎	20
4.1	if による条件分岐	20
4.2	様々な条件	21
4.3	練習	22
4.4	真理値を返す関数	22
4.5	▲再帰	24
4.6	▲条件として使われる他の値	25
4.7	▲None	25
4.8	▲オブジェクトと属性・メソッド	25
4.9	練習の解答	26

5	1-4. デバッグ	27
5.1	文法エラー : Syntax Errors	28
5.2	実行エラー : Runtime Errors	28
5.3	論理エラー : Logical Errors	28
5.4	print によるデバッグ	29
5.5	▲ assert 文によるデバッグ	30
6	2-1. 文字列型 (string)	31
6.1	文字列とインデックス	32
6.2	文字列とスライス	33
6.3	空文字列	34
6.4	文字列の検索	34
6.5	▲エスケープシーケンス	35
6.6	文字列の連結	36
6.7	文字列とメソッド	36
6.8	文字列の比較演算	38
6.9	初心者によくある誤解、変数 と文字列 の混乱	38
6.10	練習	39
6.11	練習	39
6.12	練習	39
6.13	練習	40
6.14	練習	40
6.15	練習	40
6.16	練習の解答	41
7	2-2. リスト型 (list)	43
7.1	リストとインデックス	44
7.2	多重リスト	45
7.3	リストに対する関数・演算子・メソッド	45
7.4	破壊的 (インプレース) な操作と非破壊的な生成	47
7.5	タプル型 (tuple)	51
7.6	多重代入	52
7.7	リストやタプルの比較演算	53
7.8	for 文による繰り返しとリスト、タプル	54
7.9	for 文による繰り返しと文字列	55
7.10	練習	56
7.11	練習	56
7.12	練習	56
7.13	練習	57
7.14	練習	57
7.15	練習	57
7.16	▲比較演算子 ==, != と is, is not	58
7.17	練習の解答	59
8	2-3. 条件分岐	61
8.1	インデントによる構文	61
8.2	“if” … “else” による条件分岐	62
8.3	“if” … “elif” … “else” による条件分岐	63
8.4	練習	63
8.5	練習	64
8.6	▲複数行にまたがる条件式	64
8.7	条件分岐の順番	64
8.8	練習	65
8.9	分岐の評価	65
8.10	▲ 3 項演算子 (条件式)	66
8.11	練習の解答	66
8.12	練習の解説	67
9	3-1. 辞書型 (dictionary)	68

9.1	辞書のメソッド	69
9.2	▲ keys, values, items の返り値	71
9.3	辞書とリスト	72
9.4	練習	73
9.5	練習	73
9.6	練習の解答	73
10	3-2. 繰り返し	74
10.1	“for” による繰り返し	74
10.2	for 文による繰り返しと辞書	76
10.3	練習	77
10.4	range 関数	77
10.5	range 関数とリスト	78
10.6	for 文の入れ子	79
10.7	for 文の計算量	80
10.8	練習	81
10.9	練習	82
10.10	練習	82
10.11	enumerate 関数	82
10.12	帰属演算子 “in”	83
10.13	“while” による繰り返し	83
10.14	制御構造と “return”	83
10.15	“break” 文	84
10.16	練習	84
10.17	continue 文	84
10.18	▲ for, while 繰り返し文における “else”	84
10.19	“pass” 文	85
10.20	練習	85
10.21	練習	85
10.22	練習	85
10.23	練習	86
10.24	練習	86
10.25	練習	86
10.26	練習	86
10.27	練習	87
10.28	練習	87
10.29	練習	88
10.30	練習	88
10.31	練習の解答	89
10.32	練習の解説	91
10.33	練習の解説	92
11	3-3. 内包表記	93
11.1	リスト内包表記	93
11.2	練習	94
11.3	練習	94
11.4	練習	95
11.5	内包表記のネスト	95
11.6	練習	95
11.7	練習	96
11.8	▲条件付き内包表記	96
11.9	▲集合内包表記	96
11.10	▲辞書内包表記	97
11.11	▲ジェネレータ式	97
11.12	練習の解答	98
12	3-4. 関数	99
12.1	関数の定義	99
12.2	引数	100

12.3	返値	100
12.4	複数の引数	101
12.5	変数とスコープ	101
12.6	▲キーワード引数	103
12.7	▲引数の初期値	103
12.8	▲可変長の引数	104
12.9	▲辞書型の可変長引数	104
12.10	▲引数の順番	104
12.11	▲変数としての関数	105
13	4-1. ファイル入出力の基本	106
13.1	ファイルのオープン	106
13.2	ファイルのクローズ	107
13.3	ファイル全体の読み込み	107
13.4	練習	107
13.5	ファイルに対する for 文	107
13.6	練習	109
13.7	行の読み込み	109
13.8	編集中のファイルの動作	110
13.9	ファイルに対する with 文	110
13.10	ファイルへの書き込み	110
13.11	練習	111
13.12	ファイルの読み書きにおける文字コード指定	111
13.13	改行文字の削除	113
13.14	練習の解答	113
14	4-2. コンピュータにおけるファイルやディレクトリの配置	115
14.1	カレントディレクトリ	116
14.2	相対パスと絶対パス	117
14.3	木構造によるデータ表現	118
15	4-3. モジュールの使い方	120
15.1	モジュールの import	120
15.2	from	121
15.3	as	121
15.4	練習	122
15.5	練習の解答	122
16	4-4. モジュールの作り方	123
16.1	モジュールファイル	123
16.2	Jupyter Notebook でモジュールを扱う	123
16.3	自作モジュールの使い方	124
17	5-1. Python スクリプトとコマンドライン実行	126
17.1	コマンドライン実行の具体例	127
17.2	コマンドライン引数	130
17.3	モジュールのコマンドライン実行	131
17.4	▲ソースファイル先頭部分にある宣言	132
17.5	練習の解答	133
18	5-2. NumPy	134
18.1	配列の構築	134
18.2	配列要素を生成する構築関数	137
18.3	練習	138
18.4	配列要素の操作	138
18.5	要素毎の演算	140
18.6	よく使われる配列操作	142
18.7	配列の保存と復元	143
18.8	▲真理値配列によるインデックスアクセス	144
18.9	▲線形代数の演算	144

18.10	練習の解答	145
19	6-1. 関数プログラミング	146
19.1	高階関数	146
19.2	イテレータ	148
19.3	イテレータを生成する関数	150
19.4	▲関数内関数（クロージャ）	154
19.5	練習の解答	155
20	6-2. オブジェクト指向プログラミング	157
20.1	オブジェクト指向の考え方	157
20.2	クラス定義	159
20.3	オーバーライド	160
20.4	オブジェクトとクラス	161
20.5	▲名前付きタプル	162
20.6	▲クラス属性	164
20.7	練習の解答	165
21	7-1. pandas ライブラリ	167
21.1	シリーズとデータフレーム	167
21.2	シリーズ（Series）の作成	167
21.3	データフレーム（DataFrame）の作成	168
21.4	CSV ファイルからのデータフレームの作成	169
21.5	データの参照	169
21.6	データの条件取り出し	171
21.7	列の追加と削除	172
21.8	行の追加と削除	173
21.9	データの並び替え	173
21.10	データの統計量	174
21.11	▲データの連結	174
21.12	▲データの結合	175
21.13	▲データのグループ化	175
21.14	▲欠損値、時系列データの処理	176
22	7-2. scikit-learn ライブラリ	177
22.1	機械学習について	177
22.2	教師あり学習	177
22.3	教師なし学習	177
22.4	データ	178
22.5	モデル学習の基礎	178
22.6	教師あり学習・分類の例	178
22.7	練習	179
22.8	練習の解答例	180
22.9	教師あり学習・回帰の例	180
22.10	教師なし学習・クラスタリングの例	182
22.11	練習	184
22.12	練習の解答例	184
22.13	教師なし学習・次元削減の例	185
23	▲セット型 (set)	187
23.1	セットの組み込み関数	188
23.2	集合演算	188
23.3	比較演算	189
23.4	セットのメソッド	189
23.5	練習	191
23.6	練習	191
23.7	練習	191
23.8	練習の解答	192

24 ▲簡単なデータの可視化	193
24.1 matplotlib	193
24.2 折れ線グラフ	193
24.3 散布図	194
24.4 棒グラフ	195
25 ▲再帰	197
25.1 再帰関数の例：接頭辞リストと接尾辞リスト	197
25.2 再帰関数の例：べき乗の計算	198
25.3 再帰関数の例：マージソート	198
26 ▲CSV ファイルの入出力	202
26.1 csv 形式とは	202
26.2 CSV ファイルの読み込み	202
26.3 CSV ファイルに対する for 文	204
26.4 CSV ファイルに対する with 文	204
26.5 CSV ファイルの書き込み	205
26.6 練習	211
26.7 練習	212
26.8 練習の解答	212
27 ▲JSON ファイルの入出力	214
27.1 JSON 形式とは	214
27.2 JSON ファイルのダンプとロード	215
27.3 練習	217
27.4 不要な空白や改行の除去	220
27.5 練習の解答	220
28 ▲Bokeh ライブラリ	221
28.1 線グラフ	221
28.2 散布図	224
28.3 棒グラフ	225
28.4 ヒストグラム	225
28.5 ヒートマップ	225
28.6 グラフのファイル出力	226
29 ▲Matplotlib ライブラリ	227
29.1 線グラフ	227
29.2 練習	234
29.3 練習	234
29.4 散布図	235
29.5 練習	236
29.6 棒グラフ	237
29.7 練習	238
29.8 ヒストグラム	239
29.9 練習	239
29.10 ヒートマップ	240
29.11 練習	240
29.12 グラフの画像ファイル出力	241
29.13 練習の解答	242
30 ▲正規表現	245
30.1 正規表現の基本	245
30.2 練習	250
30.3 練習	251
30.4 練習	252
30.5 練習	252
30.6 練習	253
30.7 正規表現の応用 1	254
30.8 正規表現に関する基本的な関数	255

30.9	練習	257
30.10	練習	258
30.11	正規表現の応用 2	258
30.12	メタ文字	260
30.13	練習	262
30.14	練習	262
30.15	練習	263
30.16	練習	263
30.17	正規表現のエスケープシーケンス	264
30.18	正規表現に関する関数とメソッド	264
30.19	練習	266
30.20	練習	267
30.21	練習の解答	268
31	▲イテラブルとイテレータ	272
31.1	ジェネレータ関数と無限イテレータ	272
31.2	イテラブルと for 文	273
31.3	イテレータとイテラブルの定義	274
31.4	コレクションの階層	275
32	索引など	276

CHAPTER 1

1-0. 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。

1.1 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 []:` と書かれています。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。

[]:

1.2 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

1.3 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。

1.4 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。

[]:

1.5 (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 2

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>

2.1 簡単な算術計算

Jupyter Notebook では、In []: と書いてあるセルへ Python の式を入力して、Shift を押しながら Enter を押すと、式が評価され、その結果の値が下に挿入されます。

1+1 の計算をしてみましょう。下のセルに 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、下の値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 は “/” で表され、整除 は “//” で表されます。

整除は整数同士の割り算で、商も余りも整数となります。

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

```
[6]: 7.0
```

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

```
[7]: 7
```

整除の余り を求めたいときは、別の演算子 “%” を用います。

```
[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）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。データ同士が等しいかどうかを調べる == という演算子について後で紹介します。

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

```
[13]: True
```

+ と - と * と // と % と ** では、二つの数が整数ならば結果も整数になります。二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]: 1071508607186267320948425049060001810561404811705533607443750388370351051124936122493198378815695
```

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]: 2003529930406846464979072351560255750447825475569751419265016973710894059556311453089506130880933
```

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、適当に空白を入れることができます。

```
[21]: 7 - 2 * 3
```

```
[21]: 1
```

```
[22]: (7 - 2) * 3
```

```
[22]: 15
```

```
[23]: 17 - 17//3*3
```

```
[23]: 2
```

```
[24]: 56 ** 4 ** 2
```

```
[24]: 9354238358105289311446368256
```

```
[25]: 56 ** 16
```

```
[25]: 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。

```
[26]: 16/8/2
```

```
[26]: 1.0
```

```
[27]: (16/8)/2
```

```
[27]: 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。

```
[28]: 10/2*3
```

```
[28]: 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、単項の演算子としても使えます。(これらの演算子の後に一つだけ数書かれます。)

```
[29]: -3
```

```
[29]: -3
```

```
[30]: +3
```

```
[30]: 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、二項の * 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、二項の + 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。

```
[31]: 7 **2
```

```
[31]: 49
```

```
[32]: 7* *2
```

```
File "<ipython-input-32-a8c53e2964d9>", line 1
    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。

```
[33]: 7  **2
```

```
File "<ipython-input-33-266e892b3f31>", line 1
    7  **2
      ^
SyntaxError: invalid character in identifier
```


2.7 エラー

色々試していると、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。

```
[34]: 10/0

-----
ZeroDivisionError                                Traceback (most recent call last)
<ipython-input-34-e574edb36883> 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。

```
[35]: 10/

      File "<ipython-input-35-9e74788f8c3b>", line 1
        10/
         ^
SyntaxError: invalid syntax
```

こちらのエラーは文法エラー です。つまり、入力 が Python の文法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 import）

```
[36]: import math
```

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

```
[37]: 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。

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

```
[38]: 0.0
```

```
[39]: math.pi
```

```
[39]: 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。

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

```
[40]: 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。

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

```
[41]: 1.0
```

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

```
[42]: 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。例えば、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、未定義の変数を式の中で用いると、次のようにエラーが生じます。

```
BMI # 未定義の変数
```

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 `=` による記法を、Python では代入文 (**assignment statement**) と呼びます。`=` 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を示しており、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 の値が増えたことで、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。例えば、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。

“Python
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。

“Python
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 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。

関数呼出しは演算式の一つなので、引数の位置には任意の式を記述できますし、関数呼出し自体も式中に記述できます。

```
[14]: 1.1*bmi(174.0, 119.0 * 0.454)  
[14]: 19.628947020742505
```

もう一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。コードブロックの構文ハイライトで強調される(太字緑色など)ものが予約語だと覚えておけば大体問題ありません。

3.3 練習

次のような関数を定義してください。

1. f フィート i インチをセンチメートルに変換する `feet_to_cm(f, i)` ただし、1 フィート = 12 インチ = 30.48 cm である。
2. 二次関数 $f(x) = ax^2 + bx + c$ の値を求める `quadratic(a, b, c, x)`

定義ができたら、その次のセルを実行して、True のみが表示されることを確認してください。

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

```
[18]: def check_similar(x, y):  
      print(abs(x-y)<0.000001)  
check_similar(feet_to_cm(5, 2), 157.48)  
check_similar(feet_to_cm(6, 5), 195.58)
```

```
-----  
TypeError                                Traceback (most recent call last)  
<ipython-input-18-7f7eeac1d0bd> in <module>  
      1 def check_similar(x, y):  
      2     print(abs(x-y)<0.000001)  
----> 3 check_similar(feet_to_cm(5, 2), 157.48)  
      4 check_similar(feet_to_cm(6, 5), 195.58)  
  
<ipython-input-18-7f7eeac1d0bd> in check_similar(x, y)  
      1 def check_similar(x, y):  
----> 2     print(abs(x-y)<0.000001)  
      3 check_similar(feet_to_cm(5, 2), 157.48)  
      4 check_similar(feet_to_cm(6, 5), 195.58)  
  
TypeError: unsupported operand type(s) for -: 'ellipsis' and 'float'
```

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

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

False
False
```

3.4 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、関数定義のひとまわ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` の値を参照すると、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)
<ipython-input-23-ded5ba42480f> in <module>
--> 1 s

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
```

3.5 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` は任意個の引数をとることができ、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` 関数の呼出しを単に“`print` する”とか“`print` を挿入”などと表現することになります。

3.6 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 `print` 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。

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。

3.7 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)
```

(continues on next page)

(continued from previous page)

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

3.8 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らって三角形の面積を返します。

のように、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` をもらって三角形の面積を返します。

のように、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に対しても同様の記法が用いられます。

3.9 練習

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 `det(a,b,c)`
2. 解のうち、大きくない方を求める `solution1(a,b,c)`
3. 解のうち、小さくない方を求める `solution2(a,b,c)`

2. と 3. は `det` を使って定義してください。解が実数になる場合のみを想定して構いません。

定義ができれば、その次のセルを実行して、`True` のみが表示されることを確認してください。

```
[29]: def det(a, b, c):  
      return
```

```
[30]: def solution1(a, b, c):  
      return
```

```
[31]: def solution2(a, b, c):  
      return
```

```
[32]: print(det(1,-2,1) == 0)  
print(det(1,-5,6) == 1)  
def check_similar(x,y):  
    print(abs(x-y)<0.000001)  
check_similar(solution1(1,-2,1),1.0)  
check_similar(solution2(1,-2,1),1.0)  
check_similar(solution1(1,-5,6),2.0)  
check_similar(solution2(1,-5,6),3.0)
```

```
False  
False
```

```
-----
TypeError                                Traceback (most recent call last)
<ipython-input-32-dabc82a69fbb> in <module>
      3 def check_similar(x,y):
      4     print(abs(x-y)<0.000001)
--> 5 check_similar(solution1(1,-2,1),1.0)
      6 check_similar(solution2(1,-2,1),1.0)
      7 check_similar(solution1(1,-5,6),2.0)

<ipython-input-32-dabc82a69fbb> in check_similar(x, y)
      2 print(det(1,-5,6) == 1)
      3 def check_similar(x,y):
--> 4     print(abs(x-y)<0.000001)
      5 check_similar(solution1(1,-2,1),1.0)
      6 check_similar(solution2(1,-2,1),1.0)

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

3.10 ▲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、関数の外で値を定義される変数のことです。

したがって、関数の中でグローバル変数を参照することができます。

```
[33]: g = 9.8
```

```
[34]: 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。

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

```
[35]: 1019.2
```

```
[36]: 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。

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

```
[37]: 169.86666666666667
```

以下はより簡単な例です。

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

```
[39]: foo()
```

```
[39]: 10
```

```
[40]: bar()
```

```
[40]: 3
```

```
[41]: a
```

```
[41]: 10
```

```
[42]: a = 20
```

```
[43]: foo()
```

```
[43]: 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 の中の a はグローバル変数です。

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

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

```
[45]: 5
```

```
[46]: a
```

```
[46]: 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

3.11 練習の解答

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

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

```
[49]: import math  
def det(a,b,c):  
    return b*b - 4*a*c  
def solution1(a,b,c):  
    return (-b - math.sqrt(det(a,b,c)))/(2*a)  
def solution2(a,b,c):  
    return (-b + math.sqrt(det(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 による条件分岐

制御構造については第 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、二つの入力の大きい方（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。

“Python
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。

"""Python
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。

```
[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。

<code>x < y</code>	# <code>x</code> は <code>y</code> より小さい
<code>x <= y</code>	# <code>x</code> は <code>y</code> 以下
<code>x > y</code>	# <code>x</code> は <code>y</code> より大きい
<code>x >= y</code>	# <code>x</code> は <code>y</code> 以上
<code>x == y</code>	# <code>x</code> と <code>y</code> は等しい
<code>x != y</code>	# <code>x</code> と <code>y</code> 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。

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

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

<code>i == 1 or i == 2 or i == 3</code>

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
```

4.3 練習

1. 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` という関数が用意されていますが。）
2. x が正ならば 1、負ならば -1、ゼロならば 0 を返す `sign(x)` という関数を定義してください。

定義ができたら、その次のセルを実行して、`True` のみが表示されることを確認してください。

```
[ ]:
```

```
[ ]:
```

```
[6]: print(absolute(5) == 5)
      print(absolute(-5) == 5)
      print(absolute(0) == 0)
      print(sign(5) == 1)
      print(sign(-5) == -1)
      print(sign(0) == 0)
```

```
-----
NameError                                Traceback (most recent call last)
<ipython-input-6-3f8c481ba7df> in <module>
----> 1 print(absolute(5) == 5)
      2 print(absolute(-5) == 5)
      3 print(absolute(0) == 0)
      4 print(sign(5) == 1)
      5 print(sign(-5) == -1)

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

4.4 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。

```
[7]: x = 3
```

```
[8]: x > 1
[8]: True
```

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

```
[9]: x < 1
[9]: False
```

```
[10]: x%2 == 0
[10]: False
```

そして、真理値を返す関数を定義することができます。

```
[11]: 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。

```
[12]: is_even(2)
[12]: True
```

```
[13]: is_even(3)
[13]: False
```

このような関数は、if の後に使うことができます。

```
[14]: 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。

```
[15]: is_odd(2)
[15]: False
```

```
[16]: is_odd(3)
[16]: True
```

次の関数 `tnpo(x)` は、 x が偶数ならば x を 2 で割った商を返し、奇数ならば $3*x+1$ を返します。

```
[17]: def tnpo(x):
      if even(x):
          return x//2
      else:
          return 3*x+1
```

n に 10 を入れておいて、

```
[18]: n = 10
```

次のセルを繰り返し実行してみましょう。

```
[19]: n = tnpo(n)
      n

-----
NameError                                Traceback (most recent call last)
<ipython-input-19-bb0634fb048d> in <module>
----> 1 n = tnpo(n)
      2 n

<ipython-input-17-33983ed441bb> in tnpo(x)
      1 def tnpo(x):
----> 2     if even(x):
      3         return x//2
      4     else:
      5         return 3*x+1

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

4.5 ▲再帰

関数 `tnpo(n)` は n が偶数なら $1/2$ 倍、奇数なら 3 倍して 1 加えた数を返します。

数学者 Collatz はどんな整数 n が与えられたときでも、この関数を使って数を変化させてゆくと、いずれ 1 になると予想しました。

たとえば 3 から始めた場合は $3 \Rightarrow 10 \Rightarrow 5 \Rightarrow 16 \Rightarrow 8 \Rightarrow 4 \Rightarrow 2 \Rightarrow 1$ となります。

そこで n から上の手順で数を変化させて 1 になるまでの回数を `collatz(n)` とします。たとえば `collatz(3)=7`、`collatz(5)=5`、`collatz(16)=4` です。

`collatz` は以下のように定義することができます。この関数は、自分自身を参照する再帰的な関数です。

一般に再帰とは、定義しようとする概念自体を参照する定義のことです。

```
[20]: def collatz(n):
      if n==1:
          return 0
      else:
          return collatz(tnpo(n)) + 1
```

```
[21]: collatz(3)

-----
NameError                                Traceback (most recent call last)
<ipython-input-21-b1ccdd0eb692> in <module>
----> 1 collatz(3)

<ipython-input-20-f7a6a9778c00> in collatz(n)
      3     return 0
      4     else:
----> 5     return collatz(tnpo(n)) + 1

<ipython-input-17-33983ed441bb> in tnpo(x)
      1 def tnpo(x):
----> 2     if even(x):
      3         return x//2
      4     else:
      5         return 3*x+1

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


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 - 組み込み定数 None は偽とみなされます。

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

NG

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

OK

4.7 ▲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。

```
[24]: None
```

print で無理やり表示させると以下ようになります。

```
[25]: 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。

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

NG

4.8 ▲オブジェクトと属性・メソッド

Python プログラムでは、全ての種類のデータ（数値、文字列、関数など）は、オブジェクト指向言語におけるオブジェクトとして実現されます。個々のオブジェクトは、それぞれの参照値によって一意に識別されます。

また、個々のオブジェクトはそれぞれに不変な型を持ちます。

- オブジェクト型
 - 数値型
 - * 整数
 - * 浮動小数点 など

- コンテナ型
 - * シーケンス型
 - ・ リスト
 - ・ タプル
 - ・ 文字列 など
 - * 集合型
 - ・ セット など
 - * マップ型
 - ・ 辞書 など

Python において、変数は、オブジェクトへの参照値を持っています。そのため、異なる変数が、同一のオブジェクトへの参照値を持つこともあります。また、変数に変数を代入しても、それは参照値のコピーとなり、オブジェクトそのものはコピーされません。

オブジェクトは、変更可能なものと不可能なものがあります。数値、文字列などは変更不可能なオブジェクトで、それらを更新すると、変数は異なるオブジェクトを参照することになります。一方、リスト、セットや辞書は、変更可能なオブジェクトで、それらを更新しても、変数は同一のオブジェクトを参照することになります。

個々のオブジェクトは、さまざまな属性を持ちます。これらの属性は、以下のように確認できます。

以下の例では、“__class__” という属性でオブジェクトの型を確認しています。

```
[27]: 'hello'.__class__  
[27]: str
```

この属性は“type” という関数を用いても取り出すことができます。

```
[28]: type('hello')  
[28]: str
```

属性には、そのオブジェクトを操作するために関数として呼び出すことの可能なものがあり、メソッドと呼ばれます。

```
[29]: 'hello'.upper()  
[29]: 'HELLO'
```

4.9 練習の解答

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

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

CHAPTER 5

1-4. デバッグ

デバッグについて説明します。

参考

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

プログラムにバグ（誤り）があって正しく実行できないときは、バグを取り除くデバッグの作業が必要になります。

そもそも、バグが出ないようにすることが大切です。例えば、以下に留意することでバグを防ぐことができます。

- "よい"コードを書く
 - コードに説明のコメントを入れる
 - 1行の文字数、インデント、空白などのフォーマットに気をつける
 - 変数や関数の名前を適当につけない
 - グローバル変数に留意する
 - コードに固有の"マジックナンバー"を使わず、変数を使う
 - コード内でのコピーアンドペーストを避ける
 - コード内の不要な処理は削除する
 - コードの冗長性を減らすようにする など
 - 参考
 - * [Google Python Style Guide](#)
 - * [Official Style Guide for Python Code](#)
- 関数の単体テストを行う
- 一つの関数には一つの機能・タスクを持たせるようにする

など

エラーには大きく分けて、文法エラー、実行エラー、論理エラーがあります。以下、それぞれのエラーについて対処法を説明します。また `print` を用いたデバッグについても紹介します。

5.1 文法エラー : Syntax Errors

1. まず、エラーメッセージを確認しましょう
2. エラーメッセージの最終行を見て、それが `SyntaxError` であることを確認しましょう
3. エラーとなっているコードの行数を確認しましょう
4. そして、当該行付近のコードを注意深く確認しましょう

よくある文法エラーの例： - クォーテーションや括弧の閉じ忘れ - コロンのつけ忘れ - `=` と `==` の混同 - インデントの誤り - 全角の空白

など

```
[1]: print("This is the error")

File "<ipython-input-1-clb5b7f1f4a3>", line 1
    print("This is the error")
    ^
SyntaxError: EOL while scanning string literal
```

```
[2]: 1 + 1

File "<ipython-input-2-6ald737d6e97>", line 1
    1 + 1
    ^
SyntaxError: invalid character in identifier
```

5.2 実行エラー : Runtime Errors

1. まず、エラーメッセージを確認しましょう
2. エラーメッセージの最終行を見て、そのエラーのタイプを確認しましょう
3. エラーとなっているコードの行数を確認しましょう
4. そして、当該行付近のコードについて、どの部分が実行エラーのタイプに関係しているか確認しましょう。もし複数の原因がありそうであれば、行を分割、改行して再度実行し、エラーを確認しましょう
5. 原因がわからない場合は、`print` を挿入して処理の入出力の内容を確認しましょう

よくある実行エラーの例： - 文字列やリストの要素エラー - 変数名・関数名の打ち間違い - 無限の繰り返し - 型と処理の不整合 - ゼロ分割 - ファイルの入出力誤り など

```
[3]: print(1/0)

-----
ZeroDivisionError                                Traceback (most recent call last)
<ipython-input-3-2fc232d1511a> in <module>
----> 1 print(1/0)

ZeroDivisionError: division by zero
```

5.3 論理エラー : Logical Errors

プログラムを実行できるが、意図する結果と異なる動作をする際 1. 入力に対する期待される出力と実際の出力を確認しましょう 2. コードを読み進めながら、期待する処理と異なるところを見つけましょう。必要であれば、`print` を挿入して処理の入出力の内容を確認しましょう

5.4 print によるデバッグ

print を用いたデバッグについて紹介しましょう。以下の関数 median(x, y, z) は、x と y と z の中間値（真ん中の値）を求めようとするものです。x と y と z は相異なる数であると仮定します。

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

```
[5]: median(3, 1, 2)
```

```
[5]: 1
```

このようにこのプログラムは間違っています。最初の if 文で $x > y$ のときに x と y を交換しようとしているのですが、それがうまく行っていないようです。そこで、最初の if 文の後に print を入れて、x と y の値を表示させましょう。

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

```
[7]: median(3, 1, 2)
```

```
1 1
```

```
[7]: 1
```

x と y が同じ値になってしまっています。そこで、以下のように修正します。

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

```
[9]: median(3, 1, 2)
```

```
1 3
```

```
[9]: 2
```

正しく動きました。print は削除してもよいのですが、今後のために # を付けてコメントアウトしておきます。

```
[10]: def median(x,y,z):  
      if x>y:  
          w = x  
          x = y  
          y = w  
      #print(x,y)  
      if z<x:  
          return x  
      if z<y:  
          return z  
      return y
```

```
[11]: median(3,1,2)
```

```
[11]: 2
```

5.5 ▲ assert 文によるデバッグ

論理エラーを見つける上で有用なのが、assert 文です。これは引数となる条件式が偽であった時に、AssertionError が発生してプログラムが停止する仕組みです。次に例を示します。

```
[12]: import math  
def sqrt(x):  
    assert x >= 0  
    return math.sqrt(x)
```

```
sqrt(2)  
sqrt(-2)
```

```
-----  
AssertionError                                Traceback (most recent call last)  
<ipython-input-12-c9e6b0bd682b> in <module>  
      5  
      6 sqrt(2)  
----> 7 sqrt(-2)  
  
<ipython-input-12-c9e6b0bd682b> in sqrt(x)  
      1 import math  
      2 def sqrt(x):  
----> 3     assert x >= 0  
      4     return math.sqrt(x)  
      5  
  
AssertionError:
```

ここで定義した sqrt 関数は、平方根を取る関数です。非負の数しかとらないことを前提とした関数なので、assert x >= 0 と前提をプログラムとして記述しています。sqrt(2) の呼出しでは、この前提を満たし、問題なく計算されます。しかし、sqrt(-2) の呼出しでは、この前提を満たさないため、assert 文が AssertionError を出しています。このエラーメッセージによって、どの部分のどのような前提が満たされなかったかが簡単に分かります。これは、論理エラーの原因の絞り込みに役立ちます。

CHAPTER 6

2-1. 文字列型 (string)

文書処理などに必要な文字列型について説明します。

参考

- <https://docs.python.org/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、文字列型 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あつかえます。

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'
```

上の変数が確かに文字列型（str）であることは、組み込み関数 **“type”** によって確認できます。

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

```
[3]: str
```

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

```
[4]: str
```

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。

1-1 で学んだ数値を文字列に変換したい場合、次のようにおこないます。

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

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

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。

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

```
[6]: 5
```

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` から 3 番目の文字を得たい場合は、以下のようにします：

```
[7]: 'hello'[2]
```

```
[7]: 'l'
```

変数に対しても同様です。多くの場合は変数に対しておこないます。

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

```
[8]: '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。
(言語によっては文字列ではなく別の型として扱われるものもありますので注意してください。)

文字列型に対しインデックスを用いて新しい値を要素として代入することはできません。(次のセルはエラーとなります)

これは Python では文字列そのものを変更することはできないためです。文字列を加工する場合はあらたに別の文字列を作成します。

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

```
-----
TypeError                                Traceback (most recent call last)
<ipython-input-9-e17d071bd021> in <module>
      1 word1 = 'hello'
--> 2 word1[0] = 'H'

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

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)


```
[10]: word1[100]
```

```
-----  
IndexError                                Traceback (most recent call last)  
<ipython-input-10-bd01561922a5> 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。例えば、文字列の最後の文字を取得するには、-1 を指定します。

```
[11]: word1[-1]
```

```
[11]: 'o'
```

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

Index	h	e	l	l	o
Positive	0	1	2	3	4
Negative	-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 加えた値を指定します。例えば、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。

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

```
[12]: '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、次のようにおこなえます。

```
[13]: digits1[0:3]
```

```
[13]: '012'
```

しかし、最初の 0 は省略しても同じ結果となります。

```
[14]: digits1[:3]
```

```
[14]: '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。

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

```
[15]: '3456789'
```

```
[16]: digits1[3:5]
```

```
[16]: '34'
```

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

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

```
[17]: '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 おきに 9 より小さい文字を並べた部分文字列を得ます。

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

```
[18]: '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。

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

```
[19]: '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。

空文字列は、次のように例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。）

```
[20]: price = '2,980 円'
      price.replace(',', '')
```

```
[20]: '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、例えば、最初に指定したインデックス `x` に対して、`x` 以下のインデックスの値（ただし、同じ等号をもつとし、3 番目の値を用いずに）を指定するとどうなるでしょうか？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下さい）。

```
[21]: 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。

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。

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

```
[22]: True
```

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

```
[23]: False
```

実際のプログラムでは文字列型の変数を用いることが多いでしょう。

```
[24]: word1 = 'hello'
      substr1 = 'lo'

      substr1 in word1
```

```
[24]: True
```

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

```
[25]: False
```

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クォート ' 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。

例えば、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。

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

      File "<ipython-input-26-1ce57246475b>", line 1
        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。

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

```
[27]: "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。

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

```
[28]: "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。

```
[29]: 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。

```
[30]: 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 \
```

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

6.6 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。

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

```
[32]: 'hello world'
```

“*” 演算子で文字列の繰り返し回数を指定できます。

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

```
[33]: 'hellohellohello'
```

6.7 文字列とメソッド

文字列に対する操作をおこなうため、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。

文字列には以下のようなメソッドが用意されています。

6.7.1 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。

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

```
[34]: 'he123123o'
```

6.7.2 分割

“split” メソッドは、指定した区切り文字列 B で、文字列 A を分割して、リストと呼ばれるデータに格納します。具体的には、次のように使用します。

リストについては 2-2 で扱います。

```
[35]: fruits1 = 'apple,banana,cherry'
      fruits1.split(',')
[35]: ['apple', 'banana', 'cherry']
```

6.7.3 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。

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。

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

```
[36]: 3
```

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

```
[37]: 2
```

以下はエラーとなります。

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

```
-----
ValueError                                Traceback (most recent call last)
<ipython-input-38-f7e472192833> 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 が返されます。

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

```
[39]: -1
```

6.7.4 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。

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

```
[40]: 2
```

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

```
[41]: 3
```

6.7.5 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。

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

[42]: 'dna'

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

[43]: 'Hello world!'

[44]: lower_text = 'hello world!'
      lower_text.upper() #全ての文字を大文字にする

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

6.8 文字列の比較演算

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

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

True
False

[46]: print('abc' != 'abc')
      print('ab' != 'abc')

False
True

[47]: print('abc' <= 'abc')
      print('abc' < 'abc')
      print('ab' < 'abc')

True
False
True
```

6.9 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、変数を文字列を混乱する例が見られます。例えば、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）。

```
[48]: 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。

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

```
[49]: '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。

```
[50]: str2 = 'abc'  
print(func('str2'))
```

```
STR2
```

6.10 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) を作成して下さい。(練習の正解はノートの一冊最後にあります。)

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

```
[51]: 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下さい。

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

```
False
```

6.11 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 を作成して下さい。

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

```
[53]: 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下さい。

```
[54]: 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
```

6.12 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_bp pair を作成して下さい。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_bp pair(str_atgc) を作成して下さい。

```
[55]: 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下さい。

```
[56]: print(atgc_bppair('AAGCCCCATGGTAA') == 'TTCGGGGTACCATT')  
  
False
```

6.13 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` を作成して下さい。

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

```
[57]: 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下さい。

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

6.14 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` を作成して下さい。ただし、`str_res` の先頭は大文字のアルファベットとして下さい。

次のセルの ... のところを書き換えて `remove_clause(str_atgc)` を作成して下さい。

```
[59]: def remove_clause(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下さい。

```
[60]: print(remove_clause("It's being seen, but you aren't observing.") == 'But you aren  
      ↪ 't observing.")  
  
File "<ipython-input-60-7c2b7a5cb001>", line 1  
    print(remove_clause("It's being seen, but you aren't observing.") == 'But you_  
      ↪ aren't observing.")  
      ^  
SyntaxError: unexpected EOF while parsing
```

6.15 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` を作成して下さい。

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


```
[61]: 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下さい。

```
[62]: 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
```

6.16 練習の解答

```
[63]: 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')
```

```
[64]: 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?')
```

```
[65]: def atgc_pair(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_pair('AAGCCCCATGGTAA')
```

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

```
[67]: 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.")
```

```
[68]: 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
```

(continues on next page)

(continued from previous page)

```
return False  
#check_lower('down down down')  
#check_lower('There were doors all round the hall, but they were all locked')
```

CHAPTER 7

2-2. リスト型 (list)

複数のデータ要素をまとめて取り扱うリスト型とタプル型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、リスト型では構成する要素としてあらゆるデータ型を指定できます。

他の言語ではリストに相当するものとして 配列 やアレイなどがあります。

リストを記述するには、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型を要素とするリストを作成しています。

さらに、文字列と同様に組み込み関数 `type` で変数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
```

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データ型を指定でき、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。
具体的な例として、後述する `append` メソッドの項を参照して下さい。

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

7.1 リストとインデックス

文字列の場合と同様、リストからインデックスが指定する要素を取りだせます。
リストの `x` 番目の要素を取得するには次のようにします。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'
```

[7]: 文字列と同様に、スライスを使った範囲指定も可能です。

```
File "<ipython-input-7-9782bc15463b>", line 1
    文字列と同様に、スライスを使った範囲指定も可能です。
    ^
SyntaxError: invalid character in identifier
```

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

文字列の場合とは異なり、リストは変更可能なデータ型です。
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。

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

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

7.2 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。次は二重リストの例です。

```
[10]: 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おこないます。
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。

```
[11]: lns[1][0]
```

```
[11]: 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。

```
[12]: lns[2]
```

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

以下のようにリストの要素として、リスト型の変数を指定することもできます。

```
[13]: lns2 = [lns, ['x', 1, [11, 12, 13]], ['y', [100, 120, 140]] ]  
      lns2[0]
```

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

```
[14]: lns2[1][2]
```

```
[14]: [11, 12, 13]
```

7.3 リストに対する関数・演算子・メソッド

7.3.1 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。

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

```
[15]: 6
```

```
[16]: numbers[2:4] # スライス
```

```
[16]: [20, 30]
```

7.3.2 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。

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

[17]: [0, 10, 20, 30, 40, 50, 'a', 'b', 'c']
```

```
[18]: numbers*3

[18]: [0, 10, 20, 30, 40, 50, 0, 10, 20, 30, 40, 50, 0, 10, 20, 30, 40, 50]
```

要素がすべて 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。

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

[19]: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

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。

```
[20]: 10 in numbers

[20]: 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。例えば、
は

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。

```
[21]: 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
```

7.3.3 指定した要素のインデックス取得、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。

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

[22]: 2
```

“count” メソッドは指定した要素の数を返します。

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

[23]: 3
```

7.3.4 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。引数になにも指定しなければ昇順でとなります。

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

```
[25]: numbers
```

```
[25]: [10, 20, 30, 40, 50, 60]
```

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

```
[26]: ['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。

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

```
[27]: [60, 50, 40, 30, 20, 10]
```

7.3.5 並べ替え (“sorted” 組み込み関数)

関数 `soted` 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。

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

```
[28]: [10, 20, 30, 40, 50, 60]
```

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

```
[29]: ['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。

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

```
[30]: [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下さい。

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

```
[31]: [[10, 30], [20, 5], [30, 10], [40, 20]]
```

7.4 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` メソッドは元のリストの値が変更されています。一方、`sorted` 関数は元のリストの値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。

```
[32]: 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)
      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]
```

この様に、`sort` メソッドは元のリストの値を書き換えてしまいます。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、その返り値（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。

```
[33]: numbers = [30, 50, 10, 20, 40, 60]
      numbers1 = sorted(numbers)
      print('sorted 関数の返り値:', numbers1)

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

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

7.4.1 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。

7.4.2 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。

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

[34]: [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、という様な状況でしばしば用いられます。例えば、リスト `ln1 = [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 `ln2` は次の様に構成することができます。

```
[35]: 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])
```

(continues on next page)

(continued from previous page)

```
positives.append(numbers1[3])
positives.append(numbers1[5])
positives
```

```
[35]: [10, 20, 30, 40]
```

7.4.3 ▲リストにリストを追加する

“**extend**” メソッドはリストの最後尾に指定したリストを付け加えます。

```
[36]: numbers = [10, 20, 30, 40, 50]
      numbers.extend([200, 300, 400, 200]) # 1n + [200, 300, 400, 200] と同じ
      numbers
```

```
[36]: [10, 20, 30, 40, 50, 200, 300, 400, 200]
```

7.4.4 ▲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。

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

```
[37]: [10, 1000, 20, 30, 40, 50]
```

7.4.5 ▲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。

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。

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

```
[38]: [10, 20, 40, 20]
```

```
[39]: 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
```

```
[39]: [10, 40, 20]
```

```
[40]: 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)
<ipython-input-40-b29e5e87cf76> in <module>
----> 1 numbers.remove(100) # リストに含まれない値を指定するとエラー

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

7.4.6 ▲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。

```
[41]: 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。

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

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

7.4.7 ▲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下さい。

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

```
[43]: [10, 20, 40, 50]
```

スライスを使うことも可能です。

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

```
[44]: [10, 20, 50]
```

7.4.8 ▲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。

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

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

7.4.9 ▲copy

リストを複製します。複製をおこなったあとで、一方のリストに変更を加えたとしても、もう一方のリストは影響を受けません。

```
[46]: 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下さい。

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

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

7.5 タプル型 (tuple)

タプル型は、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）。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、は適用できません。

タプルを作成するには、次のように丸括弧 (...) で要素を囲みます。

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

[48]: (1, 2, 3)
```

```
[49]: type(numbers3)

[49]: tuple
```

実は、丸括弧なしでもタプルを作成できます。

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

[50]: (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。

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

[51]: (1,)
```

`t = (1)`だと、`t = 1`と同じです。

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

[52]: 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。

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

[53]: ()
```

リストや文字列と類似したインデックスや組み込み関数をつかった操作が可能です。

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

[54]: 2
```

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

[55]: 3
```

```
[56]: numbers3[1:3] # スライス

[56]: (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。

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

[57]: [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。

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

[58]: (1, 2)
```

7.6 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。

```
[59]: numbers = [0, 10, 20, 30, 40]
      [a, b, c, d, e] = numbers
      b

[59]: 10
```

以下の様にしても同じ結果を得られます。

```
[60]: a, b, c, d, e = ln
      b

-----
ValueError                                Traceback (most recent call last)
<ipython-input-60-a397c27cf848> in <module>
----> 1 a, b, c, d, e = ln
      2 b

ValueError: not enough values to unpack (expected 5, got 4)
```

実は、多重代入は文字列においても実行可能です。

```
[61]: a, b, c, d, e = 'hello'
      d
```

```
[61]: 'l'
```

```
[62]: numbers3 = (1, 2, 3)
      (x,y,z) = numbers3
      y
```

```
[62]: 2
```

これは次の様に記述することもできます。

```
[63]: 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。

```
[64]: 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
```

7.7 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。

```
[65]: print([1, 2, 3] == [1, 2, 3])
      print([1, 2] == [1, 2, 3])
```

```
True
False
```

```
[66]: print((1, 2, 3) == (1, 2, 3))
      print((1, 2) == (1, 2, 3))
```

```
True
False
```

```
[67]: print([1, 2, 3] != [1, 2, 3])
      print([1, 2] != [1, 2, 3])
```

```
False
True
```

```
[68]: print((1, 2, 3) != (1, 2, 3))
      print((1, 2) != (1, 2, 3))

False
True
```

```
[69]: print([1, 2, 3] <= [1, 2, 3])
      print([1, 2, 3] < [1, 2, 3])
      print([1, 2] < [1, 2, 3])

True
False
True
```

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

True
False
True
```

7.8 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` の要素すべてに対して、実行文を繰り返すには次のように書きます。

for 行の **in** 演算子の右辺に処理対象となるリスト `ls` が、左辺に変数 `value` が書かれます。
`ls` の要素は最初、すなわち `ls[0]` から、一つずつ `value` に代入され、実行文の処理を開始します。
実行文の処理が終われば、`ls` の次の要素が `value` に代入され、処理を繰り返します。
`ls` の要素がなくなる、すなわち `len(ls)` 回、繰り返せば **for** 文の処理を終了します。

ここで、**in** 演算子の働きは、先に説明したリスト要素の有無を検査する **in** とは働きが異なることに、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下さい。

次に具体例を示します。実行文では3つの要素を持つリスト `ls` から一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 `vvalue` を用いて取り出した要素にアクセスしています。

```
[71]: 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。

```
[72]: 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。

```
[73]: for value in [0,1,2]:
      print('For loop:', value)

      File "<ipython-input-73-774a845550fc>", line 2
        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。例えば、次の様な内容を実行しようとしていたとします。

```
[74]: 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、次の様な結果になる場合がありますので注意して下さい。

```
[75]: 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おこなえます。

```
[76]: for value in (0,1,2):
      print('For loop:', value)

For loop: 0
For loop: 1
For loop: 2
```

7.9 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おこなう場合は次のように書きます。

ここで、c にはとりだされた一文字（の文字列）が代入されています。

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。

```
[77]: str1 = 'Apple and pen'
      for c in str1:
          print(c.upper())

A
P
P
L
```

(continues on next page)

(continued from previous page)

```
E  
  
A  
N  
D  
  
P  
E  
N
```

7.10 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` を作成して下さい。

以下のセルの ... のところを書き換えて `sum_list(ln)` を作成して下さい。(練習の正解はノートが一番最後にあります。)

```
[78]: 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下さい。

```
[79]: print(sum_list([10, 20, 30]) == 60)  
      print(sum_list([-1, 2, -3, 4, -5]) == -3)  
  
False  
False
```

7.11 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` を作成して下さい。

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

```
[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下さい。

```
[81]: print(reverse_totuple([1, 2, 3, 4, 5]) == (5, 4, 3, 2, 1))  
  
False
```

7.12 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` を作成して下さい (ただし、0 は偶数として扱うものとします)。

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

```
[82]: 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下さい。


```
[83]: 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
```

7.13 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、次の様なリスト `list_count` を返す関数 `atgc_countlist` を作成して下さい。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 の名前を格納したリストとします。

以下のセルの ... のところを書き換えて `atgc_countlist(str_atgc)` を作成して下さい。

```
[84]: 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下さい。

```
[85]: print(sorted(atgc_countlist('AAGCCCCATGGTAA')) == sorted([[5, 'A'], [2, 'T'], [3,
      ↪ 'G'], [4, 'C']]))
```

```
-----
TypeError                                Traceback (most recent call last)
<ipython-input-85-4f8891a83a08> in <module>
--> 1 print(sorted(atgc_countlist('AAGCCCCATGGTAA')) == sorted([[5, 'A'], [2,
      ↪ 'T'], [3, 'G'], [4, 'C']]))

TypeError: 'NoneType' object is not iterable
```

7.14 練習

英語の 1 文からなる文字列 `str_engsentence` が引数として与えられたとき、`str_engsentence` 中に含まれる単語数を返す関数 `count_words` を作成して下さい。ただし、文はピリオドで終了し単語は空白で区切られるものとします。

以下のセルの ... のところを書き換えて `count_words(str_engsentence)` を作成して下さい。

```
[86]: def count_words(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下さい。

```
[87]: print(count_words('From Stettin in the Baltic to Trieste in the Adriatic an iron_
      ↪curtain has descended across the Continent.') == 18)

False
```

7.15 練習

文字列 `str1` が引数として与えられたとき、`str1` を反転させた文字列を返す関数 `reverse_string` を作成して下さい。

以下のセルの ... のところを書き換えて `reverse_string(str1)` を作成して下さい。

```
[88]: def reverse_string(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下さい。

```
[89]: print(reverse_string('No lemon, No melon') == 'nolem oN ,nomel oN')  
  
False
```

7.16 ▲比較演算子 ==, != と is, is not

先に紹介した比較演算子と似た機能の演算子として、is および is not があります。比較演算子 == あるいは != は左辺と右辺のオブジェクトの中身が、それぞれ等しいあるいは等しくないかを判定します。一方、is および is not は左辺と右辺のオブジェクトそのものが、等しいあるいはそうではないかを判定します。オブジェクトについては 1-3 で簡単に紹介されていますが、詳細については 6-2 オブジェクト指向の回で説明します。

これらの違いをリストを使って説明します。

リスト a を作成、それを b に代入します。b の中身はもちろん a と同じです。

```
[90]: a = [1, 2, 3]  
b = a  
print(b)  
print(a == b)  
  
[1, 2, 3]  
True
```

is 演算子で a, b を比較すると True、すなわち同じオブジェクトであることがわかります。念のためオブジェクトの識別値を得る組み込み関数 id の結果も併せて示します。（オブジェクトの識別値とは、1.3 で説明されているオブジェクトの参照値を整数に変換したものです。）

```
[91]: print(a is b)  
print(id(a), id(b))  
  
True  
4496531336 4496531336
```

リスト a と同じ中身のリスト c を作って比較してみます。（組み込み関数 list は新しいリストを作成します。）

== 演算子による比較結果は True にもかかわらず、is 演算子の結果は False となります。もちろん id の結果も異なります。

```
[92]: c = list(a)  
print(c)  
print(a == c)  
print(a is c)  
print(id(a), id(c))  
  
[1, 2, 3]  
True  
False  
4496531336 4497053256
```

オブジェクトの中身が等しいとオブジェクトそのものと同じかどうかの違いは意味がないように思われるかもしれません。両者の違いを具体例で示してみます。

上で作られたリスト a, b, c の a の要素を書き換え、内容を確認します。

```
[93]: a[1] = 100  
print(a)
```

```
[1, 100, 3]
```

予想どおり a は書き換わっています。b, c についてはどうでしょうか？

```
[94]: print(b)
      print(c)
```

```
[1, 100, 3]
[1, 2, 3]
```

a の操作によって b の中身が変わっています (a と b はオブジェクトそのものが等しいため)。一方で c は不変でした (オブジェクトは異なり a と中身が等しかったため)。

このようにオブジェクトの中身が等しいとオブジェクトそのものが同じかどうかによってがふるまいの違いが生じ、予期しない結果となることもあります。

さらに、以下のセルを確認してください

```
[95]: a = []
      b = []
      print(a == b)
      print(a is b)
```

```
True
False
```

[] という式は、新しいリストを作成して返すからです。

多重リスト、たとえば二重リストは、リストの参照値から成るリストです。以下の例では、リスト a が二つ並んだ二重リストを作成しています。

```
[96]: a = [1, 2]
      b = [a, a]
      print(b)
      print(b[0])
      print(b[0] is b[1])
      print(b[0][1])
```

```
[[1, 2], [1, 2]]
[1, 2]
True
2
```

b[0] と b[1] が同じオブジェクトであることに注意してください。どちらも、リスト a と同じオブジェクトです。

ここで、以下のような代入を行ってみましょう。

```
[97]: b[0][1] = 20
```

すると、b の中身は以下ようになります。

```
[98]: print(b)
```

```
[[1, 20], [1, 20]]
```

7.17 練習の解答

```
[99]: def sum_list(ln):
      int_sum = 0
      for value in ln:
          int_sum += value
```

(continues on next page)

(continued from previous page)

```
    return int_sum
#sum_list([10, 20, 30])
```

```
[100]: def reverse_totuple(ln):
        ln.reverse()
        tup = tuple(ln)
        return tup
#reverse_totuple([1, 2, 3, 4, 5])
```

```
[101]: def remove_evenindex(ln):
        ln2 = ln[1::2]
        return ln2
#remove_evenindex(['a', 'b', 'c', 'd', 'e', 'f', 'g'])
```

```
[102]: def atgc_countlist(str_atgc):
        lst_bpname = ['A', 'T', 'G', 'C']
        list_count = []
        for value in lst_bpname:
            int_bpcnt = str_atgc.count(value)
            list_count.append([int_bpcnt, value])
        return list_count
#atgc_countlist('AAGCCCCATGGTAA')
```

```
[103]: def count_words(str_engsentences):
        list_str1 = str_engsentences.split(' ')
        return len(list_str1)
#count_words('From Stettin in the Baltic to Trieste in the Adriatic an iron_
↳curtain has descended across the Continent.')
```

```
[104]: def reverse_string(str1):
        return str1[::-1]
#reverse_string('No lemon, No melon')

#別解
#def reverse_string(str1):
#    ln1 = list(str1)
#    ln1.reverse()
#    str2 = ''.join(ln1)
#    return str2
#reverse_string('No lemon, No melon')
```

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>

条件分岐を行う制御構造 “**if**” 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に加え、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):` の後から第一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二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二レベルの実行文は 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空白文字 4 文字が広く利用されています。講義でもこの書式を利用します。Jupyter-notebook 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。

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。省略した場合は、「式」が偽の時には `if` 直後のグループが実行されないのみになります。

```
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、下の二つのプログ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
    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。

例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` 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、すなわちすべての条件が `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、二つの値は同じで、もう一つの値は他の二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てします。

- 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、一つの要素を除いて、残りは要素の値はすべて同じとします。

その一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二つの方法を示します。一つ目は、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。二つ目は、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 条件分岐の順番

`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おこなわれません。

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ム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 分岐の評価

if 文に与える条件が or および and で結合される複合条件の場合、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。

例えば、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すすみます。

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)
                                         (continues on next page)
```

(continued from previous page)

```
<ipython-input-9-9a741ddc984d> in <module>
      4 del x          # x を未定義に
      5
--> 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` を一行に書くこともできます。

```
sign = 'positive or zero' if x >= 0 else 'negative'
```

これは、以下と等価です。

```
if x >= 0 :
    sign = 'positive or zero'
else:
    sign = 'negative'
```

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。

```
[11]: def exception3(x, y, z):
      if x==y:
          return z
      elif x==z:
          return y
      else:
          return x
```

```
[12]: 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:
```

(continues on next page)

(continued from previous page)

```
    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。

```
[13]: 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. 辞書型 (dictionary)

キーと値を対応させるデータ構造、辞書、について説明します。

参考

- <https://docs.python.org/ja/3/tutorial/datastructures.html#dictionaries>

辞書型は、キー (**key**) と 値 (**value**) を対応づけたデータです。キーとしては文字列・数値・タプルなど不変なデータ型を使うことができますが、変更可能なリスト・辞書を使うことはできません。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オブジェクトを指定できます。

例えば、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
```

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、
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)
<ipython-input-4-f7d700f1de78> 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、値を登録するには代入を用います。

```
[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出来ません。
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出来ません。

```
[6]: ppap[1]

-----
KeyError                                Traceback (most recent call last)
<ipython-input-6-a560878f1de9> 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出来ます。具体的には、次のように削除します。

```
[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もできます。このような空のデータ型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 辞書のメソッド

辞書のメソッドを紹介しておきます。

9.1.1 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分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。

```
[12]: 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)
<ipython-input-12-a70752670fa0> 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、その値を「指定したキーが含まれていない場合」に返る値とすることが出来ます。

```
[13]: 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
```

9.1.2 ▲キーがない場合に登録をおこなう

“**setdefault**” メソッドは、指定したキーが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。

```
[14]: 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}
```

9.1.3 ▲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。

```
[15]: ppap = {'apple' : 3, 'pen' : 5}
print(ppap.pop('pen'))
print(ppap)
```

```
5
{'apple': 3}
```

9.1.4 ▲全てのキー、値の削除

“clear” メソッドは全てのキー、値を削除します。その結果、辞書は空となります。

```
[16]: ppap = {'apple': 3, 'pen': 5}
      ppap.clear()
      ppap
[16]: {}
```

9.1.5 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。

```
[17]: ppap = {'apple': 3, 'pen': 5}
      list(ppap.keys())
[17]: ['apple', 'pen']
```

9.1.6 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。

```
[18]: list(ppap.values())
[18]: [3, 5]
```

9.1.7 キー、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。

```
[19]: list(ppap.items())
[19]: [('apple', 3), ('pen', 5)]
```

9.2 ▲keys, values, items の返り値

keys, values, items メソッドの一連の説明では、返り値を「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 メソッドの返り値をそれぞれ 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。一、二回目の 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。

```
[20]: 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)]
```

9.2.1 ▲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。

```
[21]: 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}
```

9.3 辞書とリスト

冒頭に述べたように、辞書では値としてあらゆるデータ型を使用できます。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。リストの要素にアクセスするには数字インデックスをさらに指定します。

```
[22]: numbers = {'dozens': [10, 20, 40], 'hundreds': [100, 101, 120, 140]}
print(numbers['dozens'])
print(numbers['dozens'][1])

[10, 20, 40]
20
```

逆に、辞書を要素にするリストを作成することも出来ます。

```
[23]: 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

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` を作成して下さい。

以下のセルの ... のところを書き換えて `reverse_lookup(list1)` を作成して下さい。

```
[24]: 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下さい。

```
[25]: print(reverse_lookup(['apple', 'pen', 'orange']) == {'apple': 0, 'orange': 2, 'pen'  
      ↪': 1})  
  
False
```

9.5 練習

辞書 `dic1` と文字列 `str1` が引数として与えられたとき、辞書 `dic2` を返す関数 `handle_collision` を作成して下さい。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。* `handle_collision` では、`dic1` から次の様な処理を行って `dic2` を作成するものとします。1. `dic1` に `str1` の長さの値 `len` 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 `ln` を作成し、`dic1` にキー `len`、`len` に対応する値 `ln` を登録します。2. `dic1` に `str1` の長さの値 `len` 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)` を作成して下さい。

```
[26]: 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下さい。

```
[27]: print(handle_collision({3: ['ham', 'egg'], 6: ['coffee', 'brandy'], 9: ['port wine'  
      ↪'], 15: ['curried chicken']}, 'tea') == {3: ['ham', 'egg', 'tea'], 6: ['coffee',  
      ↪'brandy'], 9: ['port wine'], 15: ['curried chicken']})  
  
False
```

9.6 練習の解答

```
[28]: def reverse_lookup(list1):  
      dic1 = {} # 空の辞書を作成する  
      for value in range(len(list1)):  
          dic1[list1[value]] = value  
      return dic1  
#reverse_lookup(['apple', 'pen', 'orange'])
```

```
[29]: def handle_collision(dic1, str1):  
      if dic1.get(len(str1)) is None:  
          ln = [str1]  
      else:  
          ln = dic1[len(str1)]  
          ln.append(str1)  
          dic1[len(str1)] = ln  
      return dic1  
#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>

繰り返しをおこなう制御構造 `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” 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` 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))

dog 3
cat 3
mouse 5
```

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。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` 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
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, 112 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 matplotlib を参照してください。

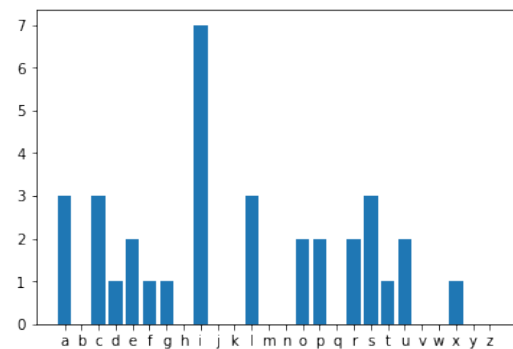
```
[5]: import matplotlib.pyplot as plt
```

```
plt.plot(height)
```

```
[5]: [<matplotlib.lines.Line2D at 0x1167bf198>]
```

```
[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 3-3 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は for 文を用います。辞書 dic1 の全ての key に対して、実行文を繰り返すには次のように書きます。

for 行の in 演算子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、キーを一つずつ取り出し、key に代入しています。

その後、key に対応する値にアクセスしています。

```
[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）値を一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: cat
value: dog
value: elephant
```

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 演算子の左辺に複数の変数を指定し多重代入をおこなっています。

```
[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、次の様な辞書 dic2 を返す関数 reverse_lookup2 を作成して下さい。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 を作成して下さい。

```
[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下さい。

```
[12]: print(reverse_lookup2({'apple': 3, 'pen': 5, 'orange': 7}) == {3: 'apple', 5: 'pen', 7: 'orange'})
```

```
False
```

10.4 range 関数

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。

if 文と同様、実行文の前にはスペースが必要であることを注意して下さい。

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!
```

(continues on next page)

(continued from previous page)

```
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. 引数を二つあるいは三つ与えると: 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。

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、1~9 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 range 関数とリスト

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与え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。

```
[18]: seq_list = list(range(5))
      print(seq_list)

[0, 1, 2, 3, 4]
```

10.6 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。

```
[19]: 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
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下さい。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。

```
[20]: 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]

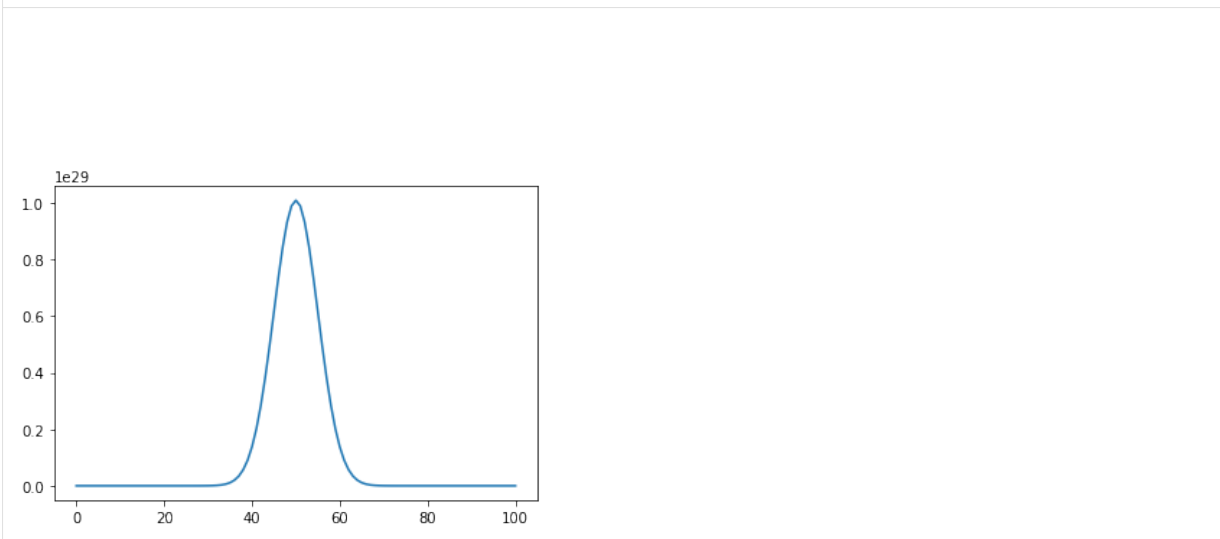
      C[:10]

[20]: [[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。

```
[21]: plt.plot(C[100])
```

```
[21]: [<matplotlib.lines.Line2D at 0x116b5e8d0>]
```



10.7 for 文の計算量

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。

```
[22]: 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。


```
[23]: 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))
```

```
[24]: variance(d100)
```

```
[24]: 120.87162367915938
```

```
[25]: variance(d10000)
```

```
[25]: 102.52515990437892
```

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。

局所変数を用いて `variance` の定義を書き直してみましょう。

```
[26]: def variance(d):
      av = average(d)
      s = 0
      for x in d:
          s = s + (x-av)**2
      return s/len(d)
```

```
[27]: variance(d100)
```

```
[27]: 120.87162367915938
```

```
[28]: variance(d10000)
```

```
[28]: 102.52515990437892
```

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、一重のループを二回実行しているだけだからです。

10.8 練習

次のような関数 `sum_lists` を作成して下さい。- `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` を作成して下さい。

```
[29]: 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下さい。

```
[30]: print(sum_lists([[20, 5], [6, 16, 14, 5], [16, 8, 16, 17, 14], [1], [5, 3, 5, 7]]))
      ↪ == 158)
      False
```

10.9 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` を作成して下さい。

- `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` を作成して下さい。

```
[31]: 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下さい。

```
[32]: 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
```

10.10 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下さい。例えば、`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` を作成して下さい。

```
[33]: 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下さい。

```
[34]: print(sum_n(1, 3) == 6)  
  
False
```

10.11 enumerate 関数

`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:

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。

たとえば、リスト要素とその順番の辞書が欲しい場合は以下のように書くことができます:

```
[35]: 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}
```

10.12 帰属演算子 “in”

Python では for ループでリストを展開する in とは別に、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
```

10.13 “while” による繰り返し

while 文では条件式が 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。

```
[36]: 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下さい。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。

```
[37]: total = 0
for x in range(11):
    total += x

print(x, total)

10 55
```

10.14 制御構造と “return”

return 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なかで return が呼ばれた場合、ただちに関数の処理を終了し、その後の処理は起こわれません。

以下の関数 simple_lsearch は与えられたリスト、list1 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 list1 の各要素に対して繰り返しを実行する様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 文ですべ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。

```
[38]: def simple_lsearch(lst, myitem):
    for item in lst:
        if item == myitem:
            return True
    return False
```

10.15 “break” 文

break 文は for もしくは while ループ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、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。

```
[39]: 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
```

10.16 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 を作成して下さい。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り値として返して下さい。str2 を含まない場合、-1 を返して下さい。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（5-1 で学習します）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 を作成して下さい。

```
[40]: 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下さい。

```
[41]: 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
```

10.17 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。

10.18 ▲ for, while 繰り返し文における “else”

for 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。

for 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。

10.19 “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。例えば以下の、if ~ elif ~ else プログラムはエラーとなります。

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。

10.20 練習

以下のプログラムでは 1 秒おきに print が永遠に実行されます。

10 回 print が実行された後に while ループ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、Jupyter の Interrupt (割り込み) ボタンが使えます。

```
[ ]:
```

10.21 練習

整数の要素からなるリスト `ln` を引数として取り、`ln` から奇数の値の要素のみを取り出して作成したリストを返す関数 `remove_evenelement` を作成して下さい（ただし、0 は偶数として扱うものとします）。

以下のセルの ... のところを書き換えて `remove_evenelement(ln)` を作成して下さい。

```
[42]: def remove_evenelement(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下さい。

```
[43]: print(remove_evenelement([1, 2, 3, 4, 5]) == [1, 3, 5])  
  
False
```

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` を作成して下さい。ただし、同じ単語を要素と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)` を作成して下さい。

```
[44]: 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下さい。

```
[45]: 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つのリストの偶数番目のインデックスの要素を値を入れ替えて、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` を作成して下さい（ただし、0は偶数として扱うものとします）。

以下のセルの ... のところを書き換えて `swap_lists(ln1, ln2)` を作成して下さい。

```
[46]: 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下さい。

```
[47]: 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

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` を作成して下さい。ただし、`ln` の `i` (`i` は 0 以上 `int_size-1` 以下の整数) 番目の要素は `i` とします。

以下のセルの ... のところを書き換えて `construct_list(int_size)` を作成して下さい。

```
[48]: 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下さい。

```
[49]: print(construct_list(10) == [0, 1, 2, 3, 4, 5, 6, 7, 8, 9])  
False
```

10.25 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` を作成して下さい。

以下のセルの ... のところを書き換えて `count_capitalletters(str1)` を作成して下さい。

```
[50]: 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下さい。

```
[51]: print(count_capitalletters('Que Ser^^c3^a1, Ser^^c3^a1') == 3)  
False
```

10.26 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` を作成して下さい。

以下のセルの ... のところを書き換えて `identify_codons(str_augc)` を作成して下さい。

```
[52]: 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下さい。

```
[53]: print(identify_codons('CCCCCGGCACCT') == ['CCC', 'CCG', 'GCA', 'CCT'])  
  
False
```

10.27 練習

正数 `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` を作成して下さい。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` を作成して下さい。

```
[54]: 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下さい。

```
[55]: 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
```

10.28 練習

リスト `list1` が引数として与えられ、次の様な文字列 `str1` を返す関数 `sum_strings` を作成して下さい。

`list1` が `k` 個（ただし、`k` は正の整数）の要素をもつとします。`list1` の要素が文字列でなければ文字列に変換して下さい。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` を作成して下さい。

```
[56]: 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下さい。

```
[57]: print(sum_strings(['a', 'b', 'c', 'd']) == 'a, b, c and d')  
      print(sum_strings(['a']) == 'a')  
  
False  
False
```

10.29 練習

辞書 `dic1` と長さ 1 以上 10 以下の文字列 `str1` が引数として与えられたとき、辞書 `dic2` を返す関数 `handle_collision2` を作成して下さい。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。* `handle_collision2` では、`dic1` から次の様な処理を行って `dic2` を作成するものとします。0. `str1` の長さを `size` とします。1. `dic1` に `size` がキーとして登録されていない場合、`dic1` にキー `size`、`size` に対応する値 `str1` を登録します。2. `dic1` に `size` がキーとして登録されている場合、`i` の値を `size+1`, `size+2`, ... と 1 つずつ増やしていき、`dic1` にキーが登録されていない値 `i` を探します。キーが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、`dic1` を `dic2` として下さい。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下さい。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 `dic2` として下さい。

以下のセルの ... のところを書き換えて `handle_collision2(dic1, str1)` を作成して下さい。

```
[58]: 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下さい。

```
[59]: print(handle_collision2({6: 'Styles', 4: 'Link', 7: 'Ackroyd'}, 'Big Four') == {6:  
      ↳ 'Styles', 4: 'Link', 7: 'Ackroyd', 8: 'Big Four'})  
print(handle_collision2({6: 'Styles', 4: 'Link', 7: 'Ackroyd', 8: 'Big Four', 10:  
      ↳ 'Blue Train', 9: 'End House'}, 'Edgware') == {6: 'Styles', 4: 'Link', 7: 'Ackroyd'  
      ↳ ', 8: 'Big Four', 10: 'Blue Train', 9: 'End House', 1: 'Edgware'})  
print(handle_collision2({6: 'Styles', 4: 'Link', 7: 'Ackroyd', 8: 'Big Four', 10:  
      ↳ 'Blue Train', 9: 'End House', 1: 'Edgware', 2: 'Orient', 3: 'Three Act', 5:  
      ↳ 'Clouds'}, 'ABC') == {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  
False
```

10.30 練習

整数を第 1 要素と文字列を第 2 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、次の様な辞書 `dic1` を返す関数 `handle_collision3` を作成して下さい。* 各子リスト `list2` に対して、`dic1` のキーは `list2` を構成する整数の値とし、そのキーに対応する値は `list2` を構成する文字列の値とします。* 2 つ以上の子リストの第 1 要素が同じ整数の値から構成されている場合、`list1` においてより小さいインデックスをもつ子リストの第 2 要素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)` を作成して下さい。

```
[60]: 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下さい。

```
[61]: 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
```


10.31 練習の解答

```
[62]: 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})

[63]: def sum_lists(list1):
    total = 0
    for list2 in list1: # for j in range(len(list1)):と list2 = list1[j] としても良い
        #print(list2)
        for i in range(len(list2)):
            #print(i, list2[i])
            total += list2[i]
    return total

[64]: 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
    #みて下さい)
    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], '+', list1[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]])

[65]: 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様にする) #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この条件が成立
            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)

[66]: def remove_evelement(ln):
    ln2 = []
    for j in range(len(ln)):
        if ln[j] % 2 == 1:
            ln2.append(ln[j])
```

(continues on next page)

(continued from previous page)

```
    return ln2
#remove_evenelement([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
```

```
[67]: 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.')
```

```
[68]: 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'])
```

```
[69]: def construct_list(int_size):
    ln = int_size * [0]
    for i in range(int_size):
        ln[i] = i
    return ln
#construct_list(10)
```

```
[70]: def count_capitalletters(str1):
    int_count = 0
    for i in range(len(str1)):
        str2 = str1[i].upper()
        str3 = str1[i].lower()
        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')
```

[70]: 3

```
[71]: 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('CCCCGGGCACCT')
```

```
[72]: 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を行う
        str1 = list1[i] + str1
```

(continues on next page)

(continued from previous page)

```
    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')
```

```
[73]: 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'])
```

```
[74]: def handle_collision2(dic1, str1):
    size = len(str1)
    for i in range(size, 11):
        if dic1.get(i) is None: # == None でも良い
            dic1[i] = str1
            return dic1
    for i in range(1, size):
        if dic1.get(i) is None: # == None でも良い
            dic1[i] = str1
            return dic1
    return dic1
#dic1 = {}
#ln1 = ['Styles', 'Link', 'Ackroyd', 'Big Four', 'Blue Train', 'End House',
#      ↪ 'Edware', 'Orient', 'Three Act', 'Clouds', 'ABC', 'Cards']
#for i in range(len(ln1)):
#    dic1 = handle_collision2(dic1, ln1[i])
#    print(dic1)
```

```
[75]: 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 でも良い
            dic1[list2[0]] = list2[1]
    return dic1
#handle_collision3([[3, 'Richard III'], [1, 'Othello'], [2, 'Tempest'], [3, 'King_
#      ↪ John'], [4, 'Midsummer'], [1, 'Lear']])
```

10.32 練習の解説

下のセルは、range() 関数を利用した解答例です。range() 関数が生成する整数列には stop の値が含まれないことに注意してください。

```
[76]: def sum_n(x, y):
    sum = 0
    for i in range(x, y + 1):
```

(continues on next page)

(continued from previous page)

```
    sum = sum + i
    return sum
sum_n(1,3)
```

[76]: 6

10.33 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。

[77]: `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
```

CHAPTER 11

3-3. 内包表記

制御構造のうち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>

11.1 リスト内包表記

Python では内包表記 (Comprehension(s)) が利用できます。
以下のような整数の自乗を要素にも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]
```

squares 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、以下のように一行で書け、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 を適用すると以下のようになります。

```
[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']
```

11.2 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成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)  
<ipython-input-5-f3a6534c2f2c> 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
```

11.3 練習

コンマで区切られた 10 進数から成る文字列が変数 `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 進数から成る文字列を、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)  
<ipython-input-6-fda9e2213556> 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
```

11.4 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` を内容表記と関数 `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` を作成して下さい。

```
[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下さい。

```
[8]: print(var([3,4,1,2]) == 1.25)  
  
False
```

11.5 内包表記のネスト

また内包表記をネスト（入れ子）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。そのリストの要素一つ一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、上の二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` が二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すべての空でない部分文字列から成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']
```

11.6 練習

次のような関数 `sum_lists` を作成して下さい。- `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` を作成して下さい。

```
[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下さい。

```
[13]: print(sum_lists([[20, 5], [6, 16, 14, 5], [16, 8, 16, 17, 14], [1], [5, 3, 5, 7]]))  
      ↪ == 158)  
  
False
```

11.7 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` を作成して下さい。

- `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` を作成して下さい。

```
[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下さい。

```
[15]: 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
```

11.8 ▲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, 4, 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。

11.9 ▲集合内包表記

内包表記はセット (集合) 型、`{}`、で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, 4, 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。

11.10 ▲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': 4, 'elephant': 8, 'giraffe': 7}
```

`length_dic` として `{'cat': 3, 'dog': 3, 'elephant': 8, 'giraffe': 7}` が得られます。

長さで文字列を逆にするとうなるでしょうか。

```
[19]: length_rdic = {len(w): w for w in words}
print(length_rdic)

{3: 'cat', 4: 'dog', 8: 'elephant', 7: 'giraffe'}
```

11.11 ▲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、イテレータと呼ばれる、シーケンス（リストやタプル等）の元となるオブジェクトを構築します。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、シーケンスを引数にと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良いです。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。

イテレータとジェネレータについては、付録：イテラブルとイテレータに説明があります。

11.12 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)]
```

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>

12.1 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、関数の定義をしてみましょう。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
```

関数の一般的な定義は以下の通りです。

```
def 関数名 (引数):
    関数本体
```

1 行目はヘッダと呼ばれ、*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。

12.2 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。

12.3 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` を実行したことに同じになります。

`return` の後に式がない場合は、`None` が返されます。`return` 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'
```

12.4 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
```

12.5 変数とスコープ

関数の引数や関数内の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)
<ipython-input-10-e7ce9f43504a> in <module>
      6
      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()
```

(continues on next page)

(continued from previous page)

```
#グローバル変数 greeting_global を参照  
greeting_global
```

```
Hello
```

```
[11]: 'Hello'
```

関数の引数や関数内でグローバル変数と同じ名前の変数を定義すると、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。

```
[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。例えば、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)  
<ipython-input-13-2e610cbc3e32> in <module>  
      4     print(greeting_global)  
      5  
>>> 6 greeting()  
  
<ipython-input-13-2e610cbc3e32> 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)  
      5  
  
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。

12.5.1 ▲ 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。

12.6 ▲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
```

12.7 ▲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
```

12.8 ▲可変長の引数

引数の前に*を付けて定義すると、複数の引数をタプル型として受け取ることができます。

```
[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')
```

可変長引数を使って、シーケンス型のオブジェクト（リストやタプルなど）の各要素を複数の引数として関数に渡す場合は、*をそのオブジェクト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')
```

12.9 ▲辞書型の可変長引数

引数の前に**を付けて定義すると、複数のキーワード引数を辞書型として受け取ることができます。

```
[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'}
```

辞書型の可変長引数を使って、辞書型のオブジェクト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、**をそのオブジェクト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'}
```

12.10 ▲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)
```

(continues on next page)

(continued from previous page)

```
#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'}
```

12.11 ▲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>

13.1 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（オブジェクト）が入ります。

'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、ノートブックの一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。ノートブックの一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（4-2 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。

モードには次のような種類があります。

記号	モード
r	読み込み
w	書き込み
a	追記
•	読み書き両方を指定したい場合に使用

書き込みについては後でも説明します。

13.2 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下さい。)

`close()` の呼出しは重要ですが、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。

13.3 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。

```
[3]: f = open('sample.txt', 'r')
f.read()
```

```
[3]: '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.
→\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'
```

```
[4]: f.close()
```

13.4 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()` の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。

```
[5]: def number_of_characters(name):
    ...
```

```
[6]: print(number_of_characters('sample.txt') == 446)

False
```

13.5 ファイルに対する for 文

ファイルのオブジェクトは、イテレータと呼ばれるオブジェクトの一種です。iterate は繰り返すという意味ですね。イテレータは、その要素を一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` にファイルの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一行を読んで文字列として返します。

```
[7]: f = open('sample.txt', 'r')
print(next(f))
print(next(f))
f.close()
```

```

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.
```

さらに、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 f を for 文の in の後に指定することができます。

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。

```
[8]: f = open('sample.txt', 'r')
for line in f:
    print(line)
f.close()
```

```

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.
```

ファイルの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。

(リストに対する for 文とは状況が異なりますので注意してください。 リストはイテラブルオブジェクトですがイテレータではないからです。 ファイルのオブジェクトは既にイテレータになっています。)

```
[9]: f = open('sample.txt', 'r')
print('---- 最初 ----')
for line in f:
    print(line)
print('---- もう一度 ----')
for line in f:
    print(line)
f.close()
```

```

--- 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、ファイルのオブジェクトをクローズしてから、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。

13.6 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。

```
[10]: def last_line(name):  
      ...
```

以下のセルによってテストしてください。

```
[11]: print(last_line('sample.txt')== "Excepteur sint occaecat cupidatat non proident,  
      ↳sunt in culpa qui officia deserunt mollit anim id est laborum.\n")  
  
False
```

13.7 行の読み込み

ファイルのオブジェクトには、“`readline()`” というメソッドを適用することもできます。

`f` をファイルの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一行を読んで文字列として返します。文字列の最後に改行文字が含まれます。

`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。(for 文はこのエラーを検知しています。つまり、`next(f)` が `StopIteration` を返したら for ループから抜け出します。)

以下のようにして `readline` を使ってファイルを読んでみましょう。

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。

```
[12]: f = open('sample.txt', 'r')  
  
[13]: f.readline()  
[13]: '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'  
  
[14]: f.readline()  
[14]: 'Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu_  
      ↳fugiat nulla pariatur.\n'  
  
[15]: f.readline()  
[15]: 'Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia_  
      ↳deserunt mollit anim id est laborum.\n'  
  
[16]: f.readline()  
[16]: ''  
  
[17]: f.close()
```

13.8 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。

```
[18]: f = open('test.txt', 'r')
```

下のセルを実行した後だと削除できます。

```
[19]: f.close()
```

13.9 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” 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。

```
[20]: 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.
```

13.10 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。

```
[21]: 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。

```
[22]: 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。

```
[23]: 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、低水準の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。

```
[24]: 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。

13.11 練習

二つのファイル名 infile, outfile を引数として、infile の半角英文字をすべ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。例えば 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。

```
[25]: def file_upper(infile, outfile):
      ...
```

以下のセルによってテストしてください。

```
[26]: 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)
<ipython-input-26-2d3cb47ae79e> 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'
```

13.12 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、シフト 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。

```
[27]: # 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)
<ipython-input-27-fd6181b50dd4> 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())
      4
      5 # Windows ならこちらでエラー

~/anaconda3/lib/python3.7/codecs.py in decode(self, input, final)
    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` のオプションに `encoding='utf-8'` (文字コードに UTF-8 を指定する場合) をつけることにより指定できます。

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つかわれます。本授業でも UTF-8 を推奨します。

```
[28]: # 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:
```

(continues on next page)

(continued from previous page)

```
print(f.read())

# 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

13.13 改行文字の削除

ファイルとテキストモードで開くと、`str` 型の文字列行として読み込まれます。
`str` 型の文字列の末尾にある改行文字が不要な場合は `rstrip()` というメソッドを呼び出すことにより削除することができます。
ここで、カッコ () の中はそのテキストファイルで使われている改行文字を指定します。一般的に、Windows は `\r\n`、MacOS や Linux は `\n` です。また昔の MacOS で作られたテキストファイルは `\r` となっているものもあります。

```
[29]: with open('text/novel.txt', 'r', encoding='utf-8') as f:
        for line in f:
            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:
    for line in f:
        print(line.rstrip('\n'))
```

二人の若い紳士が、すつかりイギリスの兵隊のかたちをして、びか／＼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、ずゐ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、すつかりイギリスの兵隊のかたちをして、びか／＼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、ずゐ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。」

13.14 練習の解答

```
[30]: def number_of_characters(name):
        f = open(name, 'r')
```

(continues on next page)

(continued from previous page)

```
s = f.read()
f.close()
return len(s)
```

```
[31]: def last_line(name):
      f = open(name, 'r')
      for line in f:
          pass
      f.close()
      return line
```

```
[32]: def file_upper(infile, outfile):
      with open(infile, 'r') as f:
          with open(outfile, 'w') as g:
              g.write(f.read().upper())
```

以下のように一つの 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())
```

```
[ ]:
```

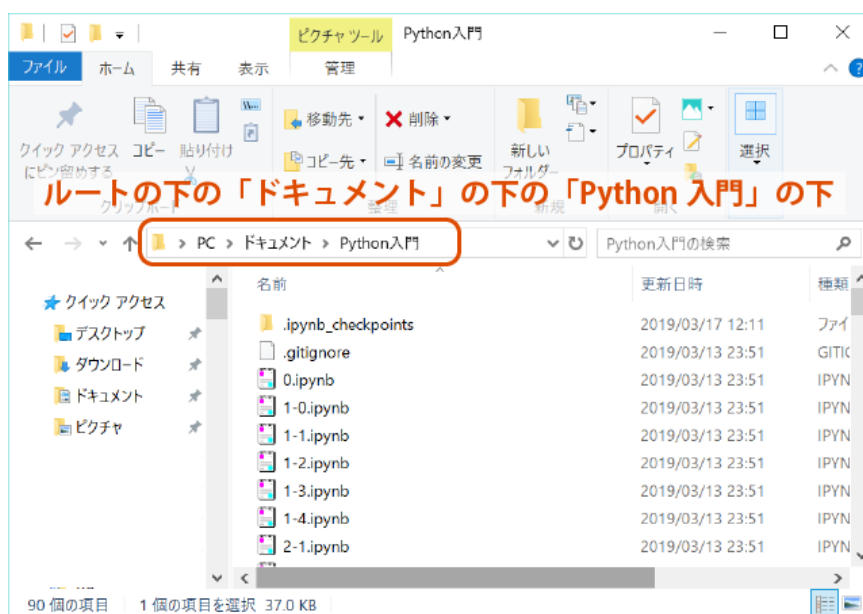
CHAPTER 14

4-2. コンピュータにおけるファイルやディレクトリの配置

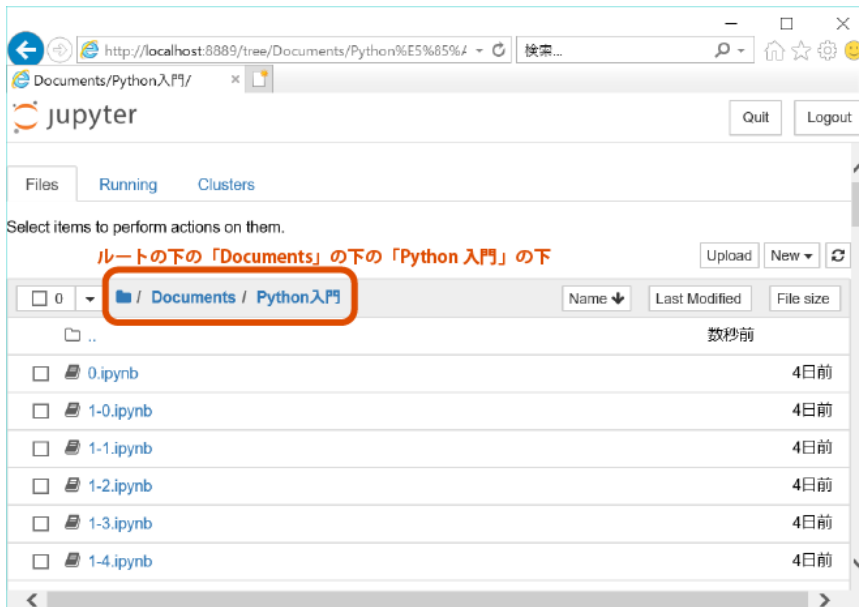
木構造のデータ形式について説明します。

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。

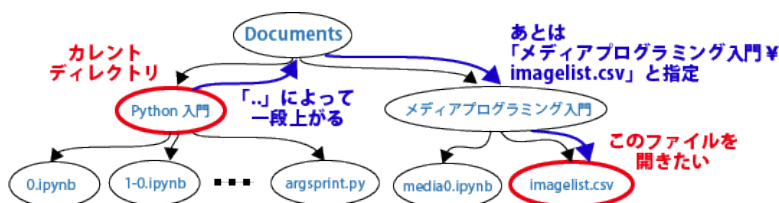
みなさん、Windows ではエクスプローラ、Mac では Finder を使ってファイルを階層的に保存していますよね。下の例では、Windows で「ドキュメント (Documents)」という名前のフォルダの中に「Python 入門」というフォルダを作り、その下にこの教材を置いた時の、エクスプローラの様子を表しています。



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グラフであらわすこともできます。まるで木を逆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。

14.1 カレントディレクトリ

4-1 で sample.txt という名前のファイルをオープンするときに、以下のように書きました。

```
f = open('sample.txt', 'r')
```

このとき、この sample.txt というファイルはどこにあるのでしょうか？

実は、プログラムを実行するときは、どこかのディレクトリをカレントディレクトリとしています。Jupyter Notebook では、その notebook が置かれているディレクトリをカレントディレクトリとします。sample.txt が、このノートブックと同じディレクトリの中に置かれているならば（実際、置かれています）、上のようにして sample.txt を開くことができました。

一方、novel.txt はこの notebook と同じフォルダではなく、そこに置かれた text という名前のフォルダの中に置かれているファイルです。ですから、そのままファイル名だけを指定して開こうとすると以下のようにエラーが出て、ファイルのオープンに失敗します。

```
f = open('novel.txt', 'r', encoding='utf-8')
```

ではどうやったら「カレントディレクトリの下で text の下」にある「novel.txt」を開けるのでしょうか？ これは次のように行います。

```
f = open('text/novel.txt', 'r', encoding='utf-8')
```

このようにすることによって、「カレントディレクトリの下で text の下にある novel.txt を開いてください」と指示することができます。

これは、カレントディレクトリから「novel.txt」までの経路（行き方）を表したもののなのでパス とも呼びます。

```
[1]: f = open('text/novel.txt', 'r', encoding='utf-8')
      print(f.read())
      f.close()
```

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、びか／＼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、ずるぶる痛快だらうねえ。くる／＼まはつて、それからどたつと倒れるだらうねえ。」

14.2 相対パスと絶対パス

text/novel.txt という表現では、カレントディレクトリから「novel.txt」までのパスを表しています。

ここで、Jupyter Notebook では、カレントディレクトリは notebook の場所になるので、どの場所に置かれた notebook を開いているかによってカレントディレクトリが変わり、それに応じて、同じファイルでもパスが変わります。

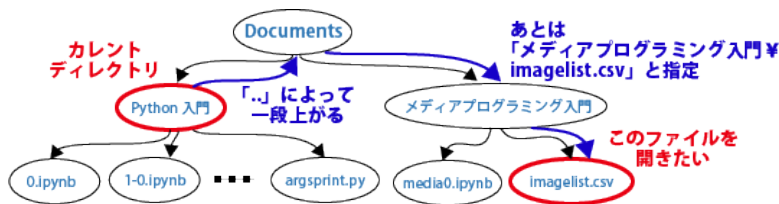
このようなパスの表現を相対パス と呼びます。

一方、Windows の場合 C:\Users\hagiya\Documents\Python 入門\novel1.txt、MacOS の場合 /Users/hagiya/Documents/Python 入門/novel1.txt のように、ルートからのパスを記した場合、カレントディレクトリの場所に関わらず、常に同じファイルを指すことができます。

このようなパスの表現を絶対パス と呼びます。

ところで、カレントディレクトリより下にあるファイルは、そこまでに入るディレクトリ名を/で区切って書けばいいですが、 その下にはないファイルを指すにはどうしたらいいでしょうか？

たとえば下の図のようにカレントディレクトリが C:\Users\hagiya\Documents\Python 入門 (MacOS なら/Users/hagiya/Documents/Python 入門) のとき、
C:\Users\hagiya\Documents\メディアプログラミング入門\imagelist.csv (MacOS なら/Users/hagiya/Documents/メディアプログラミング入門/imagelist.csv) を開きたい場合はどうしたらいいでしょう？



実は、一つ上のディレクトリを.. (コンマ2つ) で表現することができます。
上の例だと、

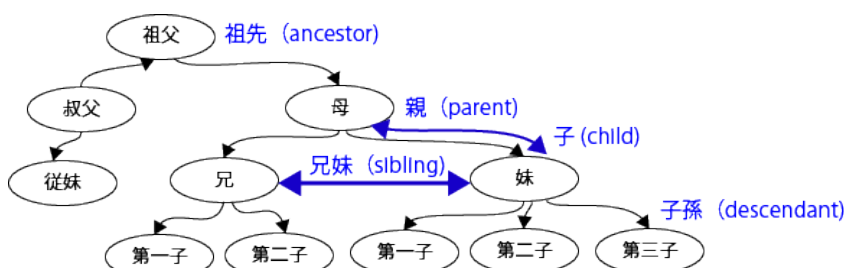
```
f = open('../メディアプログラミング入門/imagelist.csv', 'r')
```

とすれば、C:\Users\hagiya\Documents\メディアプログラミング入門\imagelist.csv (MacOS なら/Users/hagiya/Documents/メディアプログラミング入門/imagelist.csv) を開くことができます。

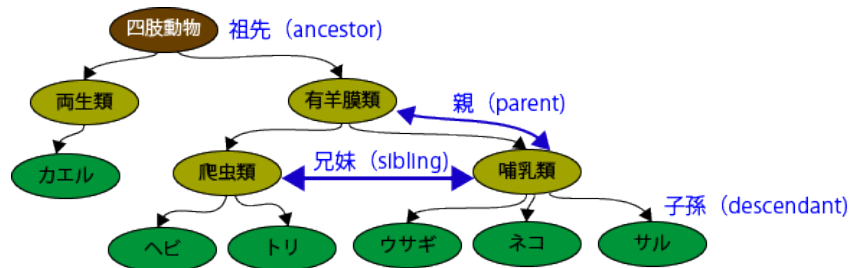
.. によって、「Python 入門」から一段上の「Documents」に戻り、そこから「メディアプログラミング入門」の下に「imagelist.csv」と辿っているわけです。

14.3 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 (siblings)」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。データのグラフ構造におけるこのような表現は、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。

4-3. 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 `import`

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）。具体的には次の様に記述します。

例えば、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（値）を使う場合には、
とする必要があるということです。

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 (値)」の様に、「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” を以下の様を書くことで利用することができます。

例えば、次の様になります。

```
[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 import 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。

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。例えば、5-3 において学習する NumPy というモジュールは、次の様に、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 `import` 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。

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))

File "<ipython-input-7-aldfc8f18650>", line 1
    import ...
    ^
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
```

4-4. 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/interpreter.html>
- <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・クラス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 Jupyter Notebook ファイル（拡張子 `.ipynb`）は、コードセルに 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 の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。

16.2 Jupyter Notebook でモジュールを扱う

Jupyter Notebook でモジュールファイルを扱うには大きく二種類の方法があります。

1. Jupyter Notebook で直接モジュールファイルを開く
2. Jupyter Notebook ファイルをモジュールファイルに変換する

16.2.1 Jupyter Notebook でモジュールファイルを開く

Jupyter Notebook で直接モジュールファイルを作成するには、(Jupyter Notebook 起動時に表示される) ファイルマネージャ画面で、

```
New ⇒ Text File
```

を選択して、エディタ画面を表示させます。

その後、

```
File ⇒ Rename
```

を選択するか、ファイル名を直接クリックして .py 拡張子をもつファイル名として保存します。実際には、コードセルの上で動作を確認したプログラムをクリップボードにコピーして、このエディタにペーストするという方法が現実的と思われます。

16.2.2 Jupyter Notebook 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 Jupyter Notebook ファイルを .py としてセーブするには、

```
File ⇒ Download as ⇒ Python(.py)
```

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#で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。

環境によっては、.py ではなく .html ファイルとして保存されるかもしれませんが、ファイル名を変更すればモジュールファイルとして利用できます。

この方法は、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。Jupyter Notebook 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。

ここでは Jupyter Notebook でモジュールファイルを作成する方法を紹介しましたが、使い慣れているエディタがあればそちらを使ってもかまいません。

16.3 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**”文を用いて `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。

```
# 階乗 n! を返す
def fact(n):
    prod = 1
    for i in range(1, n + 1):
        prod *= i
    return prod
```

```
[1]: import factorial

factorial.fact(6)
```

```
[1]: 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。

```
[2]: from factorial import fact
```

```
fact(6)
```

```
[2]: 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。

```
[3]: 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。

```
[4]: import factorial as f
```

```
f.fact(6)
```

```
[4]: 720
```

5-1. 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。

例えば、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

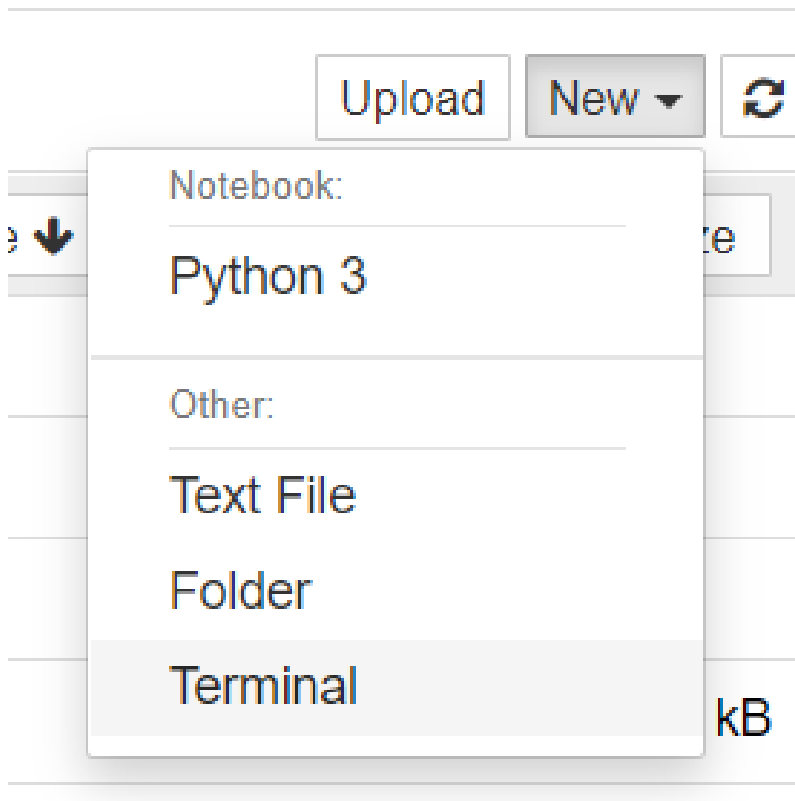
```
>>> python3 sample.py
```

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。

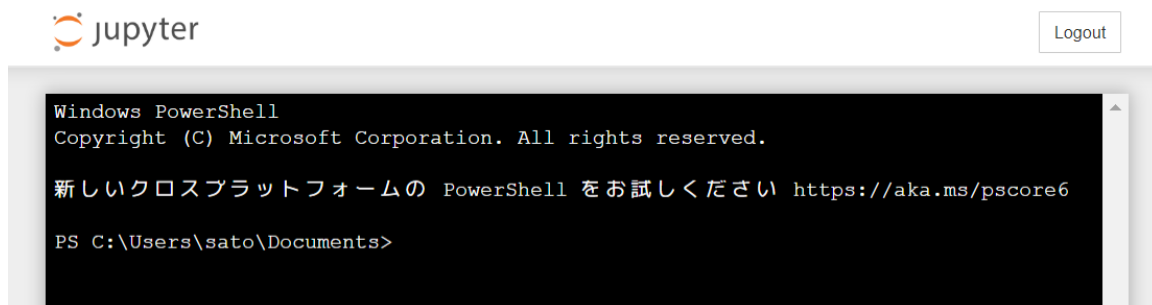
このようにスクリプトをシェルから実行することを、**コマンドライン実行** と呼びます。

17.1 コマンドライン実行の具体例

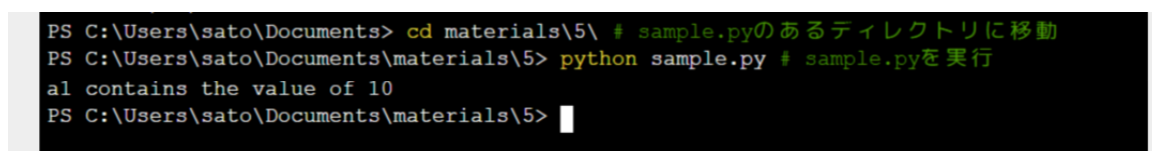
sample.py をコマンドライン実行する具体例を、実行環境毎に説明します。



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。



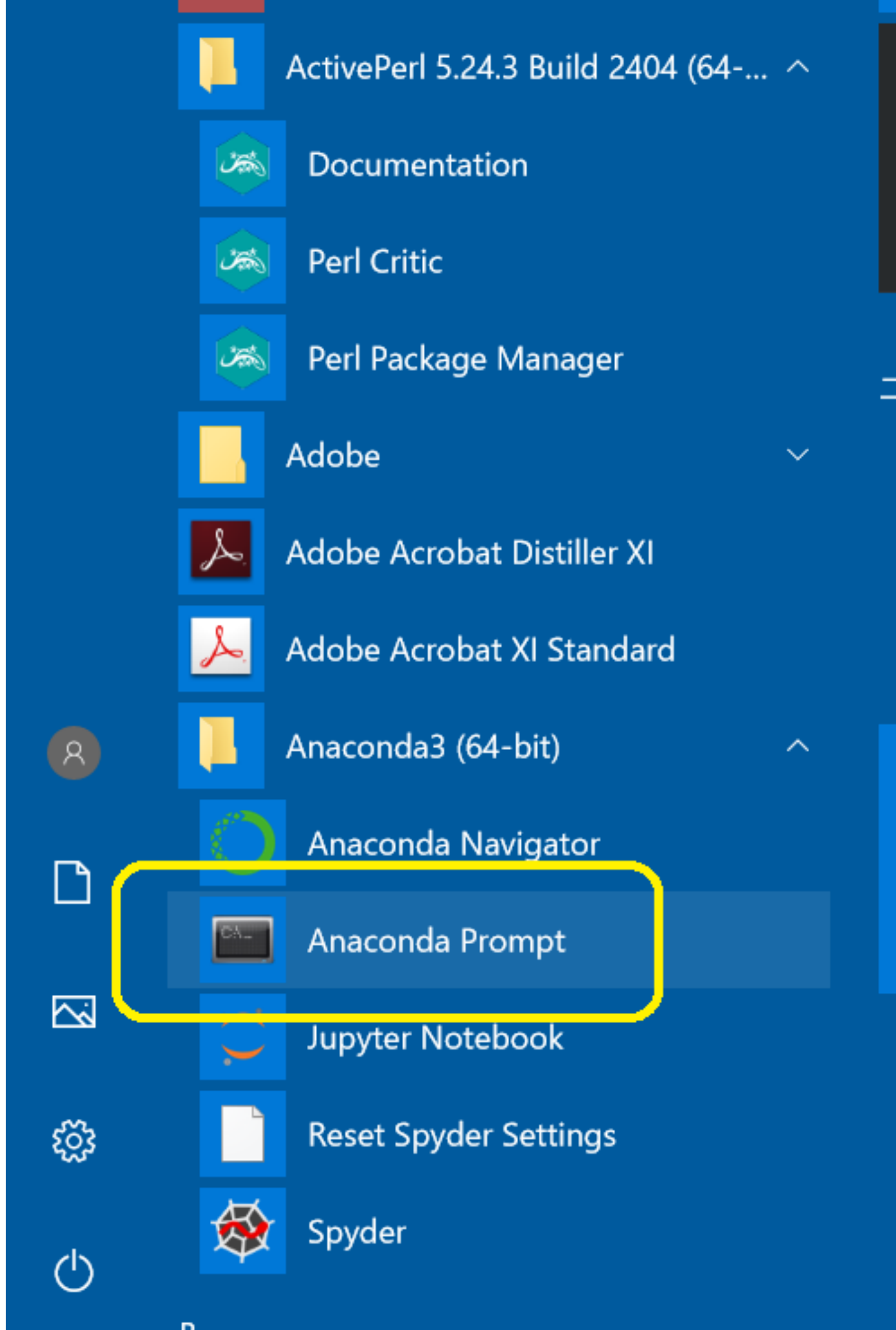
次の例では、cd コマンドで sample.py が存在するディレクトリ materials/5/に移動し、その上で sample.py を実行しています。

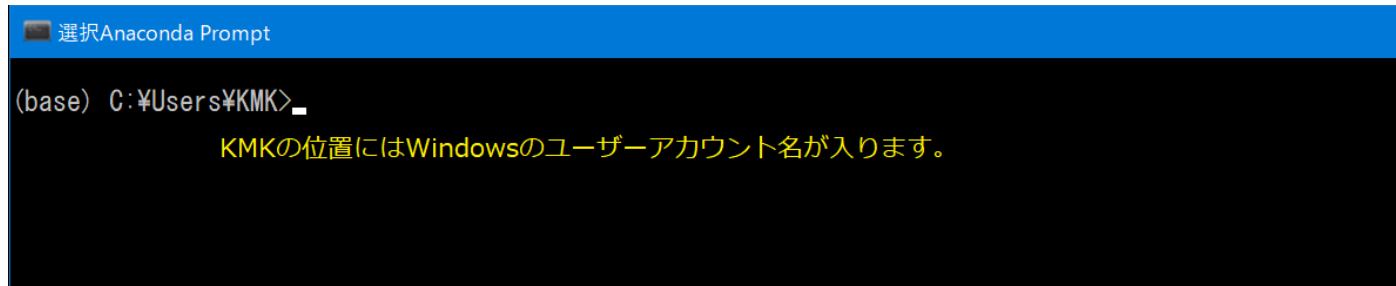


Jupyter Notebook 上で開かれるターミナルは、環境によって違います。デフォルトでは、Windows 10 では PowerShell が起動し、macOS ならば bash が起動するでしょう。

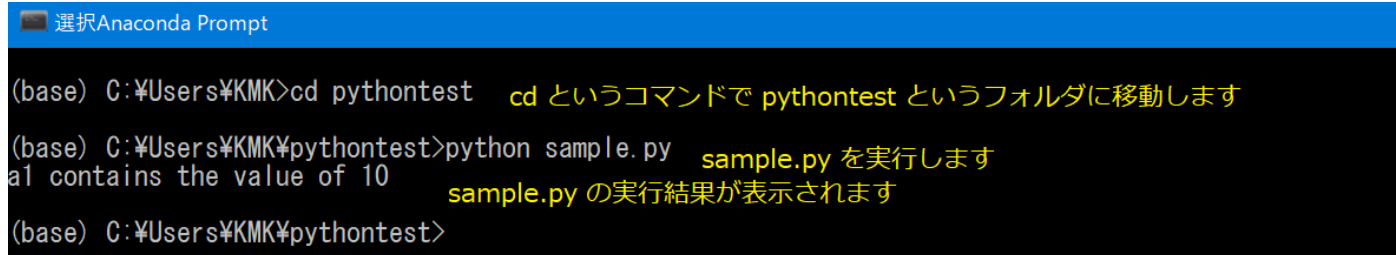
17.1.2 Windows での実行方法

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



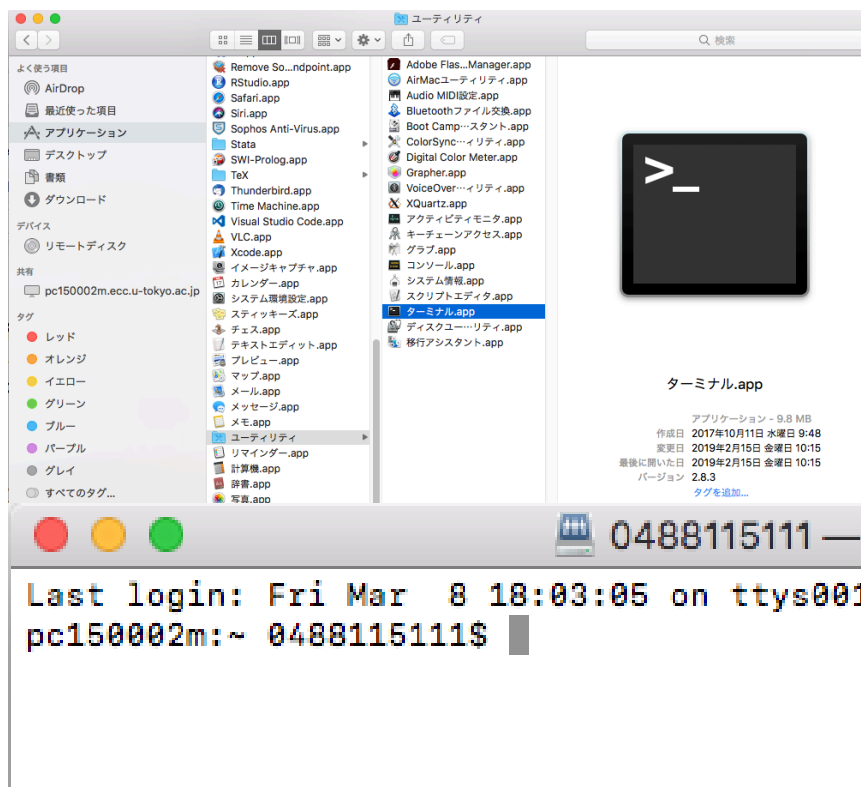


Windows のユーザーアカウント名のついたフォルダ（画像では、KMK）の中に pythontest というフォルダを作成し、その中に sample.py を格納した場合の実行例を示します。



17.1.3 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）



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。

17.2 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。

例えば、`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。例えば、以下の様に 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
```

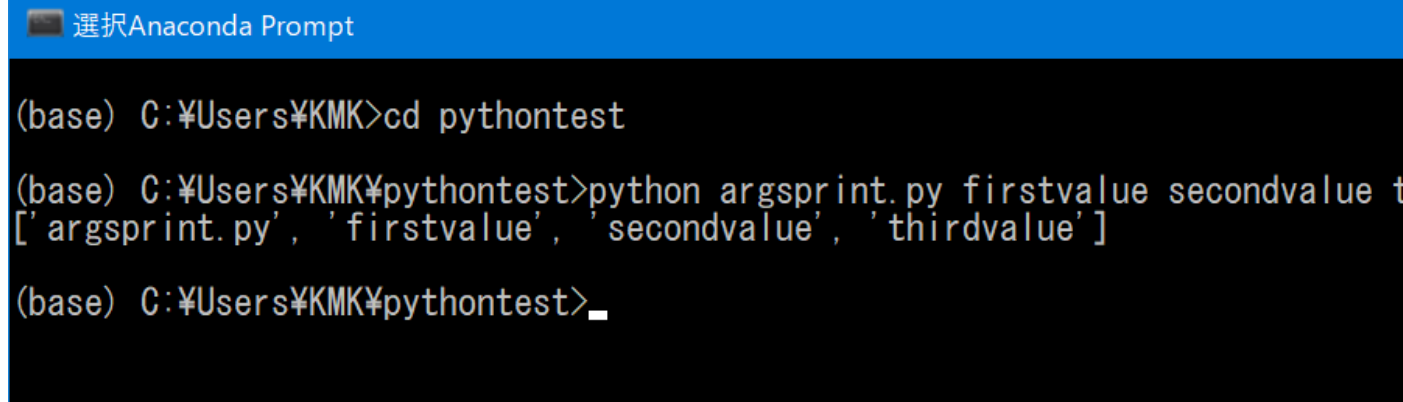
このとき、この 3 つの文字列が `argsprint.py` にコマンドライン引数として与えられることになります。

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` を次の様なコードからなるファイルとしましょう。

```
import sys
print(sys.argv) # リスト sys.argv の中身を印字
```

この様な `argsprint.py` を先の例の様に実行すると、以下の画像の様な結果が得られます。リスト `sys`



17.2.1 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よ。

17.2.2 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。

17.2.3 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` を作成せよ。

17.2.4 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` を作成せよ。

17.2.5 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` を作成せよ。

例えば、次のように実行すると、

```
>>> python sum.py 1 2 3
```

6 と印字される。

尚、コマンドライン引数は文字列型であることに注意せよ。

```
[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'
```

17.3 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 `import` 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、`import` 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、`import` 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。例えば、次に示す `factorial.py` モジュールを考えます。

```
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 `import` したときに、`import` 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・印字しようとしてしまいます。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。例えば、次のように記述すると、

```
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がひとまとめにできて、保守しやすくなります。

17.4 ▲ソースファイル先頭部分にある宣言

17.4.1 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、歴史的事情からシフト 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。

尚、*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。

17.4.2 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 Python 3 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、例えば、次のようになります。

```
#!/usr/bin/env python3
# -*- coding: shift_jis -*-
```

17.5 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])
```

```
/Users/ikob/anaconda3/lib/python3.7/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)
```

```
2
```

```
[7]: #sum.py
import sys
s = 0
for v in sys.argv[1:]:
    s += int(v)
print(s)
```

```
-----
ValueError                                Traceback (most recent call last)
<ipython-input-7-cb059ca7b2d6> in <module>
      3 s = 0
      4 for v in sys.argv[1:]:
--> 5     s += int(v)
      6 print(s)

ValueError: invalid literal for int() with base 10: '-f'
```

CHAPTER 18

5-2. NumPy

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 `import` 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。

18.1 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` 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。第 6 回教材で改めて説明しますが、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。

18.1.1 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]
```

18.1.2 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ず、リストの配列と見做されます。本教材では、このようなデータは扱いません。

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]
```

尚、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。

18.1.3 ▲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。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）

18.2 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。

18.2.1 arange

“`numpy.arange()`” は、組込み関数 `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]
```

18.2.2 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.          ]
```

18.2.3 zeros, ones

“`numpy.zeros()`” 関数は、0 からなる 1 次元配列を生成します。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.]]
```

18.2.4 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

[0.54002731 0.70762957 0.11291567 0.439479 ]
[[0.51470525 0.12223621 0.23297635]
 [0.31899735 0.9344044  0.71785833]]
```

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。

18.3 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()` を用いて定義せよ。

例えば、`range_square_matrix(3)` は、

```
[[0 1 2]
 [1 2 3]
 [2 3 4]]
```

と印字されるような 2 次元配列を返す。

18.4 配列要素の操作

18.4.1 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アクセスできます。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。

```
[21]: a = np.arange(3)
      print(a)

[0 1 2]
```

```
[22]: a[0]
```

```
[22]: 0
```

```
[23]: a[-1]
```

```
[23]: 2
```

```
[24]: 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。

```
[25]: a = np.arange(6).reshape(2,3)
      print(a)
```

```
[[0 1 2]
 [3 4 5]]
```

```
[26]: a[1,2] # 行と列のインデックスをまとめて指定
```

```
[26]: 5
```

```
[27]: a[1,2] = 6 # 要素への代入もできる
print(a)
```

```
[[0 1 2]
 [3 4 6]]
```

18.4.2 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。

```
[28]: 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]
[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。

```
[29]: 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]
```

このような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、インデックスアクセスと同様に、高い次元のスライスから順に並べて指定します。

```
[30]: 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。

18.4.3 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。

```
[31]: 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。

```
[32]: 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。

```
[33]: 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
```

```
[34]: for idx, e in np.ndenumerate(np.arange(3)):  
      print(idx, e)
```

```
(0,) 0  
(1,) 1  
(2,) 2
```

18.5 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、効率が良いです。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。

18.5.1 配列のスカラー演算

配列とスカラーとの算術演算を記述すると、要素毎のスカラー演算となります。演算結果として、新しい配列が返ります。

```
[35]: 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
```

(continues on next page)

(continued from previous page)

```
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) # 左側がスカラーでもよい
print(1 - a) # 左側がスカラーでもよい
print(2 * a) # 左側がスカラーでもよい
b = a + 1
print(1 / b) # 左側がスカラーでもよい
print(9 // b) # 左側がスカラーでもよい

[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]
```

18.5.2 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。

```
[36]: 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]]
[[-1 -1]
 [-1 -1]]
[[ 0  2]
 [ 6 12]]
[[0.          0.5          ]
 [0.66666667 0.75          ]]
[[0 1]
 [2 2]]
[[0 1]
 [2 3]]
```

(continues on next page)

(continued from previous page)

```
[[ 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。

18.5.3 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。

```
[37]: 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)) # スカラー (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。

18.6 よく使われる配列操作

18.6.1 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。

```
[38]: np.dot(np.arange(4), np.arange(1,5)) # 0*1 + 1*2 + 2*3 + 3*4
[38]: 20
```

2次元配列同士だと、行列乗算と等しいです。

```
[39]: # [[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]]
```

18.6.2 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、組込み関数 `sorted()` の配列版です。

```
[40]: 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。

```
[41]: a = np.array([3, 4, -1, 0, 2])
      print(a)
      a.sort()
      print(a)

[ 3  4 -1  0  2]
[-1  0  2  3  4]
```

18.6.3 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。

```
[42]: 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。

18.7 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。

```
[43]: 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。

```
0.000000000000000000e+00
1.000000000000000000e+00
2.000000000000000000e+00
```

2次元配列は、列が空白区切りで保存されます

```
[44]: 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。

```
0.000000000000000000e+00 1.000000000000000000e+00 2.000000000000000000e+00
3.000000000000000000e+00 4.000000000000000000e+00 5.000000000000000000e+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。

```
[45]: 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。

18.8 ▲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。

```
[46]: 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。

```
[47]: 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 に書き換え  
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、第 7 回教材で扱う Pandas ライブラリでも利用されます。

18.9 ▲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。

```
[48]: 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。

```
[49]: 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
```

```
[49]: 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](#)を参照して下さい。

18.10 練習の解答

```
[50]: def arange_square_matrix(n):  
       return np.array([np.arange(i, n+i) for i in range(n)])
```

CHAPTER 19

6-1. 関数プログラミング

関数プログラミングについて説明します。

参考 - <https://docs.python.org/ja/3/howto/functional.html> - <https://docs.python.org/ja/3/library/functions.html> - <https://docs.python.org/ja/3/library/itertools.html> - <https://docs.python.org/ja/3/library/functools.html>

関数プログラミング (**functional programming**) とは、プログラムを (数学的な) 関数の合成で記述するプログラミングスタイルです。処理を操作列と考えて命令的に記述するのではなく、処理をデータ変換を行う関数に分解して記述します。これを Python で行うときに重要になるのは、高階関数とイテレータです。したがって、Python における関数プログラミングとは、高階関数とイテレータを使いこなすことだと考えても、ほぼ差し支えありません。

19.1 高階関数

高階関数 (**higher-order function**) とは、値として関数を受け取ったり返したりする関数のことです。Python における関数はオブジェクトなので、定義した関数をそのまま渡したり返したりすることができます。

```
[1]: def inc(x):  
      return x+1  
  
      def twice(f, x):  
          return f(f(x))  
  
      def genfunc():  
          return inc  
  
      twice(genfunc(), 0)
```

```
[1]: 2
```

ここで、`twice()` は関数を受け取り、`genfunc()` は関数を返しているので、どちらも高階関数です。

組込み関数などのよく使われる関数には、関数を受け取る高階関数が多いです。そのような高階関数を使うときには、上に示した `inc()` のように、小さい関数を渡したくなるのがよくあります。この時に便利なのが、ラムダ式 (または無名関数) です。例えば、

```
lambda x: x+1
```

は、`inc()` と等価な関数オブジェクトと返します。一般に、

```
f = lambda 引数: 式
```

は

```
def f(引数):  
    return 式
```

と同等です。

ラムダ式は、`def` 記法による関数定義に比べて記述に制限が加わりますが、関数呼出しの引数の位置に関数定義を記述できるという利点があります。例えば、`twice(inc, 0)` の代わりに `twice(lambda x: x+1, 0)` と呼び出すなら、わざわざ `inc()` を定義しなくて済みます。このように、ラムダ式を有効活用すると、全体のコードが簡潔で読みやすくなります。

19.1.1 sorted

2-2 で、整列（ソート）されたリストを返す関数 “`sorted()`” を導入しました。

```
[2]: sorted([1, 3, -2, 0])
```

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

実は、`sorted()` は `key` 引数に関数を取れる高階関数です。 `key` 引数は、各要素を比較に使われる値に変換する関数を取ります。例えば、絶対値の昇順で整列したい場合、絶対値関数 `abs()` を `key` 引数に渡せばよいです。

```
[3]: sorted([1, 3, -2, 0], key=abs)
```

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

練習

文字列のキーと数値の値のペアのリスト `ls` があるとする。例えば、`ls = [('A', 1), ('B', 3), ('C', -1), ('D', 0)]`。このリスト `ls` を、値の降順で整列するように、`sorted()` を呼び出せ。

19.1.2 max, min

組込み関数 “`max()`” と “`min()`” は、それぞれ最大の要素と最小の要素を返す関数です。

```
[4]: max([1, 3, -2, 0])
```

```
[4]: 3
```

```
[5]: min([1, 3, -2, 0])
```

```
[5]: -2
```

`sorted()` と同様に、どちらも `key` 引数に、比較に使われる値に変換する関数を取れます。したがって、例えば `abs()` を渡せば、絶対値が最大と最小となる要素を返します。

```
[6]: max([1, -3, -2, 0], key=abs)
```

```
[6]: -3
```

```
[7]: min([1, -3, -2, 0], key=abs)
```

```
[7]: 0
```

練習

リスト（例えば `[1, 3, -2, 0]`）の最小の要素を返すように、`max()` を用いよ。ただし、リストの各要素は数値だと仮定して良い。

19.1.3 ▲ reduce

3-3 や 3-4 で、組込み関数の `sum()` を紹介しました。これは総和を返す組込み関数でした。

```
[8]: sum([-1, -3, 2, 4])
```

```
[8]: 2
```

総和があるならば、総乗を取るような組込み関数があるかということ、ありません。しかし、`functools` モジュールには、総和や総乗を一般化した関数 “`functools.reduce()`” があります。

`reduce()` は、第 1 引数にとる 2 引数関数を使って、第 2 引数を前から順に畳み込む関数です。前から順に畳み込むとは、具体的には、第 1 引数が `f` で、第 2 引数が `[-1, -3, 2, 4]` のとき、`f(f(f(-1, -3), 2), 4)` という演算です。したがって、総和も総乗も次のように表現できます。

```
[9]: import functools
     funtools.reduce(lambda x,y: x+y, [-1,-3,2,4])
```

```
[9]: 2
```

```
[10]: funtools.reduce(lambda x,y: x*y, [-1,-3,2,4])
```

```
[10]: 24
```

`sum()` の第 2 引数に初期値を取れるように、`reduce()` も第 3 引数に初期値を取れます。

```
[11]: sum([-1,-3,2,4], 10)
```

```
[11]: 12
```

```
[12]: funtools.reduce(lambda x,y: x*y, [-1,-3,2,4], 10)
```

```
[12]: 240
```

初期値は、第 2 引数の要素とは異なるデータ型を取ることを許されます。与える関数の第 1 引数と第 2 引数も、異なるデータ型を取ることを許されます。したがって、巧妙に初期値と引数関数を設定することで、様々な計算を `reduce()` で実現できます。

```
[13]: def enumstep(x, y):
     i, ls = x
     ls.append((i,y))
     return (i + 1, ls)
     funtools.reduce(enumstep, 'ACDB', (0, [])) [1]
```

```
[13]: [(0, 'A'), (1, 'C'), (2, 'D'), (3, 'B')]
```

ただし、このように複雑になると、素直に `for` 文で書いた方が見やすくなることも多々あります。`reduce()` の利用には、バランス感覚が重要です。

19.2 イテレータ

前述の `sorted()`、`min()`、`max()` などは、リストとタプルの両方を同様に渡して処理することができます。`for` 文で走査（全要素を訪問）するときも、リストとタプルは同様に扱えます。何故、異なるものを同じように扱えるのでしょうか。それはイテレータという仕掛けがあるからです。

イテレータ とは、要素の集まりを走査するオブジェクトです。組込み関数 “`iter()`” によって構築し、組込み関数 “`next()`” によって要素を取り出します。

```
[14]: it = iter([1,2]) # [1,2] のイテレータを構築
      next(it)
```

```
[14]: 1
```

```
[15]: next(it)
```

```
[15]: 2
```

```
[16]: next(it)
```

```
-----
StopIteration                                Traceback (most recent call last)
<ipython-input-16-bc1ab118995a> in <module>
----> 1 next(it)

StopIteration:
```

`next()` は、返す要素がないとき（走査の終了時）に、“**StopIteration**” という例外を投げます。
イテレータを `iter()` に渡すと、それ自身が返されます。

```
[17]: it = iter([1,2])
      it is iter(it)
```

```
[17]: True
```

イテレータも `for` 文で反復処理できます。

```
[18]: it = iter([1,2])
      for x in it:
          print(x)
```

```
1
2
```

`for` 文では、`StopIteration` を検知して反復を自動的に終了しています。

ここで重要なのは、リストやタプルなどのデータ構造は、全てイテレータを経由して走査するということです。つまり、リストやタプルなど異なるものから、イテレータという同様に操作できるオブジェクトを構築して利用することで、同じように走査できるようになったわけです。

```
[19]: it = iter((1,2)) # (1,2) のイテレータを構築
      for x in it:
          print(x)
```

```
1
2
```

ここで注意すべきことは、イテレータは1回の走査にしか使えない、使い捨てのオブジェクトだということです。同じデータ構造を複数回走査したいときには、走査する度にイテレータを構築する必要があります。

```
[20]: next(it) # (1,2) のイテレータ it は走査が終了したまま
```

```
-----
StopIteration                                Traceback (most recent call last)
<ipython-input-20-aacc4a5859b3> in <module>
----> 1 next(it) # (1,2) のイテレータ it は走査が終了したまま

StopIteration:
```

```
[21]: for x in it:
      print('これは呼び出されない')
```

ここで憶えておくべきことは、イテレータ自体は元のデータ構造をコピーしないということです。要素を1つ1つ訪問するという反復処理を実現するオブジェクトであり、`iter()` や `next()` は、構築元のデータ構造のサイズ(要素数)に依存しないコストで通常実装されます。例えば、リストの先頭要素を除いた残りの部分を走査するとき、

```
for x in ls[1:]:  
    何かの処理
```

と残りの部分をスライスとしてコピーするよりも、

```
it = iter(ls)  
next(it) # 先頭要素を捨てる  
for x in it:  
    何かの処理
```

とイテレータで直接走査の方が効率的です。これは、サイズが小さいデータ構造を扱うときには問題になりませんが、大きいものを扱うときには気を付けるべきことです。

ここまで、イテレータの使い方は、`next()` で要素を取り出すか、`for` 文で反復するだけでしたが、実は `sorted()`、`max()`、`min()` などに渡すことができます。文字列、タプル、リスト、辞書、イテレータなど、イテレータを介して要素を走査可能なオブジェクトを総称してイテラブルと呼びます。`sorted()`・`max()`・`min()` は、イテラブルを受け取る関数です。

イテレータ `it` の中身を印字して確認したいときには、`print(*it)` と、イテレータを展開して可変長引数として `print()` を呼び出すのが簡潔で便利です。ただし、中身を確認した後の `it` はもう利用できないこと、大量の要素を生成するイテレータには不向きであることに留意してください。

```
[22]: it = iter(range(4))  
next(it) # 先頭の 0 を捨てる  
print(*it)  
1 2 3
```

イテレータとイテラブルの詳細については、付録 6-iterable を参照してください。

19.2.1 練習

与えられたイテラブルの先頭要素を除いた残りの部分の最大値を返す関数 `tailmax()` を、イテレータを使って、`for` 文を使わずに、上の例に倣って効率的に実装せよ。

19.3 イテレータを生成する関数

Python の組み込み関数や標準ライブラリには、イテレータを返す関数が数多くあります。その中には、関数を受け取る高階関数もあります。イテレータを生成・消費する関数の適用に分解してプログラムを記述することで、イテレータを介した関数プログラミングが行えるようになります。

19.3.1 map

組み込み関数の“`map()`”は、第1引数に取った関数を、第2引数に取ったイテラブルの各要素に適用した結果を走査するイテレータを返します。

```
[23]: print(*map(lambda x: x + 1, [1,-3,2,0]))  
2 -2 3 1
```

より正確には、第1引数には、 n 引数関数 ($n \geq 1$) を取ることができ、第2引数以降に n 個のイテラブルを渡すことができます。この時、一番小さい要素数に合わせて、結果のイテレータは切り詰められます。

```
[24]: # 異なるイテラブルを受け取れる
print(*map(lambda x,y: x + y, [1,-3,2,0], (4,7,-6,5)))

5 4 -4 5
```

```
[25]: # 結果のイテレータが切り詰められる
print(*map(lambda x,y: x + y, range(1,10,2), range(1000000)))

1 4 7 10 13
```

map() とラムダ式を組み合わせるよりも、3-3 で紹介した内包表現（ジェネレータ式を含む）の方が簡潔になることも少なくありません。

```
[26]: print([x + 1 for x in [1,-3,2,0]]) # リスト内包

[2, -2, 3, 1]
```

```
[27]: print(*(x + 1 for x in [1,-3,2,0])) # ジェネレータ式（イテレータを返す）

2 -2 3 1
```

一方、既定義の関数を引数に渡すときには、map() の方が簡潔になります。その時々で、内包表記と比べてみて、より分かりやすい方を採用しましょう。

練習

第 1 引数で与えられた要素数まで、第 2 引数に与えられたイテラブルを走査するイテレータを返す関数 take() を、for 文を使わずに、map() と range() を用いて定義せよ。例えば、take(2, 'ACDB') は、AC を走査するイテレータを返す。

19.3.2 filter

組込み関数“filter()”は、第 1 引数に単項述語（真理値を返す 1 引数関数）を、第 2 引数にイテラブルを取り、その単項述語を真にする要素だけを順に生成するイテレータを返します。

```
[28]: print(*filter(lambda x: x % 2 == 0, range(8)))

0 2 4 6
```

filter() は、制御構造の観点で見ると、continue 文によるスキップを含んだ for 文に対応します。continue を含んだ for 文を使うときには、代わりに filter() を使うことができないか考えてみると良いでしょう。

map() と同様に、素直に内包表現に書き換えられます。

```
[29]: print([x for x in range(8) if x % 2 == 0]) # リスト内包

[0, 2, 4, 6]
```

```
[30]: print(*(x for x in range(8) if x % 2 == 0)) # ジェネレータ式（イテレータを返す）

0 2 4 6
```

filter() とラムダ式を組み合わせるときや、filter() と map() を組み合わせるときは、内包表記を使った方が簡潔で分かりやすくなることが多いです。

練習

ファイルオブジェクト（行単位のイテレータ）f を取って、その中の非空白行を順に生成するイテレータを返す関数 compactlines() を、for 文を使わずに、filter() を用いて定義せよ。

ヒント：s が空白文字（改行を含む）からなる文字列であるとき、s.strip() は空文字列を返す。

19.3.3 enumerate

3-2 で紹介した組込み関数の “`enumerate()`” は、実はイテレータを返します。

```
[31]: it = enumerate('ACDB')
      print(it) # リストやタプルではない
      <enumerate object at 0x10efd52d0>
```

```
[32]: print(*it)
      (0, 'A') (1, 'C') (2, 'D') (3, 'B')
```

つまり、for 文や内包表現に限定されず、イテレータを消費する関数と共に使えます。

そして、`enumerate()` は、イテラブルを引数に取ります。つまり、イテレータも渡せます。したがって、計算結果のイテレータの各要素に番号付けすることにも利用できます。

`enumerate()` の第 2 引数には番号付けの初期値を渡せます。

```
[33]: print(*enumerate('ACDB', 1))
      (1, 'A') (2, 'C') (3, 'D') (4, 'B')
```

`enumerate()` は、番号付けという汎用的なデータ変換を行う関数だったのです。

19.3.4 ▲ zip

組込み関数 “`zip()`” は、`map()` の第 1 引数の関数が、タプル構築に固定されたものです。

```
[34]: print(*zip(range(1,10,2), range(1000000)))
      (1, 0) (3, 1) (5, 2) (7, 3) (9, 4)
```

上に示したように、結果のイテレータの切り詰めも、同様に行われます。

`zip()` は、`map()` の特殊形でしかないのですが、`map()` を内包表記に書き換えるときや、結果のイテレータを for 文で反復するときに、特に役立ちます。

```
[35]: print([x + y for x, y in zip(range(1,10,2), range(1000000))]) # リスト内包
      [1, 4, 7, 10, 13]
```

```
[36]: print(*(x + y for x, y in zip(range(1,10,2), range(1000000)))) # ジェネレータ式 (イテレータを返す)
      1 4 7 10 13
```

```
[37]: for x, y in zip(range(1,10,2), range(1000000)): # for 文で反復処理
      print(x + y)
      1
      4
      7
      10
      13
```

`map()` とラムダ式を使うか、`zip()` と内包表記を使うかは、より分かりやすい方を、その時々で判断して選択しましょう。

練習

イテラブルを取って、隣接要素対のイテレータを返す関数 `adjpairs()` を、`for` 文を使わずに `zip()` を使って定義せよ。例えば、`adjsum([1,-3,2,0])` は、 $(1, -3)$ $(-3, 2)$ $(2, 0)$ を走査するイテレータを返すことになる。

19.3.5 ▲ reversed

組込み関数 “`reversed()`” は、シーケンス（文字列、リスト、タプルなど）を受け取って、それを逆順に走査するイテレータを返します。

```
[38]: print(*reversed('ABCD'))
```

```
D C B A
```

```
[39]: print(*reversed([0,1,-2,3]))
```

```
3 -2 1 0
```

```
[40]: print(*reversed((0,1,-2,3)))
```

```
3 -2 1 0
```

`reversed()` は、イテレータを取れないことに留意してください。シーケンスの詳細については、付録 `iterable.ipynb` を参照してください。

練習

与えられたシーケンスを真ん中で折り畳んで閉じ合わせた（`zip` した）結果をイテレータで返す関数 `clamshell()` を、`for` 文を使わずに、`reversed()` と `take()` を使って定義せよ。ただし、シーケンスの長さが奇数であるとき、中央の要素は結果から除外されるものとする。例えば、`clamshell('ABCDE')` は、 (A, E) (B, D) を走査するイテレータを返す。

19.3.6 ▲ chain

`itertools` モジュールから、特に有用なものを 1 つ紹介します。“`itertools.chain()`” は、与えられた任意個のイテラブルを連結します。

```
[41]: import itertools
# 異なるデータ型を連結したイテレータ
it = itertools.chain(range(3),      # range オブジェクト
                     'abc',        # 文字列
                     list('def'),  # リスト
                     tuple('ghi'), # タプル
                     iter(range(3))) # イテレータ

print(*it)
```

```
0 1 2 a b c d e f g h i 0 1 2
```

ここで重要なのは、これら異なる 5 種類のデータ型に対する `+` の連結は、いずれもエラーになるということです。`chain()` は、それぞれのイテラブルから作り出したイテレータを連結するので、データ型の違いが問題になりません。

関連して、与えられた 1 つのイテラブルの要素を連結する関数 “`itertools.chain.from_iterable()`” もあります。

```
[42]: print(*itertools.chain.from_iterable([range(3),      # range オブジェクト
                                           'abc',          # 文字列
                                           list('def'),     # リスト
```

(continues on next page)

(continued from previous page)

```
tuple('ghi'),      # タプル
iter(range(3))])) # イテレータ

0 1 2 a b c d e f g h i 0 1 2
```

chain() が典型的によく使われる局面は、ファイルデータの操作です。

```
f = itertools.chain(open('input1'), open('input2'))
```

この f は、ファイル input1 と input2 を連結したファイルのように振る舞うイテレータです。これは新しいファイルを作らないので、実際にファイル連結を行うよりも、大変効率が良いです。また、

```
it = itertools.chain.from_iterable(f) # fはファイルのようなイテレータ
```

とすると、f の各行を、全て連結したような文字列のイテレータ it を構築できます。これは、''.join(f) で文字列結合したり、(f がファイルオブジェクトの場合) f.read() でファイル全体を読み込んで文字列を構成するよりも、大変効率が良いです。

“itertools” モジュールは、他にも有用なイテレータ生成関数を数多く提供しています。詳しくは、[公式ドキュメント](#)を参照してください。

19.4 ▲関数内関数 (クロージャ)

関数内で定義された関数（ラムダ式を含む）からは、外側のローカル変数を参照できます。

```
[43]: def outer(x):
      def inner():
          return x
      return inner
```

```
f = outer(1)
f()
```

[43]: 1

```
[44]: g = outer(2)
      g()
```

[44]: 2

グローバル変数がそうであるように、外側の関数のローカル変数についても、内側の関数からは再定義が（基本的に）できません。しかし、外側の関数では再定義できるので、注意が必要です。

```
[45]: def outer(x):
      def inner():
          return x
      x = -x # inner() が参照する変数 x を再定義
      return inner
```

```
f = outer(1)
f()
```

[45]: -1

それ故に、関数を返す高階関数を記述するときには、変数定義に対してとても慎重になる必要があります。そういう事情も含めて、関数を返す高階関数を正しく定義するのは難しいので、自分で定義せずに既存の関数を使うだけにするのが無難でしょう。

19.5 練習の解答

```
[46]: ls = [('A', 1), ('B', 3), ('C', -1), ('D', 0)]
      sorted(ls, key=lambda x: -x[1])
```

```
[46]: [('B', 3), ('A', 1), ('D', 0), ('C', -1)]
```

```
[47]: max([1, 3, -2, 0], key=lambda x: -x)
```

```
[47]: -2
```

```
[48]: def tailmax(xs):
      it = iter(xs)
      next(it) # 先頭要素を捨てる
      return max(it)

      print(tailmax([3, -4, 2, 1]) == 2)
      print(tailmax((3, -4, 2, 1)) == 2)
      print(tailmax('ACDC') == 'D')
```

```
True
True
True
```

```
[49]: def take(n, xs):
      return map(lambda x, i: x, xs, range(n))

      print(*take(2, 'ACDB'))
```

```
A C
```

```
[50]: def compactlines(f):
      return filter(lambda x: x.strip() != '', f)

      fname = 'compactlines_input.txt'
      fcontents = """

      to be

      compacted.

      """
      print(fcontents, file=open(fname, 'w'))
      print(list(compactlines(open(fname))))
```

```
['to be\n', 'compacted.\n']
```

```
[51]: def adjpairs(xs):
      it = iter(xs)
      next(it) # 1つ前にずらす
      return zip(xs, it)

      print(*adjpairs([1, -3, 2, 0]))
```

```
(1, -3) (-3, 2) (2, 0)
```

```
[52]: def clamshell(xs):
      return take(len(xs)//2, zip(xs, reversed(xs)))

      print(*clamshell('ABCDE'))
```

```
('A', 'E') ('B', 'D')
```

6-2. オブジェクト指向プログラミング

オブジェクト指向プログラミングについて説明します。

参考 - <https://docs.python.org/ja/3/tutorial/classes.html> - <https://docs.python.org/ja/3/reference/datamodel.html> - <https://docs.python.org/ja/3/library/collections.html>

これまでなんとなく用いてきた、オブジェクト指向プログラミングの諸概念を改めて説明し、クラスの簡単な使い方を説明します。

20.1 オブジェクト指向の考え方

オブジェクト指向とは何かを考えるために、まずは簡単なプログラミングの例と状況を導入します。

例えば、セミナーなどの参加者名簿を管理したいとします。学生データは、名前と ID 番号のペアで表現されるとします。

```
[1]: taro = ('東大太郎', 1234567890)
     jiro = ('東大二郎', 2345678901)
```

教員データは、名前と役職と ID 番号の 3 つ組で表現されるとします。

```
[2]: hagiya = ('萩谷昌己', '教授', 9876543210)
```

これらをまとめたリストで名簿を管理するとします。

さて、参加者に名前入りのバッジを配ることになりました。バッジには、学生の場合は名前だけ、教員の場合は名前の後に役職を付けるとします。参加者リストをもらって、バッジ（に記入する文字列）のリストを返す関数 `badgelist()` を定義することを考えます。例えば、次のように定義すれば、上に示した例については用が足ります。

```
[3]: def badgelist(participants):
     ls = []
     for x in participants:
         if len(x) == 2: # 学生 (?)
             ls.append(x[0])
         if len(x) == 3: # 教員 (?)
             ls.append(x[0] + ' ' + x[1])
     return ls
```

(continues on next page)

(continued from previous page)

```
badgelist([taro, jiro, hagiya])
```

```
[3]: ['東大太郎', '東大二郎', '萩谷昌己 教授']
```

しかし、この `badgelist()` の定義は、主に次の2点で問題があります。

- サイズ (`len()`) が2ないし3だというだけで、学生ないし教員だと決め打っている。
- 全ての参加者のバッジの作り方を知らないと言定義できない。

どちらの問題も、参加者の種類を拡張しようとしたときに困ったことが起きます。2つ組や3つ組で表現される別の種類のデータを、参加者として加えるとき、このままでは条件判定に失敗します。仮に問題が起きないように条件判定を変えとしても、参加者の種類が増えるたびに `badgelist()` の定義を変更する必要が生じます。したがって、`badgelist()` と参加者の種類を別々に管理することができません。

このような状況を、保守性 (**maintainability**) が低いとか、モジュール性 (**modularity**) が低いと言います。

これを解決する方法として、高階関数を利用するという手があります。バッジを作る関数を受け取る関数として、`badgelist()` を定義すればよいのです。ただし、学生と教員のバッジの作り方が違うことを考慮して、要素毎にバッジの作り方を与えるようにします。つまり、`badgelist()` が受け取るリストは、参加者データとバッジを作る関数の組のリストとします。

```
[4]: def badgelist(participants):  
      return [badge(x) for x, badge in participants]  
  
def studentbadge(x):  
    return x[0]  
def facultybadge(x):  
    return x[0] + ' ' + x[1]  
  
badgelist([(taro, studentbadge), (jiro, studentbadge), (hagiya, facultybadge)])
```

```
[4]: ['東大太郎', '東大二郎', '萩谷昌己 教授']
```

ここでは、学生データからバッジを作る関数 `studentbadge()` と、教員データからバッジを作る関数 `facultybadge()` を定義しています。

`badgelist()` の定義が単純化し、分かりやすくなったことが一目瞭然でしょう。実は、それだけではありません。

例えば、参加者に企業人という種類を加えることを考えます。企業人データは、名前と所属と役職の3つ組で表現され、バッジには所属と役職を名前に前置することとします。このとき、次のように、企業人データからバッジを作る関数 `industrialbadge()` を追加で定義するだけで済みます。

```
[5]: iwata = ('岩田聡', 'HAL 研究所', '代表取締役社長')  
  
def industrialbadge(x):  
    return x[1] + ' ' + x[2] + ' ' + x[0]  
  
badgelist([(taro, studentbadge), (jiro, studentbadge), (hagiya, facultybadge),  
          ↪(iwata, industrialbadge)])
```

```
[5]: ['東大太郎', '東大二郎', '萩谷昌己 教授', 'HAL 研究所 代表取締役社長 岩田聡']
```

`badgelist()` を一切変更することなく、参加者の種類を増やすことができました。また、各種バッジの作り方は、独立した関数の形になっているので、意味の区切りが明快で、独立に修正できます。

このような状況を、保守性が高いとか、モジュール性が高いと言います。

勘が良い人は気付いたでしょうが、`studentbadge()・facultybadge()・industrialbadge()` が、これまでメソッドと呼んできたものの実体です。

オブジェクト指向プログラミング (**object-oriented programming, OOP**) とは、データとそれを操作する関数 (すなわちメソッド) を結びつけるプログラミングスタイルです。オブジェクトに対する操作を、

そのオブジェクトのメソッドに任せることで、オブジェクトの実体（実装）に強く依存しない記述が可能になります。その結果として、保守性やモジュール性が高められます。

20.2 クラス定義

前述の例では、タプルとしてデータと関数を直接結びつけていました。その結果、それを受け取る高階関数の側は、与えられたオブジェクトの内部構造（どれが操作対象のデータで、どれが望んだ操作を与える関数か）を知る必要がありました。これは、保守性やモジュール性の観点で望ましくありません。この問題を解消するのが、クラスです。

クラスとは、メソッドという形で、データの種類に対して名前付きで関数を結びつける言語機能です。

Python プログラミングにおいてクラスは重要ですが、クラスを一から定義するというのは、結構大変です。有用なクラスを正しく定義するには、雑多な Python の専門知識が沢山必要になります。正直に言って、妥当なクラス設計は、平均的なプログラマには手に余るものです。

そういうわけで、日常的なプログラミングでは、既存のクラスを自分の目的に叶うように拡張する形を取ることが多いです。具体例として、前述のタプルによって表現されていた学生・教員・企業人を、tuple を拡張した Student・Faculty・Industrial クラスとして定義して、利用する例を示します。

```
[6]: class Student(tuple):
      def badge(self):
          return self[0]

      class Faculty(tuple):
          def badge(self):
              return self[0] + ' ' + self[1]

      class Industrial(tuple):
          def badge(self):
              return self[0] + ' ' + self[1] + ' ' + self[2]

      def badgelist(participants):
          return [x.badge() for x in participants]

badgelist([Student(taro), Student(jiro), Faculty(hagiya), Industrial(iwata)])

[6]: [' 東大太郎', ' 東大二郎', ' 萩谷昌己 教授', ' 岩田聡 HAL 研究所 代表取締役社長']
```

各クラス定義の内側に、badge という名前で、前述の studentbadge()・facultybadge()・industrialbadge() の定義が移動しました。データと関数を組にする式（例えば (taro, studentbadge)）は、データを引数としてクラスを関数形式で呼び出す式（例えば Student(taro)）に置き換わりました。Student(taro) は、タプル taro に対応する Student 型のデータを構築します。これは、list(taro) で、タプル taro に対応するリスト（list 型データ）を構築することと同様です。

上の例からわかるように、メソッドは単なる関数ですが、引数の渡し方が異なります。x.badge() というメソッド呼出しは、暗黙にメソッド badge() の第 1 引数として x が渡されます。メソッドの操作対象であるドットの左側のオブジェクトのことを、レシーバと呼びます。つまり、メソッドとは、レシーバを常に第 1 引数として取る関数でしかありません。

レシーバが渡される第 1 引数には、Python の慣習として、self という変数名が選ばれます。しかし、self という変数名が必要ではなく、実はメソッド定義に def 構文を使う必要もありません。既存の関数を、クラス属性（クラス内の変数）として定義すれば、メソッドとして機能します。例えば、次のクラス定義は、上の例と同等です。

```
[7]: class Student(tuple):
      badge = studentbadge

      class Faculty(tuple):
          badge = facultybadge
```

(continues on next page)

(continued from previous page)

```
class Industrial(tuple):
    badge = industrialbadge

badgelist([Student(taro), Student(jiro), Faculty(hagiya), Industrial(iwata)])
```

```
[7]: ['東大太郎', '東大二郎', '萩谷昌己 教授', 'HAL 研究所 代表取締役社長 岩田聡']
```

以上のように、既存のクラスを拡張して新しいクラスを定義することを、**継承 (inheritance)** と呼びます。上の例では、`tuple` を親クラスと呼び、`Student`・`Faculty`・`Industrial` を子クラスと呼びます。子クラスは、親クラスの全てのメソッドを、共有する形で引き継ぎます。

`badge()` メソッドの例は、一見すると人工的な例のように見えますが、身近で実用的なものです。実は、Python における全てオブジェクトは“`__str__()`”というメソッドを持っており、それはオブジェクトを `str()` で文字列に変換するときに呼び出されます。この文字列変換は、例えば `print()` が引数を印字するときに利用されます。つまり、任意のオブジェクトを単に `print` するだけで、読みやすく印字されるのは、`badge()` の例と同じく、クラスの恩恵だったのです。

20.2.1 練習

`Student`・`Faculty`・`Industrial` の `badge()` メソッドを、`__str__()` メソッドに名前替える前と後で、各参加者を `print` せよ。

20.3 オーバーライド

先の練習の結果からわかる通り、子クラスのメソッド定義は、親クラスで既に定義されているメソッドを上書きします。これをメソッドの**オーバーライド (override)** と呼びます。これによって、継承によってメソッド実装の大部分を親クラスと共通化しつつ、一部のメソッドだけ子クラスで変更して振舞いを変えるという拡張が可能になります。

上書きと言っても、メソッド呼出しで子クラスのメソッド定義が優先されるというだけで、親クラス自体が変更される訳ではありません。組込み関数“`super()`”の返すオブジェクト越しに、親のメソッド定義も呼び出すことができます。これを利用することで、親クラスのメソッドに処理を追加するような形で、子クラスのメソッドを定義できます。次に示す `Counter` クラスは、与えられたキーが存在しないときに 0 を返すように辞書を拡張したクラスです。

```
[8]: class Counter(dict):
      def __getitem__(self, k):
          if k in self: # キー k に対応する値が存在するとき
              return super().__getitem__(k) # k に対応する値を返す
          else:
              return 0

      c = Counter()
      c['A'] = c['A'] + 1 # 右辺の c['A'] では 0 が返る
      c['A'] += 1       # 上の代入文と同等
      c['B'] += 1
      c
```

```
[8]: {'A': 2, 'B': 1}
```

```
[9]: c['C']
```

```
[9]: 0
```

“`__getitem__()`”は特殊メソッドであり、`x[k]` という式は、`x.__getitem__(k)` というメソッド呼出しとして解釈されます。`super().メソッド名()` という記法で、親クラスのメソッドを呼び出せます。注意すべきは、`super()` が返すオブジェクト自体は `Counter` でも `dict` でもないので、`super()[k]` で `dict`

クラスの `__getitem__()` は呼び出されないことです。もし、`self[k]` とすると、`__getitem__()` の再帰呼出しにより無限ループに陥ります。

実は、`dict` クラスの `__getitem__()` メソッドでは、与えられたキーが存在しないときに、特殊メソッド “`__missing__()`” が呼び出されるように定義されています。したがって、次の `Counter` の定義は、上に示した定義と同等です。

```
[10]: class Counter(dict):
      def __missing__(self, k):
          return 0
```

```
c = Counter()
c['A'] += 1
c['B'] += 1
c
```

```
[10]: {'A': 1, 'B': 1}
```

このように、単にオーバーライド（もしくは特定のメソッドを実装）するだけで、既存のメソッドの振舞いをカスタマイズできるように設計されているクラスは、少なくありません。これが、クラスに基くオブジェクト指向設計の恩恵です。

尚、この `Counter` クラスは、単純な拡張ではありますが、実用的な例です。ヒストグラムや確率分布など、キーに対する統計量を保持する場合には、キーが存在しないときには `0` が返るのが自然だからです。そして、この `Counter` を少し機能拡張したものが、`collections` モジュール内の “`collections.Counter`” クラスとして提供されています。実用上は、そちらを利用するのが良いでしょう。

20.3.1 練習

上に示した `Counter` クラスでは、キー `k` に対応する値が `0` になっても、項目は削除されない。

```
[11]: c = Counter()
      c['A'] += 1
      c['B'] += 1
      c['A'] -= 1
      c
```

```
[11]: {'A': 0, 'B': 1}
```

さて、特殊メソッド “`__setitem__()`” は、`x[k] = v` という代入文に対応して、`x.__setitem__(k, v)` と呼び出される。`Counter` に対して、`__setitem__()` を適切に定義することで、キーに対応する値が `0` になった項目が、自動的に削除されるようにせよ。例えば、上の例では、最終的な `c` の値は `{B: 1}` になる。

20.4 オブジェクトとクラス

これまで、「オブジェクト」という用語をカジュアルに使ってきましたが、ここでは改めてその意味を定義します。Python では、《プログラム中で値として操作可能な全てのもの》をオブジェクトと呼びます。

《値として操作可能》という部分が重要です。「全てがオブジェクト」と言われることがありますが、それは正確ではありません。例えば、`if` 文は、値として操作可能ではないので、オブジェクトではありません。また、式は値として操作可能ではないのでオブジェクトではないですが、式の評価結果は値なのでオブジェクトです。同様に、関数定義 (`def f(): ...`) は値として操作可能ではないのでオブジェクトではないですが、それから得られる関数 (`f`) はオブジェクトです。

《値として操作可能》というやや仰々しい定義を憶えなくても、卑近な判別法があります。前述のように、全てのオブジェクトは `__str__()` メソッドを持つので、`print` 可能なものがオブジェクトと考えれば十分です。

これまで「データ型」とか「型」と述べていたものは、Python では全てクラスで表現されています。Python に限らず、プログラミング言語において型というと、値の種類を意味します。クラスという言葉機能は、

データの種類に関数を結びつけるものですが、結果として生じる個々のクラスは、関数が結びつけられたデータの種類を意味します。したがって、Python では、型とクラスは同義と見做して差し支えないです。

任意のオブジェクトの型は、組み込み関数 `“type()”` で取得できます。

```
[12]: type('hello')
```

```
[12]: str
```

```
[13]: type(0)
```

```
[13]: int
```

```
[14]: type((1,))
```

```
[14]: tuple
```

オブジェクト指向プログラミング一般の文脈では、クラスはオブジェクトを生成する機能も含意します。実際、`tuple` クラスは、それを関数のように呼び出すことで、タプルを構成できます。

```
[15]: tuple([1, 'a'])
```

```
[15]: (1, 'a')
```

このとき、クラスは、それが表現する型に含まれる具体例を生成していると考えることができます。したがって、あるクラス `A` が生成したオブジェクトのことを、`A` のインスタンスと呼ぶこともあります。この言葉遣いに慣れない人は、「クラス `A` のインスタンス」を「`A` 型の値」と読み替えても差し支えありません。Python では、オブジェクトがあるクラスのインスタンスかどうかを判定する組み込み関数 `“isinstance()”` が提供されています。

```
[16]: (isinstance(1, int),
      isinstance(1, tuple),
      isinstance([1, 'a'], list),
      isinstance([1, 'a'], tuple))
```

```
[16]: (True, False, True, False)
```

クラスがインスタンスを構築する際に、インスタンスの初期化のために呼ばれる特殊なメソッドのことを、`コンストラクタ` と呼びます。`str・int・list・tuple・Student` 等は、クラスですが、それを関数形式で呼び出したときには、実際にはそのクラスのコンストラクタが呼び出されます。

コンストラクタは単なるメソッドなので、親クラスから引き継がれます。例えば、`Student` のコンストラクタは、`tuple` のコンストラクタです。なので、`Student(taro)` で構築されるオブジェクトは、タプルの `taro` と同様に初期化されます。しかし、`Student` は、`tuple` のメソッドに加えて、`badge()` メソッド（実体は `studentbadge()`）も持ちます。結果として、`Student(taro)` は、`taro` に `badge()` メソッドを加えたオブジェクトを構築することになり、`(taro, studentbadge)` に相当する計算になっていました。

さて、ここ思い出してほしいのは、`print` 可能なものは全てオブジェクトだということです。先ほど、`type()` を使って取得したクラスは、印字されました。つまり、クラスもまたオブジェクトです。

型（クラス）が値（オブジェクト）であるというのは、プログラミング言語としての Python の大きな特徴です。

20.5 ▲名前付きタプル

タプルとは、変更不可なデータ列であり、その要素は、位置（インデックス）でしか区別されませんでした。しかし、辞書のように、それぞれの要素に名前がついていると便利なことがあります。それを実現するのが、名前付きタプルです。`collections` モジュール内の関数 `“collections.namedtuple()”` は、引数に応じた名前付きタプルのクラスオブジェクトを生成します。

例として、学生を表す名前付きタプルを定義します。

```
[17]: from collections import namedtuple #名前付きタプル型を生成する関数
Student = namedtuple('Student', ('name', 'id')) #name, id という名前付き要素 (属性) をもつ
Student クラス
taro = Student(' 東大太郎', 1234567890)
taro
```

```
[17]: Student(name=' 東大太郎', id=1234567890)
```

```
[18]: Student(id=1234567890, name=' 東大太郎') # キーワード引数でも構築できる
```

```
[18]: Student(name=' 東大太郎', id=1234567890)
```

```
[19]: taro.name # 名前によるアクセス (属性アクセス)
```

```
[19]: ' 東大太郎'
```

```
[20]: taro[0] # インデックスアクセス
```

```
[20]: ' 東大太郎'
```

単に文字列と整数のタプルよりも、名前付きタプルで構成することで、要素の意味がハッキリと分かります。そして、属性アクセスを使えば、データの中で何に着目しているのかが、明示されます。少し大袈裟に言えば、プログラムコードが、ドキュメントのように読めるようになります。

20.5.1 名前付きタプルの用途

さて、名前付きタプルが、単に名前でアクセスできるだけならば、初めから辞書を使えばよいのではと思うことでしょう。しかし、名前付きタプルの嬉しさは、それがタプルとして使えるということにあります。

```
[21]: x, y = taro
print(x, y)

東大太郎 1234567890
```

```
[22]: list(taro)
```

```
[22]: [' 東大太郎', 1234567890]
```

変更不可能オブジェクトなので、辞書のキーにできます。

```
[23]: d = {}
d[taro] = 0
d
```

```
[23]: {Student(name=' 東大太郎', id=1234567890): 0}
```

属性の再定義はできませんが、

```
[24]: taro.name = ' 京大太郎'
```

```
-----
AttributeError                                Traceback (most recent call last)
<ipython-input-24-20bbb0e2b06c> in <module>
--> 1 taro.name = ' 京大太郎'

AttributeError: can't set attribute
```

`__replace()` メソッドによって、属性値を入れ替えた新しい名前付きタプルを生成できます。

```
[25]: taro.__replace(name=' 京大太郎')
```

```
[25]: Student(name=' 京大太郎', id=1234567890)
```

`namedtuple()` が返すオブジェクトは、単なるクラスなので、継承によって拡張することができます。

```
[26]: class Student(namedtuple('Student', ('name', 'id'))):  
        def badge(self):  
            return self.name  
  
        taro = Student(' 東大太郎', 1234567890)  
        taro
```

```
[26]: Student(name=' 東大太郎', id=1234567890)
```

```
[27]: taro.badge()
```

```
[27]: ' 東大太郎'
```

`namedtuple()` が返した `Student` クラスを、同名の `Student` クラスが継承しています。これは、クラスにおけるデータ部分を、名前付きタプルで表現するイディオムです。先に示した `tuple` を拡張した場合に比べて、メソッド定義も、印字される文字列も、自己説明的で分かりやすいです。

更新不可能な名前付きタプルは、更新可能な辞書よりも、一見すると不便に思えるかもしれません。しかし、更新不可能であるからこそ、データの意味が明快になり、プログラムの理解を助けます。更新を想定しないデータは、名前付きタプルで表現するのが賢明です。

“collections” モジュールは、要素の集まりを表現する有用なクラスを、他にも数多く提供しています。詳しくは、[公式ドキュメント](#)を参照してください。

20.6 ▲クラス属性

同一クラス内のインスタンスで、データを共有したいことがあります。例えば、先の例における教員を細分化して、教授を表す `Professor` クラスを定義することを考えましょう。このとき、`Professor` のインスタンスは、その役職が全て共通であるので、タプルの要素として役職を保持するのは効率が悪く、教授を表現するクラスとして不自然です。こういう時に役立つのが、クラス属性です。

クラス属性とは、クラス内に定義される変数です。あるクラスが持つクラス属性は、そのクラスのインスタンス全てに共有されます。役職がクラス属性として共有された `Professor` クラスは、次のように定義できます。

```
[28]: class Professor(tuple):  
        title = '教授'  
        def badge(self):  
            return self[0] + ' ' + self.title  
  
        hagiya = ('萩谷昌己', 9876543210)  
        print(Professor(hagiya))  
        print(Professor(hagiya).badge())  
  
(' 萩谷昌己', 9876543210)  
萩谷昌己 教授
```

実は、これまでメソッドと呼んできた、クラス内で定義された関数も、クラス属性の一種です。なので、メソッド内で定義した関数は、同一クラスの全てのインスタンスで共有されています。

ここで、メソッドの仕組みをより正確に説明します。クラス `A` のインスタンス `x` があったとして、`x.a` と属性アクセスしたときに、クラス属性 `A.a` が関数であった場合には、`a` がメソッドであると解釈されて、関数 `A.a` の第 1 引数を `x` に固定した新たなオブジェクトが返ります。それが、`x.a()` で呼び出されるメソッドの実体です。つまり、メソッドとは、クラス属性の関数とインスタンスを組にしたものです。したがって、最初に示した高階関数を使った例における `(taro, studentbadge)` と、メソッドは、意味的に対応します。

この仕組みを理解しておくと、メソッドを関数として取り出して高階関数に渡すなど、関数プログラミングとオブジェクト指向プログラミングを素直に組み合わせたプログラミングスタイルを取れるようになります。

最後に、クラス属性は、(組込みのデータ型は例外として) 基本的に上書き可能です。したがって、クラスオブジェクトを更新することで、そのクラスのインスタンス全て(作成済みのものを含めて)の振舞いを変更することができます。*クラスオブジェクトを更新するときは、よく注意しましょう。*

```
[29]: Professor.title = 'Professor'
Professor.badge = lambda self: self.title + ' ' + self[0]
print(Professor(hagiya).badge())
```

Professor 萩谷昌己

20.7 練習の解答

```
[30]: taro = ('東大太郎', 1234567890)
jiro = ('東大二郎', 2345678901)
hagiya = ('萩谷昌己', '教授', 9876543210)
iwata = ('岩田聡', 'HAL 研究所', '代表取締役社長')
```

```
class Student(tuple):
    badge = studentbadge
class Faculty(tuple):
    badge = facultybadge
class Industrial(tuple):
    badge = industrialbadge
```

```
for p in [Student(taro), Student(jiro), Faculty(hagiya), Industrial(iwata)]:
    print(p)
```

```
class Student(tuple):
    __str__ = studentbadge
class Faculty(tuple):
    __str__ = facultybadge
class Industrial(tuple):
    __str__ = industrialbadge
```

```
for p in [Student(taro), Student(jiro), Faculty(hagiya), Industrial(iwata)]:
    print(p)
```

```
('東大太郎', 1234567890)
('東大二郎', 2345678901)
('萩谷昌己', '教授', 9876543210)
('岩田聡', 'HAL 研究所', '代表取締役社長')
東大太郎
東大二郎
萩谷昌己 教授
HAL 研究所 代表取締役社長 岩田聡
```

```
[31]: class Counter(dict):
        def __missing__(self, k):
            return 0
        def __setitem__(self, k, v):
            super().__setitem__(k, v)
            if self[k] == 0:
                del self[k]

c = Counter()
c['A'] += 1
c['B'] += 1
c['A'] -= 1
c
```

```
[31]: {'B': 1}
```

[]:

CHAPTER 21

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 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)
```

(continues on next page)

(continued from previous page)

```
# 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    0.897425
1    0.511649
2    0.131253
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
12	6	7	8
13	9	10	11

	c1	c2	c3
0	0.661544	0.882424	0.701662
1	0.977650	0.836378	0.114746
2	0.460659	0.089860	0.799603
3	0.120629	0.594279	0.393181

	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 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()` 関数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)
```

	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]
```

	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
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 の多次元配列のブールインデックス参照と同様に、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	0.326727
1	4.9	3.0	1.4	0.2	setosa	0.204543
2	4.7	3.2	1.3	0.2	setosa	0.342928
3	4.6	3.1	1.5	0.2	setosa	0.294254
4	5.0	3.6	1.4	0.2	setosa	0.112838
5	5.4	3.9	1.7	0.4	setosa	0.050505
6	4.6	3.4	1.4	0.3	setosa	0.449940
7	5.0	3.4	1.5	0.2	setosa	0.082641
8	4.4	2.9	1.4	0.2	setosa	0.556019
9	4.9	3.1	1.5	0.1	setosa	0.702711

“del” ステートメントを用いると、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
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	0.099809
1	4.9	3.0	1.4	0.2	setosa	0.588242
2	4.7	3.2	1.3	0.2	setosa	0.475183
3	4.6	3.1	1.5	0.2	setosa	0.254094
4	5.0	3.6	1.4	0.2	setosa	0.328797

“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”** () 関数を用いると、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。この時に、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、sort_index() 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
```

(continues on next page)

(continued from previous page)

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
```

(continues on next page)

(continued from previous page)

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
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、二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 0x11f496dd8>
```

```
[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
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/documentation.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。

22.1 機械学習について

機械学習では、観測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 ライブラリ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学習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を評価を行う
- 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、すべてのモデルは 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一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。train_test_split() はデータセットを学習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学習データ (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、学習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学習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

# 学習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)

# ロジスティック回帰モデル: 最適化には L-BFGS 法、多クラス分類をサポート
model=LogisticRegression(solver='lbfgs', multi_class='auto')

# ***scikit-learn のバージョンが 0.19 以前の環境ではこちらを実行してください***
# model=LogisticRegression()

model.fit(X_train, y_train) # モデルを学習データに適合
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

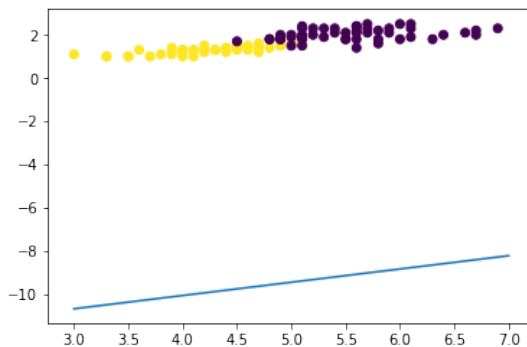
w2 = model.coef_[0,1]
```

(continues on next page)

(continued from previous page)

```
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(np.int)
plt.scatter(iris2['petal_length'],iris2['petal_width'],c=y_c);
```



22.8 練習の解答例

```
[6]: 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)

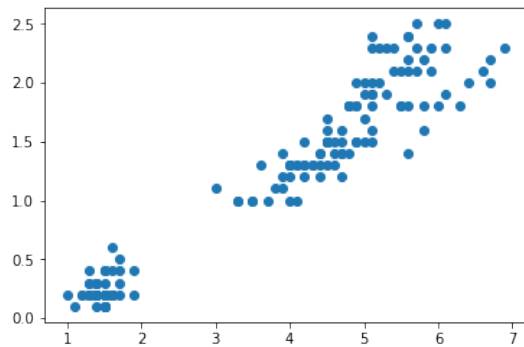
model=LogisticRegression(solver='lbfgs', multi_class='auto')
# ***scikit-learnのバージョンが0.19以前の環境ではこちらを実行してください***
# model=LogisticRegression()
model.fit(X_train, y_train)
y_model=model.predict(X_test)
accuracy_score(y_test, y_model)

[6]: 0.9666666666666667
```

22.9 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, からもう一つの特徴, 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。

```
[7]: 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一つである線形回帰 (“LinearRegression”) クラスをインポートしています。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。

データセットを学習データ (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学習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っています。

```
[8]: from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error

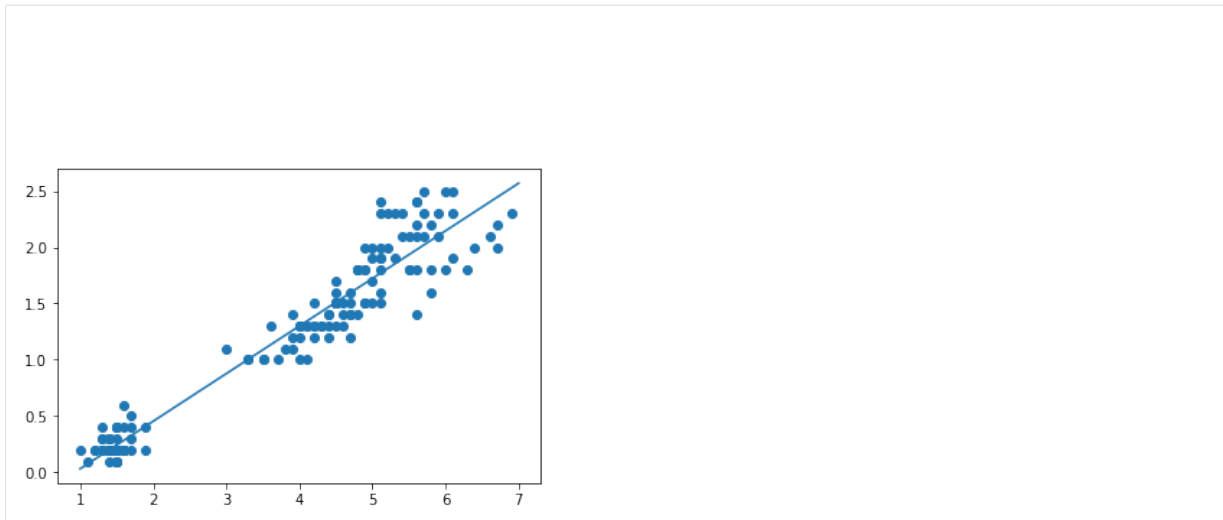
# 学習データとテストデータに分割
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_
    ↳state=1)

model=LinearRegression() # 線形回帰モデル
model.fit(X_train,y_train) # モデルを学習データに適合
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
```

```
[8]: 0.03974445760904275
```

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。

```
[9]: 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);
```



22.10 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つの特徴量, `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一つである“**KMeans**”クラスをインポートしています。

特徴量データ (`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。

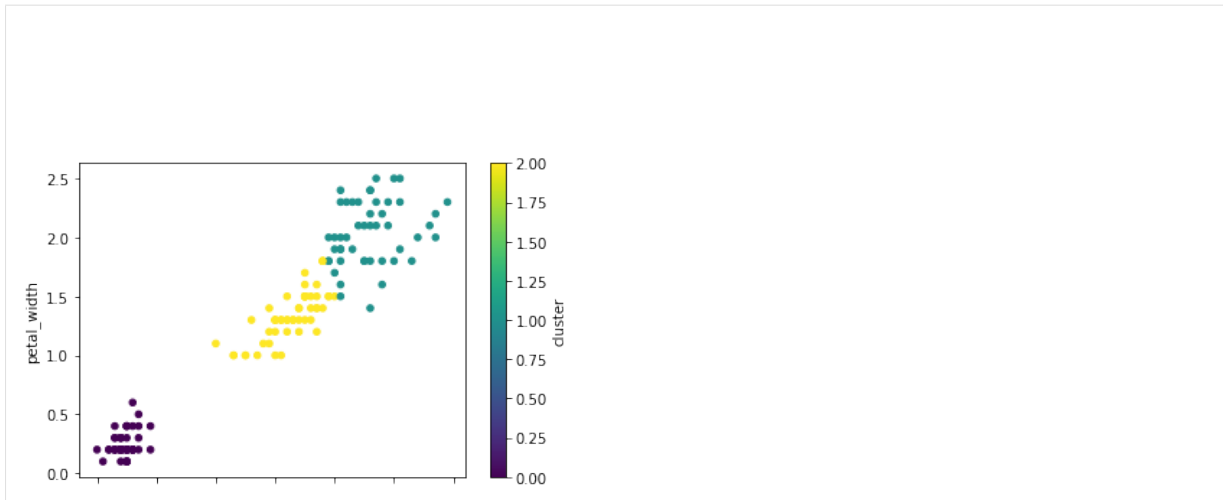
学習された各花データのクラス情報を元のデータセットのデータフレームに列として追加し、クラスごとに異なる色でデータセットを可視化しています。2つの特徴量, `petal_length` と `petal_width`, に基づき、3つのクラスが得られていることがわかります。

```
[10]: from sklearn.cluster import KMeans

iris = pd.read_csv('iris.csv')
X_iris=iris[['petal_length', 'petal_width']].values

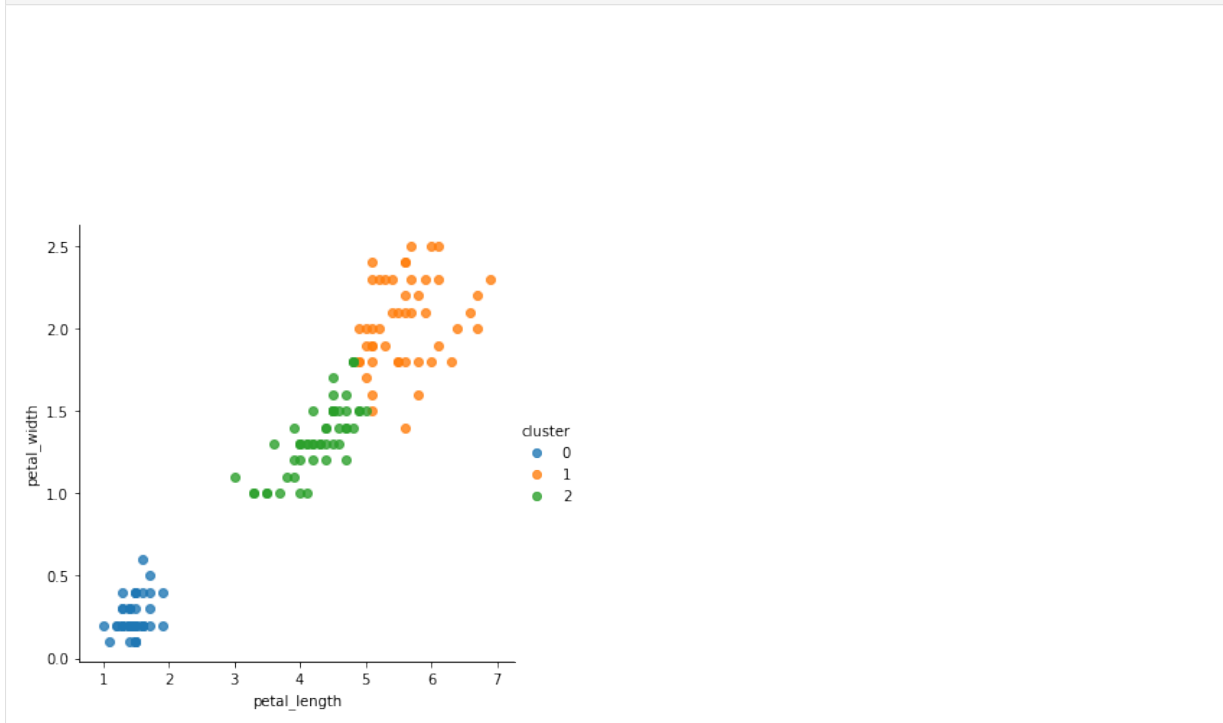
model = KMeans(n_clusters=3) # k-means モデル
model.fit(X_iris) # モデルをデータに適合
y_km=model.predict(X_iris) # クラスタを予測

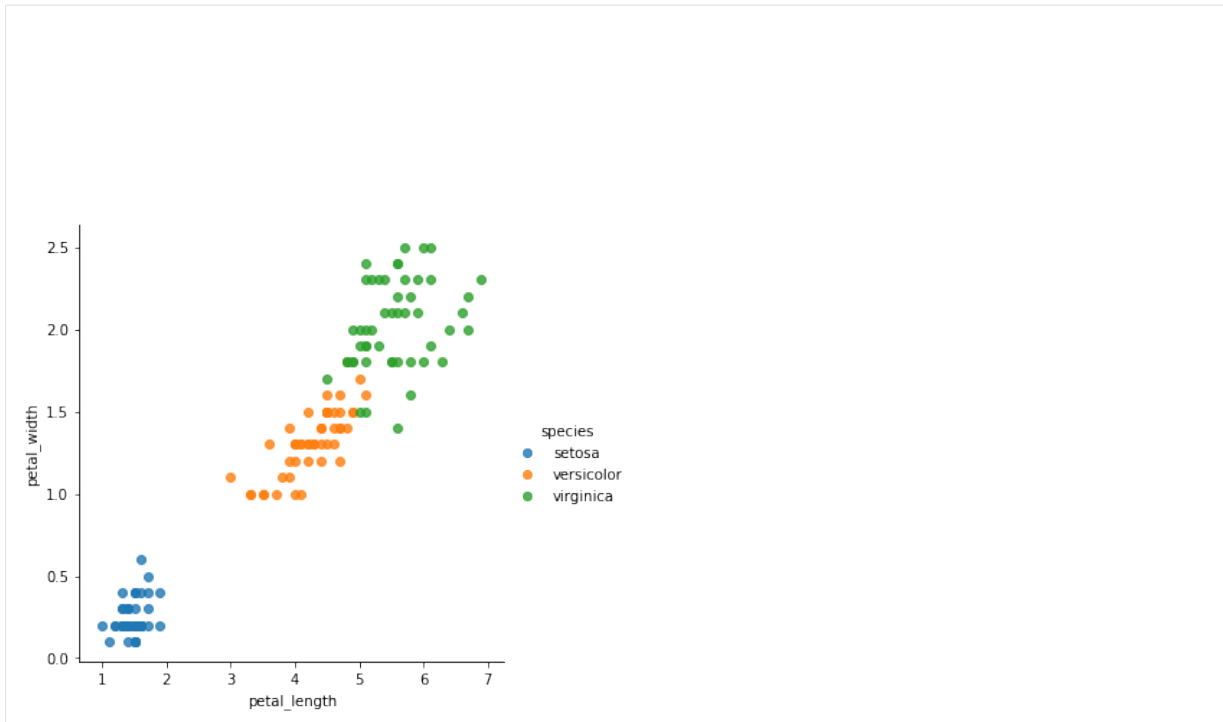
iris['cluster']=y_km
iris.plot.scatter(x='petal_length', y='petal_width', c='cluster', colormap='viridis
→');
```



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。

```
[11]: import seaborn as sns
sns.lmplot('petal_length', 'petal_width', hue='cluster', data=iris, fit_reg=False);
sns.lmplot('petal_length', 'petal_width', hue='species', data=iris, fit_reg=False);
```





22.11 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。

```
[12]: 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
```

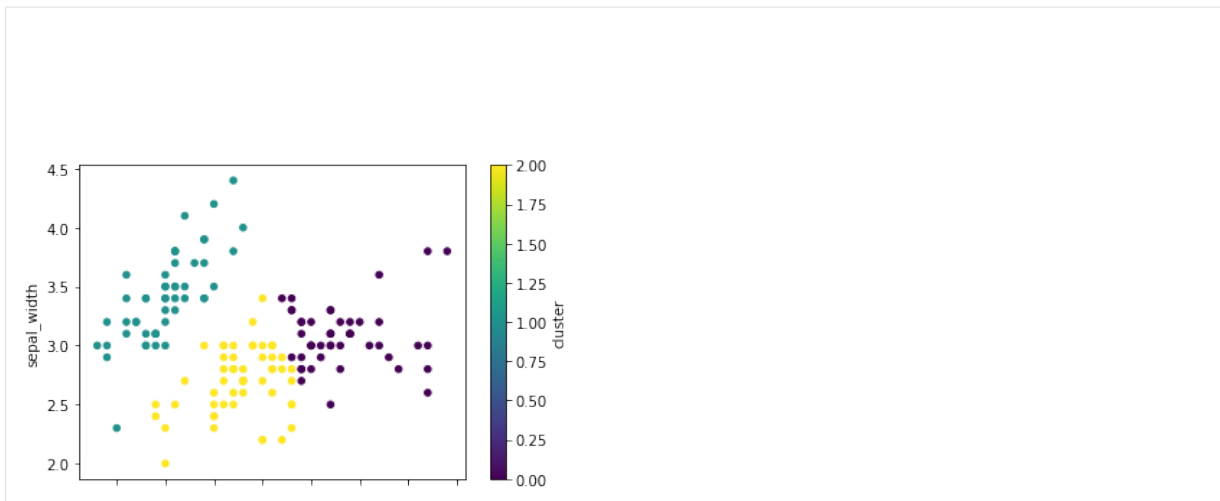
22.12 練習の解答例

```
[13]: 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
→');
```

22.13 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一つである“PCA”クラスをインポートしています。

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。

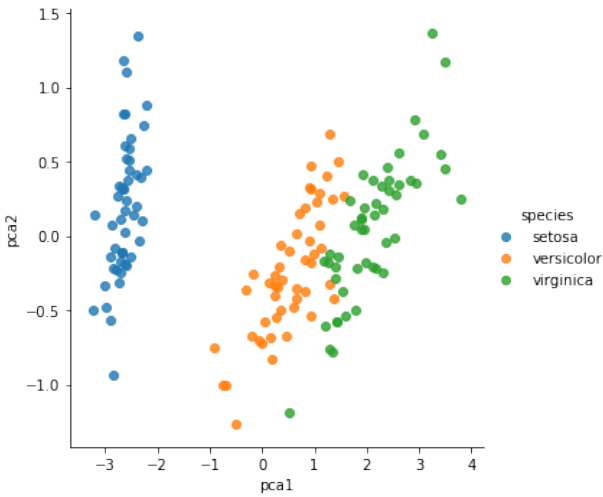
学習された各次元の値を元のデータセットのデータフレームに列として追加し、データセットを削減して得られた次元の空間において、データセットを花の種類ごとに異なる色で可視化しています。削減された次元の空間において、花の種類をグループ分けできていることがわかります。

```
[14]: 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
```

```
[15]: import seaborn as sns
iris['pca1']=X_2d[:,0]
iris['pca2']=X_2d[:,1]
sns.lmplot('pca1', 'pca2', hue='species', data=iris, fit_reg=False);
```



CHAPTER 23

▲セット型 (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>

セット型 はリストと同様に複数の要素から構成されるデータ型です。セット型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値を囲みます。リストと似ていますが、リストは : 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
```

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('aabdceabda')
[8]: {'a', 'b', 'c', 'd', 'e'}

[9]: set({'apple' : 3, 'pen' : 5})
[9]: {'apple', 'pen'}
```

23.1 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
[11]: 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
```

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。

```
[14]: set1[0]

-----
TypeError                                Traceback (most recent call last)
<ipython-input-14-c38563f1af7a> in <module>
--> 1 set1[0]

TypeError: 'set' object is not subscriptable
```

23.2 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。

```
[15]: set1 = {1, 2, 3, 4}
[15]: set2 = {3, 4, 5, 6}

[16]: set1 | set2 # 和集合
[16]: {1, 2, 3, 4, 5, 6}
```

```
[17]: set1 & set2 # 積集合
```

```
[17]: {3, 4}
```

```
[18]: set1 - set2 # 差集合
```

```
[18]: {1, 2}
```

```
[19]: set1 ^ set2 # 対称差
```

```
[19]: {1, 2, 5, 6}
```

23.3 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。

```
[20]: print({1, 2, 3} == {1, 2, 3})  
print({1, 2} == {1, 2, 3})
```

```
True  
False
```

```
[21]: print({1, 2, 3} != {1, 2, 3})  
print({1, 2} != {1, 2, 3})
```

```
False  
True
```

```
[22]: print({1, 2, 3} <= {1, 2, 3})  
print({1, 2, 3} < {1, 2, 3})  
print({1, 2} < {1, 2, 3})
```

```
True  
False  
True
```

23.4 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。

23.4.1 add

指定した要素を新たにセットに追加します。

```
[23]: set1 = {1, 2, 3}  
set1.add(4)  
set1
```

```
[23]: {1, 2, 3, 4}
```

23.4.2 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。

```
[24]: set1.remove(1)  
set1
```

```
[24]: {2, 3, 4}
```

```
[25]: set1.remove(10)
```

```
-----  
KeyError                                Traceback (most recent call last)  
<ipython-input-25-19bc86444813> in <module>  
----> 1 set1.remove(10)  
  
KeyError: 10
```

23.4.3 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。

```
[26]: set1 = {1, 2, 3, 4}  
      set1.discard(1)  
      set1
```

```
[26]: {2, 3, 4}
```

```
[27]: set1.discard(5)
```

23.4.4 clear

全ての要素を削除して対象のセットを空にします。

```
[28]: set1 = {1, 2, 3, 4}  
      set1.clear()  
      set1
```

```
[28]: set()
```

23.4.5 pop

セットからランダムに1つの要素を取り出します。

```
[29]: set1 = {1, 2, 3, 4}  
      print(set1.pop())  
      print(set1)
```

```
1  
{2, 3, 4}
```

23.4.6 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。

```
[30]: set1 = {1, 2, 3, 4}  
      set2 = {3, 4, 5, 6}  
      set1.union(set2) # 和集合
```

```
[30]: {1, 2, 3, 4, 5, 6}
```

```
[31]: set1.intersection(set2) # 積集合
```

```
[31]: {3, 4}
```

```
[32]: set1.difference(set2) # 差集合
```

```
[32]: {1, 2}
```

```
[33]: set1.symmetric_difference(set2) # 対称差
```

```
[33]: {1, 2, 5, 6}
```

23.5 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（サイズ 1 の文字列）の種類を返す関数 `check_characters` を作成して下さい（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）。

以下のセルの ... のところを書き換えて `check_characters(str1)` を作成して下さい。

```
[34]: 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下さい。

```
[35]: print(check_characters('Onde a terra acaba e o mar come^^c3^^a7a') == 13)  
  
False
```

23.6 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` を作成して下さい。

以下のセルの ... のところを書き換えて `count_words2(str_engsentences)` を作成して下さい。

```
[36]: 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下さい。

```
[37]: print(count_words2('From Stettin in the Baltic to Trieste in the Adriatic an iron_  
↪curtain has descended across the Continent.') == 15)  
  
False
```

23.7 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` を作成して下さい。

以下のセルの ... のところを書き換えて `check_dicsize(dic1)` を作成して下さい。

```
[38]: def check_characters(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下さい。

```
[39]: print(check_dicsize({'apple': 0, 'orange': 2, 'pen': 1}) == 3)

-----
NameError                                Traceback (most recent call last)
<ipython-input-39-bf7aa162a4fa> in <module>
--> 1 print(check_dicsize({'apple': 0, 'orange': 2, 'pen': 1}) == 3)

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

23.8 練習の解答

```
[40]: def check_characters(str1):
      set1 = set(str1)
      return len(set1)
      #check_characters('Onde a terra acaba e o mar come^^c3^a7a')

[41]: 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.')

[41]: 15

[42]: def check_dicsize(dic1):
      return len(set(dic1))
      #check_dicsize({'apple': 0, 'orange': 2, 'pen': 1})
```

▲簡単なデータの可視化

第2回までに学んだデータ型を利用して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)

24.1 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、Jupyter Notebook 上で用意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回で学ぶデータ型だけ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ますので、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 Jupyter Notebook で表示させるには、以下をコードセルで一回だけ実行します。

%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
```

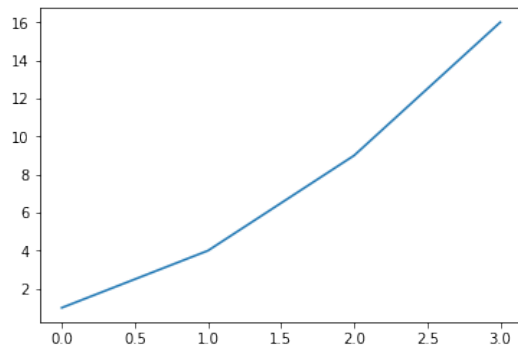
さらに matplotlib モジュールを読み込む次の処理もプログラムの冒頭でおこなう必要があります。

```
import matplotlib.pyplot as plt
```

24.2 折れ線グラフ

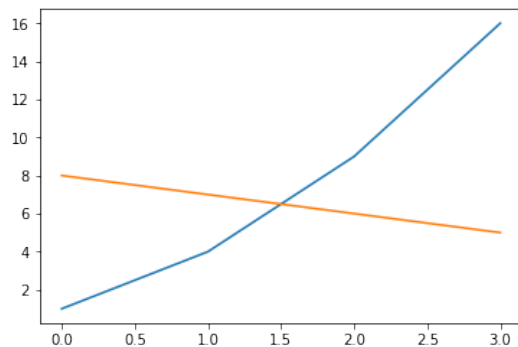
ls1 = [1, 4, 9, 16] といった数を要素とするリストを折れ線グラフで表示するには、次のようにおこないます。

```
[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()
```

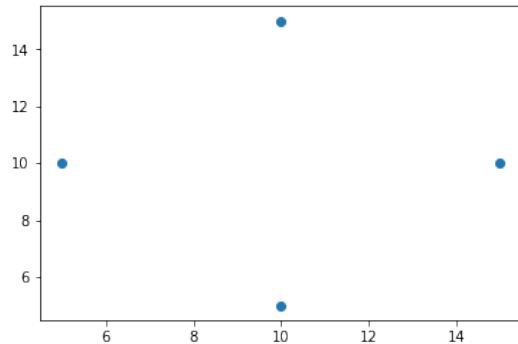


24.3 散布図

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 0x113be6ef0>
```

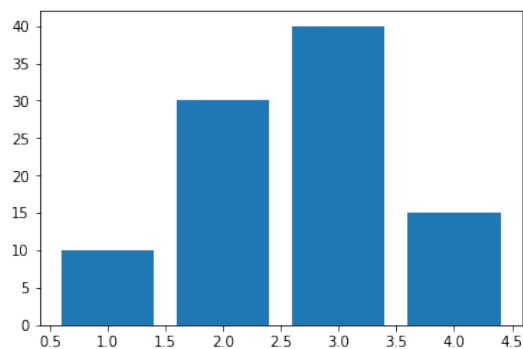


24.4 棒グラフ

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)
```

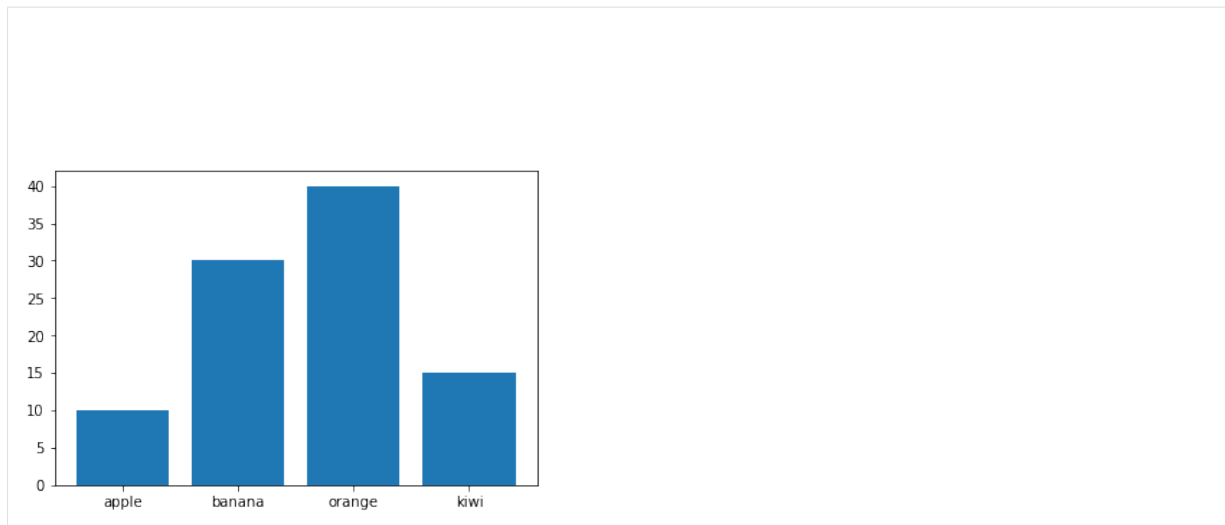
```
[5]: <BarContainer object of 4 artists>
```



第2回は文字列、辞書についても学びました。文字列を `key`、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 `key` 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>
```



▲再帰

再帰について説明します。

関数の再帰呼出しとは、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、再帰呼出しに含まれます。再帰呼出しを行う関数を、**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', 'aabc', '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 == '':
        return []
    else:
```

(continues on next page)

(continued from previous page)

```
    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。例えば、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):
        froml = False
        if i2 >= r:
            froml = True
        elif i1 < m:
            n = n + 1
            if data[i1] <= data[i2]:
                froml = True
        if froml:
            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。 - 要素が一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 は自分自身を二回呼び出していますので、繰り返しで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]: [17,
95,
314,
314,
567,
711,
772,
947,
963,
1039,
1106,
1171,
1225,
1369,
```

(continues on next page)

(continued from previous page)

```
1467,  
1483,  
1661,  
1851,  
1958,  
2029,  
2454,  
2463,  
2473,  
2489,  
2501,  
2735,  
2754,  
2796,  
3051,  
3114,  
3190,  
3213,  
3371,  
3380,  
3509,  
3615,  
3741,  
3752,  
3798,  
3907,  
4066,  
4122,  
4249,  
4682,  
4689,  
4761,  
4814,  
4853,  
5106,  
5288,  
5404,  
5525,  
5632,  
5666,  
5780,  
6004,  
6013,  
6269,  
6298,  
6449,  
6467,  
6549,  
6710,  
6825,  
6879,  
6881,  
6980,  
7109,  
7186,  
7208,  
7209,  
7279,  
7287,  
7396,  
7404,  
7605,
```

(continues on next page)

(continued from previous page)

```
7649,  
7710,  
7853,  
8090,  
8097,  
8191,  
8326,  
8414,  
8655,  
8738,  
8743,  
8942,  
9048,  
9170,  
9293,  
9358,  
9434,  
9489,  
9579,  
9634,  
9669,  
9703,  
9706,  
9707]
```

[]:

▲ 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>

26.1 csv 形式とは

CSV ファイルとは "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 では一つのセルに一つの値（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ることを考えましょう。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'

26.2 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、そのオブジェクトから、CSV リーダーを作ります。CSV リーダーとは、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 リーダーを作るには、“csv” というモジュールの “csv.reader” という関数にファイルのオブジェクトを渡します。

例えば、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)
```

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

```
[2]: _csv.reader
```

```
[3]: dir(dataReader)
```

```
[3]: ['__class__',
      '__delattr__',
      '__dir__',
      '__doc__',
      '__eq__',
      '__format__',
      '__ge__',
      '__getattr__',
      '__gt__',
      '__hash__',
      '__init__',
      '__init_subclass__',
      '__iter__',
      '__le__',
      '__lt__',
      '__ne__',
      '__new__',
      '__next__',
      '__reduce__',
      '__reduce_ex__',
      '__repr__',
      '__setattr__',
      '__sizeof__',
      '__str__',
      '__subclasshook__',
      'dialect',
      'line_num']
```

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。

```
[4]: next(dataReader)
```

```
[4]: ['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配列となって返されます。

```
[5]: next(dataReader)
```

```
[5]: ['21', '22', '23', '24', '25']
```

```
[6]: row = next(dataReader)
```

```
[7]: row
```

```
[7]: ['31', '32', '33', '34', '35']
```

```
[8]: row[2]
[8]: '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。

```
[9]: int(row[2])
[9]: 33
```

ファイルの終わりまで達したあとに `next` 関数を実行すると、下のようエラーが返ってきます。

```
[10]: next(dataReader)

-----
StopIteration                                Traceback (most recent call last)
<ipython-input-10-4f4f4fee7fd4> in <module>
----> 1 next(dataReader)

StopIteration:
```

ファイルを使い終わったら `close` することを忘れないようにしましょう。

```
[11]: f.close()
```

26.3 CSV ファイルに対する for 文

CSV リーダー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。

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。

```
[12]: 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']
```

26.4 CSV ファイルに対する with 文

以下は `with` 文を使った例です。

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

26.5 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の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（例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。

```
[14]: f = open('out.csv', 'w', encoding='utf-8', newline='')
```

```
[15]: dataWriter = csv.writer(f)
```

```
[16]: dir(dataWriter)
```

```
[16]: ['__class__',  
      '__delattr__',  
      '__dir__',  
      '__doc__',  
      '__eq__',  
      '__format__',  
      '__ge__',  
      '__getattr__',  
      '__gt__',  
      '__hash__',  
      '__init__',  
      '__init_subclass__',  
      '__le__',  
      '__lt__',  
      '__ne__',  
      '__new__',  
      '__reduce__',  
      '__reduce_ex__',  
      '__repr__',  
      '__setattr__',  
      '__sizeof__',  
      '__str__',  
      '__subclasshook__',  
      'dialect',  
      'writerow',  
      'writerows']
```

```
[17]: dataWriter.writerow([1,2,3])
```

```
[17]: 7
```

```
[18]: dataWriter.writerow([21,22,23])
```

```
[18]: 10
```

書き込みモードの場合も、ファイルを使い終わったら `close` を忘れないようにしましょう。

```
[19]: f.close()
```

読み込みのときと同様、`with` 文を使うこともできます。

```
[20]: 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])
```

26.5.1 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、2018 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。

```
[21]: import csv

with open('tokyo-temps.csv', 'r', encoding='sjis') 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。このファイルはシフト 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。'sjis' はシフト JIS を意味します。この他に、'utf-8'（8 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。

```
[22]: years
```

```
[22]: [1875,
      1876,
      1877,
      1878,
      1879,
      1880,
      1881,
      1882,
      1883,
      1884,
      1885,
      1886,
      1887,
      1888,
      1889,
      1890,
      1891,
      1892,
      1893,
      1894,
      1895,
      1896,
```

(continues on next page)

(continued from previous page)

```
1897,  
1898,  
1899,  
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,  
1956,  
1957,  
1958,
```

(continues on next page)

(continued from previous page)

```
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,  
2012,  
2013,  
2014,  
2015,  
2016,  
2017,  
2018]
```



```
[23]: july_temps
```

```
[23]: [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,  
26.1,  
23.2,  
24.6,  
26.0,  
23.4,  
25.9,  
26.3,  
21.8,  
25.7,  
26.6,  
23.9,  
24.3,
```

(continues on next page)

(continued from previous page)

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,
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,

(continues on next page)

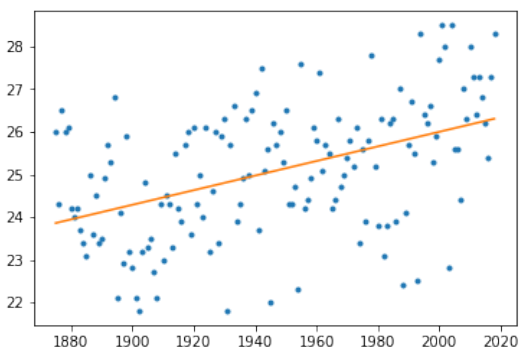
(continued from previous page)

```
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,  
27.3,  
28.3]
```

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。

```
[24]: 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))
```

```
[25]: plt.plot(years, july_temps, '.', xp, fitp(xp), '-')  
plt.show()
```



26.6 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 2018 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 の値がそのまま仮定しています。)

```
[26]: with open('tokyo-july-temps.csv', 'r', encoding='sjis') 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 == 144) # 1875 年から 2018 年まで 144 年間分のデータがあるはずです
      True
```

26.7 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。

例えば上で用いた `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。

```
[27]: def csv_matrix(name):
      ...
```

以下のセルによってテストしてください。

```
[28]: print(csv_matrix('small.csv') == [[11, 12, 13, 14, 15], [21, 22, 23, 24, 25], [31,
      ↪ 32, 33, 34, 35]])
      False
```

[]:

[]:

[]:

26.8 練習の解答

```
[29]: 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]])
```

```
[30]: 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
```

```
[ ]:
```

▲ JSON ファイルの入出力

JSON 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/json.html>

27.1 JSON 形式とは

JSON 形式 は、JavaScript Object Notation の略で、データを保存するための記録方式の一つです。特に、辞書や辞書のリストを記録することができます。

たとえばサークルのメンバーデータを作ることを考えましょう。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。ニックネームには複数の要素が入っていることに注意してください。

ID	氏名	ニックネーム	出身地
user1	鈴木一郎	イチロー, いっち	広島
user2	山田花子	はなこ, ハナちゃん	名古屋

これを JSON 形式で表すと以下ようになります。

```
"user1": {"氏名": "鈴木一郎", "ニックネーム": ["イチロー", "いっち"], "出身地": "広島"}, "user2": {"氏名": "山田花子", "ニックネーム": ["はなこ", "ハナちゃん"], "出身地": "名古屋"}
```

JSON 形式で `key:value` となっている場合、`:` で挟んだ左側が `key`、右側が `value` であるような辞書と考えてください。

また、`{}` で囲んだものは辞書、`[]` で囲んだものはリストで、辞書の中に辞書、リストの中に辞書、など、入れ子の構造にすることができます。複数の要素を列挙する場合は、`(コンマ)` で区切ります。

値の型	json の例
string	"data": "123"
number	"data": 123
boolean	"data": true
辞書	"data": {"a": "b"}
リスト	"data": [1, 2, 3]

上の JSON ファイルの例では、全体が 1 つのオブジェクトであり、その中に "user1" と "user2" というラベルのついた 2 つの辞書があり、その各辞書の中には "氏名" と "ニックネーム" と "出身地" の 3 つのオブジェクトがあり、さらに "ニックネーム" の値には、"イチロー" や "いっち" といった要素がリスト形式で記録されています。

リストや辞書は自由に構成することができます。
例えば下の例はリストのリストです。

```
[['りんご', 'みかん', 'バナナ'], ['玉ねぎ', '人参', 'ジャガイモ']]
```

下の例は辞書のリストです。

```
[{'りんご': 3, 'みかん': 5, 'バナナ': 2}, {'玉ねぎ': 4, '人参': 2, 'ジャガイモ': 1}]
```

27.2 JSON ファイルのダンプとロード

“**json**” モジュールを用いることにより、Python の各種のデータをファイルに書き出す（ダンプする）ことができ、また、ファイルからロード（読み込み）することができます。ダンプとロードには、それぞれ “**json.dumps**” と “**json.load**” を用います。

また、JSON ファイルの中身を標準出力で書き出すときは、“**json.dump**” を使います（ファイルに書き出す “**json.dumps**” とは、「dump」と「dumps」に違いがあるので注意してください）。

読み込もうとする JSON ファイルに日本語が含まれている場合、`json.dumps` でダンプする際は、オプションとして `ensure_ascii=True` を指定しないと文字化けするので注意してください。

```
[1]: import json

# 上で例に挙げた JSON 形式のデータ表現
d = {
    'user1': {
        '氏名': '鈴木一郎',
        'ニックネーム': [
            'イチロー',
            'いっち'
        ],
        '出身地': '広島'
    },
    'user2': {
        '氏名': '鈴木花子',
        'ニックネーム': [
            'はなこ',
            'ハナちゃん'
        ],
        '出身地': '名古屋'
    }
}

# d をファイルに書き出し
with open('test.json', 'w', encoding='utf-8') as f:
    # ensure_ascii=False を指定しないと文字化けします
    json.dump(d, f, ensure_ascii=False)

# JSON ファイルを読み込み
with open('test.json', 'r', encoding='utf-8') as f:
    d1 = json.load(f)

# json データをただ印刷するだけだと 1 行にまとまってしまう、データの構造が非常にわかりづらくなります。
```

(continues on next page)

(continued from previous page)

```
# 見やすくするには json.dumps が有用です。
print(d1)

# 上記のようだととても見にくいので整形して読み込み

# JSON ファイルを読み込み
# ensure_ascii=False を指定しないと文字化けします
print(json.dumps(d1, indent=2, ensure_ascii=False))

{'user1': {'氏名': '鈴木一郎', 'ニックネーム': ['イチロー', 'いっち'], '出身地': '広島'},
→ 'user2': {'氏名': '鈴木花子', 'ニックネーム': ['はなこ', 'ハナちゃん'], '出身地': '名古屋'}}
{
  "user1": {
    "氏名": "鈴木一郎",
    "ニックネーム": [
      "イチロー",
      "いっち"
    ],
    "出身地": "広島"
  },
  "user2": {
    "氏名": "鈴木花子",
    "ニックネーム": [
      "はなこ",
      "ハナちゃん"
    ],
    "出身地": "名古屋"
  }
}
```

なお、1つのJSONファイルには1つのJSON形式のデータしか記録できません。
2つ以上のJSON形式のデータを記録してしまうと、JSON形式とみなされず、エラーが起きますので注意してください。

以下の例では、'test.json' に d を JSON 形式で2回ダンプしています。よって、その'test.json' を json.load で読み込む際にエラーが出ます。
json.dump の行を一つコメントアウトして、1回だけ書き出すようにすれば、エラーが起きなくなることを確認してください。

```
[2]: with open('test.json', 'w', encoding='utf-8') as f:
      json.dump(d, f, ensure_ascii=False)
      json.dump(d, f, ensure_ascii=False) # <== 1回だけを書き出すよう、この行をコメントアウト
      してください

with open('test.json', 'r', encoding='utf-8') as f:
    d1 = json.load(f)
```

```
-----
JSONDecodeError                                Traceback (most recent call last)
<ipython-input-2-36d117b1cdf> in <module>
      4
      5 with open('test.json', 'r', encoding='utf-8') as f:
--> 6     d1 = json.load(f)

~/anaconda3/lib/python3.7/json/__init__.py in load(fp, cls, object_hook,
→ parse_float, parse_int, parse_constant, object_pairs_hook, **kw)
```

(continues on next page)

(continued from previous page)

```
294         cls=cls, object_hook=object_hook,
295         parse_float=parse_float, parse_int=parse_int,
-> 296         parse_constant=parse_constant, object_pairs_hook=object_pairs_hook,
↳ **kw)
297
298

~/anaconda3/lib/python3.7/json/__init__.py in loads(s, encoding, cls, object_hook,
↳ parse_float, parse_int, parse_constant, object_pairs_hook, **kw)
346         parse_int is None and parse_float is None and
347         parse_constant is None and object_pairs_hook is None and not
↳ kw):
-> 348         return _default_decoder.decode(s)
349     if cls is None:
350         cls = JSONDecoder

~/anaconda3/lib/python3.7/json/decoder.py in decode(self, s, _w)
338         end = _w(s, end).end()
339         if end != len(s):
-> 340             raise JSONDecodeError("Extra data", s, end)
341         return obj
342

JSONDecodeError: Extra data: line 1 column 133 (char 132)
```

27.3 練習

1. 以下のリスト内包の結果を `fib.json` というファイルに JSON フォーマットでダンプしてください。
2. ダンプしたファイルからロードして、同じものが得られることを確かめてください。

```
[3]: def fib(n):
      if (n == 0):
          return 0
      elif (n == 1):
          return 1
      else:
          return fib(n-1)+fib(n-2)

      [{ 'n': n, 'fib' : fib(n)} for n in range(0,10)]
```

```
[3]: [{ 'n': 0, 'fib': 0},
      { 'n': 1, 'fib': 1},
      { 'n': 2, 'fib': 1},
      { 'n': 3, 'fib': 2},
      { 'n': 4, 'fib': 3},
      { 'n': 5, 'fib': 5},
      { 'n': 6, 'fib': 8},
      { 'n': 7, 'fib': 13},
      { 'n': 8, 'fib': 21},
      { 'n': 9, 'fib': 34}]
```

```
[ ]:
```

以下のセルによってテストしてください。

```
[4]: with open('fib.json', 'r') as f:
      print(json.load(f) == [{ 'n': n, 'fib' : fib(n)} for n in range(0,10)])
```

```
True
```

27.3.1 東京大学授業カタログ

catalog-2018.json には、東京大学授業カタログから取り出したデータが記録されています。

具体的には、各授業の情報を納めた辞書のリストが JSON フォーマットで記録されています。これをロードするには、以下のようにします。

```
[5]: with open('catalog-2018.json', 'r', encoding='utf-8') as f:
      j = json.load(f)
```

j はリストであることを確認します。

```
[6]: type(j)
[6]: list
```

j の大きさ、すなわち授業カタログの件数、を確認します。

```
[7]: len(j)
[7]: 4910
```

j の各要素は個々の授業に対応していて、各授業の情報を辞書として含んでいます。

```
[8]: j[0]
[8]: {'year': '2018',
      'department_j': '法学部',
      'title_j': '生産システム I',
      'name_j': '藤本\u3000隆宏',
      'title': 'Production System I',
      'name': 'Takahiro Fujimoto',
      'Common_Course_Code': 'FLA-EC4810L1',
      'Semester': 'A1',
      'Period': '月曜 1 限\n          木曜 1 限\n          Mon\xa01st\n          Thu\xa01st',
      'Credits': '2',
      'Academic_Year': 'Other',
      'Open_to_other_faculties': '不可 NO',
      'Permitted_to_USTEP_Students': '不可 NO',
      'Classroom': '',
      'Language_in_Lecture': '日本語          Japanese',
      'Title': '生産システム^^e2^^85^^a0',
      'Schedule': 'ものづくりとは何か、開発と生産の流れ（プロセス）分析、プロセス分析の事例、製品と工程の統合分析、製品と生産システムの歴史（1）自動車の進化－製品工程ライフサイクル、製品と生産システムの歴史（2）アメリカ的製造システムの発展と限界、ものづくり総論のまとめ、製品開発のプロセス、製品開発の組織、開発期間とその管理、開発生産性とその管理、総合商品力と開発の組織^^ef^^bd^^a5 プロセス、製品開発のまとめ',
      'Teaching_Methods': '講義形式です。',
      'Method_of_Evaluation': '期末試験とは別に、授業中に抜き打ち小テストを 2 回行う予定です。小テストは各回 15 分が目安です。小テスト結果が 40 %、期末試験結果が 60 %の総合点で最終評価をする予定です。',
      'Required_Textbook': '藤本隆宏『生産マネジメント入門^^e2^^85^^a0^^e2^^85^^a1』を教科書とします。',
      'Reference_Books': '『人工物複雑化の時代－設計立国日本の産業競争力』藤本隆宏編、有斐閣',
      'Notes_on_Taking_the_Course': '「経営」および「経営戦略」を履修していることを強く勧めます。',
      'Others': '授業は主にスクリーンとプロジェクターを用いて行うが、その電子ファイルは学生が入手可能な状態にしておきます。URL は追って知らせます。'}
```

```
[9]: j[1]
```

```
[9]: {'year': '2018',  
      'department_j': '法学部',  
      'title_j': '特別講義\u3000国際紛争研究(外国語科目)',  
      'name_j': '藤原\u3000帰一',  
      'title': 'Introduction to International Conflicts',  
      'name': 'Kiichi Fujiwara',  
      'Common_Course_Code': 'FLA-PS4726L3',  
      'Semester': 'S1S2',  
      'Period': '火曜2限\nTue\xa02nd',  
      'Credits': '2',  
      'Academic_Year': 'Other',  
      'Open_to_other_faculties': '可 YES',  
      'Permitted_to_USTEP_Students': '可 YES',  
      'Classroom': '',  
      'Language_in_Lecture': '英語 English',  
      'Title': '特別講義\u3000国際紛争研究',  
      'Schedule': '1. Orientation\n2. The End of the Cold War and International  
Conflicts\n3. The Security Dilemma Revisited\n4. Why go to war? \n5. New Wars  
and Old Wars\n6. Ethnicity, Religion, and National Identities\n7. Failed  
States\n8. Does Intervention Work?\n9. The Constructivist Challenge \n10.  
Interdependence and International Conflicts\n11. Is Democracy the Answer?\n12. In_  
Search of International Institutions\n13. Conditions for Peace',  
      'Teaching_Methods': 'The course will be given in English: the materials are in_  
English, the lectures as well as instructions will all be given in English, and_  
you will answer my quiz in English. I look forward to your active participation_  
in class, which will compose a very important part of my grading. The reading_  
materials will be announced in class, and students must download them by_  
themselves.',  
      'Method_of_Evaluation': 'Your contributions in class will be essential,which will_  
provide 30% of your grades. There will be a written examination at the end of_  
the semester, which will be he basis for 70% of evaluation.',  
      'Required_Textbook': 'Reading materials will be given in class, which students_  
must download by themselves.',  
      'Reference_Books': 'Reading materials will be given in class, which students must_  
download by themselves.',  
      'Notes_on_Taking_the_Course': 'Please be advised that the course, including the_  
final examination, will be given in English.'}
```

各授業の担当教員の日本語の名前は、name_j というキーに対する値として格納されています。

```
[10]: j[1]['name_j']  
[10]: ' 藤原\u3000帰一'
```

姓と名は、\u3000 というコードで区切られているようです。

```
[11]: j[1]['name_j'].split('\u3000')
[11]: ['藤原', '帰一']
```

```
[12]: j[1]['Title']
```

```
[12]: '特別講義\u3000国際紛争研究'
```

title をキーに持たない授業はないようです。

```
[13]: for d in j:
        if d.get('title',-1)==-1:
            print(d)
```

27.4 不要な空白や改行の除去

ファイルから読み込んだ文字列の前後に不要な空白や改行がある場合は、組み込み関数 “`strip()`” を使用するとそれらの空白・改行を除去することができます。

```
[14]: '  This is strip.\n'.strip()
```

```
[14]: 'This is strip.'
```

`lstrip` は文頭、`rstrip` は文末の空白や改行を除去します。

```
[15]: '  This is strip.\n'.lstrip()
```

```
[15]: 'This is strip.\n'
```

```
[16]: '  This is strip.\n'.rstrip()
```

```
[16]: '  This is strip.'
```

```
[ ]:
```

```
[ ]:
```

```
[ ]:
```

27.5 練習の解答

```
[17]: with open('fib.json', 'w') as f:
      json.dump([{'n': n, 'fib': fib(n)} for n in range(0,10)], f)
```

```
[ ]:
```

▲ 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 Jupyter Notebook 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 Jupyter Notebook 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` 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) を参照して下さい。

```
[4]: data = [0, 1, 4, 9, 16]
x = range(len(data))
p = figure()
p.line(x, x, line_color='blue', legend='linear', line_dash='dashed')
p.line(x, data, line_color='green', legend='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良いです。

```
[5]: p = figure(x_axis_label='x', y_axis_label='y', title='Linear vs. Quadratic')
p.line(x, x, line_color='blue', legend='linear', line_dash='dashed')
p.circle(x, x, color='blue', line_width=5)
p.line(x, data, line_color='green', legend='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

色の使い分けをすべ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='linear', line_dash='dashed')
p.circle(x, x, color=c[0], line_width=5)
p.line(x, data, line_color=c[1], legend='quad', line_dash='dotted')
p.cross(x, data, color=c[1], size=16)
show(p)
```

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

プロットのを少なくすると、曲線は直線をつなぎ合わせることで描画される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
    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
```

(continues on next page)

(continued from previous page)

```
    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
      ↪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ーは円とし、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)
```

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、7-2 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')
      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。

```
[ ]:
```

▲ 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、5-2 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 Jupyter Notebook 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
```

29.1 線グラフ

pyplot モジュールの“**plot**” (`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` の値を位置 (リスト `A[i]`, `i`) の位置に点を打ち、各点を線でつなぎます。

例えば、次のようになります。

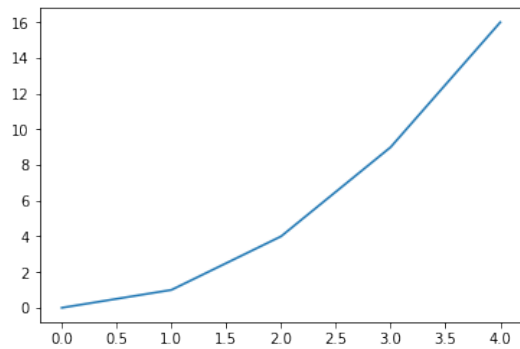
```
[2]: # plot するデータ
d = [0, 1, 4, 9, 16]

# plot 関数で描画
plt.plot(d);
```

(continues on next page)

(continued from previous page)

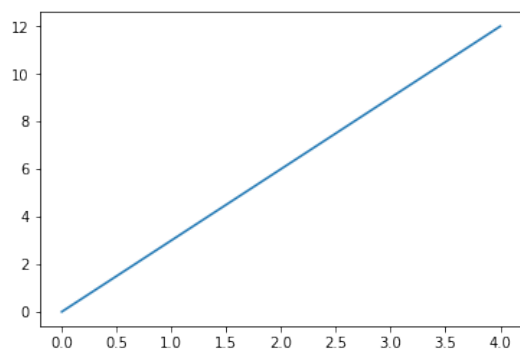
```
# セルの最後に評価されたオブジェクトの出力表示を抑制するために、以下ではセルの最後の行にセミicolon ( ;  
→ `) )` をつけています。  
# 試しにセミcolonを消した場合も試してみてください。
```



`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。

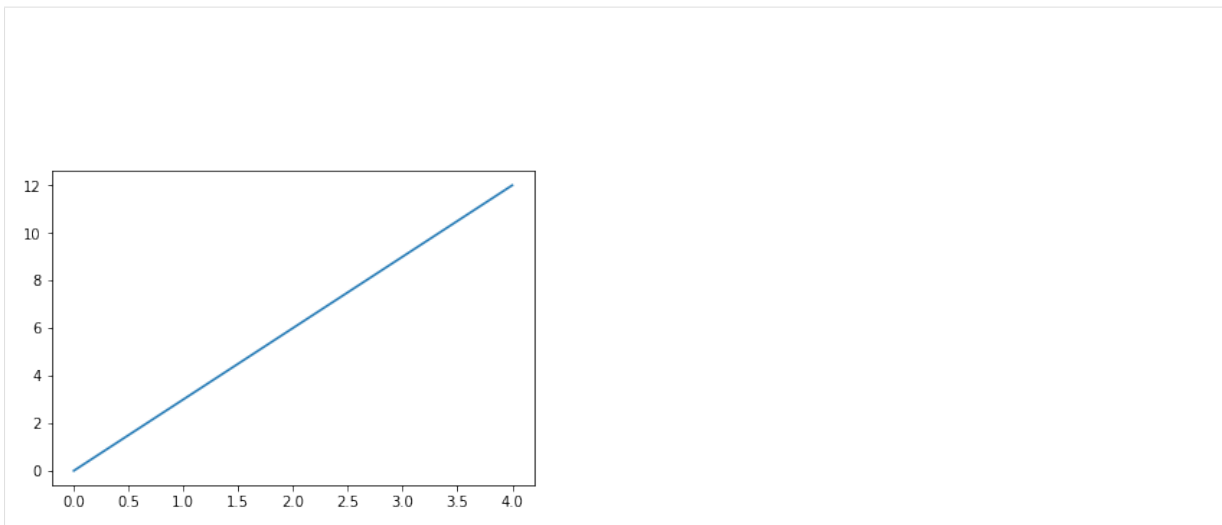
具体的には、次の様に リスト `x` と リスト `y` を引数として与えると、各 `i` に対して、(リスト `x[i]`, リスト `y[i]`) の位置に点を打ち、各点を線でつなぎます。

```
[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);
```



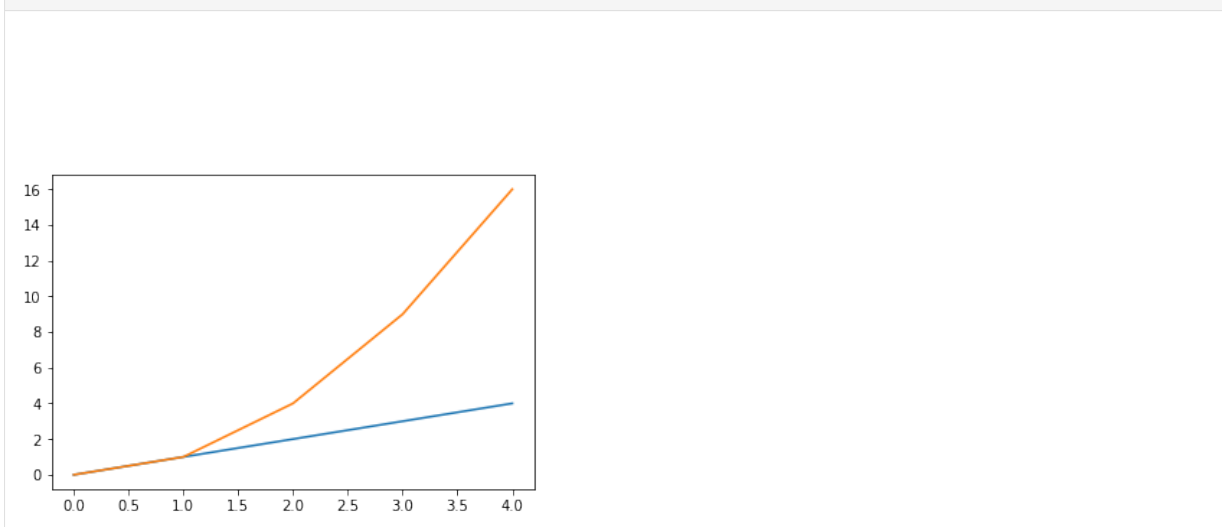
リストの代わりに NumPy の配列を与えても同じ結果が得られます。

```
[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  
y = [0, 3, 6, 9, 12]  
aryy = np.array(y) # リストから配列を作成  
  
# plot 関数で描画  
plt.plot(aryx, aryy);
```



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`

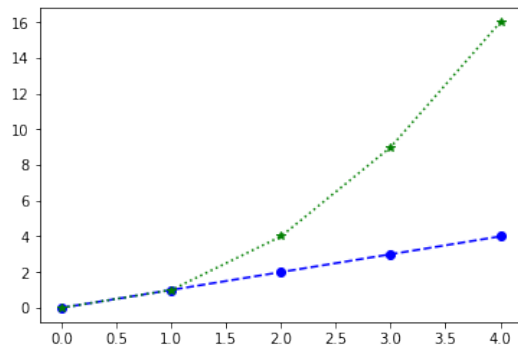
```
[6]: # plot するデータ
data = [0, 1, 4, 9, 16]
x = [0, 1, 2, 3, 4]
y = [0, 1, 2, 3, 4]

# plot 関数で描画。線の形状、色、データポイントのマーカ指定
```

(continues on next page)

(continued from previous page)

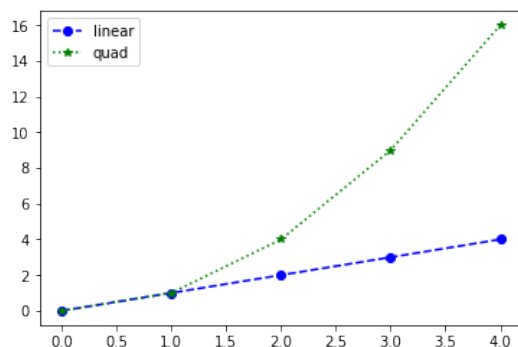
```
plt.plot(x,y, linestyle='--', color='blue', marker='o')  
plt.plot(data, linestyle=':', color='green', marker='*');
```



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();
```

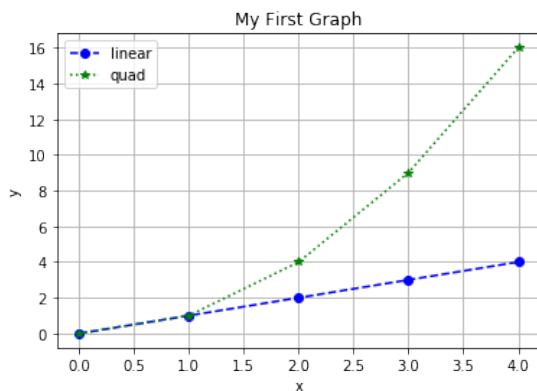


pypplot モジュールでは、以下のようにグラフのタイトルと各軸のラベルを指定して表示することができます。タイトル、x 軸のラベル、y 軸のラベル、はそれぞれ “title”() 関数、“xlabel”() 関数、“ylabel”() 関数に文字列を渡して指定します。また、“grid”() 関数を用いるとグリッドを併せて表示することもできます。グリッドを表示させたい場合は、grid() 関数に True を渡してください。

```
[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
```

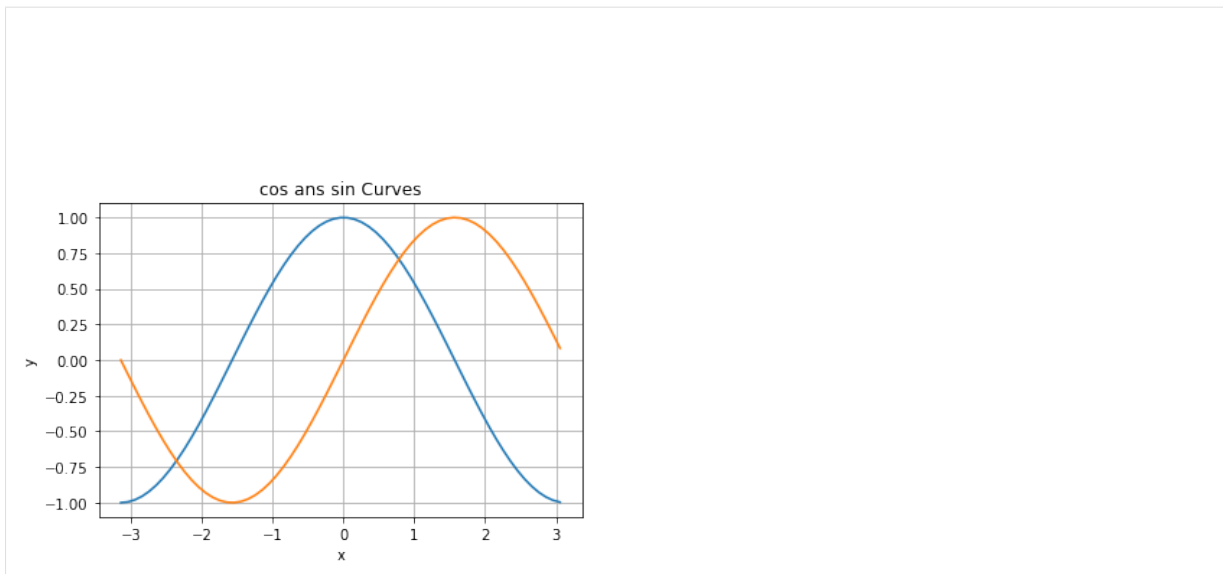


グラフを描画するときのプロット数を増やすことで任意の曲線のグラフを作成することもできます。以下では、numpy モジュールの `arange()` 関数を用いて、 $-\pi$ から π の範囲を 0.1 刻みで x 軸の値を配列として準備しています。その x 軸の値に対して、numpy モジュールの `cos()` 関数と `sin()` 関数を用いて、y 軸の値をそれぞれ準備し、cos カーブと sin カーブを描画しています。

```
[9]: # グラフの x 軸の値となる配列
x = np.arange(-np.pi, np.pi, 0.1)

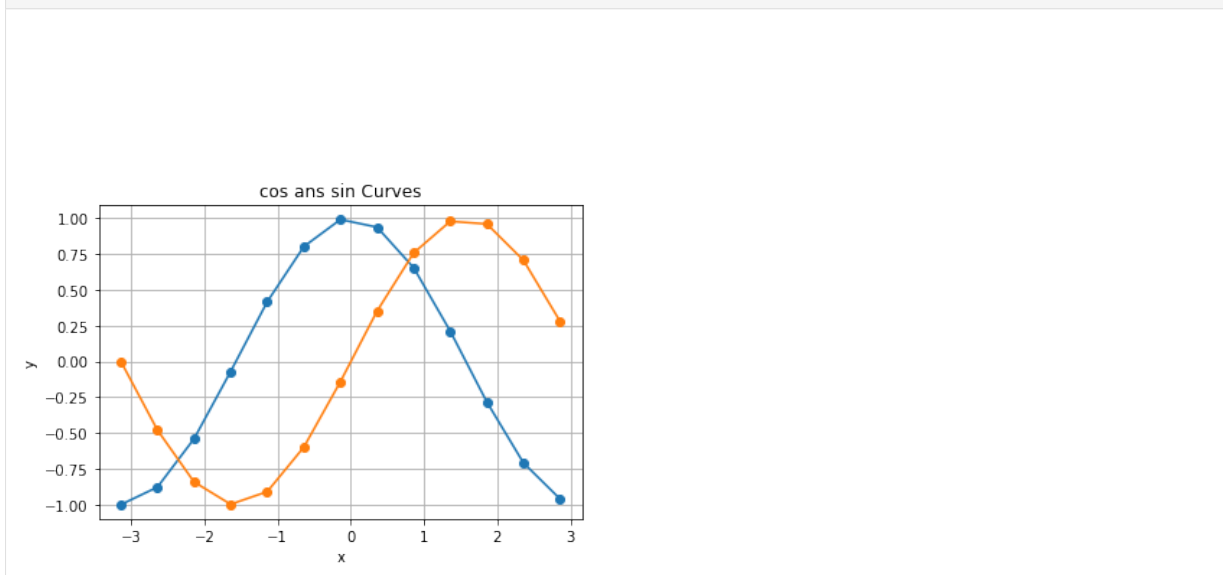
# 上記配列を cos, sin 関数に渡し、y 軸の値として描画
plt.plot(x,np.cos(x))
plt.plot(x,np.sin(x))

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る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);
```



29.1.1 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]
```

(continues on next page)

(continued from previous page)

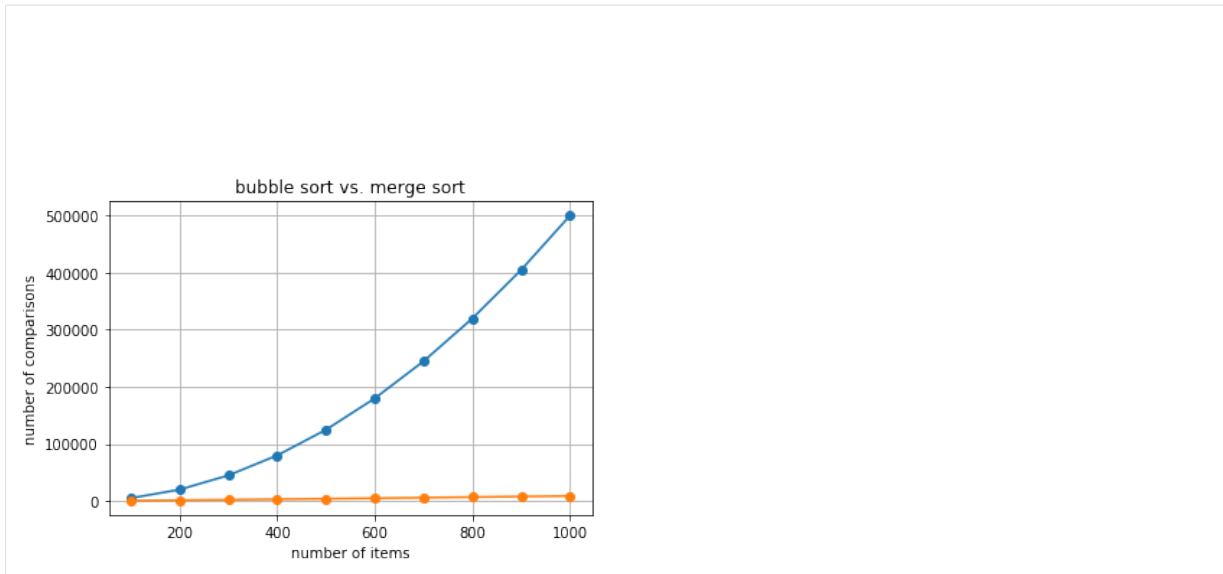
```
    return n

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):
        froml = False
        if i2 >= r:
            froml = True
        elif i1 < m:
            n = n + 1
            if data[i1] <= data[i2]:
                froml = True
        if froml:
            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
      ↪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);
```



29.2 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():
      ...

File "<ipython-input-14-859224c51e05>", line 1
    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下さい。

```
[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)
<ipython-input-15-7e0bfb663c10> 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
```

29.3 練習

4-2 で説明した様に、`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` を作成して下さい。

以下のセルの ... のところを書き換えて解答して下さい。

```
[16]: import ...
...
def plot_tokyotemps(year, month):
    ...

File "<ipython-input-16-7a1e43c0be99>", line 1
    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下さい。

```
[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)
<ipython-input-17-3c72f435fb3f> 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)
      4 print(len(res_years) == 142, len(res_temps) == 142, res_years[0] == 1875,
↳res_temps[0] == 22.3)
      5 res_years, res_temps = plot_tokyotemps(1875, 12)

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

29.4 散布図

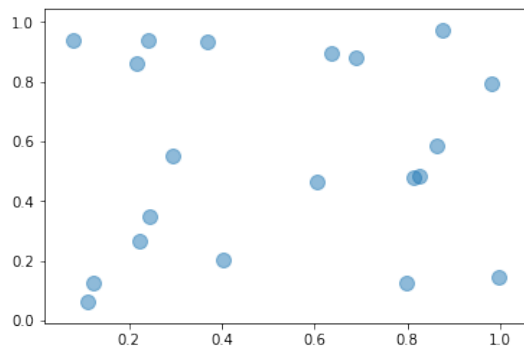
散布図は、`pypplot` モジュールの “`scatter`” () 関数を用いて描画できます。

具体的には、次の様にリスト `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。

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ています。プロットする点のマーカーの色や形状は、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、それぞれマーカーの大きさと透明度を指定することができます。

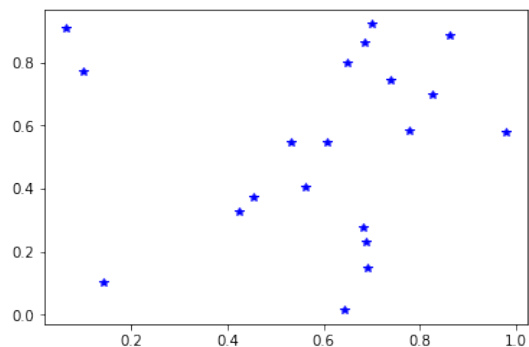
```
[18]: # グラフの x 軸の値となる配列
x = np.random.rand(20)
# グラフの y 軸の値となる配列
y = np.random.rand(20)

# 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、三番目の引数にプロットする点のマーカー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');
```



29.5 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` を作成して下さい。

以下のセルの ... のところを書き換えて解答して下さい。

```
[20]: import ...
...
def scatter_tokyotemps():
...

File "<ipython-input-20-0c22feb25120>", line 1
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下さい。

```
[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)
<ipython-input-21-98c9832c2013> 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)

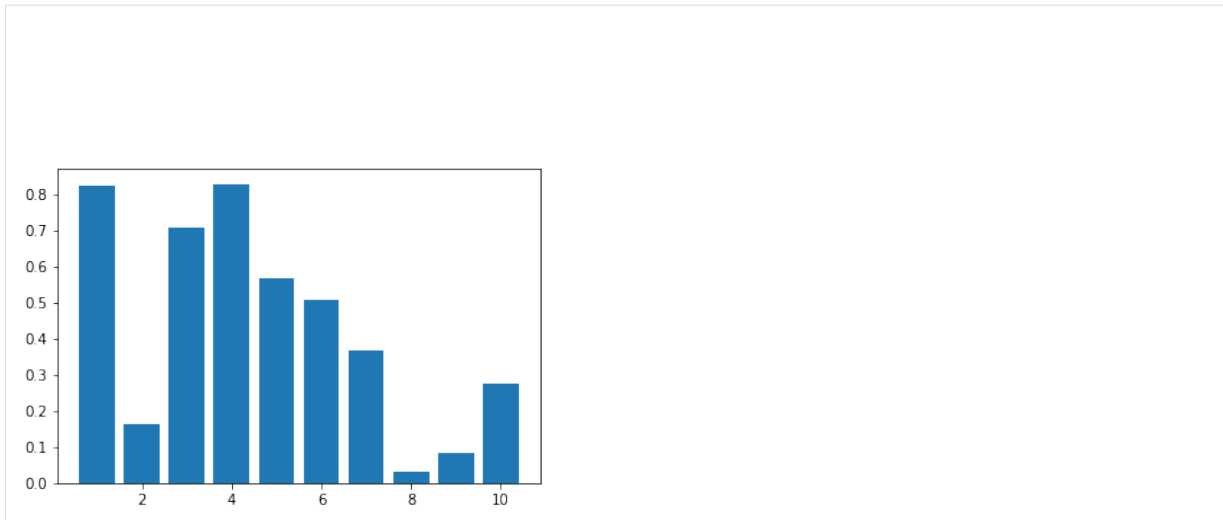
NameError: name 'scatter_tokyotemps' is not defined
```

29.6 棒グラフ

棒グラフは、matplotlib 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);
```



29.7 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 を作成して下さい。

以下のセルの ... のところを書き換えて解答して下さい。

```
[23]: import ...
...
def bar_tokyotemps(year1, month1, year2, month2):
    ...

File "<ipython-input-23-b27113b5f2c4>", line 1
    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下さい。

```
[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)
<ipython-input-24-733fe05486b6> 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)

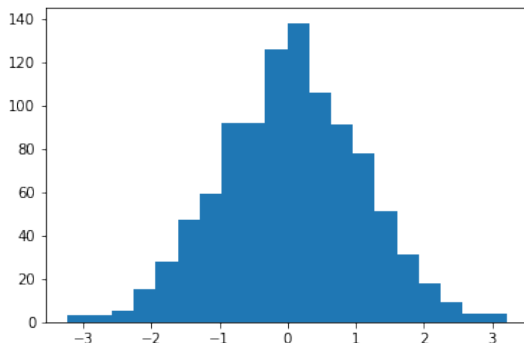
NameError: name 'bar_tokyotemps' is not defined
```

29.8 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);
```



29.9 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 を作成して下さい。

以下のセルの … のところを書き換えて解答して下さい。

```
[26]: import ...
...
def hist_tokyotemps(year1, month1, year2, month2, mybin):
    ...
```

```
File "<ipython-input-26-6b59c7306bf0>", line 1
    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下さい。

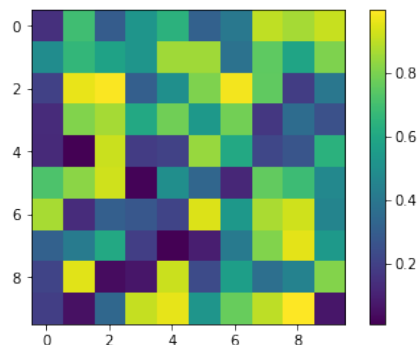
```
[27]: res_temps = hist_tokyotemps(1875, 6, 2000, 6, 50)
print(len(res_temps) == 1501, res_temps[0] == 22.3, res_temps[1500] == 22.5)
```

```
-----  
NameError                                Traceback (most recent call last)  
<ipython-input-27-0d26f8df3a7b> 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
```

29.10 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);
```



29.11 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` を作成して下さい。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下さい。


```
[29]: import ...
...
def heat_tokyotemps():
...

File "<ipython-input-29-403e5d9a71a4>", line 1
    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下さい。

```
[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)
<ipython-input-30-aaba953b7261> in <module>
----> 1 ary1 = heat_tokyotemps()
      2 print(ary1[0][0] == 2, ary1[1][1] == 2, ary1[2][0] == 28)
      3 #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
      4 #ary1 = np.flip(ary1, axis=0)
      5 #im=plt.imshow(ary1)

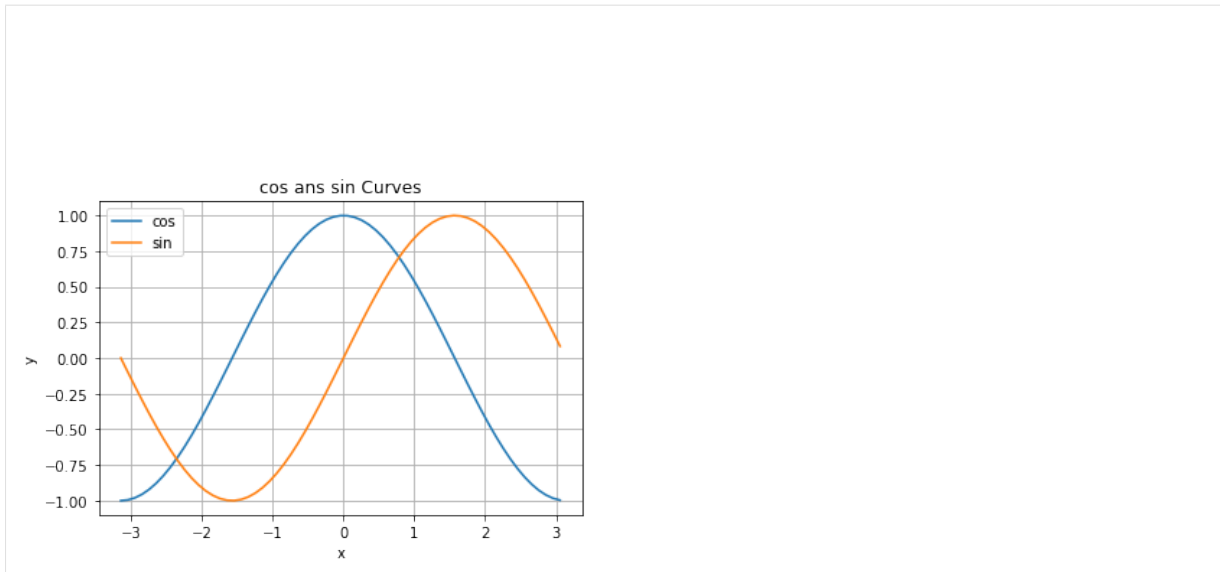
NameError: name 'heat_tokyotemps' is not defined
```

29.12 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 and 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');
```



29.13 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
                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)
```

(continues on next page)

(continued from previous page)

```
        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)
        #print(temps)
        plt.bar(months, temps)
    return months, temps
```

```
[36]: import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
import csv
```

(continues on next page)

(continued from previous page)

```
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 関数で
                # 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 `import` する必要があります。

```
[1]: import re
```

30.1 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。

例えば、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のできる関数に “**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。

```
[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。

```
[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:
```

(continues on next page)

(continued from previous page)

```
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 `print` したものを見て下さい。

`<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。例えば、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'>
<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、次の様な場合、`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。

```
[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下さい。

```
[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')
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は、正規表現の先頭にキャレット (^) を付けてください。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 (\$) を付けてください。

```
[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)
```

```
<re.Match object; span=(2, 5), match='abc'>
None
<re.Match object; span=(3, 6), match='def'>
None
```

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。

例えば、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|)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|)r', 'That colour matches your suit.') #正規表現内の () につ
      いては下で述べます
      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下さい。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下さい。

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、この様に正規表現を繋げて新しい正規表現を作る演算を接続 といいます。

また、a* も正規表現です。この正規表現では、0個以上の a からなる文字列を表しています。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)
```

(continues on next page)

(continued from previous page)

```
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（a という文字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いませんか？ しかし、a* は「0 個以上」の a からなる文字列を表すことができることに注意して下さい。その結果として、cde という文字列も「0 個」の a からなる文字列（この様な長さ 0 の文字列を空列、もしくは空文字列 と言います）を含むので cde という文字列の先頭部分（の空列）にマッチしているのです。

例えば、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接続、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。例えば、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。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('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'>
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、積 (×) < 和 (+) < べきと同じですので、直感的にも分かり易いと思います。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下さい。例えば、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出来て便利です。

```
[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'>
```

30.2 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、含まれないければ None を返す関数 check_monthstr を作成して下さい。ただし、「月を表す文字列」は次の様な文字列とします。

1. 長さ 2 の 'mm' という文字列
2. 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下さい。

```
[18]: import ...
def check_monthstr(str1):
    ...

File "<ipython-input-18-1a32806d32e9>", line 1
    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下さい。

```
[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)
<ipython-input-19-2e84418df86e> 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)
      2 print(check_monthstr('mon1521vb') == None)
      3 print(check_monthstr('00an23') == None)
      4 print(check_monthstr('13302').group() == '02')

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

30.3 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、そうでない場合は True を返す関数 `check_ACGTstr` を作成して下さい。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 を返して下さい。

以下のセルの ... のところを書き換えて解答して下さい。

```
[20]: import ...
def check_ACGTstr(str1):
    ...

File "<ipython-input-20-4144afe1298b>", line 1
    import ...
    ^
SyntaxError: invalid syntax
```

```
[21]: print(check_ACGTstr('ACGCTAGCacATcGgAaaTtGCacT') == True)
print(check_ACGTstr(':ACaacgta24FgtGH') == False)
print(check_ACGTstr('') == False)

-----
NameError                                Traceback (most recent call last)
<ipython-input-21-578e52450be5> in <module>
--> 1 print(check_ACGTstr('ACGCTAGCacATcGgAaaTtGCacT') == True)
      2 print(check_ACGTstr(':ACaacgta24FgtGH') == False)
      3 print(check_ACGTstr('') == False)

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

30.4 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、含まれいなければ `None` を返す関数 `check_timestr` を作成して下さい。ただし、「時刻を表す文字列」は次の様な文字列とします。

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下さい。

```
[22]: import ...
def check_timestr(str1):
    ...

File "<ipython-input-22-49048bc2cff9>", line 1
    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下さい。

```
[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)
<ipython-input-23-b20693e52e53> in <module>
----> 1 print(check_timestr('10:23').group() == '10:23') # group() については後半に説明が
      2 print(check_timestr('time?1023') == None)
      3 print(check_timestr('time?11:23').group() == '11:23')
      4 print(check_timestr('12:3xx1;23ah23:23') == None)

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

30.5 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、含まれいなければ `None` を返す関数 `check_ipv4str` を作成して下さい。ただし、「IPv4 を表す文字列」は次の様な文字列とします。

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下さい。

```
[24]: import ...
def check_ipv4str(str1):
    ...

File "<ipython-input-24-37c7b1e3c96b>", line 1
    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下さい。

```
[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)
<ipython-input-25-a5d78279eda5> 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)
      3 print(check_ipv4str('IP?=25:25:55:155') == None)
      4 print(check_ipv4str('IP?=255:255:255') == None)

NameError: name 'check_ipv4str' is not defined
```

30.6 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、含まれなければ `None` を返す関数 `check_monthdaystr` を作成して下さい。ただし、「月と日を表す文字列」は次の様な文字列とします。

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下さい。

```
[26]: import ...
def check_monthdaystr(str1):
    ...

File "<ipython-input-26-27e73dd1e9b1>", line 1
    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下さい。

```
[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)
<ipython-input-27-d292ac577c53> 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)
      4 print(check_monthdaystr('03/24').group() == '03/24')

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

30.7 正規表現の応用 1

Python 以外のプログラミング言語や Microsoft Word などのエディタでも正規表現は用いることができますが、その使い方は言語毎（エディタ毎）に微妙に異なります。しかし、この接続・和・閉包という3つの演算は（大抵）用いることができます。以下では、（大抵の）言語・エディタでも用いることができる便利な演算（もしくは、記法）を紹介しておきましょう。

30.7.1 文字クラス

[abc] は a|b|c と同じ意味の正規表現です（文字クラス といいます）。

```
[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。例えば、[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=''>
```

(continues on next page)

(continued from previous page)

```
<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出来ますが、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。例えば、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:'>
```

30.7.2 否定文字クラス

文字クラスで用いた括弧 (`[]`) の先頭に、キャレット (^) をつけると (すなわち、`[^]` とすると)、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を作成できます。例えば、`[^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判定されます。例えば、`[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='^'>
```

30.8 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出来ます。以下では 2 つの基本的な関数を紹介します。

30.8.1 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。

具体的には次の様すると、

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）が返り値となります。（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) # \(と\) 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」の節を参照して下さい

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\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。

30.8.2 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り値となります。

具体的には次の様に用います。

以下が典型例です。

`^[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。

30.8.3 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ス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。

```
[ ]:
```

```
[ ]:
```

```
[ ]:
```

```
[ ]:
```

30.9 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` を作成して下さい。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下さい。

以下のセルの ... のところを書き換えて解答して下さい。

```
[41]: import ...
def get_engsentences():
    ...

File "<ipython-input-41-b70aefd4c3ef>", line 1
    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下さい。

```
[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)
<ipython-input-42-24963a0722b0> in <module>
----> 1 list1 = get_engsentences()
      2 print(len(list1) == 289)
      3 print(list1[0] == 'a')
      4 print(list1[100] == 'in')
      5 print(list1[288] == 'would')

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

30.10 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 を作成して下さい。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下さい。

以下のセルの ... のところを書き換えて解答して下さい。

```
[43]: import ...
def get_engsentences2():
    ...

File "<ipython-input-43-e4d056de09d5>", line 1
    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下さい。

```
[44]: list1 = get_engsentences2()
print(len(list1) == 149)
print(list1[0] == 'a')
print(list1[100] == 'proclaim')
print(list1[148] == 'would')

-----
NameError                                Traceback (most recent call last)
<ipython-input-44-1be5eff39300> in <module>
----> 1 list1 = get_engsentences2()
      2 print(len(list1) == 149)
      3 print(list1[0] == 'a')
      4 print(list1[100] == 'proclaim')
      5 print(list1[148] == 'would')

NameError: name 'get_engsentences2' is not defined
```

30.11 正規表現の応用 2

30.11.1 ドット

．（ドット）はあらゆる文字とマッチする正規表現です。

```
[45]: 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、文字クラスの中ではドットを用いても、あらゆる文字とはマッチしません。あくまで、ピリオドとマッチします。

```
[46]: match1 = re.search('[.]', 'Hello')
      print(match1)
      match1 = re.search('[.]', '3.141592')
      print(match1)

None
<re.Match object; span=(1, 2), match='.'>
```

繰り返しになりますが、今まで扱ってきた演算子などもその様な例に該当します。文字クラスの中では多くの文字が1つの文字として扱われます。

```
[47]: match1 = re.search('[*]', '*|()')
      print(match1)
      match1 = re.search('[|]', '*|()')
      print(match1)
      match1 = re.search('[()]', '*|()')
      print(match1)

<re.Match object; span=(0, 1), match='*>
<re.Match object; span=(1, 2), match='|'>
<re.Match object; span=(3, 4), match='('>
```

30.11.2 ?

`a?` は、`a` を0個、もしくは1個含む文字列とマッチします。すなわち、`a(bc)?` は `a|abc` と同じ意味の正規表現です。

```
[48]: match1 = re.search('colou?r', 'colour')
      print(match1)
      match1 = re.search('colou?r', 'color')
      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'>
<re.Match object; span=(0, 5), match='color'>
```

•

~

`a+` は1個以上の `a` からなる文字列とマッチする正規表現です。すなわち、`a+` は `aa*` と同じ意味の正規表現です。

```
[49]: match1 = re.search('boo+', 'boooo!')
      print(match1)
      match1 = re.search('boo+', 'bo!')
      print(match1)
```

(continues on next page)

(continued from previous page)

```
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。

30.11.3 {x, y}

正規表現 R に対して、R{x, y} は R を x 回以上かつ y 回以下繰り返す文字列とマッチする正規表現です。

```
[50]: match1 = re.search('bo{3,5}', 'booooooo!')
print(match1)
match1 = re.search('bo{3,5}', 'boo!')
print(match1)
match1 = re.search('a{2,5}', 'acaad')
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='booooo'>
None
<re.Match object; span=(3, 5), match='aa'>
<re.Match object; span=(0, 5), match='1,298'>
None
```

30.12 メタ文字

以下では、良く使うメタ文字（特殊シーケンス）を紹介します。

‘\t’ は タブを表します。~~~~~

```
[51]: 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。~~~~~

```
[52]: 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。~~~~~

```
[53]: match1 = re.search('b\S', 'a      b      bc      d')
print(match1)

<re.Match object; span=(4, 6), match='bc'>
```

‘[a-zA-Z0-9]’ ~~~~~

```
[54]: match1 = re.search('\w\w', 'abcde')
print(match1)
match1 = re.search('b\w*g', 'abcdefgh')

<re.Match object; span=(0, 2), match='ab'>
```

‘\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。
~~~~~

```
[55]: match1 = re.search('gW*', '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 ' >
```

‘[0-9]’ と同じ意味です。~~~~~

```
[56]: 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’ 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。  
~~~~~

```
[57]: match1 = re.search('\D*', 'He has 10 apples.')
print(match1)

<re.Match object; span=(0, 7), match='He has ' >
```

30.13 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。

```
[58]: import ...
def sumnumbers(s):
    ...

File "<ipython-input-58-2eb053df1419>", line 1
    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下さい。

```
[59]: print(sumnumbers(' 2 33 45, 67.9') == 156)

-----
NameError                                Traceback (most recent call last)
<ipython-input-59-59e10eb64761> in <module>
----> 1 print(sumnumbers(' 2 33 45, 67.9') == 156)

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

30.14 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` を作成して下さい。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` を返して下さい。

以下のセルの `...` のところを書き換えて解答して下さい。

```
[60]: import ...
def check_ACGTstr(str1):
    ...

File "<ipython-input-60-4144afe1298b>", line 1
    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下さい。

```
[61]: print(check_ACGTstr('AcCGTAGCacATcGgAaaTtGCacT') == True)
print(check_ACGTstr(':ACaacgta24FgtGH') == False)
print(check_ACGTstr('') == False)

-----
NameError                                Traceback (most recent call last)
<ipython-input-61-578e52450be5> 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
```

30.15 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` を作成して下さい。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下さい。

```
[62]: import ...
def check_postalcode(str1):
    ...

File "<ipython-input-62-824b53a3f099>", line 1
    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下さい。

```
[63]: 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)
<ipython-input-63-07617d20e3bc> in <module>
----> 1 print(check_postalcode('113-8654') == True)
      2 print(check_postalcode('119-110') == False)
      3 print(check_postalcode('abc-defg') == False)
      4 print(check_postalcode('〒153-0041') == False)
      5 print(check_postalcode('113-86547') == False)

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

30.16 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` を作成して下さい。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下さい。

```
[64]: import ...
def check_extension(str1):
    ...

File "<ipython-input-64-58b9fba638b9>", line 1
    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下さい。

```
[65]: print(check_extension('24115') == True)
print(check_extension('46858') == False)
```

(continues on next page)

(continued from previous page)

```
print(check_extension('^e2^98^8e46666') == False)
print(check_extension('467890') == False)
```

```

NameError                                Traceback (most recent call last)
<ipython-input-65-1a10a713dbe3> in <module>
----> 1 print(check_extension('24115') == True)
      2 print(check_extension('46858') == False)
      3 print(check_extension('^e2^98^8e46666') == False)
      4 print(check_extension('467890') == False)

NameError: name 'check_extension' is not defined

```

30.17 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様に \ を前につけてやる必要があります。

```
[66]: 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、例えば、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 '>
```

特殊な意味をもつ記号は次の 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下さい。

30.18 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。

30.18.1 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。

具体的には次の様に実行します。

[illegible]

```
['had', 'had', 'had', 'had', 'had', 'had', 'had', 'had', 'had', 'had', 'had']
['Peter', 'Piper', 'picked', 'peck', 'pickled', 'peppers']
```

30.18.2 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、特殊なリストに格納して返します。

具体的には次の様に実行します。

返り値は特殊なリストであり、for 文の `in` の後ろに置いて使って下さい。

```
[68]: 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 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'>
<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'>
```

30.18.3 group

group は、正規表現にマッチした文字列を（部分的に）取り出すための、match オブジェクトのメソッドです。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様にします。i = 0 の場合は、マッチした文字列全体を取得できます。

```
[69]: 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下さい。

```
[70]: 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)
<ipython-input-70-92d13e1aa636> 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))

AttributeError: 'NoneType' object has no attribute 'group'
```

例えば、if 文でエラーを回避します。

```
[71]: 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

30.19 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、そうでない場合は True を返す関数 check_ACGTstr

を作成して下さい。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 を返して下さい。
以下のセルの ... のところを書き換えて解答して下さい。

```
[72]: import ...
def check_ACGTstr(str1):
    ...

File "<ipython-input-72-4144afe1298b>", line 1
    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下さい。

```
[73]: print(check_ACGTstr('AcCGTAGCacATcGgAaaTtGCacT') == True)
print(check_ACGTstr(':ACaacgta24FgtGH') == False)
print(check_ACGTstr('') == False)

-----
NameError                                Traceback (most recent call last)
<ipython-input-73-578e52450be5> 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
```

30.20 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 を作成して下さい。ただし、一般に w タグは、次の様な形式で記述されます。

例えば、以下の様な具合です。

以下のセルの ... のところを書き換えて解答して下さい。

```
[74]: import ...
def get_pos():
    ...

File "<ipython-input-74-c3ee74e88c39>", line 1
    import ...
    ^
SyntaxError: invalid syntax

-----
NameError                                Traceback (most recent call last)
<ipython-input-75-ec3d414c3b66> in <module>
----> 1 print(get_pos()['They '] == 'PRON')
      2 print(get_pos()['know '] == 'VERB')

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

30.21 練習の解答

```
[76]: 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' # としても良い
    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
```

```
[77]: 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 = '([0-9]|10|11)' # 文字クラスを使ってと表すことも出来ます (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出来ます (文字クラスは後で学習します)
    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
```

```
[78]: 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出来ます (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出来ます
    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出来ます
    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
```

```
[79]: 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]でも良い
    reg_0to9 = '(0|1|2|3|4|5|6|7|8|9)' # [0-9]でも良い
    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)
```

(continues on next page)

(continued from previous page)

```
# 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
return match1
```

```
[80]: import re
def sumnumbers(s):
    numbers = re.split('[^0-9]+', s)
    numbers.remove('')
    n = 0
    for number in numbers:
        n += int(number)
    return n
```

```
[81]: 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良い
        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
```

```
[82]: 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] = 'anthing 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良い)
            #word_dict[word.lower()] = 'anthing 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
```

```
[83]: 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
```

(continues on next page)

(continued from previous page)

```
#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
    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
```

```
[84]: 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
```

```
[85]: 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
```

```
[86]: import re
def get_pos():
```

(continues on next page)

(continued from previous page)

```
str_file = 'BIS.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
```

▲イテラブルとイテレータ

イテラブルとイテレータについて説明します。

参考 - <https://docs.python.org/ja/3/tutorial/classes.html> - <https://docs.python.org/ja/3/library/abc.html>

6-1 で利用法を説明したイテレータや、暗黙的に利用されるイテラブルについて、より詳しく説明します。

31.1 ジェネレータ関数と無限イテレータ

3-3 にて、リスト内包表記と似て非なるものとして、イテレータを返すジェネレータ式を説明しました。例を再掲します。

```
[1]: it = (x * 3 for x in 'abc')
      for x in it:
          print(x)
```

```
aaa
bbb
ccc
```

このジェネレータ式と同等のものを、関数形式で定義できます。

```
[2]: def gen():
      for x in 'abc':
          yield x * 3
      it = gen()
      for x in it:
          print(x)
```

```
aaa
bbb
ccc
```

関数 `gen()` は、`return` で値を返すのではなく `yield` で値を返しています。 `gen()` が返すイテレータ `it` は、`yield` された値を `next()` を適用する度に順に生成します。このような関数を、ジェネレータ関数もしくは単にジェネレータと呼びます。また、ジェネレータ関数・ジェネレータ式が返すイテレータのことを特に、ジェネレータイテレータと呼びます。

ジェネレータ関数は、ジェネレータ式と違って、途中状態を局所変数で管理できるので、より豊富なイテレータを構成できます。例えば、次の `ascend()` は、与えられた整数から 1 ずつカウントアップする無

限列のイテレータを返します。

```
[3]: def ascend(n):  
    while True:  
        yield n  
        n += 1  
  
for x in ascend(1):  
    if x == 10:  
        break # これがないと無限ループする  
    else:  
        print(x)
```

```
1  
2  
3  
4  
5  
6  
7  
8  
9
```

このように、ジェネレータ関数を使うことで、様々なイテレータを定義できるようになります。因みに、`itertools` モジュール内には、`ascend()` を少しだけ拡張した関数 `count()` が定義されています。

31.2 イテラブルと for 文

さて、イテレータとコレクションが別物であるのに、なぜ同様に `for` 文で走査できるのでしょうか。その答えは、組み込み関数 `iter()` にあります。

`iter()` にコレクションを与えると、それを前から順に走査するイテレータを返します。

```
[4]: it = iter('abc')  
print(next(it))  
print(next(it))  
print(next(it))
```

```
a  
b  
c
```

`iter()` にイテレータを渡すと、何もせずにそのイテレータを返します。

```
[5]: it = iter('abc')  
it2 = iter(it)  
it is it2
```

```
[5]: True
```

実は、`for` 文は、この `iter()` を `in` の後ろのオブジェクトに適用して得られたイテレータを使って、反復しています。つまり、

```
for x in xs:  
    print(x)
```

この `for` 文は、次のような反復処理と（変数 `it` の導入を除いて）等価です。

```
it = iter(xs)  
try:  
    while True:  
        x = next(it)
```

(continues on next page)

(continued from previous page)

```
        print(x)
    except StopIteration:
        pass
```

Python では、イテレータという反復処理を表現した抽象的なオブジェクトを通すことで、具体的なデータ型の違いを忘れて、統一的に反復処理ができるように設計されています。

`iter()` を適用可能なオブジェクトのことを、イテラブル (iterable) と呼びます。

イテラブルを受け取る関数は、イテレータもコレクションも同様に受け取って処理します。6-1 において、コレクションやイテレータを取ると説明していた関数は、正確にはイテラブルを取ります。

31.3 イテレータとイテラブルの定義

さて、イテレータとイテラブルがどんなものか分かったところで、それらをオブジェクト指向プログラミングの観点で改めて定義します。

イテラブルとは、`__iter__()` メソッドを持つオブジェクトです。`iter()` は、与えられたオブジェクトの `__iter__()` メソッドを呼び出しているだけです。したがって、`__iter__()` メソッドは、レシーバの持つ要素を走査するイテレータを返すことが期待されます。

イテレータとは、`__iter__()` メソッドと `__next__()` メソッドを持つオブジェクトです。ただし、`__iter__()` メソッドは、そのレシーバをそのまま返します。`__next__()` は、次の要素を返すか、終わりに達していたら `StopIteration` を送出します。`next()` は、与えられたオブジェクトの `__next__()` メソッドを呼び出すだけです。この `__iter__()` メソッドと `__next__()` メソッドに関する規約を、イテレータプロトコルと呼ばれます。

クラス定義にて `__iter__()` と `__next__()` を定義すれば、そのインスタンスはイテレータとなります。次の `Ascend` クラスは、前述のジェネレータ関数 `ascend()` と同等のイテレータを返します。

```
[6]: class Ascend:
      def __init__(self, n):
          self.n = n
      def __iter__(self):
          return self
      def __next__(self):
          n = self.n
          self.n += 1
          return n
it = Ascend(1)
next(it)
```

```
[6]: 1
```

```
[7]: next(it)
```

```
[7]: 2
```

ここで、`__init__()` はコンストラクタに対応する特殊メソッドであり、`Ascend` インスタンスの属性 `self.n` は、次の `next()` で返される値を保持します。`self.n = n` という代入によって、このインスタンスの属性 `self.n` が設定されます。代入の右辺の `n` はメソッドの仮引数で、`Ascend(1)` の実引数 `1` の値が渡されます。属性 `self.n` は `next()` で参照されます。

このように、メソッドによって特徴付けられた統一的な反復処理は、オブジェクト指向プログラミングの典型例です。

イテレータやイテラブルの定義にはメソッドを要求しますが、自前のイテレータを定義するときに、クラスを使う必要はありません。ジェネレータ関数を通してイテレータを定義すれば、實際上十分です。単純なイテレータであれば、ジェネレータ式でも十分でしょう。尚、ジェネレータ関数はメソッドにもできます。次の `CanaryList` は、それを走査するイテレータが、`next()` で値を生成する度に、その値を `print` するように `list` を拡張したクラスです。

```
[8]: class CanaryList(list):
      def __iter__(self):
          for x in super().__iter__():
              print(x)
              yield x

for x in CanaryList([1,2,3]):
    pass

1
2
3
```

`super().__iter__()` によって、`list` クラスの `__iter__()` メソッドがこのインスタンスに対して呼び出されます。

31.4 コレクションの階層

これまで要素の集まりのことをコレクションと呼び、文字列・リスト・タプルなどをシーケンスと呼んできました。このコレクションやシーケンスは、具体的なクラスを指したものではなかったのですが、Python では、抽象クラスという形で、その実装要件が定義されています。

抽象クラス	継承している抽象クラス	抽象メソッド	性質の説明
Container		<code>__contains__()</code>	<code>in</code> 演算子が適用できる
Sized		<code>__len__()</code>	組込み関数 <code>len()</code> が適用できる
Iterable		<code>__iter__()</code>	組込み関数 <code>iter()</code> が適用できる
Iterator	Iterable	<code>__next__()</code>	組込み関数 <code>next()</code> が適用できる
Reversible	Iterable	<code>__reversed__()</code>	組込み関数 <code>reverse()</code> が適用できる
Collection	Container Sized Iterable		
Sequence	Collection Reversible	<code>__getitem__()</code>	項目アクセス演算 (<code>x[k]</code>) が適用できる
Mapping	Collection	<code>__getitem__()</code>	項目アクセス演算 (<code>x[k]</code>) が適用できる

ここで抽象メソッドとは、その抽象クラスが実装を要求するメソッドを意味します。抽象クラスを継承する場合、その実装責任も継承します。つまり、コレクションとは、`__contains__()`・`__len__()`・`__iter__()` メソッドを持ち、`in`・`len()`・`iter()` が適用可能なオブジェクトです。この階層を見れば、`reverse()` がコレクション一般には適用できず、シーケンスを引数に取ることが一目で分かります。

多くの組込み関数や `for` 文・内包表現は、最も要件の緩い `Iterable` しか要求しないので、様々なデータ型を統一的に扱えるのです。

この階層の説明は、意図的に簡略化しています。詳細は、`collections.abc` モジュールのドキュメント <<https://docs.python.org/ja/3/library/collections.abc.html>>‘__’を参照してください。

CHAPTER 32

索引など

- search

=, 21
*, 4, 36, 45, 121, 125
**, 4
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, 132
関数本体, 99
関数名, 99
クラスオブジェクトを更新するときは、よく注意しましょう。, 165
全角の空白, 8
注意, 12, 226
+, 4, 8, 45
+=, 12
-, 4, 8
-=, 12
/, 5
//, 5
==, 21
#, 5
%, 5
__class__, 26
__getitem__(), 160
__missing__(), 161
__name__, 131
__setitem__(), 161
__str__(), 160
>, 21
>=, 21
<, 21
<=, 21
3 項演算子, 66

add, 189
and, 21
append, 48, 173
argument, 100
as, 110, 121, 125
assign, 172
assignment statement, 12
augmented assignment statement, 12

bar, 237
bokeh, 221
bokeh.models.ColumnDataSource, 225
bokeh.models.LinearColorMapper, 226
bokeh.plotting, 221
bokeh.plotting.figure(), 221
bokeh.plotting.output_file(), 226
bokeh.plotting.output_notebook(), 221
bokeh.plotting.reset_output(), 226
bokeh.plotting.show(), 221
break, 84

capitalize, 38
circle(), 222
clear, 71, 190
close(), 107
collections, 164
collections.Counter, 161
collections.namedtuple(), 162
concat, 174
continue, 84
copy, 50, 72
count, 37, 46
cross(), 222
csv, 202, 205
csv.reader, 202
csv.writer, 205
CSV ファイル, 169, 202
CSV ライター, 205
CSV リーダー, 202

DataFrame, 167
def, 13
del, 50, 69, 172
describe(), 174
difference, 190
discard, 190
drop, 172, 173

elif, 63
else, 20, 62, 63, 84
encoding, 206
enumerate, 82
enumerate(), 152
extend, 49

False, 22
filter(), 151
find, 37

fit, 179
flatten(), 136
for, 54, 74
from, 121, 125
functional programming, 146
functool.reduce(), 148

get, 70
grid, 230
groupby, 175

higher-order function, 146
hist, 239

if, 20, 61–63
iloc, 170
import, 9, 120
import 文, 124
in, 34, 46, 69, 83
in-place, 48
index, 37, 46
inheritance, 160
insert, 49
intersection, 190
isinstance(), 162
items, 71
iter(), 148
itertools, 154
itertools.chain(), 153
itertools.chain.from_iterable(), 153

json, 215
json.dump, 215
json.dumps, 215
json.load, 215
JSON 形式, 214

keys, 71
KMeans, 182

legend, 230
len, 32, 45, 69
line(), 221
LinearRegression, 181
list, 52
loc, 171
LogisticRegression, 178
lower, 38

maintainability, 158
map(), 150
match, 245
match オブジェクト, 245
math, 9
math.cos, 9
math.pi, 9
math.sin, 9
math.sqrt, 9
Matplotlib, 227
matplotlib, 193

max(), 143, 147
mean(), 143
merge, 175
min(), 143, 147
modularity, 158

next, 107, 203
next(), 148
None, 14, 25
not, 21
numpy, 134
numpy.arange(), 137
numpy.array(), 134
numpy.bool, 135
numpy.complex128, 135
numpy.dot(), 142
numpy.float64, 135
numpy.histogram(), 225
numpy.identity(), 144
numpy.int32, 135
numpy.linalg, 145
numpy.linalg.norm(), 145
numpy.linspace(), 137
numpy.loadtxt(), 143
numpy.matmul(), 144
numpy.ndarray, 134
numpy.ones(), 137
numpy.random.binomial(), 138
numpy.random.poisson(), 138
numpy.random.rand(), 138
numpy.random.randn(), 138
numpy.savetxt(), 143
numpy.sort(), 142
numpy.sqrt(), 142
numpy.zeros(), 137

object-oriented programming, 158
OOP, 158
open(), 106
or, 21
override, 160

pandas, 167
parameter, 100
pass, 85
PCA, 185
plot, 227
pop, 49, 70, 190
predict, 179
print, 16
Python スクリプト, 126

quad(), 225

range, 77
ravel(), 136
re.I, 246
re.IGNORECASE, 246
read(), 107
read_csv, 169

readline(), 109
remove, 49, 189
replace, 36
reshape(), 136
return, 13, 83
reverse, 50
reversed(), 153
rstrip('n'), 113

savefig, 241
scatter, 235
scatter(), 224
scikit-learn, 177
search, 247
Series, 167
set, 187
setdefault, 70
shebang, 132
sort, 46
sort(), 143
sort_index(), 173
sort_values(), 173
sorted, 47
sorted(), 147
split, 36, 256
StopIteration, 149
str, 31
strip(), 220
sub, 256
sum(), 143
super(), 160

title, 230
transform, 185
True, 22
tuple, 52
type, 26, 31
type(), 162

union, 190
upper, 38

values, 71
vbar(), 225

while, 83
with, 110
write(), 111

xlabel, 230

ylabel, 230

zip(), 152

余り, 5

イテラブル, 150, 274
イテラブル (iterable) , 274
イテレータ, 97, 107, 148, 203, 274
入れ子, 13, 62, 79

インスタンス, 162
インデックス, 32, 167
インデント, 13, 61
インプレース, 48
インポート, 9, 120

エスケープシーケンス, 35, 110
エラー, 9

オーバーライド, 160
オープン, 106
大文字, 38
オブジェクト, 25, 161
オブジェクト指向プログラミング, 158
親クラス, 160

返値, 13, 100
書き込みモード, 110
掛け算, 4
数え上げ, 37
型, 25
形, 136
括弧, 7
仮引数, 13, 100
カレントディレクトリ, 117
関数, 13, 99
関数プログラミング, 146
関数定義, 13
関数内関数, 154

キー (key), 68
機械学習, 177
教師あり学習, 177
教師なし学習, 177

空行, 16
空白, 8
空文字列, 34, 249
空リスト, 44
空列, 34, 249
組み込み関数, 16
クラス, 159
クラス属性, 164
クラスタリング, 182
繰り返し, 74
クローズ, 107
グローバル変数, 18, 101

継承, 160
検索, 37

子クラス, 160
高階関数, 146
コマンドライン実行, 126
コマンドライン引数, 130
コメント, 5, 16
小文字, 38
コンストラクタ, 162

再帰, 24

再帰呼出し, 197
再帰関数, 197
差集合, 188, 190
参照値, 25

ジェネレータ, 272
ジェネレータイテレータ, 272
ジェネレータ関数, 272
ジェネレータ式, 97
次元削減, 185
辞書型, 68
辞書内包表記, 97
値 (value), 68
実行エラー, 9, 27, 28
実数, 5
実引数, 100
集合内包表記, 96
集合演算, 188
条件付き内包表記, 96
条件分岐, 20, 61
シリーズ, 167
真理値, 22
真理値配列によるインデックスアクセス, 144

スライス, 33, 169

整除, 5
正規表現, 245
整数, 5
積集合, 188, 190
絶対パス, 118
セット型, 187
線形回帰, 181, 211
選択, 248

相対パス, 117
空タプル, 51

対称差, 188, 190
代入演算子, 12
代入文, 12
多次元配列, 135
足し算, 4
多重リスト, 45
多重代入, 52
タプル型, 51
単項, 8

置換, 36

データフレーム, 167
デバッグ, 16, 27

特徴量, 178

内包表記, 93
名前付きタプル, 162

偽, 22

ネスト, 62

配列, 43, 134
配列のスカラ演算, 140
配列のデータ属性, 137
配列同士の演算, 141
破壊的, 48
バグ, 16, 27
パス, 117
パターン, 245
半角の空白, 8

非破壊的, 48
比較演算, 189
比較演算子, 21
引き算, 4
引数, 13, 99
否定文字クラス, 255

ファイル, 106
ブールインデックス参照, 171
浮動小数点数, 5
分割, 36
分割統治, 197
文法エラー, 9, 27, 28

閉包, 248
べき乗, 4
べき表示, 6
変数, 11, 38

保守性, 158

真, 22
マッチする, 245

無名関数, 146

メソッド, 26, 36, 158

文字クラス, 254
文字コード, 112, 206
文字コード宣言, 132
モジュール, 9, 120, 123
モジュール性, 158
モジュール名, 124
文字列, 38
文字列の比較演算, 38
文字列型, 31

優先順位, 7
ユニバーサル関数, 142

読み込みモード, 106
予約語, 14

ライブラリ, 9
ラムダ式, 146

リスト型, 43

累算代入文, 12
ループ, 74

レシーバ, 159
接続, 248

ローカル変数, 15, 100
ロジスティック回帰, 178
論理エラー, 27, 28

和, 248
ワイルドカード, 121
和集合, 188, 190
割り算, 5