Python プログラミング入門
(学内限定)

2019 年 5 月 18 日

目次

第 1 回

1.0　Jupyter Notebook の使い方

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。

1.0.1　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。

```
In [ ]:
```

1.0.2　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

1.0.3 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。

1.0.4 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。

In [ ]:

1.0.5 （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 のインタープリタ）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 のインタープリタ）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。

```
In [ ]: while True:
            pass
```

In [ ]:

1.1 数値演算

1.1.1 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。

In [ ]:

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。

In [ ]: 7-2

In [ ]: 7*2

In [ ]: 7**2

- 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。

In [ ]: 7/2

In [ ]: 7//2

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。

In [ ]: 7/1

In [ ]: 7//1

整除の**余り**を求めたいときは、別の演算子 % を用います。

In [ ]: 7%2

1.1.2 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。

In [ ]: # 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
```

In [ ]: # 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

```
# 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
```

1.1.3 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。）

In [ ]: 7/1

In [ ]: 7//1

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。

In [ ]: 7/1 == 7//1

+ 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。

In [ ]: 2+5

In [ ]: 2+5.0

/ の結果は必ず実数となります。

In [ ]: 7/1

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。

In [ ]:

In [ ]:

In [ ]:

1.1.3.1 実数のべき表示

In [ ]: 2.0**1000

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。

In [ ]: 2.0**-1000

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。

1.1.3.2 いくらでも大きくなる整数

In [ ]: 2**1000

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。

In [ ]: 2**2**2**2**2

1.1.4 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。

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

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

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

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

```
In [ ]: 56 ** 16
```

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。

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

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

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。

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

以上のように、演算子によって式の評価の順番がか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。

```
In [ ]:
```

```
In [ ]:
```

```
In [ ]:
```

1.1.4.1 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が書かれます。)

```
In [ ]: -3
```

```
In [ ]: +3
```

1.1.5 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。

1.1.6 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。

```
In [ ]: 7 **2
```

```
In [ ]: 7* *2
```

1.1.6.1 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。

```
In [ ]: 7　**2
```

1.1.7 エラー

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

```
In [ ]: 10/0
```

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。

```
In [ ]: 10/
```

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。

1.1.8 数学関数（モジュールの import）

```
In [ ]: import math
```

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

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 で求まります。

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

```
In [ ]: math.pi
```

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。

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

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。

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

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

1.1.9 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。

```
In [ ]:
```

1.2 変数と関数の基礎

1.2.1 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。

```
In [ ]: 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。

```
In [ ]: h
```

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。

```
In [ ]: 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。

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

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。

```
In [ ]: 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。

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

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。

```
In [ ]: # 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。

1.2.1.1 代入文

= という演算子は、=の左辺に、右辺の式の評価結果の値を割り当てる文（代入文）を表します。この操作は、**代入**と呼ばれています。なお、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。

```
In [ ]: 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。

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

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。

1.2.1.2 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。

```
In [ ]: 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。

```
In [ ]: 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。

```
In [ ]:
```

1.2.2 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。

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

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

```
def 関数名(引数, ...):
```

引数（ひきすう）とは、関数が受け取る値を指し示す変数のことです。**仮引数**（かりひきすう）ともいいます。
`:`以降は関数定義の本体であり、関数の処理を記述する部分として以下の構文が続きます。

```
    return 式
```

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

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

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

では、定義した関数 `bmi` を呼び出してみましょう。

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

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種なので、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の中に記述できます。

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

もう一つ関数を定義してみましょう。

```
In [ ]: 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。

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

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。

1.2.2.1 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。

1.2.3 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。

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

```
In [ ]: 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)
```

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

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

1.2.4 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。

```
In [ ]: 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。

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

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

そして、`s` を使った式が計算されて return 文で返されます。ここで、関数定義のひとまと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。

```
In [ ]: s
```

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。

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

```
In [ ]: s
```

1.2.5 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。

```
In [ ]: 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))
```

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

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します。

1.2.6 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。

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

なお、

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

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

1.2.7 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。

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

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

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

1.2.8 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。

というように、名前だけで参照することもありますが、

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。

というように、引数を明示して参照することもあります。

ときには、

heron() は三角形の面積を返します。

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

関数には引数がゼロ個のものがあるのですが、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。

1.2.9 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。

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

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

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

```
In [ ]: 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)
```

1.2.10 ▲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。

```
In [ ]: g = 9.8
```

```
In [ ]: 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。

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

```
In [ ]: 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。

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

以下はより簡単な例です。

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

```
In [ ]: foo()
```

```
In [ ]: bar()
```

```
In [ ]: a
```

```
In [ ]: a = 20
```

```
In [ ]: foo()
```

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。

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

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

```
In [ ]: a
```

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

1.2.11 練習の解答

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

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

```
In [ ]: 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

1.3.1 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。

```
In [ ]: 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。

```
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。

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

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

たとえば、上の条件分岐は以下のように書くこともできます。

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

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

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

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

なお、Python では、max という関数があらかじめ定義されています。

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

1.3.2 様々な条件

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

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

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

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

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

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

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

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

これを i == 1 or 2 or 3 と書くことはできませんので、注意してください。
また、not によって条件の否定をとることもできます。

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

比較演算子は、以下のように連続して用いることもできます。

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

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

1.3.3 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。

In [ ]:
```
```

In [ ]:
```
```

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

1.3.4 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。

In [ ]:
```
x = 3
```

In [ ]:
```
x > 1
```

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

```
In [ ]: x < 1
```

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

そして、真理値を返す関数を定義することができます。

```
In [ ]: 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。

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

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

このような関数は、if の後に使うことができます。

```
In [ ]: 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。

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

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

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。

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

n に 10 を入れておいて、

```
In [ ]: n = 10
```

次のセルを繰り返し実行してみましょう。

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

1.3.5 ▲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 ⇒ 10 ⇒ 5 ⇒ 16 ⇒ 8 ⇒ 4 ⇒ 2 ⇒ 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。

collaz 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。

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

```
In [ ]: collatz(3)
```

1.3.6 ▲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。

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

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

1.3.7 ▲ 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。

```
In [ ]: None
```

print で無理やり表示させると以下のようになります。

```
In [ ]: print(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。

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

1.3.8 ▲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。

オブジェクト . 属性名

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。

```
In [ ]: 'hello'.__class__
```

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。

```
In [ ]: type('hello')
```

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。

```
In [ ]: 'hello'.upper()
```

1.3.9 練習の解答

```
In [ ]: def absolute(x):
            if x<0:
```

```
        return -x
    else:
        return x
```

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

1.4 デバッグ

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。

1.4.1 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： - クオ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

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

```
In [ ]: 1 +   1
```

1.4.2 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・関数名の打ち間違え - 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

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

1.4.3 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

1.4.4 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。

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

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

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。

```
In [ ]: 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
```

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

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。

```
In [ ]: 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
```

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

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残しておきます。

```
In [ ]: 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
```

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

1.4.5 ▲ 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。

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

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

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。

第 2 回

2.1　文字列

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

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（"）で囲んで作成します。

```
In [ ]: str1 = 'hello'
        str1
```

```
In [ ]: str2 = "hello"
        str2
```

ここで作成したデータが確かに文字列（str）であることは、組み込み関数 type によって確認できます。

```
In [ ]: type(str1)
```

```
In [ ]: type(str2)
```

組み込み関数　str を使えば、任意のデータから文字列を作成できます。

1-1 で学んだ数値を文字列に変更したい場合、次のようにおこないます。

```
In [ ]: str3 = str(123)
        str3
```

文字列の長さは、組み込み関数の len を用いて次のようにして求めます。

```
In [ ]: len(str1)
```

2.1.1　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 **文字列 A** から 3 番目の要素を得たい場合は、以下のようにします。

文字列 A[2]

最初の章で定義された変数 str1 におこなうには:

```
In [ ]: str1[2]
```

次のように文字列に対してもおこなえます。

```
In [ ]: 'Thanks'[2]
```

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。

なお、このようにして取得した文字は、Python のデータとしては、長さが 1 の文字列として扱われます。

しかし、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）

```
In [ ]: str1[0] = 'H'
```

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）

```
In [ ]: str1[100]
```

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。

```
In [ ]: str1[-1]
```

2.1.2 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。

```
In [ ]: str9='0123456789'
```

```
In [ ]: str9[1:4]
```

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。

```
In [ ]: str9[0:3]
```

しかし、最初の 0 は省略しても同じ結果となります。

```
In [ ]: str9[:3]
```

同様に、最後尾の文字のインデックスも、値を省略することができます。

```
In [ ]: str9[3:]
```

```
In [ ]: str9[3:5]
```

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することができます。

```
In [ ]: str9[-4:-1]
```

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することもできます。次のように str0to9[3:10:2] と指定すると、3, 5, 7, ... という 2 おきの 10 より小さいインデックスをもつ文字からなる部分文字列を得ることができます。

```
In [ ]: print(str9[3:10:2])
        str_atn = "abcdefghijklmn"
        print(str9[3:10:2])
```

そして、3 番目の値として 1 おきを指定する　str0to9[1:4:1] は str0to9[1:4] と同じです。

```
In [ ]: print(str9[1:4:1], str9[1:4])
```

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。

```
In [ ]: print(str9[8:4:-1])
```

2.1.3 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））を作成することができます。具体的には、下記のように使用します。

```
str_blank = ''
```

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。）

```
In [ ]: str_price = '2,980 円'
        str_price.replace(',', '')
```

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）。

```
In [ ]: print("空文字列 1 = ", str9[4:2])
        print("空文字列 2 = ", str9[-1:-4])
        print("空文字列 3 = ", str9[3:3])
        print("空文字列ではない = ", str9[3:-1])
```

2.1.4 文字列の検索

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

```
文字列 B in 文字列 A
```

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

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

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

2.1.5 エスケープシーケンス

文字列を作成するにはシングル ' あるいはダブルクオート " 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。

```
In [ ]: str1 = 'This is 'MINE''
        str1
```

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。

```
In [ ]: str1 = 'This is \'MINE\''
        str1
```

実は、シングルクォートで囲むのをやめてダブルクォートを使うことでエスケープシーケンスを使わずに済ますこともできます。

```
In [ ]: str1 = "This is 'MINE'"
        str1
```

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

```
In [ ]: str1 = "時は金なり\n\"Time is money\"\nTime is \\"
        print(str1)
```

この場合も 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して、\" や \n を使わずに済ますことができます。

```
In [ ]: str1 = ''' 時は金なり
        "Time is money"
        Time is \\'''
        print(str1)
```

```
In [ ]: str1 = """時は金なり
        "Time is money"
        Time is \\"""
        print(str1)
```

2.1.6 文字列の連結

文字列同士は、+ 演算子によって連結することができます。連結した結果、新しい文字列が作られます。 元の文字列は変化しません。

```
In [ ]: hello = 'hello'
        world = ' world'
        str3 = hello + world
        print("連結元 1 =", hello, ", 連結元 2 =", world)
        print("連結結果 =", str3)
```

+ 演算子では複数の文字列を連結できましたが、* 演算子では文字列の掛け算が可能です。

```
In [ ]: hello = 'hello'
        hello * 3
```

2.1.7 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、以下のようにして使用します。

文字列．メソッド名（式，...）

文字列には以下のようなメソッドが用意されています。

2.1.7.1 置換

replace 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返します。
この操作では、元の文字列は変化しません。具体的には、次のように使用します。

文字列.replace(部分文字列 A，文字列 B)

```
In [ ]: hello = 'hello'
        hello2 = hello.replace('l', '123')
        print('replace 前 = \"',hello, '\" \nreplace 後 = \"', hello2, '\"')
```

2.1.7.2 分割

split 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。

文字列 A.split(区切り文字列 B)

リストについては 2-2 で扱います。

```
In [ ]: hello_world = "hello_world_!"
        hello_world.split('_')
```

2.1.7.3 検索

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

文字列 A.index(部分文字列 B)

ただし、指定した部分文字列が文字列に複数回含まれる場合、一番最初の出現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。

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

```
In [ ]: hello.index('l')
```

以下はエラーとなります。

```
In [ ]: hello.index('a')
```

find メソッドでも指定した**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。

文字列 A.find(部分文字列 B)

ただし、指定した部分文字列が文字列に複数回含まれる場合、一番最初の出現のインデックスが返されます。
指定した部分文字列が文字列内に含まれない場合は、index メソッドと異なり-1 が返されます。

```
In [ ]: hello.find('lo')
```

```
In [ ]: hello.find('l')
```

```
In [ ]: hello.find('a')
```

2.1.7.4 数え上げ

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

文字列 A.count(部分文字列 B)

```
In [ ]: hello.count('l')
```

```
In [ ]: "aaaaaaa".count("aa")
```

2.1.7.5 大文字・小文字

lower, capitalize, upper 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。

```
In [ ]: str1 = "DNA"
        print(str1)
```

```
In [ ]: str2 = str1.lower() # s の全ての文字を小文字にする
        print("実行前 = ", str1, "\n 実行後 = ",  str2)
```

```
In [ ]: str3 = str2.capitalize() # s の先頭の文字を大文字にする
        print("capitalize 実行元 = ", str2, ", capitalize 実行結果=",  str3)
```

```
In [ ]: str4 = str3.upper() # s を構成する全ての文字を大文字にする
        print("upper 実行元 = ", str3, ", upper 実行結果=", str4)
```

2.1.8 文字列の比較演算

数値などを比較するのに用いた比較演算子を用いて、2 つの文字列を比較することもできます。

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

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

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

2.1.9 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）。

```
In [ ]: 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。

```
In [ ]: str2 = "abc"
        print(func(str2))
```

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。

```
In [ ]: str2 = "abc"
        print(func("str2"))
```

2.1.10 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番最後にあります。）

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。

```
In [ ]: 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。

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

2.1.11 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。

```
In [ ]: 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。

```
In [ ]: print(remove_punctuations("Quiet, uh, donations, you want me to make a donation to the coas
```

2.1.12 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_bppair 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_bppair(str_atgc) を作成して下さい。

```
In [ ]: 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。

```
In [ ]: print(atgc_bppair("AAGCCCCATGGTAA") == 'TTCGGGGTACCATT')
```

2.1.13 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。

```
In [ ]: 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。

```
In [ ]: print(atgc_count("AAGCCCCATGGTAA", "A") == 5)
```

2.1.14 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。

```
In [ ]: 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。

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

2.1.15 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。

```
In [ ]: 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。

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

2.1.16 練習の解答

```
In [ ]: 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")
```

```
In [ ]: 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 gua
```

```
In [ ]: 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")
```

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

```
In [ ]: 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.")
```

```
In [ ]: 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
                return True
            return False
        #check_lower("down down down")
        #check_lower("There were doors all round the hall, but they were all locked")
```

2.2 リスト

文字列を構成する要素は文字でしたが、**リスト**（または、**配列**）では構成する要素としてあらゆるデータを指定できます。

リストを作成するには、リストを構成する要素をコンマで区切り全体をかぎ括弧, [および], でくくります。次の例は数を構成要素とするリストを作成しています。

```
In [ ]: ln = [0, 10, 20, 30, 40, 50]
        ln
```

```
In [ ]: type(ln)
```

次に文字列を構成要素とするリストを作成してみます。

```
In [ ]: ls = ['a', 'b', 'c']
        ls
```

リストは複数の種類のデータを取り扱うこともできます。

```
In [ ]: ls = [10, 'a', 20, 'b', 30]
        ls
```

次のように、何も要素を格納していない**空のリスト**（**空リスト**）を作成することもできます。空のリストは使い方の例として、後述する「append」の項を参照して下さい。

```
In [ ]: ls = []
        print(ls)
```

2.2.1 リストとインデックス

文字列の場合と同様、インデックスを指定することによりリストを構成する要素を個々に取得することができます。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]

```
In [ ]: ls = ['a', 'b', 'c']
        ls[1]
```

文字列の場合とは異なり、リストの要素は代入によって変更することができます。

```
In [ ]: ls = ['a', 'b', 'c']
        ls[1] = 'hello'
        ls
```

スライスを用いた代入も可能です。

```
In [ ]: ls = ['a', 'b', 'c']
        ls[1:3] = ['x', 'y', 'z', 'w']
        ls
```

2.2.2 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リスト内の全ての要素を一度の操作で代入することができます。

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

以下の様にしても同じ結果を得られます。

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

実は、多重代入は文字列においても実行可能です。

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

2.2.3 多重リスト

リストの要素としてリストを指定することもできます。次は二重リストの例です。

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

多重リストでは複数のインデックスによって要素を指定します。
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。(インデックスは ***0*** から開始されることを思い出してください。)

```
In [ ]: lns[1][0]
```

3 番目の（内側の）リストを取り出したいときは、次のように指定します。

```
In [ ]: lns[2]
```

次のようにリストを要素として含むリストを作成することも可能です。

```
In [ ]: lns2 = [lns, ["x", 1, [11, 12, 13]], ["y", [100, 120, 140]] ]
        print(lns2[0])
        print(lns2[1][2])
```

2.2.4 リストの操作

文字列において用いた関数・演算子などリストに対しても用いることができます。

```
In [ ]: ln = [0, 10, 20, 30, 40, 50]
        len(ln) # リストの長さ（大きさ）
```

```
In [ ]: ln[2:4] # スライス
```

```
In [ ]: 10 in ln # リストに所属する特定の要素の有無
```

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、

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

は

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

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

```
In [ ]: 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])
```

```
In [ ]: ln.index(20) # 指定した要素のリスト内のインデックス #find は使えない
```

```
In [ ]: ln.count(20) # 指定した要素のリスト内の数
```

```
In [ ]: ln + ['a', 'b', 'c'] # リストの連結
```

```
In [ ]: ln * 3 # リストの積
```

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。

```
In [ ]: ln0 = [0] * 10
        ln0
```

文字列にはない関数やメソッドも用意されています。以下では、幾つか例を挙げます。

2.2.4.1 メソッド sort

sort はリスト内の要素を昇順に並べ替えます。

リスト.sort()

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

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

sort(reverse = True) とすることで要素を降順に並べ替えることもできます。

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

また、並べ替えを行う組み込み関数も用意されています。

2.2.4.2 メソッド **sorted**

この関数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ます。

---

sorted(リスト)

```
In [ ]: ln = [30, 50, 10, 20, 40, 60]
        sorted(ln)
```

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

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。

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

ついでですが、多重リストをソートするとどの様な結果が得られるか確かめてみて下さい。

```
In [ ]: ln = [[20, 5], [10, 30], [40, 20], [30, 10]]
```

```
ln.sort()
ln
```

2.2.5 破壊的（インプレース）な操作と非破壊的な生成

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

具体的には、sort メソッド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。

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

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。この様な操作を**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が、sort メソッドはリストを返さないためその様な使い方が出来ないことに注意して下さい。

```
In [ ]: ln = [30, 50, 10, 20, 40, 60]
        ln2 = sorted(ln)
        print("sorted 関数の返り値:", ln2)

        ln = [30, 50, 10, 20, 40, 60]
        ln2 = ln.sort()
        print("sort メソッドの返り値:", ln2)
```

2.2.6 リストの操作（2）

以下では、幾つかのメソッドや関数の例を挙げます。以下の例中において行う操作は破壊的であることに注意して下さい。

2.2.6.1 append

リストの最後尾に指定した要素を付け加えます。

リスト.append(追加する要素)

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

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。

```
In [ ]: ln1 = [10, -10, 20, 30, -20, 40, -30]
        ln2 = [] # 空のリストを作成する
        ln2.append(ln1[0])
        ln2.append(ln1[2])
        ln2.append(ln1[3])
        ln2.append(ln1[5])
        print(ln2)
```

2.2.6.2 extend

リストの最後尾に指定したリストの要素を付け加えます。

リスト.extend(追加するリスト)

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

2.2.6.3 insert

リストのインデックスを指定した位置に新しい要素を挿入します。

リスト.insert(インデックス, 新しい要素)

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

2.2.6.4 remove

指定した要素をリストから削除します。

リスト.remove(削除したい要素)

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

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

```
In [ ]: ln.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
        ln

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

2.2.6.5 pop

指定したインデックスの要素をリストから削除して返します。

リスト.pop(削除したい要素のインデックス)

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

インデックスを指定しない場合、最後尾の要素を削除して返します。

リスト.pop()

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

2.2.6.6 reverse

リスト内の要素の順序を逆順にします。

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

2.2.6.7 del

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。

---

del リスト [x]

```
In [ ]: ln = [10, 20, 30, 40, 50]
        del ln[1]
        ln
```

スライスを使うことも可能です。

```
del リスト[x:y]
```

```
In [ ]: ln = [10, 20, 30, 40, 50]
        del ln[2:4]
        ln
```

2.2.6.8 **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。

```
In [ ]: ln = [10, 20, 30, 40, 50]
        ln2 = ln.copy()
        del ln[1:3]
        print(ln)
        print(ln2)
```

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。

```
In [ ]: ln = [10, 20, 30, 40, 50]
        ln2 = ln
        del ln[1:3]
        print(ln)
        print(ln2)
```

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覚えておく必要があります。

2.2.7 タプル

タプルは、リストと同じようにデータの並びであり、あらゆる種類のデータを要素とすることができます。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値をくくります。

```
In [ ]: tup1 = (1, 2, 3)
        tup1
```

```
In [ ]: type(tup1)
```

実は、丸括弧なしでもタプルを作成することができます。

```
In [ ]: tup1 = 1,2,3
        tup1
```

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。

```
In [ ]: tup1 = (1,)
        tup1
```

t = (1) だと、t = 1 と同じです。

```
In [ ]: tup1 = (1)
        tup1
```

リストや文字列と類似した操作が可能です。

```
In [ ]: tup1 = (1, 2, 3, 4, 5)
        tup1[1] # インデックスの指定による値の取得
```

```
In [ ]: len(tup1) # len はタプルを構成する要素の数
```

```
In [ ]: tup1[2:5] # スライス
```

多重代入も可能です。

```
In [ ]: tup1 = (1, 2, 3)
        (x,y,z) = tup1
        y
```

これは次の様に記述することもできます。

```
In [ ]: x,y,z = tup1
        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)
```

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。

```
In [ ]: 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
```

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。

```
In [ ]: tup1[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。

```
In [ ]: list(tup1)
```

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。

```
In [ ]: ls = [1, 2]
        tuple(ls)
```

2.2.8 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。

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

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

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

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

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

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

2.2.9 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。オブジェクトの詳細は 6-1 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。

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

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子を得る組み込み関数 id の結果も併せて示します。

```
In [ ]: print(a is b)
        print(id(a), id(b))
```

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。

```
In [ ]: c = list(a)
```

```
print(c)
print(a == c)
print(a is c)
print(id(a), id(c))
```

2.2.10 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 value in ls:
    実行文
```

---

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、一つづつ 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 から一つづつ要素を取り出し、変数 value に代入しています。実行文では vbalue を用いて取り出した要素にアクセスしています。

```
In [ ]: ls = [0,1,2]


        for value in ls:
            print("For loop:", value)
```

in の後に直接リストを記述することもできます。

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

実行文 の前にスペースがないとエラーが出ます。

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

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。

```
In [ ]: for value in  [0,1,2]:
            print("During for loop:", value)
            print("During for loop, too:", value)
```

後者の print の行のスペースの数が間違ってると、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。

```
In [ ]: 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
```

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。

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

2.2.11 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。

```
for c in str1:
    実行文
```

---

str1 で与えられる文字列を一文字ずつ大文字で出力する処理は以下のようになります。

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

2.2.12 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の一番最後にあります。）

```
In [ ]: 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。

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

2.2.13 練習

整数の要素からなるリスト ln を引数として取り、ln 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。

```
In [ ]: 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。

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

2.2.14 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_eveneindex 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。

```
In [ ]: 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。

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

2.2.15 練習

ATGC の 4 種類の文字から成る文字列 str_atgc が引数として与えられたとき、次の様なリスト 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。

```
In [ ]: 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。

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

2.2.16 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。

```
In [ ]: 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。

```
In [ ]: print(count_words("From Stettin in the Baltic to Trieste in the Adriatic an iron curtain ha
```

2.2.17 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。

```
In [ ]: 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。

```
In [ ]: print(reverse_string("No lemon, No melon") == "nolem oN ,nomel oN")
```

2.2.18 練習の解答

```
In [ ]: def sum_list(ln):
            int_sum = 0
            for value in ln:
                int_sum += value
            return int_sum
        #sum_list([10, 20, 30])
```

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

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

```
In [ ]: 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")
```

```
In [ ]: 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 des
```

```
In [ ]: 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 辞書

辞書は、**キー (key)** と **値 (value)** とを対応させるデータです。キーとしては文字列・数値・タプルなどを使うことができますが、変更可能な型であるリスト・辞書を使うことができません。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（後述）を指定できます。

例えば、文字列 apple をキーとし値として数 3 を、pen をキーとし数 5 を、対応付けた辞書は次のように作成します。

```
In [ ]: dic1 = {'apple' : 3, 'pen' : 5}
        dic1
```

```
In [ ]: type(dic1)
```

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のようと同様に、

辞書 [キー 1]

とします。

```
In [ ]: dic1 = {'apple' : 3, 'pen' : 5}
        dic1['apple']
```

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。

```
In [ ]: dic1['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。

```
In [ ]: dic1 = {'apple' : 3, 'pen' : 5}
        dic1['apple'] = 5
        dic1['orange'] = 7
        dic1
```

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、リストのように値が順序を持つわけではないので、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。

```
In [ ]: dic1[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。

```
In [ ]: dic1 = {'apple': 5, 'orange': 7, 'pen': 5}
        'apple' in dic1
```

```
In [ ]: 'orange' in dic1
```

```
In [ ]: 'banana' in dic1
```

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、の数を得ることが出来ます。

```
In [ ]: dic1 = {'apple': 5, 'orange': 7, 'pen': 5}
        len(dic1)
```

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。

```
del 辞書 [削除したいキー]
```

```
In [ ]: dic1 = {'apple' : 3, 'pen' : 5, 'orange':7}
        del dic1['pen']
        dic1
```

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。

```
In [ ]: dic1 = {}
        dic1
```

2.3.1 辞書のメソッド

辞書にも様々なメソッドがあります。

2.3.1.1 pop

指定したキーおよびそれに対応する値を辞書から削除し、削除されるキーに対応付けられた値を返します。

```
In [ ]: dic1 = {'apple' : 3, 'pen' : 5, 'orange':7}
        print(dic1.pop('pen'))
        print(dic1)
```

2.3.1.2 clear

全てのキー、値を辞書から削除します。

```
In [ ]: dic1 = {'apple' : 3, 'pen' : 5, 'orange':7}
        dic1.clear()
        dic1
```

2.3.1.3 get

引数として指定したキーが辞書に含まれてる場合にはその値を取得し、指定したキーが辞書に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、エラーを回避して辞書に登録されている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インデックス、[``]、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。

```
In [ ]: dic1 = {'apple' : 3, 'pen' : 5}
        print("キー apple に対応する値 = ", dic1.get("apple"))
        print("キー orange に対応する値 = ", dic1.get("orange"))
        print("キー orange に対応する値（エラー） = ", dic1["orange"])
```

また、get に 2 番目の引数を与えると、その値を「指定したキーが辞書に含まれていない場合」に返る値とすることが出来ます。

```
In [ ]: dic1 = {'apple' : 3, 'pen' : 5}
        print("キー apple に対応する値 = ", dic1.get("apple", -1))
        print("キー orange に対応する値 = ", dic1.get("orange", -1))
```

2.3.1.4 setdefault

1 番目の引数として指定したキー（key）が辞書に含まれてる場合にはその値を取得します。key が辞書に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、key に対応する値として辞書に登録します。

```
In [ ]: dic1 = {'apple' : 3, 'pen' : 5}
        print("キー apple に対応する値 = ", dic1.setdefault("apple", 7))
        print("setdefault(\"apple\", 7) を実行後の辞書 = ", dic1)
        print("キー orange に対応する値 = ", dic1.setdefault("orange", 7))
        print("setdefault(\"orange\", 7) を実行後の辞書 = ", dic1)
```

2.3.1.5 keys

辞書に登録されている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 ループなどを使って活用できます。

```
In [ ]: dic1 = {'apple' : 3, 'pen' : 5, 'orange':7}
        list(dic1.keys())
```

2.3.1.6 values

辞書に登録されている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、for ループなどを使って活用できます。

```
In [ ]: list(dic1.values())
```

2.3.1.7 items

辞書に登録されている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。これは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。

```
In [ ]: list(dic1.items())
```

2.3.2 ▲ keys, values, items の返り値

keys, values, items 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、dic1 の keys, values, items メソッドの返り値を変数 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、dic1 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。

```
In [ ]: dic1 = {'apple' : 3, 'pen' : 5, 'orange':7}
        ks = dic1.keys()
        vs = dic1.values()
        itms = dic1.items()
        print(list(ks))
        print(list(vs))
        print(list(itms))
        dic1['kiwi']=9
        print(list(ks))
        print(list(vs))
        print(list(itms))
```

2.3.2.1 **copy**

辞書の複製を行います。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。

```
In [ ]: dic1 = {'apple' : 3, 'pen' : 5, 'orange':7}
        dic2 = dic1.copy()
        dic2['banana'] = 9
        print(dic2)
        print(dic1)
```

2.3.3 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。

```
In [ ]: dic1 = {"one": [1, 2, 3], "ten":[10, 20, 40], "hundred":[100, 101, 120, 140]}
        print(dic1["ten"])
        print(dic1["ten"][1])
```

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。

```
In [ ]: dic1 = {'apple' : 3, 'pen' : 5, 'orange':7}
        dic2 = {'cat' : 3, 'dog' : 3, 'elephant':8}
        ld = [dic1, dic2]
        print(ld[0]["pen"])
```

2.3.4 for 文による繰り返しと辞書

辞書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。 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 key in dic1.keys():
    実行文
```

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、キーを一つづつ取り出し、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。

```
In [ ]: dic1 = {'apple' : 3, 'pen' : 5, 'orange':7}
        for key in dic1.keys():
            print('Key:', key, 'Value:',dic1[key])
```

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。

```
In [ ]: for value in {'apple' : 3, 'pen' : 5, 'orange':7}.keys():
            print('Value:',value)
```

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。

```
In [ ]: for key, value in {'apple' : 3, 'pen' : 5, 'orange':7}.items():
            print('Key:', key, 'Value:',value)
```

2.3.5 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。

```
In [ ]: 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。

```
In [ ]: print(reverse_lookup(["apple", "pen", "orange"]) == {'apple': 0, 'orange': 2, 'pen': 1})
```

2.3.6 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。

```
In [ ]: 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。

```
In [ ]: print(handle_collision({3: ['ham', 'egg'], 6: ['coffee', 'brandy'], 9: ['port wine'], 15: [
```

2.3.7 練習の解答

```
In [ ]: 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"])
```

```
In [ ]: def handle_collision(dic1, str1):
            if dic1.get(len(str1)) is None:# == None でも良い
                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: ['curr
```

2.4 ▲セット

セットは、要素となるデータを集めて作られるデータであり、セットの中の要素は並んでいるわけではありません。そのため、同じデータは高々 1 つしか同じセットに属することは出来ません。

セットを作成するには、次のように波括弧で値をくくります。

```
In [ ]: set1 = {2, 1, 2, 3, 2, 3, 1, 3, 3, 1}
        set1
```

```
In [ ]: type(set1)
```

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。

```
In [ ]: set([2, 1, 2, 3, 2, 3, 1, 3, 3, 1])
```

空集合を作成する場合、次のようにします。({} では空の辞書が作成されます。)

```
In [ ]: set2 = set() # 空集合
        set2
```

```
In [ ]: set2 = {} # 空の辞書
        set2
```

set を用いて、文字列、リストやタプルなど（iterative オブジェクトと呼ばれています）からセットを作成することができます。

```
In [ ]: set([1,1,2,2,2,3])
```

```
In [ ]: set((1,1,2,2,2,3))
```

```
In [ ]: set('aabdceabdae')
```

```
In [ ]: set({'apple' : 3, 'pen' : 5})
```

2.4.1 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。

```
In [ ]: len(set1) # 集合を構成する要素数
```

```
In [ ]: x,y,z = set1 # 多重代入
        x
```

```
In [ ]: 2 in set1 # 指定した要素を集合が含むかどうかの判定
```

```
In [ ]: 10 in set1 # 指定した要素を集合が含むかどうかの判定
```

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。

```
In [ ]: set1[0]
```

2.4.2 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。

```
In [ ]: set1 = {1, 2, 3, 4}
        set2 = {3, 4, 5, 6}
```

```
In [ ]: set1 | set2 # 和集合
```

```
In [ ]: set1 & set2 # 積集合
```

```
In [ ]: set1 - set2 # 差集合
```

```
In [ ]: set1 ^ set2 # 対称差
```

2.4.3 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。

```
In [ ]: print({1, 2, 3} == {1, 2, 3})
        print({1, 2} == {1, 2, 3})
```

```
In [ ]: print({1, 2, 3} != {1, 2, 3})
        print({1, 2} != {1, 2, 3})
```

```
In [ ]: print({1, 2, 3} <= {1, 2, 3})
        print({1, 2, 3} < {1, 2, 3})
        print({1, 2} < {1, 2, 3})
```

2.4.4 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。

2.4.4.1 add

指定した要素を新たにセットに追加します。

```
In [ ]: set1 = {1, 2, 3}
        set1.add(4)
        set1
```

2.4.4.2 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。

```
In [ ]: set1.remove(1)
        set1
```

```
In [ ]: set1.remove(10)
```

2.4.4.3 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。

```
In [ ]: set1 = {1, 2, 3, 4}
        set1.discard(1)
        set1
```

```
In [ ]: set1.discard(5)
```

2.4.4.4 clear

全ての要素を削除して対象のセットを空にします。

```
In [ ]: set1 = {1, 2, 3, 4}
        set1.clear()
        set1
```

2.4.4.5 pop

セットからランダムに 1 つの要素を取り出します。

```
In [ ]: set1 = {1, 2, 3, 4}
        print(set1.pop())
        print(set1)
```

2.4.4.6 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。

```
In [ ]: set1 = {1, 2, 3, 4}
        set2 = {3, 4, 5, 6}
        set1.union(set2) # 和集合
```

```
In [ ]: set1.intersection(set2) # 積集合
```

```
In [ ]: set1.difference(set2) # 差集合
```

```
In [ ]: set1.symmetric_difference(set2) # 対称差
```

2.4.5 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。

```
In [ ]: 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。

```
In [ ]: print(check_characters("Onde a terra acaba e o mar começa") == 13)
```

2.4.6 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。

```
In [ ]: 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。

```
In [ ]: print(count_words2("From Stettin in the Baltic to Trieste in the Adriatic an iron curtain h
```

2.4.7 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。

```
In [ ]: 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。

```
In [ ]: print(check_dicsize({'apple': 0, 'orange': 2, 'pen': 1}) == 3)
```

2.4.8 練習の解答

```
In [ ]: def check_characters(str1):
            set1 = set(str1)
            return len(set1)
        #check_characters("Onde a terra acaba e o mar começa")
```

```
In [ ]: 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 des
```

```
In [ ]: def check_dicsize(dic1):
            return len(set(dic1))
        #check_dicsize({'apple': 0, 'orange': 2, 'pen': 1})
```

2.5 ▲簡単なデータの可視化

これ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)

2.5.1 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 Juputer 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。

```
In [ ]: %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
```

2.5.2 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。

```
In [ ]: 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。

```
In [ ]: 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()
```

2.5.3 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。

```
In [ ]: import matplotlib.pyplot as plt
        x = [5, 10, 15, 10]
        y = [10, 5, 10, 15]
        plt.scatter(x,y)
```

2.5.4 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。

```
In [ ]: import matplotlib.pyplot as plt
        x = [1, 2, 3, 4]
        y = [10, 30, 40, 15]
        plt.bar(x,y)
```

第 2 回は文字列、辞書についも学びました。 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。

```
In [ ]: 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()))
```

第 3 回

3.1 条件分岐

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。

3.1.1 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もちます。
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:

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

        print("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。

3.1.2 `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:
    ここのグループは「式」が真のときにのみ実行される
else:
    ここのグループは「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:
    ここのグループは「式」が真のときにのみ実行される
こ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
    ここのグループは「式 1」が真のときにのみ実行される
ここのグループは常に実行される
```

```
if 式 1:
    ここのグループは「式 1」が真のときにのみ実行される
ここのグループは常に実行される
if 式 2:
    ここのグループは「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）
ここのグループは常に実行される
if 式 3:
    ここのグループは「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）
ここのグループは常に実行される
```

3.1.3 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。

3.1.4 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。

```
In [ ]: def exception3(x,y,z):
            ...
```

次のセルで動作を確認してください。

```
In [ ]: print(exception3(1,2,2))
        print(exception3(4,2,4))
        print(exception3(9,3,9))
```

3.1.5 練習

関数 exception9(a) の引数は以下の条件を満たすとします。

- 引数 a には、長さが 9 のリストが渡されます。
- このリストの要素は整数ですが、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。

```
In [ ]: def exception9(a):
            ...
```

次のセルで動作を確認してください。

```
In [ ]: 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]))
```

3.1.6 ▲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あます。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。

```
In [ ]: ### 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.")
```

3.1.7 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。）

```
In [ ]: 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)
```

3.1.8 練習

以下のプログラムはプログラマの意図どおりに動作しません。 print の出力内容から意図を判断して条件分岐を書き換えてください。

```
In [ ]: 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")
```

3.1.9 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。

```
In [ ]: 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")
```

```
In [ ]: 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")
```

3.1.10 ▲ 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"
```

3.1.11 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。

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

```
In [ ]: def exception9(a):
            x = a[0] + a[1] + a[2]
            y = a[3] + a[4] + a[5]
            z = a[6] + a[7] + a[8]
            if x==y:
                return exception3(a[6], a[7], a[8])
            elif x==z:
                return exception3(a[3], a[4], a[5])
            else:
                return exception3(a[0], a[1], a[2])
```

3.1.12 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。

```
In [ ]: 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")
```

```
In [ ]:
```

3.2 繰り返し

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。

3.2.1 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 value in seqence:
    実行文
```

`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。

```
In [ ]: words = ["dog", "cat", "mouse"]
        for w in words:
            print(w, len(w))
```

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。

```
In [ ]: word = "supercalifragilisticexpialidocious"
        for c in word:
            print(c)
```

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。

```
In [ ]: print(ord('a'))
        print(ord('b'))
        print(ord('z'))

        print(chr(97))
```

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。

```
In [ ]: height = [0] * 26
        for c in word:
            height[ord(c) - ord('a')] += 1

        print(height)
```

height を視覚化してみましょう。詳しくは、5-3 を参照してください。

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

        plt.plot(height)
```

```
In [ ]: 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)
```

3.2.2 辞書に対する繰り返し処理

次に辞書を用いた例です。辞書では、辞書に登録されたキーが順に取り出されます。

```
In [ ]: dic1 = {'apple':3, 'pen':5, 'orange':7}
        for key in dic1:
            print(key)
```

for 文によって辞書の要素全てに同じ処理をおこないたい場合、辞書に関するメソッド keys、values、items などが利用できます。 dict_a という名前の辞書に対して： 1. キーを取り出したい場合は、dict_a.keys() 2. 値を取り出したい場合は、dict_a.values() 3. キー、値のペアを取り出したい場合は dict_a.items()

のように使います。辞書 dict_a をそのまま書くのは dict_a.keys() と書くのと等価です。

以下に例を示します。

```
In [ ]: dict_a = {
            "key1": "val1",
            "key2": "val2",
            "key3": "val3",
            "key4": "val4"
        }
```

```
    for key in dict_a.keys(): # dict_a.keys() の代わりに、dict_a としても同じ結果が得られます
        print(key)

    print()

    for val in dict_a.values():
        print(val)

    print()

    for key, val in dict_a.items():
        print(key, ":", val)
```

3.2.3 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。

```
In [ ]: 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。

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

3.2.4 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。

```
for value in range(j):
    実行文
```

---

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。

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。

```
In [ ]: for value in range(5):
            print("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。

```
In [ ]: for value in range(5):
            print(value)
```

すなわち、valur 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。

```
In [ ]: ln = ['e', 'd', 'a', 'c', 'f', 'b']
        for value in range(len(ln)):
            print(ln[value])
```

range() 関数は:

1. 引数を一つ与えると 0 から 引数までの整数列を返します。このとき引数の値は含まれないことの注意してください。
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 ～ 9 までの整数列の総和を計算、印字するプログラムです:

```
In [ ]: s = 0
        for i in range(10):
            s = s + i

        print(s)
```

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。

```
In [ ]: s = 0
        for i in range(1,10,2):
            s = s + i

        print(s)
```

3.2.5 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。

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

3.2.6 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下さい。

```
In [ ]: 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])
```

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。

```
In [ ]: 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]
```

C[100] を視覚化してみましょう。

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

3.2.7 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。

```
In [ ]: 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。

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

3.2.8 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。

```
In [ ]: 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。

```
In [ ]: 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,
```

3.2.9 練習

引数で与えられる 2 つの整数 x, y 間（x, y を含む）の整数の総和を返す関数 sum_n を for 文を利用して作成してください。 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。

```
In [ ]: 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。

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

3.2.10 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:

```
i = 0
for val in some_list:
    print(i, val)
    # 繰り返させたい処理
    i += 1
```

Python では enumerate() 関数が用意されており、上のプログラムは以下のように書き換えることができます。

```
for i, val in enumerate(some_list):
    # 繰り返させたい処理
```

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:

```
In [ ]: 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。
```

3.2.11 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
```

3.2.12 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。

```
In [ ]: x = 1
        total = 0
        while x <= 10:
            total += x
            x += 1

        print(x, total)
```

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。

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

        print(x, total)
```

3.2.13 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、その後の処理はおこなわれません。

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。

```
In [ ]: def simple_lsearch(lst, myitem):
```

```
    for item in lst:
        if item == myitem:
            return True
    return False
```

3.2.14 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。

```
In [ ]: 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)
```

3.2.15 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。

```
In [ ]: 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。

```
In [ ]: print(simple_match("location", "cat") == 2)
        print(simple_match("soccer", "cat") == -1)
        print(simple_match("category", "cat") == 0)
        print(simple_match("carpet", "cat") == -1)
```

3.2.16 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。

```
colors = ["red", "green", "blue", "black", "white"]
```

```
for c in colors:
    if(c == "black"):
            continue
    print(c)
```

3.2.17 ▲ 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。

```
colors = ["red", "green", "blue", "black", "white"]
for c in colors:
    if(c == "black"):
        continue
    print(c)
else:
    print("")
```

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。

3.2.18 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。

```
x = -1
if x < 0:
    print("'x' is positive")
elif x == 0:
    # IndentationError: expected an indented block
elif 0 < x < 5:
    print("x is positive and smaller than 5")
else:
    print("x is positive and larger than or equal to 5")
```

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。

```
x = -1
if x < 0:
    print("'x' is positive")
```

```
elif x == 0:
    # no error
    pass
elif 0 < x < 5:
    print("x is positive and smaller than 5")
else:
    print("x is positive and larger than or equal to 5")
```

3.2.19 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。

```
In [ ]: from time import sleep

        while True:
            print ("Yeah!")
            sleep(1)
```

3.2.20 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。

```
In [ ]: 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。

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

3.2.21 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。

```
In [ ]: 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。

```
In [ ]: print(collect_engwords("Unfortunately no, it requires something with a little more kick, pl
        'something', 'with', 'little', 'more', 'kick', 'plutonium'])
```

3.2.22 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。

```
In [ ]: 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。

```
In [ ]: print(swap_lists([1, 2, 3, 4, 5], ["a", "b", "c", "d", "e"]) == ([1, 'b', 3, 'd', 5], ['a',
```

3.2.23 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。

```
In [ ]: 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。

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

3.2.24 練習

文字列 str1 を引数として取り、 str1 の中に含まれる大文字の数を返す関数 count_capitalletters を作成して下さい。

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。

```
In [ ]: 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。

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

3.2.25 練習

長さが 3 の倍数である文字列 str_augc が引数として与えられたとき、 str_augc を長さ 3 の文字列に区切り、それらを順に格納したリストを返す関数 identify_codons を作成して下さい。

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。

```
In [ ]: 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。

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

3.2.26 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。

```
In [ ]: 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。

```
In [ ]: print(add_commas(14980) == "14,980")
        print(add_commas(3980) == "3,980")
        print(add_commas(298) == "298")
        print(add_commas(1000000) == "1,000,000")
```

3.2.27 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。

```
In [ ]: 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。

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

3.2.28 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。

```
In [ ]: 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。

```
In [ ]: print(handle_collision2({6: 'Styles', 4: 'Link', 7: 'Ackroyd'}, "Big Four") == {6: 'Styles'
        print(handle_collision2({6: 'Styles', 4: 'Link', 7: 'Ackroyd', 8: 'Big Four', 10: 'Blue Tra
        print(handle_collision2({6: 'Styles', 4: 'Link', 7: 'Ackroyd', 8: 'Big Four', 10: 'Blue Tra
```

3.2.29 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。

```
In [ ]: 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。

```
In [ ]: print(handle_collision3([[3, "Richard III"], [1, "Othello"], [2, "Tempest"], [3, "King John
```

3.2.30 練習の解答

```
In [ ]: 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})
```

```
In [ ]: 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
```

```
In [ ]: 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
            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]])
```

In [ ]:
```
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
            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)
```

In [ ]:
```
def remove_evenelement(ln):
    ln2 = []
    for j in range(len(ln)):
        if ln[j] % 2 == 1:
            ln2.append(ln[j])
    return ln2
#remove_evenelement([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
```

In [ ]:
```
def collect_engwords(str_engsentences):
    list_punctuation = [".", ",", ":", ";", "!", "?"]
    for j in range(len(list_punctuation)):#list_punctuation の中の文字列（この場合、句読点）を空
        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, plutoni
```

In [ ]:
```
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"])
```

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

In [ ]:
```
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
            int_count += 1
    return int_count
count_capitalletters('Que Ser^^c3^^a1, Ser^^c3^^a1')
```

In [ ]:
```
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("CCCCCGGCACCT")
```

In [ ]:
```
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
        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")
```

In [ ]:
```
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"])
```

In [ ]:
```
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", "Edgware", "Ori
#for i in range(len(ln1)):
#    dic1 = handle_collision2(dic1, ln1[i])
#    print(dic1)
```

In [ ]:
```
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"],
```

3.2.31 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。

```
In [ ]: def sum_n(x,y):
            sum = 0
            for i in range(x, y + 1):
                sum = sum + i
            return sum
        sum_n(1,3)
```

3.2.32 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。

```
In [ ]: from time import sleep

        count = 0
        while True:
            print ("Yeah!", count)
            count += 1
            if(count >= 10):
                break
            sleep(1)
```

3.3 内包表記

3.3.1 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:

```
In [ ]: squares1 = []
        for x in range(6):
            squares1.append(x**2)
        squares1
```

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。

```
In [ ]: squares2 = [x**2 for x in range(6)]
        squares2
```

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。

```
In [ ]: sum([x**2 for x in range(6)])
```

以下の内包表記は 3-2 で用いられていました。

```
In [ ]: [chr(i + ord('a')) for i in range(26)]
```

3.3.2 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。

```
In [ ]: strings = ["The", "quick", "brown"]
        [ここに内包表記を書く]
```

3.3.3 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。

```
In [ ]: str1 = "123,45,-3"
        [ここに内包表記を書く]
```

3.3.4 練習

数のリストが与えらえたとき、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。

```
In [ ]: 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。

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

3.3.5 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:

```
In [ ]: [[x*y for y in range(x+1)] for x in range(4)]
```

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。

```
In [ ]: [x*y for x in range(4) for y in range(x+1)]
```

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。

```
In [ ]: def allsubstrings(s):
            return [s[i:j] for i in range(len(s)) for j in range(i+1,len(s)+1)]

        allsubstrings("abc")
```

3.3.6 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。

```
In [ ]: 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。

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

3.3.7 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。

```
In [ ]: 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。

```
In [ ]: 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,
```

3.3.8 ▲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:

```
In [ ]: words = ["cat", "dog", "elephant", None, "giraffe"]
        length = [len(w) for w in words if w != None]
        print(length)
```

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。

3.3.9 ▲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:

```
In [ ]: words = ["cat", "dog", "elephant", "giraffe"]
        length_set = {len(w) for w in words}
        print(length_set)
```

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。

3.3.10 ▲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。

```
In [ ]: words = ["cat", "dog", "elephant", "giraffe"]
        length_dic = {w:len(w) for w in words}
        print(length_dic)
```

length_dic として{'cat': 3, 'dog': 3, 'elephant': 8, 'giraffe': 7} が得られます。
長さと文字列を逆にするとどうなるでしょうか。

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

3.3.11 ▲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。

```
In [ ]: it = (x * 3 for x in 'abc')
        for x in it:
            print(x)
```

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。

```
In [ ]: list(x ** 2 for x in range(5))
```

```
In [ ]: tuple(x ** 2 for x in range(5))
```

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。

```
In [ ]: sum(x ** 2 for x in range(5))
```

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、6-3 にて改めて説明します。

3.3.12 練習の解答

```
In [ ]: strings = ["The", "quick", "brown"]
        [len(x) for x in strings]
```

```
In [ ]: str1 = "123,45,-3"
        [int(x) for x in str1.split(",")]
```

```
In [ ]: def var(lst):
            n = len(lst)
            av = sum(lst)/n
            return sum([(x - av)*(x - av) for x in lst])/n
```

```
In [ ]: def var(lst):
            n = len(lst)
            av = sum(lst)/n
            return sum([x*x for x in lst])/n - av*av
```

```
In [ ]: def sum_lists(list1):
            return sum([sum(lst) for lst in list1])
```

```
In [ ]: def sum_matrix(list1,list2):
            return [[list1[i][j]+list2[i][j] for j in range(3)] for i in range(3)]
```

3.4 関数

3.4.1 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。

```
In [ ]: #"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。

```
In [ ]: #関数 greeting を呼び出し
        greeting()
```

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 行名はヘッダと呼ばれ、**関数名**はその関数を呼ぶのに使う名前、**引数**はその関数へ渡す変数の一覧です。変数がない場合もあります。

関数本体はインデントした上で、処理や手続きの流れを記述します。

3.4.2 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。

```
In [ ]: #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。

```
In [ ]: #関数 greeting に文字列"Hello"を渡して呼び出し
        greeting("Hello")
```

このようにして引数に渡される値のことを、**実引数**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**と呼ばれます。

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。

3.4.3 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**（かえりち）（1-2 で説明済み）を返すことができます。

返値は、return で定義します。関数の返値がない場合は、None が返されます。return が実行されると、関数の処理はそこで終了するため、次に文があっても実行はされません。また、ループなどの繰り返し処理の途中でも return が実行されると処理は終了します。関数の処理が最後まで実行され、返値がない場合は最後に return を実行したことと同じになります。

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

```
In [ ]: #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")
```

```
In [ ]: #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])
```

関数の返値を変数に代入することもできます。

```
In [ ]: #関数 greeting の返値を変数 greet に代入
        greet = greeting("Hello")
        greet
```

3.4.4 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。

```
In [ ]: #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")
```

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。

```
In [ ]: #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')
```

3.4.5 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。

```
In [ ]: #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
```

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。

```
In [ ]: #グローバル変数 greeting_global の定義
        greeting_global = "Hello"
```

```
#グローバル変数 greeting_global の値を表示する関数 greeting
def greeting():
    print (greeting_global)

greeting()

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

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に代入すると、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として扱われます。

```
In [ ]: #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
```

関数内でグローバル変数に明示的に代入するには global を使って、代入したいグローバル変数を指定します。

```
In [ ]: #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
```

3.4.6 ▲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。

```
In [ ]: #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)
```

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。

```
In [ ]: #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)
```

3.4.7 ▲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。

```
In [ ]: #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')
```

3.4.8 ▲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。

```
In [ ]: #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")
```

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。

```
In [ ]: #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)
```

3.4.9 ▲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。

```
In [ ]: #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")
```

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。

```
In [ ]: #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)
```

3.4.10 ▲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）
```

```
In [ ]: #位置引数、初期値を持つ引数、可変長引数、辞書型の可変長引数
        #それぞれを引数として受け取り、それらを表示する関数 greeting
        def greeting(greet, en='Hello', *args, **kwargs):
            print (greet)
            print (en)
            print (args)
            print (kwargs)

        #可変長引数へ渡すリスト
        greeting_list = ['Bonjour']

        #辞書型の可変長引数へ渡す辞書
        greeting_dict = {'de':"Guten Tag"}

        #関数 greeting に引数を渡して呼び出し
        greeting('Hi','Hello',*greeting_list,**greeting_dict)
```

3.4.11 ▲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。

```
In [ ]: #グローバル変数 greeting_global の定義と参照
        greeting_global = "Hello"
        type(greeting_global)
```

```
In [ ]: #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)
```

3.5 ▲再帰

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（直接的・間接的に）呼び出すことです。関数が自分自身を呼び出すことを再帰呼び出しといいます。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。

3.5.1 再帰関数の例：接頭辞リストと接尾辞リスト

```
In [ ]: # 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')
```

```
In [ ]: # 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:
                return [s] + suffixes(s[1:])

        suffixes('aabcc')
```

3.5.2 再帰関数の例：べき乗の計算

```
In [ ]: # 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)
```

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。

In [ ]:
```
# 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)
```

単純な処理においては、繰り返しの方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ってくると、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。

In [ ]:
```
# 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)
```

3.5.3 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。

In [ ]:
```
# マージソートを行い、比較回数 n を返す
def merge_sort_rec(data, l, r, work):
    n = 0
    if r - l <= 1:
        return n
    m = l + (r - l) // 2
    n1 = merge_sort_rec(data, l, m, work)
    n2 = merge_sort_rec(data, m, r, work)
    i1 = l
```

```
            i2 = m
            for i in range(l, r):
                from1 = False
                if i2 >= r:
                    from1 = True
                elif i1 < m:
                    n = n + 1
                    if data[i1] <= data[i2]:
                        from1 = True
                if from1:
                    work[i] = data[i1]
                    i1 = i1 + 1
                else:
                    work[i] = data[i2]
                    i2 = i2 + 1
            for i in range(l, r):
                data[i] = work[i]
            return n1 + n2 + n

        def merge_sort(data):
            return merge_sort_rec(data, 0, len(data), [0]*len(data))
```

merge_sort は、与え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。

In [ ]:
```
import random
a = [random.randint(1,10000) for i in range(100)]
merge_sort(a)
```

In [ ]:
```
a
```

In [ ]:

第 4 回

4.1 ファイル入出力の基本

4.1.1 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。

```
In [ ]: f = open('small.csv', '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。

'small.csv' 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、small.csv 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、big.csv 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にあるならば、'../big.csv' 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/big.csv' となります。

'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。

4.1.2 オブジェクト

Python プログラムでは、全ての種類のデータは、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。

オブジェクトの型は type という関数によって求めることができます。

たとえば、3 というデータ（オブジェクト）の型は int です。

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

Python において、変数には、オブジェクトの参照値が入ります。 では、変数 f に入っているオブジェクトの型はどうなっているでしょうか。

```
In [ ]: type(f)
```

_io.TextIOWrapper は、io (in/out の略で、様々な入出力を扱うモジュール）の中の、Text の IO（In/Out) を扱うラッパー型です。

f のオブジェクトそのものを表示させると以下のようになります。

```
In [ ]: f
```

_io.TextIOWrapper 型であるこのオブジェクトは、name (ファイル名）属性が small.csv、mode (モード) 属性が r であることを意味しています。encoding(文字コード）はこのプログラムを動かしている OS によって違うでしょう。もし Windows なら cp932（Shift-JIS のこと)、Mac なら euc_jp となっているのが一般的です。

4.1.2.1 属性

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、以下のようにして確認できます。

オブジェクト . 属性名

たとえば、以下のように f に入っているオブジェクトに対して色々な情報を問い合わせることができます。

```
In [ ]: f.name
```

```
In [ ]: f.mode
```

オブジェクトがどのような属性を持つかは、dir という関数を使って調べることができます。

```
In [ ]: dir(f)
```

dir の結果は文字列の配列です。それぞれの文字列は属性の名前です。 この中に、name や mode も含まれています。

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できるものがあり、**メソッド**と呼ばれます。

たとえば、read という属性の値を () を付けないで表示させると以下のようです。

```
In [ ]: f.read
```

この関数が、() を付けることによって呼び出されます。このとき、read() は、そのファイルを全て読み込むという働きをします。

```
In [ ]: f.read()
```

ファイル全体の内容が一続きの文字列として返されました。文字列が複数行にわたる場合は、それを一続きの文字列として表すために、改行する場所が\n という記号（改行文字）で置き換えられます。（同様に、ファイルに書き込みする際に、文字列中に改行を加えたい場合は、そこに\n を挿入します。）

これでファイルの読み込みが終わりましたので、close() というメソッドによってファイルを**クローズ**して（閉じて）おきましょう。

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

繰り返しますが、属性のうち、そのオブジェクトを操作するための関数として呼び出すことのできるものをメソッドと呼びます。メソッドは、オブジェクト指向言語で一般的に使われる用語です。

メソッドは、以下のようにして呼び出すことができます。

オブジェクト.属性名(式, ...)

この構文により、属性の値である関数が呼び出されます。その実行は、当然ながら、属性を持つオブジェクトに依存したものになります。

4.1.3 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。

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

In [ ]: print(number_of_characters('small.csv') == 45)
```

4.1.4 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よね。iterator 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。

```
In [ ]: f = open('small.csv', 'r')
        print(next(f))
        print(next(f))
        f.close()
```

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。

```
for line in f:
    ...
```

繰り返しの各ステップで、next(f) が呼び出されて、変数 line にその値が設定され、for 文の中身が実行されます。

以下の例を見てください。

```
In [ ]: f = open('small.csv', 'r')
        for line in f:
            print(line)
        f.close()
```

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。）

```
In [ ]: f = open('small.csv', 'r')
        print(' 最初')
        for line in f:
            print(line)
        print(' もう一度')
        for line in f:
            print(line)
        f.close()
```

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。

4.1.5 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 lastline(name) を定義してください。

```
In [ ]: def lastline(name):
            ...
```

以下のセルによってテストしてください。

```
In [ ]: print(lastline("small.csv") == '31,32,33,34,35\n')
```

4.1.6 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。

```
In [ ]: f = open('small.csv', 'r')
```

```
In [ ]: f.readline()
```

```
In [ ]: f.readline()
```

```
In [ ]: f.readline()
```

```
In [ ]: f.readline()
```

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

4.1.7 ファイルに対する with 文

ファイルのオブジェクトは、with 文に指定することができます。

```
with ファイル as 変数:
    ...
```

---

with の次には、open によってファイルをオープンする式を書きます。
また、as の次には、ファイルのオブジェクトが格納される変数を書きます。
with 文は処理後にファイルのクローズを自動的にやってくれますので、ファイルに対して close() を呼び出す必要がありません。

```
In [ ]: with open('small.csv', 'r') as f:
            for line in f:
                print(line)
```

4.1.8 ファイルへの書き込み

ファイルへの書き込みは以下のように write というメソッドを用いて行います。

```
In [ ]: with open("write-test.txt", "w") as f:
            f.write("hello\nworld\n")
```

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。

```
In [ ]: with open("write-test.txt", "w") as f:
            f.writelines(["hello\n", "world\n"])
```

write には文字列を指定します。writelines には文字列のリストを指定します。
どちらも、改行するためには、文字列の中に\n を入れる必要があります。

4.1.9 練習

二つのファイル名 infile, outfile を引数として、infile の英文字をすべて大文字にした結果を outfile に書き込む fileupper(infile,outfile) という関数を作成してください。

```
In [ ]: def fileupper(infile,outfile):
            ...
```

以下のセルによってテストしてください。

```
In [ ]: with open("write-test.txt", "w") as f:
            f.writelines(["hello\n", "world\n"])
        fileupper("write-test.txt", "write-test-upper.txt")
```

```
        with open("write-test-upper.txt", "r") as f:
            print(f.read() == "HELLO\nWORLD\n")
```

```
In [ ]:
```

```
In [ ]:
```

```
In [ ]:
```

4.1.10 練習の解答

```
In [ ]: def  number_of_characters(name):
            f = open(name, "r")
            s = f.read()
            f.close()
            return len(s)
```

```
In [ ]: def lastline(name):
            f = open(name, "r")
            for line in f:
                pass
            f.close()
            return line
```

```
In [ ]: def fileupper(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 を書くことができます。

```
In [ ]: def fileupper(infile,outfile):
            with open(infile, "r") as f, open(outfile, "w") as g:
                g.write(f.read().upper())
```

4.2 csv ファイルの入出力

4.2.1 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

ID	氏名	ニックネーム	出身地
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"

4.2.2 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

```
In [ ]: import csv
        f = open('small.csv', 'r')
        dataReader = csv.reader(f)
```

```
In [ ]: type(dataReader)
```

```
In [ ]: dir(dataReader)
```

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。

```
In [ ]: next(dataReader)
```

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。

```
In [ ]: next(dataReader)
```

```
In [ ]: row = next(dataReader)
```

```
In [ ]: row
```

```
In [ ]: row[2]
```

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。

```
In [ ]: int(row[2])
```

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、下のようにエラーが返ってきます。

```
In [ ]: next(dataReader)
```

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。

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

4.2.3 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。

```
for row in dataReader:
    ...
```

繰り返しの各ステップで、next(dataReader) が呼び出されて、row にその値が設定され、for 文の中身が実行されます。

```
In [ ]: f = open('small.csv', 'r')
        dataReader = csv.reader(f)
        for row in dataReader:
            print(row)
        f.close()
```

4.2.4 csv ファイルに対する with 文

以下は with 文を使った例です。

```
In [ ]: with open('small.csv', 'r') as f:
            dataReader = csv.reader(f)
            for row in dataReader:
                print(row)
```

4.2.5 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。

```
In [ ]: f = open('out.csv', 'w')

In [ ]: dataWriter = csv.writer(f)

In [ ]: dir(dataWriter)

In [ ]: dataWriter.writerow([1,2,3])

In [ ]: dataWriter.writerow([21,22,23])
```

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。

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

読み込みのときと同様、with 文を使うこともできます。

```
In [ ]: with open('out.csv', 'w') as f:
            dataWriter = csv.writer(f)
            dataWriter.writerow([1,2,3])
            dataWriter.writerow([21,22,23])
```

4.2.5.1 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。

```
In [ ]: 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。

```
In [ ]: years
```

```
In [ ]: july_temps
```

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。

```
In [ ]: 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))
```

```
In [ ]: plt.plot(years, july_temps, '.', xp, fitp(xp), '-')
        plt.show()
```

4.2.6 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。

```
In [ ]:
```

以下のセルによってテストしてください。(years と july_temps の値がそのままと仮定しています。)

```
In [ ]: with open('tokyo-july-temps.csv', 'r') 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
```

4.2.7 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。

```
In [ ]: def csv_matrix(name):
            ...
```

以下のセルによってテストしてください。

```
In [ ]: print(csv_matrix("small.csv") == [[11, 12, 13, 14, 15], [21, 22, 23, 24, 25], [31, 32, 33,
```

In [ ]:

In [ ]:

In [ ]:

4.2.8 練習の解答

```
In [ ]: with open('tokyo-july-temps.csv', 'w') as f:
            i = 0
            dataWriter = csv.writer(f)
            for i in range(len(years)):
                dataWriter.writerow([years[i],july_temps[i]])
```

```
In [ ]: 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
```

4.3 json ファイルの入出力

4.3.1 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 形式で表すと以下のようになります。 "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":"鈴木花子", "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、{}で囲んだものはオブジェクト、[] で囲んだものはリストで、オブジェクトの中にオブジェクト、リストの中にオブジェクト、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`

値の型	json の例
オブジェクト	`"data":{"a":"b"}`
配列	`"data":[1,2,3]`

csv 形式では、上記のニックネームの列のように、同じセルに複数の要素を含むデータは基本的には扱えませんが、json では扱うことができます。また「ID」「氏名」「ニックネーム」「出身地」のようなラベルは、csv 形式では 1 行目のデータとして書きこむことはできますが、『1 行目はラベルの行』とプログラム側で区別しなければ、ただの一行のデータとして扱われてしまいます。これに対し json では、「氏名」が「鈴木一郎」というように、各値に対してキーを設定することができます。

4.3.2 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.dump` と `json.load` を用います。

```
In [ ]: 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") as f:
            json.dump(d, f)

        # json ファイルを読み込み
        with open("test.json", "r") as f:
            d1 = json.load(f)
```

```
# json データのプリント
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))
```

4.3.3 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。

```
In [ ]: 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)]
```

```
In [ ]:
```

以下のセルによってテストしてください。

```
In [ ]: with open("fib.json", "r") as f:
            print(json.load(f) == [{'n': n, 'fib' : fib(n)} for n in range(0,10)])
```

4.3.3.1 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。

```
In [ ]: with open("catalog-2018.json", "r", encoding="utf-8") as f:
            j = json.load(f)
```

```
In [ ]: j
```

```
In [ ]: len(j)
```

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。

```
In [ ]: j[0]
```

```
In [ ]: j[1]
```

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。

```
In [ ]: j[1]['name_j']
```

姓と名は、\u3000 というコードで区切られているようです。

```
In [ ]: j[1]['name_j'].split('\u3000')
```

```
In [ ]: j[1]['Title']
```

title をキーに持たない授業はないようです。

```
In [ ]: for d in j:
            if d.get('title',-1)==-1:
                print(d)
```

4.3.4 ▲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。

```
In [ ]: "   This is strip.\n".strip()
```

lstrip は文頭、rstrip は文末の空白や改行を除去します。

```
In [ ]: "   This is strip.\n".lstrip()
```

```
In [ ]: "   This is strip.\n".rstrip()
```

```
In [ ]:
```

```
In [ ]:
```

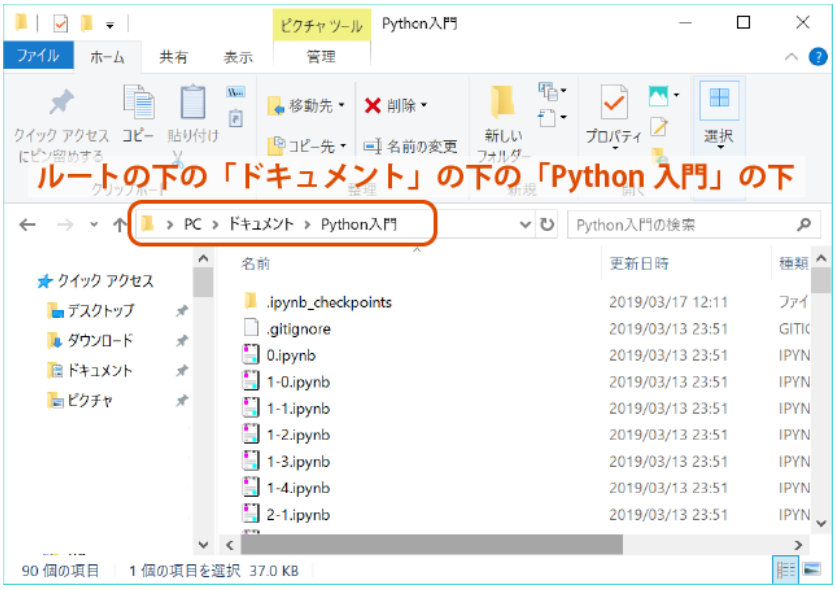
```
In [ ]:
```

4.3.5 練習の解答

```
In [ ]: with open("fib.json", "w") as f:
            json.dump([{'n': n, 'fib' : fib(n)} for n in range(0,10)], f)
```

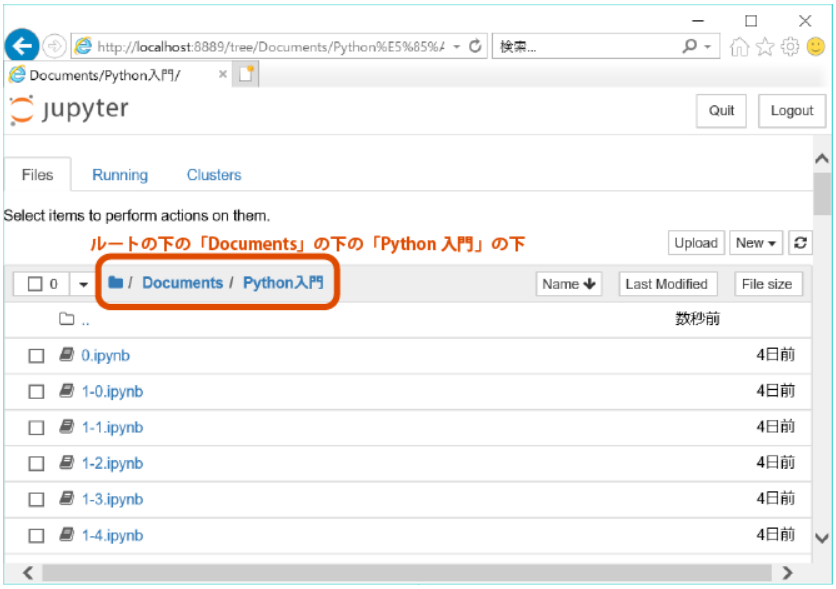
4.4 ▲木構造のデータ形式

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。



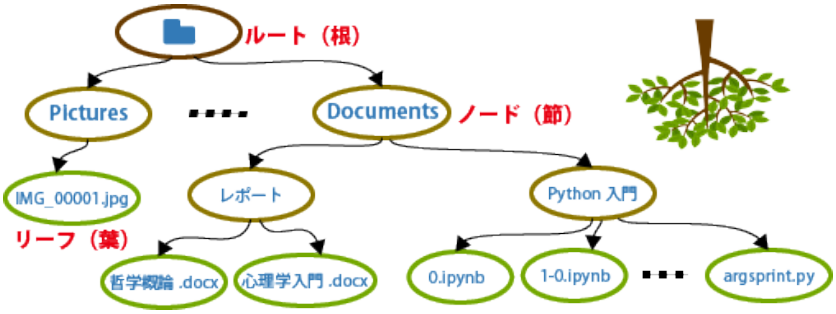
エクスプローラ

これは Jupyter Notebook では以下のように見えます。



Jupyter Notebook

このようなデータ形式は以下のようにグラフであらわすこともできます。まるで木を逆さにしたような形に見えますね。ですからこのようなデータの形式を「木構造」と呼びます。また、一番根っこにあたるデータを「ルート（根）」、先端にあたるデータを「リーフ（葉）」、その間にあるデータを「ノード（節）」と呼びます。



Jupyter Notebook tree

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。

4.4.1 カレントディレクトリ

4-1 で `small.csv` 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('small.csv', 'r')
```

このとき、この `small.csv` 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、その実行環境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。
もし `sample.csv` がその notebook と同じフォルダではなく、そこに置かれた `tmp` という名前のフォルダの中に置かれているとすると、notebook とは同じ場所に `small.csv` が置かれていないので、————————————————————— FileNotFoundError Traceback (most recent call last) <ipython-input-1-d72d4a8a8bc1> in <module> —-> 1 f = open('small.csv', 'r')

FileNotFoundError: [Errno 2] No such file or directory: 'small.csv' といったエラーが出て、ファイルのオープンに失敗するはずです。

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

```
f = open('tmp/small.csv', 'r')
```

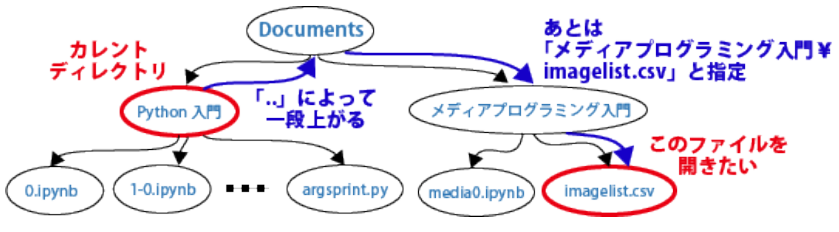
このようにすることによって、ターミナルに「カレントディレクトリの下の `tmp` の下にある `small.csv` を開いてください」と指示することができます。
これは、カレントディレクトリから「`small.csv`」までの経路（行き方）を表したものなので**パス**とも呼びます。

4.4.2 相対パスと絶対パス

`tmp/small.csv` という表現では、カレントディレクトリから「`small.csv`」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、どの場所の 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、`C:\Users\hagiya\Douments\Python 入門\small.csv` のように、ルートからパスを記した場合、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、そこまでに通るディレクトリ名をパスに書けばいいですが、その下に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\Douments\Python 入門`のとき、`C:\Users\hagiya\Douments\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？



Jupyter Notebook tree

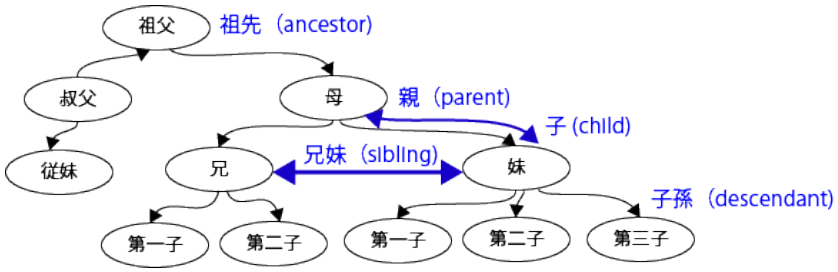
実は、一つ上のディレクトリを..（コンマ２つ）で表現することができます。上の例だと、

```
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\Douments\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の imagelist.csv」と辿っているわけです。

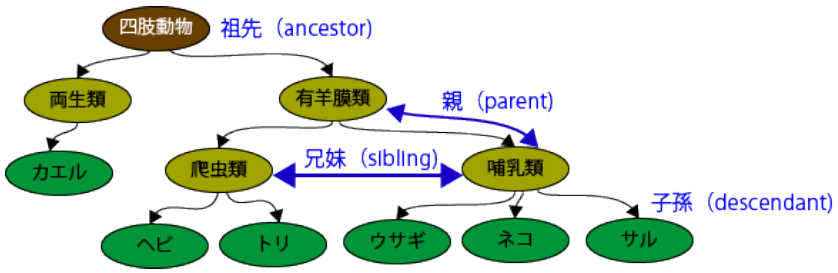
4.4.3 木構造によるデータ表現

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” ですよね。



Family tree

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（desendant)」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。



tree

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。

4.4.4 テキストによる木構造のデータ表現

上のイラストのように毎回絵を描いて木構造を表せばわかりやすいですが面倒ですよね。テキストでは以下のように表現することがあります。今度は一番左にあるのがルートで、右に行くにしたがって分岐しています。四肢動物

```
┣両生類━カエル
┃
┃
┗哺乳類┳ウサギ
       ┃
       ┃
       ┣ネコ
       ┃
       ┃
       ┗サル
┏爬虫類┳ヘビ
┃
┃
┗トリ
┗有羊膜類┫
```

4.4.5 木構造の json 表現

json 形式のメリットの一つは、木構造のような入れ子（何かの中に何かが入っているという構造）を表現できることです。上の例を json で表すと以下のようになります。 "四肢動物": "両生類": ["カエル"], "有羊膜類": "爬虫類": ["ヘビ", "トリ"], "哺乳類": ["ウサギ", "ネコ", "サル"]

第 5 回

5.1 モジュールの使い方

5.1.1 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。

`import モジュール名`

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。

```
In [ ]: 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、$\log_2 32$）
```

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、

`モジュール名. モジュールの中の関数名（値）`

とする必要があるということです。

5.1.2 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）」の様にして、「モジュール名.」を付けずにそのままの名前で、モジュールの読み込み元のプログラムで使いたい場合には、from を以下の様に書くことで利用することができます。

`from モジュール名 import モジュールの中の関数名（値）`

例えば、次の様になります。

```
In [ ]: 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、$\log_2 32$）
```

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。

```
from math import *
```

この方法ではアンダースコア _ で始まるものを除いた全ての名前が読み込まれるため、明示的に名前を指定する必要はありません。

```
In [ ]: 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
```

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。

```
In [ ]: 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）
```

5.1.3 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。

```
In [ ]: 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
```

個々の関数ごとに別の名前を付けることもできます。

```
In [ ]: 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))
```

5.1.4 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。

```
import math
print(math.sqrt(2))
print(math.sin(math.pi))
```

以下のセルを、モジュール名を付けないで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。

```
In [ ]: import ...
        ...

        print(sqrt(2))
        print(sin(pi))
```

5.1.5 パッケージ

大きなモジュールは、パッケージによって階層化されていることが多いです。パッケージはモジュールのディレクトリのようなものです。

A がパッケージの場合、A の下のモジュールを A.B という記法によって参照することができます。(階層構造はさらに深くすることもできます。)

以下の例では、matplotlib というパッケージの下にある pyplot というモジュールをインポートしています。

```
In [ ]: import matplotlib.pyplot
```

```
In [ ]: # plot するデータ
        d =[0, 1, 4, 9, 16]

        # plot 関数で描画
        matplotlib.pyplot.plot(d);
```

これは長いので matplotlib.pyplot に plt という名前を付けましょう。

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

すると、matplotlib.pyplot.plot を plt.plot として参照することができます。

```
In [ ]: # plot するデータ
        d =[0, 1, 4, 9, 16]
```

```
        # plot 関数で描画
        plt.plot(d);
```

以下のようにすれば、matplotlib.pyplot.plot を plot として参照することができます。

```
In [ ]: from matplotlib.pyplot import plot
```

```
In [ ]: # plot するデータ
        d =[0, 1, 4, 9, 16]

        # plot 関数で描画
        plot(d);
```

以下では、よく使われるモジュールについて簡単に説明しています。

5.1.6 math

math モジュールについて詳しくは以下を参照してください。
https://docs.python.jp/3/library/math.html

5.1.7 random

random は、乱数を生成する関数から成るモジュールです。詳しくは、以下を参照してください。
乱数とは、一般にある範囲の数の値の中から無作為に選ばれる数の値のことを指します。
https://docs.python.jp/3/library/random.html

```
In [ ]: import random
```

random.random は、ゼロ個の引数の関数で、0.0 以上 1.0 未満の実数を一様にランダムに選んで返します。すなわち、一様分布にしたがって乱数を生成します。

```
In [ ]: random.random()
```

```
In [ ]: random.random()
```

このように、呼び出すごとに異なる数が返ります。
random.gauss を random.gauss(mu,sigma) として呼び出すと、 平均 mu、標準偏差 sigma の正規分布（ガウス分布）にしたがって乱数を生成して返します。

```
In [ ]: random.gauss(0,1)
```

```
In [ ]: random.gauss(0,1)
```

random.randint を random.randint(from,to) として呼び出すと、 from から to までの整数を等確率で返します。 to も含まれることに注意してください。

```
In [ ]: random.randint(1,10)
```

```
In [ ]: random.randint(1,10)
```

```
In [ ]: random.randint(0,1)
```

```
In [ ]: random.randint(0,1)
```

乱数を使用するときに、乱数として値を使いたい様な場合があります。まずリスタートを押してから、次を実行してみて下さい。

```
In [ ]: for i in range(5):
            print(random.randint(1,10))
```

その上で改めてリスタートを押して上のセルを実行してみて下さい。最初の実行時とは違う値が表示されるはずです。

例えば、乱数を使うコードを書いていて、「乱数で決定されるある値ではエラーが発生するけれどある値では発生しない」という場合、エラーを解消するには同じ乱数の値が出てくれた方が都合が良いですね。そこで次の様な関数 random.seed を使うことでそれを実現することが出来ます。具体的には、以下の様に使用します。

---

random.seed(a=整数):

整数に何でも良いので整数を指定します。すると、その指定された整数を基準にして乱数が初期化されますので、同じ値を乱数として使用することが出来る様になります。試してみましょう。

```
In [ ]: random.seed(a=0) # 0 を指定しました
        for i in range(5):
            print(random.randint(1,10))
```

リスタートを押しても押さなくても、以下を実行すると、上のセルと同じ値が表示されます。

```
In [ ]: random.seed(a=0) # 0 を指定しました
        for i in range(5):
            print(random.randint(1,10))
```

5.1.8 練習

実行すると 1/5 の確率で 0 を、 3/10 の確率で 1 を、1/2 の確率で 2 を返す関数 paperrockscissors 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。

```
In [ ]: import ...
        def paperrockscissors():
            ...
```

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。

```
In [ ]: list1 = [0] * 3
        for i in range(100000):
            list1[paperrockscissors()] += 1
        print(abs(list1[0]-20000)<500, abs(list1[1]-30000)<500, abs(list1[2]-50000)<500)
```

5.1.9 練習

二次元のランダムウォークをシミュレートしてみましょう。

以下のような関数 brownian(n) を定義してください。
まず、

```
xs = []
ys = []
```

として、変数 xs と ys を空リストで初期化します。
x と y という二つの変数を用意して、それぞれ 0.0 で初期化します。

```
x = 0.0
y = 0.0
```

i を 0 から n-1 まで動かします。
各ステップで、xs と ys の最後にそれぞれ x と y を追加します。
その後、x と y を以下のように更新します。
random.random() を使って 0 以上 1 未満の実数を一つ選んで、それに円周率の二倍を掛けて、角度を求めます。
そして、その角度の cos と sin を求めて、それぞれを x と y に足し込みます。
円周率は、math.pi で得られます。cos と sin は、math.sin と math.cos で計算できます。
brownian(n) は、最終的に xs と ys のタプルを返してください。

```
return xs,ys
```

以下のようなライブラリが必要になります。

```
In [ ]: import random
        import math
        import matplotlib.pyplot as plt
        %matplotlib inline
```

以下で brownian(n) を定義してください。

```
In [ ]: def brownian(n):
            ...
            return xs,ys
```

以下のようにして 100 ステップのランダムウォークをプロットしましょう。

```
In [ ]: xs,ys = brownian(100)
        plt.plot(xs,ys);
```

以下は、n ステップのランダムウォーク後の原点からの距離を、m 回求めてリストにして返す関数です。

```
In [ ]: def brownian_dists(m,n):
            ds =  []
            for i in range(m):
                xs,ys = brownian(n)
                ds.append((xs[n-1]**2 + ys[n-1]**2)**0.5)
            return ds
```

```
In [ ]: ds = brownian_dists(1000,1000)
```

原点からの距離のヒストグラムを表示します。

```
In [ ]: plt.hist(ds, bins=50);
```

ステップ数が増えるにしたがって原点からの平均距離がどのように増加するかを観察しましょう。時間がかかります。

```
In [ ]: ad = [0.0]
        for i in range(1,100):
            ds = brownian_dists(100,100*i)
            #print(sum(ds)/len(ds))
            ad.append(sum(ds)/len(ds))
```

```
In [ ]: plt.plot(ad);
```

ステップ数の平方根に比例するようです。

5.1.10 time

time は、時間に関する関数から成るモジュールです。詳しくは、以下を参照してください。
https://docs.python.jp/3/library/time.html

```
In [ ]: import time
```

time.time 関数は、ある定まった起点から現在までの秒数を実数として返します。

```
In [ ]: time.time()
```

次のコードでは、最初に現在の時刻を t という変数に記憶しておき、少し時間のかかる計算を行った後、 現在の時刻から t を引いて得られる秒数を出力しています。これで計算にかかった秒数を知ることができます。

```
In [ ]: t = time.time()

        def fib(n):
            if (n == 0):
                return 0
            elif (n == 1):
                return 1
            else:
                return fib(n-1)+fib(n-2)
        print(fib(32))

        print(time.time() - t)
```

time.localtime 関数は、現在の時刻の情報をオブジェクトとして返します。

```
In [ ]: time.localtime()
```

このオブジェクトの属性を参照することにより、時刻に関する情報を得ることができます。属性名は上のセルの結果に含まれています。

```
In [ ]: t = time.localtime()
        t.tm_year
```

5.1.11 練習

fib(n) にかかる時間を返す fibtime(n) という関数を定義してください。

```
In [ ]: def fibtime(n):
            ...
```

以下を実行してグラフをプロットしてください。

```
In [ ]: import matplotlib.pyplot as plt
        %matplotlib inline
        plt.plot([fibtime(i) for i in range(20,35)]);
```

```
In [ ]: import matplotlib.pyplot as plt
        %matplotlib inline
        plt.plot([fibtime(i) for i in range(20,35)]);
```

5.1.12 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。

```
In [ ]: from math import sqrt, sin, pi
        print(sqrt(2))
        print(sin(pi))
```

```
In [ ]: import random
        def paperrockscissors():
            prc1 = [0, 0, 1, 1, 1, 2, 2, 2, 2, 2]
            int1 = random.randint(0, 9)
            return prc1[int1]
```

```
In [ ]: def brownian(n):
            xs = []
            ys = []
            x = 0.0
            y = 0.0
            for i in range(n):
                xs.append(x)
                ys.append(y)
                theta = random.random()
                x += math.cos(2*math.pi*theta)
                y += math.sin(2*math.pi*theta)
            return xs,ys
```

```
In [ ]: def fibtime(n):
            t = time.time()
            fib(n)
            return time.time() - t
```

```
        def fibtime(n):
            t = time.perf_counter()
            fib(n)
            return time.perf_counter() - t
```

```
In [ ]:
```

5.2 NumPy ライブラリ

NumPy ライブラリを用いることにより、Python 標準のリストよりも効率的に多次元の配列（行列）を扱うことができます。これにより高速な行列演算が可能になるため、行列演算を行う科学技術計算などでよく活用されています。以下では、NumPy ライブラリの配列の基本的な操作や機能を説明します。

5.2.1 配列の作成

NumPy ライブラリを使用するには、まず numpy 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 np と別名をつけてコードの中で使用します。

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

NumPy の配列は numpy モジュールの array() 関数で作ります。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。

```
In [ ]: # リストから配列作成
        list1 = [1,2,3,4,5]
        list_to_array = np.array(list1)
        print("リスト", list1, "から作成した配列: ", list_to_array)
        # タプルからの配列作成
        tuple1 = (1,2,3,4,5)
        tuple_to_array = np.array(tuple1)
        print("タプル", tuple1, "から作成した配列: ", tuple_to_array)
```

リストとは異なり、要素が , ではなく空白で区切られて表示されます。

NumPy の配列は ndarray オブジェクトと呼ばれるデータ型によって実現されています。データ型を調べるには type 関数を使います。上で作成した配列で確かめてみましょう。

```
In [ ]: # 配列の型
        print(type(list_to_array))
        print(type(tuple_to_array))
```

<class 'numpy.ndarray'> と表示されたはずです。これは ndarray オブジェクトを意味しています。（class の意味は第 6 回で学習します。）

配列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
int32	整数を表す型
float64	実数を表す型
complex128	複素数を表す型

型名	説明
bool	真理値（ True 、もしくは False）を表す型

NumPy の配列はリストと異なり、要素の型を混在させることはできません。
配列の要素の型は

（配列）.dtype

という値に格納されています。調べてみましょう。

```
In [ ]: print(list_to_array.dtype)
```

int32 と表示されたと思います。 list_to_array と dtype の間にある . については、第 6 回で勉強しますが、一般に . の後に続く値のことを、前の値の属性と呼びます。ここでは、「list_to_array の dtype 属性」と呼びます。
array() 関数では、配列の作成時に第 2 引数 dtype の値を指定することで、配列の要素の型を指定することができます。

```
In [ ]: list_to_array = np.array([-1,0,1,2,3], dtype="int32") # 配列要素の型の指定
        #list_to_array = np.array([-1,0,1,2,3], dtype="int")#としても同じ
        print(list_to_array.dtype, list_to_array)# 配列要素の型の確認
        list_to_array = np.array([-1,0,1,2,3], dtype="float64")
        #list_to_array = np.array([-1,0,1,2,3], dtype="float")#としても同じ
        print(list_to_array.dtype, list_to_array)
        list_to_array = np.array([-1,0,1,2,3], dtype="complex128")
        #list_to_array = np.array([-1,0,1,2,3], dtype="complex")#としても同じ
        print(list_to_array.dtype, list_to_array)
        list_to_array = np.array([-1,0,1,2,3], dtype="bool")
        print(list_to_array.dtype, list_to_array)
```

5.2.2 練習

2 つの正の整数 int1 と int2 を引数として取り、NumPy の配列 arr1 を返す関数 construct_array を作成して下さい。 ただし、 arr1 の大きさ 4 の要素を int32 とする配列であり、次の様なリスト list1 から作成される。また、 list1 の 1 番目の要素は int1 + int2 を、2 番目の要素は int1 - int2 を、3 番目の要素は int1 * int2 を、4 番目の要素は int1 / int2 を格納しているとします。
以下のセルの ... のところを書き換えて解答して下さい。

```
In [ ]: import ...
        def construct_array(int1, int2):
            ...
```

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。

```
In [ ]: print(construct_array(9, 2) == np.array([11, 7, 18, 4]))
```

5.2.3 多次元配列の作成

多次元配列は、配列の中に配列がある入れ子の配列です。NumPy は多次元配列を効率的に扱うことができます。NumPy では、`array()` 関数の引数にリストが入れ子になった多重リストを与えると多次元配列が作成できます。

```
In [ ]: # 多次元配列の作成
        mul_array = np.array([[1,2,3],[4,5,6]])
        print(mul_array)
```

NumPy において扱う多次元配列は一般に行列を扱うことを想定しています。すなわち、多重リストの各リストの大きさは同じであることを想定しています。行列では、値の横の並びを**行**と呼び、値の縦の並びを**列**と呼びます。

例えば、先の多重配列 `mul_array` は 2 行 3 列の行列となっています（2 × 3 行列とも言います）。

`shape` 属性で、配列（行列）が何行何列かを調べることができます。また、`ndim` 属性で、何次元の配列か（`shape` 属性の大きさ）を調べることができます。`size` 属性では、配列の要素の個数を調べることができます。

```
In [ ]: # 多次元配列の行数と列数
        print(mul_array.shape)

        # 多次元配列の次元数
        print(mul_array.ndim) # 一般に len(mul_array.shape) に等しい

        # 多次元配列の要素数
        print(mul_array.size)
```

`reshape()` メソッドを使うと、`reshape`(行数、列数）と指定して、1 次元配列を多次元配列に変換することができます。`reshape()` で変換した多次元配列の操作の結果は元の配列にも反映されることに注意してください。

`ravel()` メソッドまたは `flatten()` 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。

```
In [ ]: mydata = [1,2,3,4,5,6]
        a1 = np.array(mydata)

        # 2 行 3 列の多次元配列に変換
        a2 = a1.reshape(2,3)
        print(a1)
        print(a2)

        # 1 行 1 列の要素に代入（後述）
        a2[0,0]=0
        print(a1)
        print(a2)

        # 多次元配列を 1 次元配列に戻す
        print(a2.ravel())
```

5.2.4 練習

空でないリスト list1 と正の整数 int1 を引数として取り、NumPy の配列 ary1 を返す関数 construct_copymatrix を作成して下さい。 ただし、 ary1 は、 int1 × len(list1) の大きさの多重配列です。なお、各列に対して、その列の j 番目の要素の値は list1 の j 番目の要素の値に等しいものとします。

以下のセルの ... のところを書き換えて解答して下さい。

```
In [ ]: import ...
        def construct_copymatrix(list1, int1):
            ...
```

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。

```
In [ ]: print(construct_copymatrix([2,4,6,8], 3) == np.array([[2,4,6,8],[2,4,6,8],[2,4,6,8]]))
```

5.2.5 練習

空でないリスト list1 と正の整数 int1 を引数として取り、NumPy の配列 ary1 を返す関数 construct_copymatrix2 を作成して下さい。 ただし、ary1 は、len(list1) × int1 の大きさの多重配列です。なお、各行に対して、その行の全ての値は list1 の j 番目の要素の値に等しいものとします。

以下のセルの ... のところを書き換えて解答して下さい。

```
In [ ]: import ...
        def construct_copymatrix2(list1, int1):
            ...
```

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。

```
In [ ]: print(construct_copymatrix2([2,4,6,8], 3) == np.array([[2,2,2],[4,4,4],[6,6,6],[8,8,8]]))
```

5.2.6 様々な配列の作り方

5.2.6.1 arange() 関数

arange() 関数を用いると、開始値から一定の刻み幅で生成した値の要素からなる配列を作成できます。具体的には、arange() 関数の第 1 引数には開始値 val1 、第 2 引数には終了値 val2 、第 3 引数には刻み幅 val3 を指定します。その返り値は、val1, val1+val3, val1+2*val3, val1+3*val3, ... という値を格納した配列となり、配列の最後の値は val1+x*val3 となる値です。ただし、 x は val2 > val1+x*val3 を満たす最大の整数です。つまり、終了値は生成される値に含まれないことに注意してください。arange() 関数では dtype で数値の型も指定できますが、省略すると開始値、終了値、刻み幅に合わせて型が選ばれます。

numpy.arange(開始値、終了値、刻み幅、dtype=型)

開始値を省略すると 0 、刻み幅を省略すると 1 がそれぞれ初期値となります。

```
In [ ]: # 0 から 1 刻みで 5 つの要素を持つ配列
        ary1 = np.arange(5)
        print(ary1)
```

```
        # 0 から 0.1 刻みで 1 未満の値の要素を持つ配列
        ary2 = np.arange(0, 1, 0.1)
        print(ary2)

        # 0 から 1 刻みで 4 つの要素を持つ配列を 2 行 2 列の多次元配列に変換
        ary3 = np.arange(4)
        ary3 = ary3.reshape(2, 2)
        #ary3=np.arange(4).reshape(2,2) #の様に一行で記述しても同じ
        print(ary3)
```

5.2.6.2 linspace() 関数

linspace() 関数を用いると、分割数を指定することで値の範囲を等間隔で分割した値の要素からなる配列を作成できます。linspace() 関数の第 1 引数には開始値、第 2 引数には終了値、第 3 引数には分割数を指定します。

```
In [ ]: # 0 から 100 の値を 11 分割した値を要素に持つ配列
        ary1 = np.linspace(0, 100, 11)
        print(ary1)
```

5.2.6.3 zeros() 関数

zeros() 関数を用いると、すべての要素が 0 の配列を作成することができます。zero() 関数の第 1 引数には 0 の個数を（多次元配列の場合は行数と列数をタプルで）指定し、第 2 引数の dtype に数値の型を指定します。

```
In [ ]: # 5 つの 0 要素からなる配列
        zero_array1 = np.zeros(5, dtype=int)
        print(zero_array1)

        # 3 行 4 列の 0 要素からなる多次元配列
        zero_array2 = np.zeros((3, 4), dtype=int)
        print(zero_array2)
```

5.2.6.4 ones() 関数

ones() 関数を用いると、すべての要素が 1 の配列を作成することができます。ones() 関数の第 1 引数には 1 の個数を（多次元配列の場合は行数と列数をタプルで）指定し、第 2 引数の dtype に数値の型を指定します。

```
In [ ]: # 4 行 3 列の 1 要素からなる多次元配列
        one_array = np.ones((4, 3), dtype=int)
        print(one_array)
```

5.2.6.5 random.rand() 関数

random.rand() 関数を用いると、乱数の配列を作成することができます。random.rand() 関数では、引数で与えた個数の乱数が 0 から 1 の間の値で生成されます。この他にも、random.randn() 関数、random.binomial() 関数、random.poisson() 関数を用いると、それぞれ正規分布、二項分布、ポアソン分布から乱数の配列を作成することができます。

```
In [ ]: # 5 つのランダムな値の要素からなる多次元配列
        rand_array = np.random.rand(5)
        print(rand_array)
```

5.2.7 練習

開始値 val1 、終了値 val2 、刻み幅 val3 を引数として取り、次の様な返り値を返す関数 arange_plus を作成して下さい。 その返り値は、val1, val1+val3, val1+2*val3, val1+3*val3, ... という値を格納した配列となり、配列の最後の値は val1+x*val3 となる値です。ただし、 x は val2 >= val1+x*val3 を満たす最大の整数です。

以下のセルの ... のところを書き換えて解答して下さい。

```
In [ ]: import ...
        def arange_plus(val1, val2, val3):
            ...
```

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。

```
In [ ]: print(arange_plus(10, 30, 2) == np.array([10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30]))
```

5.2.8 配列要素の操作

5.2.8.1 インデックス

NumPy の配列の要素を利用するには、リストの場合と同様に 0 から始まる**インデックス**を使います。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。

```
In [ ]: ary1 = np.array([1,3,5,7,9])
        print(ary1)

        # 配列 ary1 のインデックス 0 の要素
        print(ary1[0])

        # 配列 ary1 のインデックス -1 (終端) の要素
        print(ary1[-1])

        # 配列 ary1 のインデックス -1 の要素に代入
        ary1[-1]=0
        print(ary1)
```

多次元配列では、**配列名 [行, 列]** のように行と列のインデックスをそれぞれ指定します。この時、通常の配列のインデックスと同じくそれぞれ 0 から始まります。例えば、下記の 2 × 3 の多重配列の場合、行については 0 行と 1 行を、列については 0 列から 2 列までを指定可能です。また、多重リストと同様に、**配列名 [行] [列]** のようにしても同じです。

```
In [ ]: ary1 = np.array([[1,2,3],[4,5,6]])
        print(ary1)

        # 1 行 2 列の要素
```

```
print(ary1[1, 2])
# 1 行 2 列の要素
print(ary1[1][2])

# 1 行 2 列の要素に代入
ary1[1,2] = 0
print(ary1[1][2])
```

5.2.8.2 スライス

リストと同様に、NumPy の配列でも、**array[開始位置:終了位置:ステップ]** のように**スライス**を用いて配列の要素を抜き出すことができます。リストと同じく、スライスの開始位置や終了位置は省略が可能です。

```
In [ ]: ary1 = np.array([0, 10, 20, 30, 40])

        # 配列 ary1 のインデックス 1 からインデックス 3 までの要素をスライス
        print(ary1[1:4])

        # 配列 ary1 のインデックス 1 から終端までの要素をスライス
        print(ary1[1:])

        # 配列 ary1 の先頭から終端から 3 番目までの要素をスライス
        print(ary1[:-2])

        # 配列 ary1 の先頭から 1 つ飛ばしで要素をスライス
        print(ary1[::2])

        # 配列 ary1 の終端から先頭までの要素をスライス
        print(ary1[::-1])
```

NumPy の配列では、配列からスライスで抜き出した要素に値をまとめて代入することができます。配列においてスライスに対する変更は元の配列にも反映されることに注意してください。

```
In [ ]: ary1 = np.array([0, 10, 20, 30, 40])

        # 配列 ary1 のインデックス 1 からインデックス 3 までの要素に 0 を代入
        ary1[1:4] = -10
        print(ary1)
```

多次元配列のスライスでは、**array[行のスライス，列のスライス]** のように行と列のスライスのそれぞれの指定をコンマで区切って指定します。

```
In [ ]: ary1 = np.array( [1,2,3,4,5,6,7,8,9])
        ary1 = ary1.reshape(3,3)
        print(ary1)

        # 多次元配列 a の先頭行から 2 行目、先頭列から 2 列目までの要素をスライス
        print(ary1[:2,:2])
```

```
        # 多次元配列 a の 2 行目から終端行、2 列目から終端列までの要素をスライス
        print(ary1[1:,1:])
```

5.2.8.3 要素の順序取り出し

リストと同様に、for...in 文を用いて、配列の要素を順番に取り出すことができます。

```
In [ ]: ary1 = np.array([1,2,3,4,5,6])
        # 配列 ary1 から要素の取り出し
        for num in ary1:
            print(num)
```

多次元配列になっている場合も同様です。

```
In [ ]: ary2 = np.array([1,2,3,4,5,6])
        ary2 = ary2.reshape(2,3)
        i = 1
        # 多次元配列 ary2 から行の取り出し
        for row in ary2:
            print("多重配列 ary2 の", i, "番目の要素（配列）:", row)
            i += 1
            j = 1
            # 多次元配列 ary2 の行の要素の取り出し
            for num in row:
                print ("多重配列 ary2 の", i, "番目の要素（配列）の", j, "番目の要素:", num)
                j += 1
```

enumerate() 関数を使うと、リストと同じく、取り出しの繰り返し回数も併せて数えることができます。

```
In [ ]: ary1 = np.array([1,2,3,4,5,6])
        # 配列 ary1 から繰り返し回数と要素の取り出し
        print("ary1: ")
        for i, num in enumerate(ary1):
            print(i+1, "番目の要素: ", num)

        ary2 = np.array([1,2,3,4,5,6])
        ary2 = ary2.reshape(2,3)
        print("ary2: ")
        # 多次元配列 ary2 から繰り返し回数と要素（行）の取り出し
        for i, num in enumerate(ary2):
            print(i+1, "番目の要素: ", num)
```

多次元配列の要素の取り出しでは enumerate() 関数の代わりに ndenumerate() 関数を用います。ndenumerate() 関数は取り出した要素とともに、その要素の位置を行と列のタプルで返します。

```
In [ ]: ary2 = np.array([1,2,3,4,5,6])
        ary2 = ary2.reshape(2,3)
        # 多次元配列 ary2 から要素の位置（行と列をタプルで表現する）と対応する要素の取り出し
```

```
        for i, num in np.ndenumerate(ary2):
            print(i, num)
```

5.2.8.4 要素の並び替え

配列の要素の並び替えには、ndarray オブジェクトの sort() メソッド、または NumPy ライブラリの sort() 関数を使います。sort() メソッドは、メソッドを呼び出した自身の配列の要素を並び替えるので破壊的です。

```
In [ ]: ary1 = np.array([5,3,1,4,2])
        # 配列 ary1 の要素を並び替え
        ary1.sort() # ndarray オブジェクトの sort() メソッドで並べ替え
        print("ary1:", ary1)
```

一方、sort() 関数は引数で与えた配列の要素を並び替えた新しい配列を返しますので非破壊的です。sort() 関数の引数にリストやタプルを指定し、それらの並び替えを行った結果を配列として取得することもできます。

```
In [ ]: ary2 = np.array([5,3,1,4,2])
        # 配列 ary2 の要素を並び替えた結果から新たな配列 ary3 を作成
        ary3 = np.sort(ary2) # NumPy ライブラリの sort() 関数で並べ替え

        print("ary2:", ary2)
        print("ary3:", ary3)
```

5.2.9 配列の演算

NumPy の配列では、配列のすべての要素に数値演算を適用するブロードキャストという機能により、要素が数値である配列の演算を簡単に行うことができます。

```
In [ ]: ary1 = np.array([1,2,3,4])
        print(ary1)

        # 配列 ary1 のすべての要素に 1 を加算
        ary2 = ary1 + 1
        print(ary2)

        # 配列 ary2 のすべての要素に 2 を乗算
        ary3 = ary3 * 2
        print(ary3)

        # 配列 ary3 のすべての要素に 2 を除算
        ary4 = ary3 / 2
        print(ary4)

        # 配列 ary5 のすべての要素を二乗
        ary5 = ary4 ** 2
        print(ary5)
```

この他に、NumPy には**ユニバーサル関数**と呼ばれる、配列を入力として、そのすべての要素を操作した結果を配列として返す関数が複数あります。ユニバーサル関数については以下を参照してください。

- ユニバーサル関数の一覧

ndarray オブジェクトのメソッドを用いて、要素の合計、平均値、最大値、最小値を、それぞれ sum()、mean()、max()、min() で求めることができます。各メソッドは引数を指定しなければ配列のすべての要素に適用されます。多次元配列の場合、引数に 0 を指定すると、各列にメソッドを適用した結果の配列、引数に 1 を指定すると各行にメソッドを適用した結果の配列が返ります。

```
In [ ]: ary1 = np.array([1,2,3,4,5,6])
        ary1 = ary1.reshape(2,3)
        print(ary1)

        # 多次元配列 ary1 のすべての要素の平均
        print(ary1.mean())

        # 多次元配列 ary1 の各列の要素の平均
        print(ary1.mean(0))

        # 多次元配列 ary1 の各行の要素の平均
        print(ary1.mean(1))
```

この他の NumPy の数学・統計関連のメソッド・関数については以下を参照してください。

- 数学関数
- 統計関数

5.2.10 配列同士の演算

行数と列数が同じ配列同士の四則演算は、各要素同士の演算となります。

```
In [ ]: A1 = np.array([1,2,3,4]).reshape(2,2)
        B1 = np.array([2,4,6,8]).reshape(2,2)

        # 配列の要素同士の足し算
        C1 = A1 + B1
        print(C1)

        # 配列の要素同士の引き算
        D1 = B1 - A1
        print(D1)

        # 配列の要素同士の掛け算
        E1 = A1 * B1
        print(E1)

        # 配列の要素同士の割り算
```

```
F1 = B1 // A1
print(F1)
```

5.2.11 練習

NumPy の 2 次元の多重配列 ary1 を引数として取り、次の様な整数 int1 を返り値を返す関数 get_minmax を作成して下さい。ただし、int1 は次の様に求めます。

1. ary1 の各行を構成する配列に対して、それぞれ最小値を求めます。
2. 1 で求めた値の中で最大の値が int1 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。

```
In [ ]: import ...
        def get_minmax(ary1):
            ...
```

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。

```
In [ ]: print(get_minmax(np.array([[10, 30, 55], [0, 40, 15], [35, 50, 75], [15, 66, 20]])) == 35)
```

5.2.12 NumPy の関数

5.2.12.1 dot()

dot は、2 つの配列を引数に取り、その内積を返します。
具体的には次の様に用います。

```
numpy.dot(配列 A, 配列 B)
```

```
In [ ]: ary1 = np.array([2, 3, 1])
        ary2 = np.array([1, 2, 1])
        np.dot(ary1, ary2)
```

5.2.12.2 linalg.norm()

linalg.norm は、配列を引数に取り、原点からその配列の値によって表される点までの距離（ノルム）を返します。
具体的には次の様に用います。

```
numpy.linalg.norm(配列 A)
```

```
In [ ]: ary1 = np.array([2, 2, 1])
        np.linalg.norm(ary1)
```

5.2.12.3 sqrt()

sqrt は、0 以上の数を引数に取り、その平方根を返します。
具体的には次の様に用います。

```
numpy.sqrt(数)
```

```
In [ ]: print(np.sqrt(2))
        import math # math を使った場合と同じ結果が得られます
        print(math.sqrt(2))
```

配列を引数に取ることも出来ます。

```
In [ ]: ary1 = np.array([2, 2, 1])
        np.sqrt(ary1)
```

5.2.13 文字列型の配列

配列の要素が文字列の時は、第 2 引数 dtype に “<U” を指定すると、要素の文字列の最長値に合わせて、文字列の長さが決まります。また、dtype に “<U” と数値を続けて指定（例えば、“<U5”) すると、文字列の長さはその数値の固定長となります。

```
In [ ]: # 配列要素の文字列の長さを最長値の文字列要素に合わせる
        str_array = np.array(['a','bb','ccc'], dtype="<U")
        str_array
```

```
In [ ]: # 配列要素の文字列の長さ 2 に合わせる
        str_array = np.array(['a','bb','ccc'], dtype="<U2")
        str_array
```

5.2.14 ▲配列要素の追加、挿入、削除

5.2.14.1 append() 関数

NumPy の配列の要素の追加には append() 関数を使います。append() 関数の第 1 引数には配列を指定し、第 2 引数にはその配列に追加する値を指定します。リストやタプルで複数の値を同時に指定することもできます。NumPy の append() 関数は、要素を追加した新しい配列が返り、元の配列は変化しないことに注意してください。

```
In [ ]: a1 = np.array([1,2])

        # 配列 a1 に値 3 を要素として追加
        a2 = np.append(a1, 3)

        # 配列 a2 に値 4,5 を要素として追加
        a3 = np.append(a2,[4,5])
```

```
print(a1)
print(a2)
print(a3)
```

多次元配列に要素を追加する場合は、append 関数の axis 引数に対して、行追加であれば 0、列追加であれば 1 を渡します。追加する要素は、追加先の配列の行または列と同じ次元の配列である必要があります。

```
In [ ]: mul_array1 = np.array([[1,2,3],[4,5,6]])

        #  多次元配列 mul_array1 に行 [7,8,9] を追加
        mul_array2 = np.append(mul_array1, [[7,8,9]], axis =0)

        #  多次元配列 mul_array2 に列 [0,0,0] を追加
        mul_array3 = np.append(mul_array2, np.array([[0,0,0]]).T, axis =1)

        print(mul_array1)
        print(mul_array2)
        print(mul_array3)
```

5.2.14.2 insert() 関数

NumPy の配列の要素の挿入には insert() 関数を使います。insert() 関数の第 1 引数には配列、第 2 引数には要素を挿入する位置、第 3 引数にはその配列に追加する値を指定します。値は、リストやタプルで複数を同時に指定することもできます。NumPy の insert() 関数は、要素を追加した新しい配列が返り、元の配列は変化しないことに注意してください。

```
In [ ]: a1 = np.array([1,3])

        # 配列 a1 のインデックス 1 に値 2 を要素として追加
        a2 = np.insert(a1, 1, 2)

        # 配列 a2 のインデックス 3 に値 4,5 を要素として追加
        a3 = np.insert(a2, 3, [4,5])

        print(a1)
        print(a2)
        print(a3)
```

多次元配列に要素を挿入する場合は、axis 引数に対して、行追加であれば 0、列追加であれば 1 を渡します。挿入する要素は、挿入先の配列の行または列と同じ次元の配列である必要があります。

```
In [ ]: mul_array1 = np.array([[1,2,3],[7,8,9]])

        #  多次元配列 mul_array1 に行 [4,5,6] を追加
        mul_array2 = np.insert(mul_array1, 1, [[4,5,6]], axis =0)

        print(mul_array1)
        print(mul_array2)
```

5.2.14.3 delete() 関数

NumPy の配列の要素の削除には delete() 関数を使います。delete() 関数の第 1 引数には配列、第 2 引数には削除する要素の位置を指定します。delete 関数でも append() 関数、insert() 関数と同様に、元の配列は変化しないことに注意してください。

```
In [ ]: a1 = np.array([0,1,2])

        # 配列 a1 のインデックス 2 の要素を削除
        a2 = np.delete(a1,2)

        print(a1)
        print(a2)
```

5.2.15 ▲要素の条件取り出し

条件式を用いて、配列の要素の中から条件に合う要素のみを抽出し、要素の値を変更したり、新たな配列を作成することができます。配列と比較演算を組み合わせることで、比較演算が配列の個々の要素に適用されます。

条件式のブール演算では、and、or、not の代わりに&、|、~を用います。

```
In [ ]: a1 = np.array([1,2,-3,-4,5,-6,-7])

        # 0 未満で 2 で割り切れる値を持つ要素に 0 を代入
        a1[(a1<0) & (a1%2==0)]=0
        print(a1)
```

以下の例において、print(a1>0) とすると [True True False False True False False] というブール値の配列が返ってきていることがわかります。True は条件（この場合は要素が正）に対して真な要素（この場合は 1,2,5）に対応しています。配列要素の条件取り出しでは、このブール値の配列を元の配列に渡して、条件に対して真な要素のインデックスを参照していることになります。これを**ブールインデックス参照**と呼びます。

```
In [ ]: a1 = np.array([1,2,-3,-4,5,-6,-7])

        # 0 より大きい値を持つ要素は True, それ以外は False のブール値配列
        print(a1>0)

        # 配列 a1 の 0 より大きい値を持つ要素から新たな配列 a2 の作成
        a2 = a1[a1>0]
        print(a2)
```

5.2.16 ▲ブロードキャスト

行数と列数が異なる配列や行列同士の四則演算では、足りない行や列の値を補うブロードキャストが行われます。以下の例では、配列 A と演算に対して、配列 B の 2 行目が足りないため、B の 1 行目と同じ値で 2 行目を補い演算を行なっています。このようなブロードキャストが機能するのは、B の行数または列数が A のそれらと同じ場合、または配列 B が 1 行・1 列の場合です。

```
In [ ]: A = np.array([1,2,3,4]).reshape(2,2)
        B = np.array([2,4])

        # 行列 B をブロードキャストして行列 A と足し算
        C = A+B
        print(C)
```

5.2.17 ▲行列の演算

dot() 関数を使うとベクトルの**内積**や**行列積**を計算することができます。この時、それぞれの配列の行数と列数、または列数と行数が同じである必要があります。

```
In [ ]: A = np.array([1,2,3,4]).reshape(2,2)
        B = np.array([2,4,6,8]).reshape(2,2)

        # 行列積
        C = np.dot(A,B)
        print(C)
```

単位行列は identity() 関数または eye() 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。

```
In [ ]: # 3 行 3 列の単位行列
        E = np.identity(3, dtype=int)
        print(E)
```

transpose() 関数または配列の T 属性で、配列の行と列の要素を入れ替えた配列を得ることができます。この時、元の配列の形状を変えているだけで元の配列を直接変更していないことに注意してください。

```
In [ ]: A = np.array([1,2,3,4,5,6]).reshape(2,3)

        # 配列の行と列の入れ替え
        print(np.transpose(A))
        print(A.T)
        print(A)
```

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機能は、numpy.linalg モジュールで提供されています。同モジュールについては以下を参照してください。- 線形代数関連関数

5.2.18 練習の解答

```
In [ ]: import numpy as np
        def construct_array(int1, int2):
            list1 = [int1 + int2, int1 - int2, int1 * int2, int1 / int2]
            arr1 = np.array(list1, dtype="int32")
            return arr1
```

```
In [ ]: import numpy as np
        def construct_identitymatrix(int1):
```

```
        list1 = []
        list2 = [0] * int1
        #print(list2)
        for i1 in range(int1):
            list2[i1] = [0] * int1
            list2[i1][i1] = 1
            list1.append(list2[i1])
            #print(list1)
        idmtrx = np.array(list2)
        return idmtrx
    #別解
    #def construct_identitymatrix(int1):
    #    idmtrx = np.identity(int1, dtype=int)
    #    return idmtrx
    #construct_identitymatrix(4)
```

In [ ]:
```
import numpy as np
def construct_copymatrix(list1, int1):
    list2 = []
    for i in range(int1):
        list3 = []
        for j in list1:
            list3.append(j)
        list2.append(list3)
    ary1 = np.array(list2)
    return ary1
#別解
#def construct_copymatrix(list1, int1):
#    list2 = []
#    for i in range(int1):
#        list2.append(list1)
#    ary1 = np.array(list2)
#    return ary1
#
#別解 2
#def construct_copymatrix(list1, int1):
#    list2 = list1*int1
#    print(list2)
#    ary1 = np.array(list2)
#    ary1 = ary1.reshape(int1, len(list1))
#    return ary1
```

In [ ]:
```
import numpy as np
def construct_copymatrix2(list1, int1):
    list2 = []
    for i in list1:
```

```
            list3 = [i] * int1
            list2.append(list3)
        ary1 = np.array(list2)
        return ary1
    #別解
    #def construct_copymatrix2(list1, int1):
    #    list2 = []
    #    for i in range(int1):
    #        list2.append(list1)
    #    print(list2)
    #    ary1 = np.array(list2)
    #    return ary1.T # Tという属性を使うと、ary1の行と列を入れ替えた行列を取得できます
    #
    #別解2
    #def construct_copymatrix2(list1, int1):
    #    list2 = list1*int1
    #    print(list2)
    #    ary1 = np.array(list2)
    #    ary1 = ary1.reshape(int1, len(list1))
    #    return ary1.T # Tという属性を使うと、ary1の行と列を入れ替えた行列を取得できます
```

```
In [ ]: import numpy as np
    def arange_plus(val1, val2, val3):
        ary1 = np.arange(val1, val2+val3, val3)
        return ary1
```

```
In [ ]: import numpy as np
    def get_minmax(ary1):
        ary2 = ary1.min(1)
        #print(ary2)
        ary3 = ary2.max()
        #print(ary3, type(ary3))
        return ary3
    #get_minmax(np.array([[10, 30, 55], [0, 40, 15], [35, 50, 75], [15, 66, 20]]))
```

5.3 ▲ Matplotlib ライブラリ

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。

Matoplotlib 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。

```
In [ ]: import numpy as np
        import matplotlib.pyplot as plt
        %matplotlib inline
```

5.3.1 線グラフ

pyplot モジュールの plot() 関数を用いて、リストの要素の数値を y 軸の値としてグラフを描画します。y 軸の値に対応する x 軸の値は、リストの各要素のインデックスとなっています。

具体的には、次の様にすることで 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。

```
plt.plot(リスト A)
```

例えば、次の様になります。

```
In [ ]: # plot するデータ
        d =[0, 1, 4, 9, 16]

        # plot 関数で描画
        plt.plot(d);
        # セルの最後に評価されたオブジェクトの出力表示を抑制するために、以下ではセルの最後の行にセミコロン (`;
        # 試しにセミコロンを消した場合も試してみて下さい。
```

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 X[i]) の位置に点を打ち、各点を線でつなぎます。

```
plt.plot(リスト X, リスト Y)
```

```
In [ ]: # 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。

```
In [ ]: # 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。

```
In [ ]: # 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

```
In [ ]: # plot するデータ
        data =[0, 1, 4, 9, 16]
        x =[0, 1, 2, 3, 4]
        y =[0, 1, 2, 3, 4]

        # plot 関数で描画。線の形状、色、データポイントのマーカ指定
        plt.plot(x,y, linestyle='--', color='blue', marker='o')
        plt.plot(data, linestyle=':', color='green', marker='*');
```

plot() 関数の label 引数にグラフの各線の凡例を文字列として渡し、legend() 関数を呼ぶことで、グラフ内に凡例を表示できます。legend() 関数の loc 引数で凡例を表示する位置を指定することができます。引数で指定可能な値は以下を参照してください。

- lengend() 関数

```
In [ ]: # plot するデータ
        data =[0, 1, 4, 9, 16]
        x =[0, 1, 2, 3, 4]
        y =[0, 1, 2, 3, 4]

        # plot 関数で描画。線の形状、色、データポイントのマーカ指定
        plt.plot(x,y, linestyle='--', color='blue', marker='o', label="linear")
        plt.plot(data, linestyle=':', color='green', marker='*', label="quad")
        #凡例を表示
        plt.legend();
```

pyplot モジュールでは、以下のようにグラフのタイトルと各軸のラベルを指定して表示することができます。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。

```
In [ ]: # 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。

```
In [ ]: #  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。

```
In [ ]: 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);
```

5.3.1.1 グラフの例：ソートアルゴリズムにおける比較回数

In [ ]:
```
import random

def bubble_sort(lst):
    n = 0
    for j in range(len(lst) - 1):
        for i in range(len(lst) - 1 - j):
            n = n + 1
            if lst[i] > lst[i+1]:
                lst[i + 1], lst[i] = lst[i], lst[i+1]
    return n

def merge_sort_rec(data, l, r, work):
    if l+1 >= r:
        return 0
    m = l+(r-l)//2
    n1 = merge_sort_rec(data, l, m, work)
    n2 = merge_sort_rec(data, m, r, work)
    n = 0
    i1 = l
    i2 = m
    for i in range(l, r):
        from1 = False
        if i2 >= r:
            from1 = True
        elif i1 < m:
            n = n + 1
            if data[i1] <= data[i2]:
                from1 = True
        if from1:
            work[i] = data[i1]
            i1 = i1 + 1
        else:
            work[i] = data[i2]
            i2 = i2 + 1
    for i in range(l, r):
        data[i] = work[i]
    return n1+n2+n

def merge_sort(data):
    return merge_sort_rec(data, 0, len(data), [0]*len(data))
```

In [ ]:
```
x = np.arange(100, 1100, 100)
bdata = np.array([bubble_sort([random.randint(1,10000) for i in range(k)]) for k in x])
```

```
        mdata = np.array([merge_sort([random.randint(1,10000) for i in range(k)]) for k in x])
```

```
In [ ]: 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);
```

5.3.2 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。

```
In [ ]: import ...
        ...
        def plot_exp():
            ...
```

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。

```
In [ ]: res_x = plot_exp()
        print(len(res_x) == 41, int(res_x[0]) == -2, int(res_x[9]) == -1)
```

5.3.3 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。

```
In [ ]: import ...
        ...
        def plot_tokyotemps(year, month):
            ...
```

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。

```
In [ ]: res_years, res_temps = plot_tokyotemps(1875, 7)
        print(len(res_years) == 142, len(res_temps) == 142, res_years[0] == 1875, res_temps[0] == 2
        res_years, res_temps = plot_tokyotemps(1875, 6)
        print(len(res_years) == 142, len(res_temps) == 142, res_years[0] == 1875, res_temps[0] == 2
        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
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
res_years, res_temps = plot_tokyotemps(1876, 6)
print(len(res_years) == 141, len(res_temps) == 141, res_years[0] == 1876, res_temps[0] == 1
res_years, res_temps = plot_tokyotemps(1900, 6)
print(len(res_years) == 117, len(res_temps) == 117, res_years[0] == 1900, res_temps[0] == 1
```

5.3.4 散布図

散布図は、pyplot モジュールの scatter() 関数を用いて描画できます。

具体的には、次の様に **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 X[i]**) の位置に点を打ちます。

```
plt.scatter(リスト X, リスト Y)
```

以下では、ランダムに生成した 20 個の要素からなる配列 x, y の各要素の値の組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。

```
In [ ]: #  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。

```
In [ ]: x = np.random.rand(20)
        y = np.random.rand(20)
        plt.plot(x, y, '*', color='blue');
```

5.3.5 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。

```
In [ ]: import ...
```

```
...
def scatter_tokyotemps():
    ...
```

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。

```
In [ ]: res_months, res_temps = scatter_tokyotemps()
        print(len(res_months) == 1700, len(res_temps) == 1700, res_months[0] == 6, res_months[1] ==
        print(res_temps[0] == 22.3, res_temps[1] == 26.0, res_temps[12] == 18.5, res_temps[13] == 2
```

5.3.6 棒グラフ

棒グラフは、pyplot モジュールの bar() 関数を用いて描画できます。以下では、ランダムに生成した 10 個の要素からなる配列 y の各要素の値を縦の棒グラフで表示しています。x は、x 軸上で棒グラフのバーの並ぶ位置を示しています。ここでは、numpy モジュールの arange() 関数を用いて、1 から 10 の範囲を 1 刻みで x 軸上のバーの並ぶ位置として配列を準備しています。

```
In [ ]: # 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);
```

5.3.7 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。

```
In [ ]: import ...
        ...
        def bar_tokyotemps(year1, month1, year2, month2):
            ...
```

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。

```
In [ ]: 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
```

5.3.8 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。

```
In [ ]: # 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);
```

5.3.9 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。

```
In [ ]: import ...
        ...
        def hist_tokyotemps(year1, month1, year2, month2, mybin):
            ...
```

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。

```
In [ ]: res_temps = hist_tokyotemps(1875, 6, 2000, 6, 50)
        print(len(res_temps) == 1501, res_temps[0] == 22.3, res_temps[1500] == 22.5)
```

5.3.10 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。

```
In [ ]: # 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);
```

5.3.11 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 ℃以上、 y+1 ℃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。

```
In [ ]: import ...
        ...
        def heat_tokyotemps():
            ...
```

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。

```
In [ ]: 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下さい
        #ary1 = np.flip(ary1, axis=0)
        #im=plt.imshow(ary1)
```

5.3.12 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。

```
In [ ]: x = np.arange(-np.pi, np.pi, 0.1)
        plt.plot(x,np.cos(x), label='cos')
        plt.plot(x,np.sin(x), label='sin')
        plt.legend()
        plt.title("cos ans sin Curves")
        plt.xlabel("x")
        plt.ylabel("y")
        plt.grid(True)

        # savefig 関数でグラフを画像保存
        plt.savefig('cos_sin.png');
```

5.3.13 練習の解答

```
In [ ]: 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
```

```
In [ ]: import numpy as np
        import matplotlib.pyplot as plt
        %matplotlib inline
        import csv

        def plot_tokyotemps(year, month):
            with open('tokyo-temps.csv', 'r', encoding='sjis') as f:
                dataReader = csv.reader(f) # csv リーダを作成
                n=0
                # 1875 年 6 月が 47 行目なので、指定された year 年 6 月のデータの行番号をまず求める
                init_row = (year - 1875) * 12 + 47
                # その上で、year 年 month 月のデータの行番号を求める
                init_row = init_row + month - 6
                years = [] # 年
                temps = [] # 平均気温
                for row in dataReader: # csv ファイルの中身を 1 行ずつ読み込み
                    n = n+1
                    if n >= init_row and (n - init_row) % 12 == 0: # init_row 行目からはじめて 12 か月
                        years.append(year)
                        temp = float(row[1]) # float 関数で実数のデータ型に変換する
                        temps.append(temp)
                        year = year + 1
            #print(years)
            #print(temps)
            plt.plot(years, temps)
            return years, temps
```

```
In [ ]: 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
```

In [ ]:
```
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 内を実行
                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
```

In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
import csv
```

```
        def hist_tokyotemps(year1, month1, year2, month2, mybin):
            with open('tokyo-temps.csv', 'r', encoding='sjis') as f:
                dataReader = csv.reader(f) # csv リーダを作成
                n=0
                months = [] #
                temps = [] # 平均気温
                init_row = (year1 - 1875) * 12 - 6 + month1 + 47
                end_row = (year2 - 1875) * 12 - 6 + month2 + 47
                for row in dataReader: # csv ファイルの中身を 1 行ずつ読み込み
                    n = n+1
                    if n >= init_row and n <= end_row: # init_row 行目から、end_row 行まで if 内を実行
                        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

In [ ]: 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
                        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
```

5.4 Python 実行ファイルとモジュール

5.4.1 Python プログラムファイル

この授業ではこれまで、プログラムを Jupyter ノートブック（拡張子.ipynb）のコードセル (Code) に書き込むスタイルを採ってきました。Jupyter ノートブックでは、コードセルに加え、文書セル (Markdown) および出力結果を json 形式で保存しています。この形式は学習には適していますが、Python で標準的に使われるプログラムファイル形式ではありません。

Python の標準のプログラムファイル形式（拡張子 .py）では Python プログラム、すなわち、Jupyter ノートブックにおけるコードセルの内容をファイルに記述します。

例えば、次のコードセルを実行してみて下さい。

```
In [ ]: a1 = 10
        print("a1 contains the value of", a1)
```

この内容と全く同じコードを記述した Python プログラムファイル 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、以下のようにします。

```
python sample.py
```

あるいは

```
python3 sample.py
```

では、以下の説明を参考に各自の環境で sample.py を実行してみて下さい。

5.4.1.1 Windows での実行方法（自身で Anaconda3 をインストールした場合）

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 ⇒ Anaonda3(64-bit) ⇒ Anaconda Prompt

下記の様なウインドウが表示されます。

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

例では、cd というコマンドで sample.py を格納したフォルダ pythontest に移動し、その上で sample.py を実行しています。

5.4.1.2 Windows での実行方法（ECCS の Windows 環境）

以下をクリックしてターミナルを開きます。 Start メニュー ⇒ Cygwin64 Terminal（デスクトップにショートカットアイコンがある場合があります）下記の様なウインドウが表示されます。

cygwin64 というフォルダの中に home フォルダがあります。更に、そのフォルダの中に、ユーザー ID のフォルダ（例の画像では 0488115111 というフォルダ）があります。その中に、 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。

この場合、python ではなく、 /cygdrive/c/Anaconda3/python を実行します。

5.4.1.3 MacOS での実行方法

Application ⇒ Utilities ⇒ Terminal.app を起動します。 アプリケーション ⇒ユーティリティ ⇒ ターミナル .app を起動します。（日本語の場合）

下記の様なウインドウが表示されます。

ダウンロードフォルダ (Downloads) に `sample.py` を格納した場合の実行例を示します。

例では、cd というコマンドで `sample.py` を格納した Downloads フォルダに移動し、その上で `sample.py` を実行しています。

5.4.2 プログラムファイルの文字コード

Python の標準プログラムファイル形式では、ヘッダ行が定義されており、プログラムで使用する文字コードや、Unix 環境では Python インタープリタのコマンドなどを記述します。Python の標準の文字コードは utf-8（8 ビットの Unicode）ですが、これに代えて Windows などで使われてきたシフト JIS (shift_jis) を利用する場合は、先頭行に以下を記述します。

```
# -*- coding: shift_jis -*-
```

例えば、次の画像の a.py というコードを実行するとエラーが出ます。

そこで、上記の 1 文を先頭行に追加するとエラーが起こらなくなります。

Unix ではプログラムスクリプトの先頭行（**shebang** 行）には、そのスクリプトを読み込み実行するコマンドを指定します。このケースでは先頭行は例えば、以下のようになります。

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

shebang 行では、コマンドを絶対パスで指定します。`/usr/bin/env python3` と指定すると、 env というコマンドは python3 インタープリタを環境変数から探して実行しますので、 python3 自身の絶対パスを指定する必要はありません。

5.4.3 Jupyter Notebook で Python プログラムファイル (.py) を扱う

Jupyter Notebook で Python プログラム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 Python プログラムファイル (.py) を開く
2. Jupyter ノートブックを Python プログラムファイル (.py) に変換する

5.4.3.1 Jupyter Notebook で Python プログラムファイルを開く

Jupyter Notebook で直接に Python の標準のプログラム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。

5.4.3.2 Jupyter ノートブックを Python プログラムファイル (.py) に変換する

講義で利用している Jupyter ノートブックを .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 Python プログラムファイルがダウンロードファイルとして生成されます。

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 Python プログラムファイルとして利用できます。

後者の方法は、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とはならないことに注意する必要があります。

ここでは Jupyter Notebook で Python プログラム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。

5.4.4 プログラムへの引数の渡し方

Python プログラムファイル (.py) の実行時には、実行するプログラム名の後に文字列を書き込むことにより、実行するプログラムに引数を与えることができます。

例えば、通常 argsprint.py というファイルを実行する場合、上で見た様に以下の様に実行します。

```
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 に引数として与えられることになります。

この引数は、sys というモジュールの 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 の中身を表示
```

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.argv に 2

番目の要素として文字列 firstvalue が、3 番目の要素として文字列 secondvalue が、4 番目の要素として文字列 thirdvalue が格納されていることを確認して下さい。また、リストの最初の要素には、実行したプログラム名（ここでは argsprint.py ）が格納されることに注意してください。

引数を変更したり、引数の数を増やしたり減らしたりして、表示がどう変わるか調べてみて下さい。

なお、 sys.argv は、下記の様に from sys import argv として、 argv として利用することも多い様です。

```
from sys import argv
print(argv)
```

5.4.5 練習

上記のように argsprint.py というプログラムファイルを作成して、実行して下さい。

5.4.6 練習

sample2.py というプログラムファイルを作成して下さい。このプログラムは実行すると、実行した際に与えた 1 番目の引数の値（仮に value と呼びます）を、

```
a1 contains the value of value
```

と表示 (print) します。

5.4.7 練習

showname.py というプログラムファイルを作成して下さい。このプログラムは実行すると、実行するファイル名を表示します。

5.4.8 練習

argsnum.py というプログラムファイルを作成して下さい。このプログラムは実行すると、引数の数が 3 個以下の場合には引数の数を表示し、引数の数が 4 個以上の場合は、too many と表示します。

5.4.9 練習

argssum.py というプログラムファイルを作成して下さい。このプログラムは引数に整数を（文字列として）与えて実行すると、引数の数を全て足した数を表示します。例えば、

```
python argssum.py 1 2 3
```

と実行すると、 6 と表示します。（文字列を整数に変換するには、int という関数を使って下さい。）

```
In [ ]: str1 = "100"
        str2 = "200"
```

```
print(str1+str2) # 文字列の結合
print(int(str1)+int(str2)) # int を使って文字列を整数に変換する
```

5.4.10 モジュール

プログラムが大きくなるとそれを複数のファイルに分割した方がプログラム開発・維持が簡単になります。また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にもファイル分割が必要となります。Python ではプログラムを**モジュール**の単位で複数のファイルに分割することができます。

以下が記述された fibonacci.py というモジュールを読み込む場合を説明します。

```
def fib(n):
    if (n == 0):
        return 0
    elif (n == 1):
        return 1
    else:
        return fib(n-1)+fib(n-2)
```

ここで定義されている関数を利用するには、import を用いて import **モジュール名** と書きます。**モジュール名**は、Python プログラムファイルの名前から拡張子.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。

```
In [ ]: import fibonacci

        print(fibonacci.fib(10))
```

from や as の使い方も既存のモジュールと全く同じです。

モジュール内で定義されている名前を読み込み元のプログラムでそのまま使いたい場合は、from を用いて以下のように書くことができます。

```
In [ ]: from fibonacci import fib

        print(fib(10))
```

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。

```
In [ ]: from fibonacci 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。

```
In [ ]: import fibonacci as f

        print(f.fib(10))
```

5.4.11 モジュールの検索パス

自身でつくるモジュールはプログラムファイルと同じディレクトリに置けば（当面は）問題ありません。

実際には、インポートされるモジュールは Python を実行するために事前に指定されたフォルダの場所（検索パスと言います）に置く必要があります。検索パスの情報は sys.path という変数から取得できます。

```
In [ ]: import sys
        sys.path
```

5.4.12 ▲モジュールファイルの実行

モジュールファイルは、他のプログラムで利用する関数・クラスを定義するだけではなく、それ自身を Pythoh プログラムとして実行したい場合もあります。

例えば、上で扱った fibonacci.py を Pythoh プログラムで実行する場合、引数に整数（ int1 としましょう）を指定すると、 fib(int1) という値を表示する様に変更しましょう。

そのためにはどうすれば良いか考えてみて下さい。例えば、次の様なコードを考える人がいるかも知れません。

```
import sys
def fib(n):
    if (n == 0):
        return 0
    elif (n == 1):
        return 1
    else:
        return fib(n-1)+fib(n-2)

int1 = int(sys.argv[1]) # 引数で与えられた整数の形をした文字列を整数に変換する
print(fib(int1))
```

では、このコードを保存したファイルの名前を fibonacci2.py として用意しましたので、まず Pythoh プログラムとして実行してみて下さい。例えば、下記の様な結果が得られるはずです。

では、次にモジュールファイルとして実行してみましょう。

```
In [ ]: import fibonacci2
        print(fibonacci2.fib(10))
```

今度はエラーが出たはずです。では、これをどう修正すれば良いのかというと、次の様に修正します。

```
import sys
def fib(n):
    if (n == 0):
        return 0
    elif (n == 1):
        return 1
```

```
    else:
        return fib(n-1)+fib(n-2)

if __name__ == "__main__":
    int1 = int(sys.argv[1])
    print(fib(int1))
```

このコードを保存したファイルの名前を fibonacci3.py として用意しましたので、再び Pythoh プログラムとして実行してみて下さい。

画像の様に、fibonacci2.py のときと同じ結果が得られます。では、次にモジュールファイルとして実行してみましょう。

```
In [ ]: import fibonacci3
        print(fibonacci3.fib(10))
```

今度は上手く行ったはずです。

つまり、Pythoh プログラムとして実行する場合にのみ実行するコードを、以下の様に if __name__ == "__main__": の中に記述する様にすれば良いということです。

関数・クラスの定義

```
if __name__ == "__main__":
    プログラムの実行文
```

Python では、コマンドによって直接実行されたファイルは"__main__" という名前のモジュールとして扱われます。プログラムの中でモジュールの名前は__name__ という変数に入っていますので、__name__ == "__main__" という条件により、直接実行されたファイルかどうかが判定できます。

例えば、用意した maintest.py というファイルは次の様なソースコードになっています。

```
print("maintest: ", __name__)
```

これを Pythoh プログラムとして実行すると、次の様な結果になります。

この場合、変数 __name__ には __main__ という値が格納されていることが分かります。

一方、モジュールファイルとして次のセルを実行してみて下さい。すると、__main__ ではなく、 maintest とモジュール名が表示されることを確認することが出来ます。（上手く表示されない場合は、カーネルのリスタートボタンを押してから実行してみて下さい。）

```
In [ ]: import maintest
```

5.4.13 ▲パッケージ

パッケージは Python のモジュール名を、区切り文字 . を利用して構造化する方法です。多くの関数・クラスを提供する巨大なモジュールで利用されています。

以下は、foo パッケージのサブモジュール bar.baz をインポートする例です。

```
import foo.bar.baz
```

あるいは

```
from foo import bar.baz
```

パッケージはファイルシステムのディレクトリによって構造化されています。Python はディレクトリに `__init__.py` という名前のファイルがあればディレクトリパッケージとして取り扱います。最も簡単なケースでは `__init__.py` はただの空ファイルで構いませんが、 `__init__.py` にパッケージのための初期化コードを入れることもできます。

上の例 `foo.bar.baz` は、以下のディレクトリ構造となります。

```
foo/
    __init__.py
    bar/
        __init__.py
        baz.py
```

5.4.14 練習の解答

各セルのコードを保存した .py のファイルを作成して下さい。

In [ ]:
```
#sample2.py
import sys
a1 = sys.argv[1]
print("a1 contains the value of", a1)
```

In [ ]:
```
#showname.py
import sys
print(sys.argv[0])
```

In [ ]:
```
#argsnum.py
import sys
num = len(sys.argv) - 1 # sys.argv の先頭は実行するファイルの名前であって引数ではないので、1 減らす
if num <= 3:
    print(num)
else:
    print("too many")
```

In [ ]:
```
#argssum.py
import sys
sum1 = 0
for i in range(1, len(sys.argv)):
    sum1 += int(sys.argv[i])
print(sum1)
```

5.5 正規表現

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。

```
In [ ]: import re
```

5.5.1 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。

```
re.match(正規表現 A, 文字列 B)
```

```
In [ ]: 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)
```

match では、マッチが成立している場合、**match オブジェクト**と呼ばれる特殊なデータを返します。マッチが成立しない場合、None を返します。

つまり、マッチする部分文字列を含む場合、返値は None ではないので、if 文などの条件で真とみなされます。したがって以下のようにして条件分岐することができます。

```
if re.match(正規表現, 文字列):
    ...
```

```
In [ ]: if re.match("abc", "abcde"): #マッチする
            print("正規表現 abc が文字列 abcde にマッチする")
        else:
            print("正規表現 abc が文字列 abcde にマッチしない")
        if re.match("abc", "ababc"): #マッチしない
            print("正規表現 abc が文字列 ababc にマッチする")
        else:
            print("正規表現 abc が文字列 ababc にマッチしない")
```

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。

```
In [ ]: match1 = re.match("abc", "ABCdef")
        print(match1)
        match1 = re.match("ABC", "abcdef")
        print(match1)
```

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。

```
In [ ]: 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)
```

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。

```
In [ ]: match1 = re.match("def", "abcdef")
        print(match1)
        match1 = re.match("xyz", "abcdef")
        print(match1)
```

文字列の先頭しか調べられないのでは、いかにも不便です。

そこで、関数 search は、指定した正規表現 A が文字列 B に（文字列の先頭以外でも）マッチするかどうか調べることができます。

---

re.search(正規表現 A, 文字列 B)

```
In [ ]: match1 = re.search("abc", "abcdef")
        print(match1)
        match1 = re.search("abc", "ababcd")
        print(match1)
        match1 = re.search("def", "abcdef")
```

```
        print(match1)
```

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。

```
In [ ]: 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)
```

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。

```
if re.search(正規表現, 文字列):
    ...
```

```
In [ ]: 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")
```

文字列の先頭からのマッチを調べたいときときは、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。

```
In [ ]: 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)
```

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。

```
In [ ]: 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
        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
        print(match1)
        match1 = re.search("colo(u|)r", "That colour matches your suit.") #正規表現内の（）については
        print(match1)
```

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**連接**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。

```
In [ ]: match1 = re.search("a*", "abcdef")
        print(match1)
        match1 = re.search("a*", "aabbcc")
        print(match1)
        match1 = re.search("a*", "cde")
        print(match1)
        match1 = re.search("bo*", "boooo!")
        print(match1)
```

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でしたか？　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**（くうれつ）、もしくは**空文字列**（くうもじれつ）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。

```
In [ ]: 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)
```

これまでに紹介した連接、和、閉包という 3 つの演算を組み合わせることで様々な正規表現を記述することができますが、これらの演算には結合の強さが存在します。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、連接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。

```
In [ ]: 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)
```

演算の結合の強さは、和 < 連接 < 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連接、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。

```
In [ ]: 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)
```

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。

```
In [ ]: 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)
```

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。

```
In [ ]: 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)
```

5.5.2 練習

文字列 str1 を引数として取り、 str1 の中に「月を表す文字列」が含まれているかどうか調べて、含まれていればマッチしたときの match オブジェクトを、含まれいなければ None を返す関数 check_monthstr を作成して下さい。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。

```
In [ ]: import ...
        def check_monthstr(str1):
            ...
```

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

```
In [ ]: print(check_monthstr("10").group() == '10') # group() については後半に説明があります（オプショナ
        print(check_monthstr("mon1521vb") == None)
        print(check_monthstr("00an23") == None)
        print(check_monthstr("13302").group() == '02')
```

5.5.3 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。

```
In [ ]: import ...
        def check_ACGTstr(str1):
            ...
```

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

```
In [ ]: print(check_ACGTstr("AcCGTAGCacATcGgAaaTtGCacT") == True)
        print(check_ACGTstr(":ACaacgta24FgtGH") == False)
        print(check_ACGTstr("") == False)
```

5.5.4 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。

```
In [ ]: import ...
        def check_timestr(str1):
            ...
```

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

```
In [ ]: 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
        print(check_timestr("time?1023") == None)
        print(check_timestr("time?11:23").group() == '11:23')
        print(check_timestr("12:3xx1;23ah23:23") == None)
```

5.5.5 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。

```
In [ ]: import ...
        def check_ipv4str(str1):
            ...
```

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

```
In [ ]: 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)
```

5.5.6 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、含まれいなければ 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。

```
In [ ]: import ...
        def check_monthdaystr(str1):
            ...
```

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

```
In [ ]: 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")
```

5.5.7 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、この連接・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。

5.5.7.1 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）。

```
In [ ]: 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)
```

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。

```
In [ ]: 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.",
        print(match1)
```

ただし、文字クラスの中で連接、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。

```
In [ ]: 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)
```

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。

```
In [ ]: 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)
```

5.5.7.2 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。

```
In [ ]: 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)
```

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が判定されます。例えば、[d^ef] は、d, ^. e, f のいずれかにマッチします。

```
In [ ]: 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)
```

5.5.8 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はなく、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。

5.5.8.1 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、

---

re.sub(正規表現 R，置換する文字列 B，元になる文字列 A)

R とマッチする A 中の全ての文字列を B と置き換えることができます。置き換えられた結果の文字列（新たに作られて）が返り値となります。（もちろん、もとの文字列 A は変化しません。）

```
In [ ]: 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?")
        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")
        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
```

```
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。

```
In [ ]: re.sub(r'[ \t\n][ \t\n]*', ' ', 'Hello,\n    World!\tHow 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）。

```
In [ ]: re.sub(r'<[^>]*>', '', '<body>\nClick <a href="a.href">this</a>\n</body>\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。

5.5.8.2 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。

`re.split(正規表現 R, 元になる文字列 A)`

以下が典型例です。

```
re.split(r'[^a-zA-Z][^a-zA-Z]*', 'Hello, World! How are you?')
```

`[^a-zA-Z][^a-zA-Z]*` という正規表現は、英文字以外の文字が 1 回以上繰り返されている、というパターンを表します。この正規表現を Python の式の中で用いるときは、`r'[^a-zA-Z][^a-zA-Z]*'` という構文を用います。 先頭の `r` については、以下の説明を参照してください。

```
In [ ]: 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)
```

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。

5.5.8.3 ▲ 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。

```
In [ ]: r'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。

```
In [ ]: r'[ \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。

```
In [ ]: r'\\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。

```
In [ ]:
```

In [ ]:

In [ ]:

In [ ]:

5.5.9 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。

```
In [ ]: import ...
        def get_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。

```
In [ ]: list1 = get_engsentences()
        print(len(list1) == 289)
        print(list1[0] == "a")
        print(list1[100] == "in")
        print(list1[288] == "would")
```

5.5.10 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。

```
In [ ]: import ...
        def get_engsentences2():
            ...
```

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。

```
In [ ]: list1 = get_engsentences2()
        print(len(list1) == 149)
        print(list1[0] == "a")
        print(list1[100] == "proclaim")
        print(list1[148] == "would")
```

5.5.11 ▲正規表現の応用 2

5.5.11.1 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。

```
In [ ]: 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)
```

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。

```
In [ ]: match1 = re.search("[.]", "Hello")
        print(match1)
        match1 = re.search("[.]", "3.141592")
        print(match1)
```

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。

```
In [ ]: match1 = re.search("[*]", "*|()")
        print(match1)
        match1 = re.search("[|]", "*|()")
        print(match1)
        match1 = re.search("[)]", "*|()")
        print(match1)
```

5.5.11.2 ?

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。

```
In [ ]: match1 = re.search("colou?r", "colour")
        print(match1)
        match1 = re.search("colou?r", "color")
        print(match1)
        match1 = re.search("colou?r", "color")
        print(match1)
```

5.5.11.3 +

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。

```
In [ ]: match1 = re.search("boo+", "boooo!")
        print(match1)
        match1 = re.search("boo+", "bo!")
        print(match1)
        match1 = re.search("a+", "abcdef")
        print(match1)
        match1 = re.search("a+", "aabbcc")
        print(match1)
```

```
match1 = re.search("a+", "cde")
print(match1)
match1 = re.search("[^a-zA-Z]+", "abc12345efg67hi89j0k")
print(match1)
match1 = re.search("[a-zA-Z]+", "abc12345efg67hi89j0k")
print(match1)
```

上記の例を * を使う形に書き換えてみて下さい。

5.5.11.4 {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。

```
In [ ]: match1 = re.search("bo{3,5}", "boooooo!")
        print(match1)
        match1 = re.search("bo{3,5}", "boo!")
        print(match1)
        match1 = re.search("a{2,5}", "bacaad")
        print(match1)
        match1 = re.search("[0-9]{1,3},[0-9]{3,3}", "1,298 円")
        print(match1)
        match1 = re.search("[0-9]{1,3},[0-9]{3,3}", "298 円")
        print(match1)
```

5.5.12 ▲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。

5.5.12.1 \t

\t は タブを表します。

```
In [ ]: match1 = re.search("b\t", "a        b        c        d")
        print(match1)
```

5.5.12.2 \s

\s は空白文字（スペース、タブ、改行など）を表します。

```
In [ ]: match1 = re.search("b\s", "a        b        c        d")
        print(match1)
        match1 = re.search("a\s\s\s", "a          b          c        d")
        print(match1)
        match1 = re.search("b\s*", "a          b          c        d")
        print(match1)
```

5.5.12.3 \S

\S は \s 以外の全ての文字を表します。

```
In [ ]: match1 = re.search("b\S", "a        b        bc        d")
        print(match1)
```

5.5.12.4 \w

[a-zA-Z0-9_] と同じ意味です。

```
In [ ]: match1 = re.search("\w\w", "abcde")
        print(match1)
        match1 = re.search("b\w*g", "abcdefgh")
```

5.5.12.5 \W

\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。

```
In [ ]: match1 = re.search("g\W*", "ab defg  hi jklm no p")
        print(match1)
        match1 = re.search("\W\w*\W", "ab defg  hi jklm no p")
        print(match1)
```

5.5.12.6 \d

[0-9] と同じ意味です。

```
In [ ]: 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)
```

5.5.12.7 \D

\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。

```
In [ ]: match1 = re.search("\D*", "He has 10 apples.")
        print(match1)
```

5.5.13 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。

```
In [ ]: import ...
        def sumnumbers(s):
            ...
```

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。

```
In [ ]: print(sumnumbers(" 2  33 45, 67.9") == 156)
```

5.5.14 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。

```
In [ ]: import ...
        def check_ACGTstr(str1):
            ...
```

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

```
In [ ]: print(check_ACGTstr("AcCGTAGCacATcGgAaaTtGCacT") == True)
        print(check_ACGTstr(":ACaacgta24FgtGH") == False)
        print(check_ACGTstr("") == False)
```

5.5.15 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。

```
In [ ]: import ...
        def check_postalcode(str1):
            ...
```

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

```
In [ ]: 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)
```

5.5.16 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。

```
In [ ]: import ...
        def check_extension(str1):
            ...
```

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

```
In [ ]: print(check_extension("24115") == True)
        print(check_extension("46858") == False)
        print(check_extension("^^e2^^98^^8e46666") == False)
        print(check_extension("467890") == False)
```

5.5.17 ▲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。

```
In [ ]: 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
        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
        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、この様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)
```

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。

5.5.18 ▲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。

5.5.18.1 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。

```
re.findall(正規表現 R, 文字列 A)
```

```
In [ ]: list1 = re.findall("had", "James while John had had had had had had had had had had a b
```

```
        #James, while John had had "had", had had "had had"; "had had" had had a better effect on t
        print(list1) #全ての had を抜き出す
        list1 = re.findall("p[^ .]*", "Peter Piper picked a peck of pickled peppers.", re.I)
        print(list1)# p で始まる全ての単語を取得する，大文字小文字を区別しない
```

5.5.18.2 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。

`re.finditer(正規表現 R, 文字列 A)`

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。

```
In [ ]: print("1:正規表現 had の結果：")
        iter1 = re.finditer("had", "James while John had had had had had had had had had had had a
        #James, while John had had "had", had had "had had"; "had had" had had a better effect on t
        for list1 in iter1:
            print(list1) #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 list1 in iter1:
            print(list1)# p で始まる全ての単語を取得する，大文字小文字を区別しない
```

5.5.18.3 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様になっています。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。

`match オブジェクト.group(i)`

```
In [ ]: 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
        match1 = re.search("([^@]*)@[^.]*(\.[^.]*)?\.u-tokyo\.ac\.jp", 'accountname@test.ecc.u-toky
        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
        match1 = re.search("([^@]*)@[^.]*(\.[^.]*)?\.u-tokyo\.ac\.jp", 'accountname@test.u-tokyo.ac
        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
        match1 = re.search("href=\"([^\"]*)\"", '<a href="http://www.u-tokyo.ac.jp" target="_blank"
        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
```

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。

```
In [ ]: 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))
```

例えば、if 文でエラーを回避します。

```
In [ ]: 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")
```

5.5.19 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。

```
In [ ]: import ...
        def check_ACGTstr(str1):
            ...
```

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

```
In [ ]: print(check_ACGTstr("AcCGTAGCacATcGgAaaTtGCacT") == True)
        print(check_ACGTstr(":ACaacgta24FgtGH") == False)
        print(check_ACGTstr("") == False)
```

5.5.20 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。

```
In [ ]: import ...
        def get_pos():
            ...
```

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。

```
In [ ]: print(get_pos()['They '] == 'PRON')
        print(get_pos()['know '] == 'VERB')
```

5.5.21 練習の解答

```
In [ ]: 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
```

```
In [ ]: import re
        def check_timestr(str1):
            reg_hour = "((0(0|1|2|3|4|5|6|7|8|9))|10|11)" #「時」部分の正規表現
            #reg_hour = "((0[0-9]|10|11)" # 文字クラスを使ってと表すことも出来ます（文字クラスは後で学習し
            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
```

```
In [ ]: 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
            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
            #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
            #print(reg_000_255)
            reg_ip = reg_000_255 + ":" + reg_000_255 + ":" + reg_000_255 + ":" + reg_000_255 # aaa
            #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
```

```
In [ ]: 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 21 から 29 になる場合
            reg_day_01to29 = reg_day_01to09 + "|" + reg_day_10to19 + "|" + reg_day_20to29 # dd が 01
```

```
        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
        reg_monthday_30 = reg_month_30 + "/(" + reg_day_01to30 + ")" # mm と dd を組み合わせる（01
        reg_monthday_29 = "02/(" + reg_day_01to29 + ")" # mm と dd を組み合わせる（01-29 の場合は mm
        # 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
```

```
In [ ]: import re
        def sumnumbers(s):
            numbers = re.split("[^0-9]+", s)
            numbers.remove('')
            n = 0
            for number in numbers:
                n += int(number)
            return n
```

```
In [ ]: 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
            #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
```

```
In [ ]: 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
```

```
            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
                #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
```

In [ ]:
```
import re
def check_ACGTstr(str1):
    reg_ACGT = "(A|C|G|T)+" # A,C,G,T を表す正規表現  # + →* だと空文字列がマッチしてしまう
    #reg_ACGT = "(A|C|G|T)(A|C|G|T)*" # A,C,G,T を表す正規表現
    #reg_ACGT = "[ACGT]+" # A,C,G,T を表す正規表現
    match1 = re.search(reg_ACGT, str1, re.I) # re.I を入れて、大文字と小文字を区別しない
    if match1 != None and str1 == match1.group(): # str1 全体とマッチした文字列が等しいかチェッ
        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
```

In [ ]:
```
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
```

In [ ]:
```
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
```

In [ ]:
```
import re
def get_pos():
    str_file = "B1S.xml"
    with open(str_file, "r", encoding="utf-8") as f:
        str_script = f.read() # ファイルの中身を 1 つの文字列に格納する
    #print(str_script)
    itr1 = re.finditer("<w[^>]*pos=\"([^>\"]*)\"[^>]*>([^<]*)</w>", str_script) # 正規表現を
    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
```

第 6 回

6.1 関数プログラミング

関数プログラミングとは、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。

6.1.1 高階関数

高階関数（こうかいかんすう）（**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。

```
In [ ]: def inc(x):
            return x+1

        def twice(f, x):
            return f(f(x))

        def genfunc():
            return inc

        twice(genfunc(), 0)
```

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ことがよくあります。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。

6.1.1.1 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。

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

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。

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

■6.1.1.1.1 **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。

6.1.1.2 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。

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

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

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。

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

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

■6.1.1.2.1 **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。

6.1.1.3 ▲ 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。

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

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 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。

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

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

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。

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

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

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。

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

ただし、このように複雑になってくると、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。

6.1.2 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、コレクション（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。

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

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

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

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。

イテレータも for 文で反復処理できます。

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

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、リストやタプルを含めたコレクションは、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。

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

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。同じコレクション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。

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

```
In [ ]: 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コレクション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、通常 iter() や next() はコレクションのサイズ（要素数）に依存しないコストで実装されます。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する方が効率的です。これは、サイズが小さいコレクション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。

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。

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

イテレータの定義は、6-3 で改めて説明します。

6.1.2.1 練習

与えられたコレクション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。

6.1.3 イテレータを生成する関数

Python の組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。

6.1.3.1 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。

```
In [ ]: it = enumerate('ACDB')
        print(it) # リストやタプルではない

In [ ]: print(*it)
```

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、コレクションだけでなく、イテレータも渡せます。つまり、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。

```
In [ ]: print(*enumerate('ACDB',1))
```

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。

6.1.3.2 map

組込み関数の map() は、第 1 引数に取った関数を、第 2 引数に取ったコレクションやイテレータ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。

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

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 個のコレクションやイテレータ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。

```
In [ ]: # 異なるコレクション/イテレータを受け取れる
        print(*map(lambda x,y: x + y, [1,-3,2,0], (4,7,-6,5)))

In [ ]: # 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)))
```

map() とラムダ式を組み合わるよりも、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。

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

In [ ]: 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）
```

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。

■6.1.3.2.1 **練習** 第 1 引数で与えられた要素数まで、第 2 引数に与えられたコレクション/イテレータ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、A C を走査するイテレータを返す。

6.1.3.3 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。

```
In [ ]: print(*zip(range(1,10,2), range(1000000)))
```

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。

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

```
In [ ]: 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
```

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

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。

■6.1.3.3.1 練習 コレクションを取って、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。

6.1.3.4 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 引数にコレクションもしくはイテレータを取り、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。

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

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。

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

```
In [ ]: 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）
```

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。

■6.1.3.4.1 練習 文字列には、文字列のコレクション/イテレータを取る join() メソッドがある。s.join(ss) としたとき、ss の各要素の文字列を、s を間に挟んで連結した結果の文字列を返す。

```
In [ ]: ','.join(['A','B','C','D'])
```

```
In [ ]: ','.join(iter('ABCD'))
```

この join() メソッドと filter() を用いて、与えられた文字列の全ての改行'\n' と空白' ' を除去した文字列を返す関数 condense() を、for 文を使わずに定義せよ。例えば condence('The Zen of Python\n') は、'TheZenofPython' を返す。

6.1.3.5 takewhile, dropwhile

itertools モジュール内の関数 takewhile() は、filter() と同様の引数を取り、単項述語が偽を返すまで走査するイテレータを返します。

```
In [ ]: import itertools
        print(*itertools.takewhile(lambda x: x != 4, range(8)))
```

同じく itertools モジュール内の関数 dropwhile() は、同様の引数を取り、takewhile() の残りを走査するイテレータを返します。

```
In [ ]: print(*itertools.dropwhile(lambda x: x != 4, range(8)))
```

takewhile() と dropwhile() は、制御構造の観点で見ると、break 文による途中脱出を含んだ for 文に対応します。つまり、内部的には反復を途中で打ち切って結果を返しています。これは、第 2 引数を走査しきる filter() と異なる点であり、実行効率に顕著に影響します。

```
In [ ]: print(*filter(lambda x: x < 4, range(10000000)))
```

```
In [ ]: print(*itertools.takewhile(lambda x: x < 4, range(10000000)))
```

同じ結果を返していますが、filter() よりも takewhile() が速いことが実感できるでしょう。大量の（もしくは際限なく）要素を生成するイテレータを使うときには、takewhile() と dropwhile() は、特に役立ちます。

■6.1.3.5.1 **練習** 第 1 引数に整数 k、第 2 引数に整列可能なコレクション/イテレータを取り、その中の上位 k 位までを走査するイテレータを返す関数 topk() を、for 文を使わずに takewhile() を用いて定義せよ。ただし、同率順位を考慮するものとする。例えば、topk(3, [1,4,3,2,3,4]) は、4 4 3 3 を走査するイテレータを返すことになる。

6.1.3.6 ▲ accumulate

itertools モジュール内の関数 accumulate() は、sum() の途中結果をイテレータで返す関数です。

```
In [ ]: import itertools
        print(*itertools.accumulate([1]*8))
```

```
In [ ]: print(*itertools.accumulate(range(8)))
```

func 引数に 2 引数関数を渡すことができ、この場合は reduce() の途中結果をイテレータで返すことに相当します。

```
In [ ]: print(*itertools.accumulate([1,-5,2,-6,0,3,7], func=max))
```

6.1.3.7 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。

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

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

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

reversed() は、コレクション一般やイテレータを取れないことに留意してください。

■6.1.3.7.1 **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。

6.1.3.8 ▲ product

itertools モジュール内の関数 product() は、任意個のコレクション/イテレータを取って、それらの直積を取った結果を走査するイテレータを返します。

```
In [ ]: import itertools
        print(*itertools.product('ABCD', range(2)))
```

```
In [ ]: print(*itertools.product('AB', range(2), 'CD'))
```

制御構造の観点で見ると、for 文のネストや、for 句が連なった内包表記に対応します。それらを、ネストしない 1 つの for 文や、for 句 1 つの内包表記で記述するときに役立ちます。

6.1.4 ▲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。

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

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

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

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。

```
In [ ]: 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()
```

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。

6.1.5 練習の解答

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

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

```
In [ ]: 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')
```

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

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

```
In [ ]: 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]))
```

```
In [ ]: def condence(ss):
            return ''.join(filter(lambda s: s not in ('\n',' '), ss))

        condence('The Zen of Python\n')
```

```
In [ ]: def topk(k, xs):
            xs = sorted(xs, reverse=True)
            return itertools.takewhile(lambda x: x >= xs[k-1], xs)

        print(*topk(3, [1,4,3,2,3,4]))
```

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

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

6.2 オブジェクト指向プログラミング

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。

6.2.1 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。

```
In [ ]: 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。

```
In [ ]: 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。

```
In [ ]: 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

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

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っていないと定義できない。

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るとしても、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、学生と教員のバッチ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。

```
In [ ]: 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)])
```

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。

```
In [ ]: 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, indu
```

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。

6.2.2 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。

```
In [ ]: 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)])
```

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、次のようにクラス定義は、上の例と同等です。

```
In [ ]: class Student(tuple):
            badge = studentbadge

        class Faculty(tuple):
            badge = facultybadge

        class Industrial(tuple):
            badge = industrialbadge

        badgelist([Student(taro), Student(jiro), Faculty(hagiya), Industrial(iwata)])
```

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。

6.2.2.1 練習

Student・Faculty・Industrial の badge() メソッドを、__str__() メソッドに名前替えする前と後で、各参加者を print せよ。

6.2.3 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。

```
In [ ]: 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
```

```
In [ ]: c['C']
```

__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。

```
In [ ]: class Counter(dict):
            def __missing__(self, k):
                return 0


        c = Counter()
        c['A'] += 1
        c['B'] += 1
        c
```

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 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。

6.2.3.1 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。

```
In [ ]: c = Counter()
        c['A'] += 1
        c['B'] += 1
        c['A'] -= 1
        c
```

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。

6.2.4 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。クラスという言語機能は、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、組込み関数 type() で取得できます。

```
In [ ]: type('hello')
```

```
In [ ]: type(0)
```

```
In [ ]: type((1,))
```

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。

```
In [ ]: tuple([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込み関数 isinstance() が提供されています。

```
In [ ]: (isinstance(1,int),
         isinstance(1,tuple),
         isinstance([1,'a'],list),
```

```
        isinstance([1,'a'],tuple))
```

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。

6.2.5 ▲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 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。

```
In [ ]: 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 Stud
        taro = Student('東大太郎', 1234567890)
        taro
```

```
In [ ]: Student(id=1234567890, name='東大太郎') # キーワード引数でも構築できる
```

```
In [ ]: taro.name # 名前によるアクセス（属性アクセス）
```

```
In [ ]: taro[0] # インデックスアクセス
```

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。

6.2.5.1 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。

```
In [ ]: x, y = taro
        print(x, y)
```

```
In [ ]: list(taro)
```

変更不可能オブジェクトなので、辞書のキーにできます。

```
In [ ]: d = {}
        d[taro] = 0
```

```
        d
```

属性の再定義はできませんが、

```
In [ ]: taro.name = '京大太郎'
```

_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。

```
In [ ]: taro._replace(name='京大太郎')
```

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。

```
In [ ]: class Student(namedtuple('Student', ('name', 'id'))):
            def badge(self):
                return self.name

        taro = Student('東大太郎', 1234567890)
        taro
```

```
In [ ]: taro.badge()
```

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。

6.2.6 練習の解答

```
In [ ]: 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)
```

```
In [ ]: 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
```

```
In [ ]:
```

6.3 ▲イテレータとイテラブル

6-1 で利用法を説明したイテレータや、暗黙的に利用してきたイテラブルについて、より詳しく説明します。

6.3.1 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。

```
In [ ]: it = (x * 3 for x in 'abc')
        for x in it:
            print(x)
```

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。

```
In [ ]: def gen():
            for x in 'abc':
                yield x * 3
        it = gen()
        for x in it:
            print(x)
```

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。

```
In [ ]: 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)
```

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。

6.3.2 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。

```
In [ ]: it = iter('abc')
        print(next(it))
        print(next(it))
        print(next(it))
```

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。

```
In [ ]: it = iter('abc')
        it2 = iter(it)
        it is it2
```

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)
        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。

6.3.3 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。

```
In [ ]: 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)
```

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

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。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、実際上十分です。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。

```
In [ ]: class CanaryList(list):
            def __iter__(self):
                for x in super().__iter__():
                    print(x)
                    yield x
```

```
for x in CanaryList([1,2,3]):
    pass
```

6.3.4 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		__contains__()	in 演算子が適用できる
Sized		__len__()	組込み関数 len() が適用できる
Iterable		__iter__()	組込み関数 iter() が適用できる
Iterator	Iterable	__next__()	組込み関数 next() が適用できる
Reversible	Iterable	__reversed__()	組込み関数 reverse() が適用できる
Collection	Container Sized Iterable		
Sequence	Collection Reversible	__getitem__()	項目アクセス演算（x[k]）が適用できる
Mapping	Collection	__getitem__()	項目アクセス演算（x[k]）が適用できる

ここの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を参照してください。

第 7 回

7.1 pandas ライブラリ

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 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。

```
In [ ]: import pandas as pd
        import numpy as np
```

7.1.1 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。

7.1.2 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。

```
In [ ]: # リストからシリーズの作成
        s1 = pd.Series([0,1,2])
        print(s1)

        # 配列からシリーズの作成
        s2 = pd.Series(np.random.rand(3))
        print(s2)

        # 辞書からシリーズの作成
        s3 = pd.Series({0:'boo',1:'foo',2:'woo'})
        print(s3)
```

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。

7.1.3 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。

```
In [ ]: # 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','
        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',
        print(d3)
```

7.1.4 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。

```
In [ ]: # 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)
```

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。

```
In [ ]: 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
```

7.1.5 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。

```
In [ ]: # データフレームの先頭 5 行のデータ
        iris_d[:5]
```

```
In [ ]: # データフレームの終端 5 行のデータ
        iris_d[-5:]
```

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。

```
In [ ]: # 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)
```

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。

```
In [ ]: # 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)
```

7.1.5.1 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。

```
In [ ]: # データフレームの 2 行のデータ
        iris_d.iloc[1]
```

```
In [ ]: # データフレームの 2 行,2 列目のデータ
        iris_d.iloc[1, 1]
```

```
In [ ]: # 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]
```

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。

```
In [ ]: # データフレームの行インデックス 5 のデータ
        iris_d.loc[5]
```

```
In [ ]: # 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']
```

```
In [ ]: # 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']]
```

7.1.6 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。

```
In [ ]: # 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)]
```

7.1.7 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。

```
In [ ]: # 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)
```

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。

```
In [ ]: # 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)
```

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。

```
In [ ]: # 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)
```

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。

```
In [ ]: # 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)
```

7.1.8 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。

```
In [ ]: # 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:]
```

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。

```
In [ ]: # 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:]
```

7.1.9 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。

```
In [ ]: # 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_wid
        sorted_iris.head(10)
```

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。

```
In [ ]: # 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_wid
        sorted_iris.head(10)
```

7.1.10 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

```
In [ ]: # 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()
```

7.1.11 ▲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。

```
In [ ]: # 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
```

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。

```
In [ ]: # 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)
```

7.1.12 ▲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。

```
In [ ]: # '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], ['spe
        # '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], ['spec

        # 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
```

7.1.13 ▲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。

```
In [ ]: # 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')
```

```
In [ ]: # グループごとの先頭 5 行を表示
        iris_d.groupby('species').head(5)
```

```
In [ ]: # 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()
```

7.1.14 ▲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。

7.2.1 機械学習について

機械学習では、観察されたデータをよく表すようにモデルのパラメータの調整を行います。パラメータを調整することでモデルをデータに適合させるので、「学習」と呼ばれます。学習されたモデルを使って、新たに観測されたデータに対して予測を行うことが可能になります。

7.2.2 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。

7.2.3 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。

7.2.4 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。

```
In [ ]: import pandas as pd
        iris = pd.read_csv('iris.csv')
        iris.head(5)
```

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。

```
In [ ]: from sklearn.datasets import load_iris
        iris = load_iris()
        X_iris = iris.data
        y_iris = iris.target
```

7.2.5 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

7.2.6 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_irist）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なっています。97% の精度で花の 4 つの特徴から 3 つの花の種類を分類できていることがわかります。

```
In [ ]: 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_s

        model=LogisticRegression() # ロジスティック回帰モデル
        model.fit(X_train, y_train) # モデルを学習データに適合
```

```
y_predicted=model.predict(X_test) # テストデータでラベルを予測
accuracy_score(y_test, y_predicted) # 予測精度の評価
```

7.2.7 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。

```
In [ ]: 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。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。

```
In [ ]: import numpy as np
        import matplotlib.pyplot as plt
        %matplotlib inline

        w2 = model.coef_[0,1]
        w1 = model.coef_[0,0]
        w0 = model.intercept_[0]

        line=np.linspace(3,7)
        plt.plot(line, -(w1*line+w0)/w2)
        y_c = (y_iris=='versicolor').astype(np.int)
        plt.scatter(iris2['petal_length'],iris2['petal_width'],c=y_c);
```

7.2.8 教師あり学習・回帰の例

以下では、アイリスデータセットを用いて花の特徴の 1 つ, petal_length, からもう一つの特徴, 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。

```
In [ ]: 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なっています。

```
In [ ]: 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) # 予測精度の評価
```

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。

```
In [ ]: 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);
```

7.2.9 教師なし学習・クラスタリングの例

以下では、アイリスデータセットを用いて花の 2 つの特徴量, petal_lengh 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_irist）を用意し、引数 n_clusters にハイパーパラメータとしてクラスタ数、ここでは 3、を指定して KMeans クラスのインスタンスを作成しています。そして、fit() メソッドによりモデルをデータに適合させ、predict() メソッドを用いて各データが所属するクラスタの情報（y_km）を取得しています。

学習された各花データのクラスタ情報を元のデータセットのデータフレームに列として追加し、クラスタごとに異なる色でデータセットを可視化しています。2 つの特徴量, petal_lengh と petal_width, に基づき、3 のクラスタが得られていることがわかります。

```
In [ ]: 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 つクラスタと 3 つの花の種類の分布を 2 つの特徴量, petal_lengh と petal_width, の空間で比較してみると、クラスタと花の種類には対応があり、2 つの特徴量から花の種類をクラスタとしてグループ分けできていることがわかります。可視化には seaborn モジュールを用いています。

```
In [ ]: 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);
```

7.2.10 練習

アイリスデータセットの 2 つの特徴量、sepal_length と sepal_width、を元に、KMeans モデルを用いて花のデータをクラスタリングしてください。クラスタの数は任意に設定してください。

```
In [ ]: from sklearn.cluster import KMeans

        iris = pd.read_csv('iris.csv')
        X_iris=iris[['sepal_length', 'sepal_width']].values

        ### your code here
```

7.2.11 教師なし学習・次元削減の例

以下では、アイリスデータセットを用いて花の 4 つの特徴量を元に花のデータを**次元削減**する手続きを示しています。ここでは次元削減を行うモデルの一つである PCA クラスをインポートしています。

特徴量データ（X_irist）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。

```
In [ ]: 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
```

```
In [ ]: 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);
```

索引