Python プログラミング入門
(学内限定)

2018 年 9 月 22 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、行の終わりまでが**コメント**になります。コメントは空白と同等に扱われます。

```
In [ ]: 7%2  # 7 を 2 で割った余りが返ります。
```

```
In [ ]: 7**2  # 49 になるはずです。
```

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。

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。すると、h という式を評価することができます。

In [ ]: h

さらに

In [ ]: w = 104.0

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

h を身長、w を体重と考えると、上の式によって BMI（ボディマス指数）を求めることができます。
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。
実は、値に名前を付ける、という説明は正しくありません。
変数とは、値をしまっておく箱のようなものと考えた方がよいです。
なぜなら、次のように、変数の値を更新することができるからです。

In [ ]: w = 104.0-10

この後で、前と同じ式を評価してみましょう。 すなわち、もう一度 BMI を求めてみましょう。

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

1.2.1.1 代入

= という演算子は、その左の変数に、その右の式の値をしまう、ということを意味します。 この操作は、**代入**と呼ばれています。
= の右の式の中に、= の左の変数が出て来てもかまいません。

In [ ]: w = w-10

上の操作は、w の値を 10 減らす、という意味を持ちます。
もう一度 BMI を求めてみましょう。

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

1.2.1.2 代入演算子

上の例のように変数の値を減らすには、-= という演算子を用いることができます。これは、- と = を結合させた演算子です。

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

同様に、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

次に、身長と体重をもらって、BMI を求める**関数**を定義してみましょう。

関数定義など、複数行の Code 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、関数が受け取る入力をしまう変数のことです。**仮引数**（かりひきすう）ともいいます。

def の行の後に、

```
return 式
```

という構文が続きます。この構文は return で始まり、return 文と呼ばれます。

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。

この関数を、入力とともに呼び出すと、 return の後の式が計算されて、関数がその値を返します（これを**返値**（かえりち）と言います）。

次の式を評価してみましょう。

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

関数を呼び出す式は、より大きな式の一部とすることもできます。

```
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、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 に出会わずにで関数定義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。

1.2.3 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、三辺の長さを足して 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有効な変数となります。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 によって関数の値となります。
なお、s 代入の行も return と同様にインデントされていることに注意してください。
Python では、関数の中で代入などにより値が設定された変数は、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。
関数を呼び出した後で、s の値を参照すると、

```
In [ ]: s
```

というように、エラーになってしまいます。

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

```
In [ ]: s
```

上の例で、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。

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、その値を関数の外から確認することはできません。
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ためには、print で始まる print 命令（print 文）を用いることができます。（実は Python3 では print は関数なので、これは関数呼び出しに他なりませんが、その重要さから print 命令や print 文という言い方をすることにします。）

```
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)
```

print 命令は、プログラムの誤り（**バグ**）を見つけて、プログラムを修正（**デバッグ**）する最も基本的な方法です。

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。
print で値を表示しても、関数の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 命令は None という値を返しますので、これでは関数は常に None という値が返してしまいます。

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)`（det を使って定義してください。）
3. 解のうち、小さくない方を求める `solution2(a,b,c)`

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 [ ]:

In [ ]:

In [ ]:

```
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 ▲ローカル変数とグローバル変数

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 真偽値を返す関数

ここで、真偽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。
真偽値とは、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て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、真偽を返す関数を定義することができます。

```
In [ ]: def 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偽値を返します。

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偽値を return によって返しています。

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

```
In [ ]: 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 odd(x):
            if 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 [ ]: odd(2)
```

```
In [ ]: 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 は偽、その他は真を表します。

```
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 文を用いたデバッグについて紹介します。

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 文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 文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 文によるデバッグ

print 文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 文を入れて、`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 文は削除してもよいのですが、今後のために# を付けてコメントにして残しておきます。# を頭に付けると、その行は空白行と同じになります。

```
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)
```

```
In [ ]:
```

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。
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 [ ]: str1 = "hello"
        str1
```

この様に作成したデータが確かに文字列（`str`）であることを、次の様なおまじない（関数）によって確かめることができます。

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

Python が最初から用意してくれている関数（その様な関数を組み込み関数と言います）`str` を用いて、任意のデータから文字列を作成することができます。

```
In [ ]: str2 = str(123)
        str2
```

文字列の長さは、組み込み関数の `len` を用いて次の様に求めることができます。

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

2.1.1 文字列とインデックス

次の様にして、文字列を構成する個々の文字（要素）を取得することができます。例えば、3 番目の要素を取得するには、以下の様にします。

文字列 [2]

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

```
In [ ]: 'Thanks'[4]
```

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、この様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。

```
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。

```
In [ ]: str1[10]
```

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取得できます。例えば、文字列の一番後ろ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）を取得することができます。
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表示したい場合は次の様にします。

```
In [ ]: str1[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、通常次の様に記述します。

```
In [ ]: str1[0:3]
```

しかし、0 の値は省略しても同じ結果を得ることができます。

```
In [ ]: str1[:3]
```

それと同様に、最後尾の文字のインデックスを指定する場合もその値を省略することができます。

```
In [ ]: str1[3:]
```

```
In [ ]: str1[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 [ ]: str1[-4:-1]
```

実は 3 番目の値を指定することで、とびとびの部分文字列を取得することができます。例えば、`str1[3:10:2]` と指定すると、$3, 5, 7, \ldots$ 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 [ ]: str1 = "0123456789"
        print(str1[3:10:2])
        str1 = "abcdefghijklmn"
        print(str1[3:10:2])
```

つまり、 `str1[1:4]` と、 `str1[1:4:1]` は同じ文字列を得ることができるのです。

```
In [ ]: str1 = "0123456789"
        print(str1[1:4], str1[1:4:1])
```

その 3 番目の値に -1 を指定することもできます。この様にすると元の文字列の逆向きの文字列を容易に取得でき

ます。

```
In [ ]: str1 = "0123456789"
        print(str1[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、次の様に使用します。

```
str1 = ''
```

空文字列は、例えば、文字列中からある部分文字列を取り除く場合に使用します。

```
In [ ]: str1 = "2,980 円"
        str2 = str1.replace(",", "")
        print(str2)
```

文字列 str1 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、つまり、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）を指定した場合、空文字列を得ることができます（エラーが出たり、None にはならないことに注意して下さい）。

```
In [ ]: str1 = "abcdefgh"
        print("空文字列 1 = ", str1[4:2])
        print("空文字列 2 = ", str1[-1:-4])
        print("空文字列 3 = ", str1[3:3])
        print("空文字列ではない = ", str1[3:-1])
```

2.1.4 文字列の検索

文字列 A が特定の文字列 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、次の様に使用します。

```
文字列 B in 文字列 A
```

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

```
In [ ]: 'lo' in str1
```

```
In [ ]: 'a' in str1
```

2.1.5 エスケープシーケンス

文字列にシングルクォートを含めるには、**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。

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

具体的には、文字列中に \' と記述すると、' が表示されます。

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

実は、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 [ ]: str1 = 'hello'
        str2 = ' world'
        str3 = str1 + str2
        print("連結結果 =", str3)
        print("連結元 1 =", str1, ", 連結元 2 =", str2)
```

+ 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 [ ]: str1 = 'hello'
        str1 * 3
```

2.1.7 文字列とメソッド

文字列に対しては、様々なメソッド（関数の様なもの）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，...）

例えば、次の様なメソッドがあります。

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 で置き換えます。ただし、元の文字列は変化しません。具体的には、次の様に使用します。

文字列.replace(部分文字列 A，文字列 B)

```
In [ ]: str1 = 'hello'
        str2 = str1.replace('l', '123')
        print("元の文字列 = ", str1, " replace 後の文字列 = ", str2)
```

2.1.7.2 分割

split は、指定した文字列 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、次の様に使用します。

文字列 A.split(文字列 B)

リストについてはこの後で勉強します。

```
In [ ]: str1 = "hello:world:!"
        str1.split(':')
```

2.1.7.3 検索

index により、指定した部分文字列が文字列内のどこに存在するか調べることができます。具体的には、次の様に使用します。

文字列.index(部分文字列)

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、エラーが出ます。

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

```
str1.index('lo')
```

```
In [ ]: str1.index('l')
```

```
In [ ]: str1.index('a')
```

別のメソッド find でも指定した部分文字列が文字列内のどこに存在するか調べることができます。

文字列.find(部分文字列)

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内に含まれない場合には-1 が返されます。

```
In [ ]: str1.find('lo')
```

```
In [ ]: str1.find('l')
```

```
In [ ]: str1.find('a')
```

2.1.7.4 数え上げ

count により、指定した部分文字列が文字列内にいくつ存在するか調べることができます。

文字列.count(部分文字列)

```
In [ ]: str1.count('l')
```

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

2.1.7.5 大文字・小文字

メソッド 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("lower 実行元 = ", str1, ", lower 実行結果=",  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 変数と文字列に関する誤解

変数を文字列と勘違いする例が偶に見られます。例えば、文字列を引数に取る次の様な関数 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 を呼び出してみましょう。 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))
```

良いでしょうか？ このとき、決して以下の例の様には func を呼び出してはいけません。この例では変数 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 を呼び出してしまっています。

```
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。）

以下のセルの ... 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。

以下のセルの ... 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、次の様な文字列 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 は T に、T は A に、G は C に、C は G に置き換えて作成します。

以下のセルの ... 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。

以下のセルの ... 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。

以下のセルの ... 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。

以下のセルの ... 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")
```

In [ ]:

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でくくります。以下は、数を構成要素とするリストです。

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

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

以下は、文字列を構成要素とするリストです。

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

複数の種類のデータを含むことも可能です。

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

次の様にして、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は、後述する「リストの操作（2）」の「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。

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。

```
In [ ]: lns = [[1, 2, 3], [10, 20, 30, 40], ['a', 'b']]
        print(lns[1][2])
        print(lns[2][0])
```

勿論、その様なリストを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以下の関数などを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 sort

このメソッド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 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**であるといいます。一方、sorted 関数は破壊的ではありません。

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

リスト内の要素を削除する演算子です。具体的には、以下のようにインデックスで指定した要素を削除します。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。複製を作っておくと、複製元のリストが破壊されたとしても、複製したリストは影響を受けません（破壊されません）。

```
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。タプルは、文字列と同じように一度設定した要素を変更できません。加えて、タプルには適用可能なメソッドがありません。

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` は、タプルをリストに変換することができます。

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

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 for 文による繰り返し

同じ（様な）文を繰り返し実行したい場合には **for** を用います。次の様な使い方が最も基本的な使い方です。

```
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!")
```

実行分 の前にスペースがないとエラーが出ます。

```
In [1]: for value in range(5):
        print("Hi!")


          File "<ipython-input-1-e08541670a1c>", line 2
        print("Hi!")
            ^
    IndentationError: expected an indented block
```

エラーが出れば意図した通りにプログラムが組めていないのにすぐ気が付きますが、エラーが出ないために意図したプログラムが組めていないことに気が付かないことがあります。例えば、次の様な内容を実行しようとしていたとします。

```
In [3]: for value in range(3):
            print("ここは for の繰り返しの中です")
            print("ここも for の繰り返しの中です")
```

```
ここは for の繰り返しの中です
ここも for の繰り返しの中です
ここは for の繰り返しの中です
ここも for の繰り返しの中です
ここは for の繰り返しの中です
ここも for の繰り返しの中です
```

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 [6]: for value in range(3):
            print("ここは for の繰り返しの中です")
        print("ここも for の繰り返しの中です") #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
```

```
ここは for の繰り返しの中です
ここは for の繰り返しの中です
ここは for の繰り返しの中です
ここも for の繰り返しの中です
```

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)
```

この value の値を用いることでリストの要素を順番に用いることも出来ます。

```
In [ ]: ln = ['e', 'd', 'a', 'c', 'f', 'b']
        for value in range(6):
            print(ln[value])
```

この様にリストと組み合わせて使う場合、リストの大きさを上述の関数 len で取得して用いることがよくあります。

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

2.2.10 練習

整数の要素からなるリスト ln を引数として取り、ln の要素の総和を返す関数 sum_list を作成して下さい。
以下のセルの ... のところを書き換えて sum_list(ln) を作成して下さい。(練習の正解はノート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)
```

2.2.10.1 range

in の後ろにある range は実は組み込み関数です。1 つの整数を引数として与えていましたが、次の様に 2 つの引数を与えることも出来ます。

```
range(a, b)
```

---

2 つの引数を取った場合にどうなるか確認してみましょう。

```
In [ ]: ln = ['a', 'b', 'c', 'd', 'e', 'f']
        for value in range(2, 5):
            print("value の値 = ", value, ", ln の要素 = ", ln[value])
```

分かりましたか？ range(x, y) とすると、 x から y-1 までの値が value に代入されることになります。つまり、1番目の引数を 0 に指定した range(0, y) と range(y) は全く同じ役割を果たします。

range は実は 3 つ目の引数を取ることも出来ます。試してみましょう。

```
In [ ]: ln = ['a', 'b', 'c', 'd', 'e', 'f']
        for value in range(0, 6, 2):
            print("value の値 = ", value, ", ln の要素 = ", ln[value])
```

これは分かり難いかも知れません。 range(x, y, z) とすると、value には x, x+z, x+2z, ... という値が順に代入されます。ただし、 y-1 より大きい値は代入されません。つまり、3 番目の引数を 1 に指定した range(x, y, 1) は range(x, y) と同じ機能を持ちます。（これらの仕組みは文字列のスライスの引数の仕組みに類似しています）

この 3 番目の引数は、例えば逆順にリストの値を用いるときに効果を発揮します。

```
In [ ]: ln = ['a', 'b', 'c', 'd', 'e', 'f']
        for value in range(5, -1, -1):
            print("value の値 = ", value, ", ln の要素 = ", ln[value])
```

なお、for 文については第 3 回に、より詳しく学習します。

2.2.11 練習

整数の要素からなるリスト ln を引数として取り、ln 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])
```

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 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))
```

2.2.13 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])
```

2.2.14 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,
```

2.2.15 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'])
```

2.2.16 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 中に含まれる単語数を返す関数 count_words 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_words(str_engsentences) を作成して下さい。

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

2.2.17 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',
```

2.2.18 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])
```

2.2.19 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)
```

2.2.20 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'])
```

2.2.21 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")
```

2.2.22 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")
```

2.2.23 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')
```

2.2.24 練習の解答

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

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

```
In [ ]: def remove_evenindex(ln):
            ln2 = []
            for j in range(1, len(ln), 2):
                ln2.append(ln[j])
            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 j in range(4):
                int_bpcnt = str_atgc.count(lst_bpname[j])
                list_count.append([int_bpcnt, lst_bpname[j]])
            return list_count
        #atgc_countlist("AAGCCCCATGGTAA")
```

```
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 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 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 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")
```

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 [ ]:

2.3 辞書

辞書は、「キー」と「値」とを対応させるデータです。キーとしてはリストと辞書自身以外のほとんどのオブジェクト（数値、文字列、タプルなどのことをいいます。オブジェクトについては後で勉強します）を指定できます。値としてはあらゆる種類のデータを指定できます。

例えば、文字列 apple をキーとし、それに数 3 を対応付ける。更に、文字列 pen をキーとし、それに数 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)
```

辞書に登録されているキーに対する値を取り出すには、

辞書 [キー]

とします。

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

キーが辞書に登録されていない場合、エラーになります。

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

キーに対する値を変更したり、新たな要素を値として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
```

関数 len によって、辞書に登録されているキーの数を得ることが出来ます。

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

演算子 del によって、登録されているキー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、次の様に空の辞書を作ることもできます。

```
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

指定したキーを辞書から削除し、削除された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。

```
In [1]: 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"])
```

```
キー apple に対応する値 =  3
キー orange に対応する値 =  None
```

```
---------------------------------------------------------------------------

KeyError                                  Traceback (most recent call last)

<ipython-input-1-7c06a6ff86e4> in <module>()
  2 print("キー apple に対応する値 = ", dic1.get("apple"))
  3 print("キー orange に対応する値 = ", dic1.get("orange"))
----> 4 print("キー orange に対応する値 = ", dic1["orange"])


KeyError: 'orange'
```

また、get に 2 番目の引数を与えると、その値を「指定したキーが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リストとして返します。ただし、普通のリストではなくて、dict_keys という特殊なリストです。 dict_keys は、同じ第 2 回で述べた for ループなどを使って活用できます（詳しくは第 3 回に説明します）。

```
In [ ]: dic1 = {'apple' : 3, 'pen' : 5, 'orange':7}
        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特殊なリスト (dict_values) で返します。 dict_values も for ループなどで活用します。

```
In [ ]: 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特殊なリスト（dict_items）で返します。dict_items も for ループなどで活用します。

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

2.3.1.8 copy

辞書の複製を行います。リストの場合と同様に、複製元の辞書が破壊されても複製した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)
```

2.3.2 辞書とリスト

一番最初に述べた様に、リストをキーに対応する値とする辞書を作成可能です。

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

一方、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"])
```

2.3.3 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})
```

2.3.4 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: [
```

2.3.5 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
```

2.3.6 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
```

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
```

```
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"],
```

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})
```

第 3 回

3.1 条件分岐

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 行目までつづきます。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 文ではインデントが戻されています。 5 行目から再びはじ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。

条件が複雑になってくると if 文のブロックにさらに if 文を記述する、条件式を**入れ子（ネスト）**とすることがあります。この場合は、インデントはさらに深くなります。

そして、下の 2 つのプログラムの動作はあきらかに異なることに注意が必要です。

```
if 式 1:
    ....
    if 式 2:
        ....
        if 式 3:
            ....
```

```
if 式 1:
    ....
if 式 2:
    ....
if 式 3:
    ....
```

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 では空白行に Tab を入力すれば、自動的にこの書式のインデントに変換されます。また、インデントを戻すときは Shift-Tab が便利です。

3.1.2 #によるコメント

プログラムが複雑になってくると、プログラム中のコメントが理解の助けになることがあります。

Python プログラム行では、ハッシュ記号 (#) につづく文字は無視されます。これを利用してプログラムにコメントを記述することができます。

行全体をコメントとするには、行頭に # をおきコメントを書きます。
また、実行文に続けて # を置き短いコメントを書くこともできます。
さらに、実行文の冒頭に # を入れ、その行を無効化（コメントアウト）することもよくおこなわれます。

以下に、関数 bmax にコメントをいれた例を示します:

```
In [ ]: #
        # Copyright (c) foo bar, All right reserved
        #
        def bmax(a,b):
            if a > b:                    # a は b より大きいか?
        #       print("start a > b") # 文全体をコメントアウト
                return a
            else:
        #       print("start a <= b")
                return b

        print("Hello World")           # 関係ないけど Hello World と印字してみる。
```

3.1.3 if ... elif ... else による条件分岐

if による条件分岐は第 1 回の関数 bmax で紹介しましたが、ここではもう少し詳しく説明していきます。

Python の if 文は以下のように記述します:

```
if 式 1 :
    実行文 1(のグループ、以下同じ)
elif 式 2 :
    実行文 2
elif 式 3 :
```

```
    実行文 3
else:
    実行文 e
```

if ... elif ... else では、条件は上から順に評価され、式が True 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。
そして、elif および else を省略することも可能です。

3.1.4 ▲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 2 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 \ を記述する方法です。

```
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.")
```

3.1.5 条件分岐の順番

条件分岐では、if あるいは elif 文いずれかの条件式が True の場合、以降の elif 文の条件式の評価はおこなわれません。

以下のプログラムは正常に終了しますが、6 行目の条件、x > 0 を満たさない場合、例えば 4 行目で x = 0 とするとエラーとなります。

```
In [ ]: y = 0                   # del y のエラーを抑制するためのおまじない
        del y                   # 変数 y を未定義に

        x = 0

        if x > 0:
            print("x is a positive")
        elif y > 0:
            print("x is not a positive, but y is a positive")
```

3.1.6 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書き換えてください。

```
In [ ]: x = -1
        if x < 3:
            print ("x is larger or equal to 2, and less than 3")
        elif x < 2:
            print ("x is larger 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")
```

3.1.7 分岐の評価

if 文に与える条件が or, 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、
2 番目の式 b == 0 を評価することなくつづ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。

以下のセルで示す二つの例のうち、最初のものは変数 x が未定義のためエラーとなります。一方 2 つ目は正常に実行されます。これは、複合条件のうち y > 5 は評価されることなく、変数 y が未定義にもかかわらずプログラムは正常に終了します。

```
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")
```

3.1.8 ▲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"
```

3.1.9 練習の解説

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 True の場合『直後の実行文グループのみ』が処理されます。

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

3.2 制御構造のうち繰り返し

3.2.1 for による繰り返し

関数 range によって繰り返し回数を指定する for 文については 2-2 でも説明しました。 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)
```

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 繰り返し (ループ) を抜け、完了メッセージを印字します。

次は文字列の例です。文字列は先頭から 1 文字ずつ文字列が取り出されます。

```
In [ ]: word = "Supercalifragilisticexpialidocious"
        for w in word:
            print(w)
```

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。辞書のキーに対応する値を順に取り出す方法については次の節を参照して下さい。

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

3.2.2 辞書に対する繰り返し処理

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 という名前の辞書から： 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。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: 'orang
```

3.2.4 range 関数

2-2 で説明した range 関数は特定の回数の繰り返し処理が必要なとき利用します。 range() 関数は:

1. 引数をひとつ与えると 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 2 つあるいは 3 つ与えると: 最初の引数を数列の開始 (start)、2 番目を停止 (stop) 、3 番目を数列の刻み (step) とする整数列を返します。
   3 番目の引数は省略可能で、既定値は 1 となっています。

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 [ ]: sum = 0
        for i in range(10):
            sum = sum + i

        print (sum)
```

3.2.4.1 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。1 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。

3.2.5 練習

リスト list1 が引数として与えられたとき、次の様な整数を返す関数 sum_lists を作成して下さい。ただし、list1 の各要素はリストであり、そのリストの要素は実数です。

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)
```

3.2.6 練習

リスト list1 と list2 が引数として与えられたとき、次の様なリスト list3 を返す関数 sum_matrix を作成して下さい。ただし、list1, list2, list3 は、3 つの要素をもちます。また、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実数です。また、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,
```

3.2.6.1 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)
```

3.2.7 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。

```
In [ ]: def sum_n(x,y):
            return 0
```

関数が完成すれば、以下のセルを実行して True が印字されることを確認してください。

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

3.2.8 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)
    # くりかえさせたい処理
    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):
    # くりかえさせたい処理
```

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。
```

3.2.9 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, 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
```

3.2.10 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様になります。

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

        print(x, total)
```

3.2.11 break 文

break 文は for および 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)
```

3.2.12 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_search 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)
```

3.2.13 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)
```

3.2.14 ▲ 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。

3.2.15 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:
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:
    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")
```

3.2.16 練習

以下のプログラムでは 1 秒おきに print() 文が永遠に実行されます。
10 回 print() 文が実行された後に 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)
```

3.2.17 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（これがない場合を試してみ
            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)
```

3.2.18 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)
```

3.2.19 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
```

3.3.2 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 [ ]: table =  [[x*y for y in range(3)] for x in range(3)]
        vector = [x*y for y in range(3) for x in range(3)]
```

上のプログラムでは、 table として [[0, 0, 0], [0, 1, 2], [0, 2, 4]] が、vector として、[0, 0, 0, 0, 1, 2, 0, 2, 4] が得られます。

3.3.3 練習

文字列リストが strings で与えられたとき、文字列の長さをリストで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]
```

3.3.4 ▲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。

3.3.5 ▲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。

3.3.6 ▲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_dir = {w:len(w) for w in words}
        print(length_dir)
```

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

3.3.7 ▲ジェネレータ式

タプル () に対して内包表記を利用することはできません。ただし、丸括弧 () で内包表記（厳密には違う）を記述するとタプルではなくジェネレータオブジェクトがつくられます。ジェネレータについては第 6 回で説明します。

したがって、以下のプログラムは動作すしますがオブジェクト squares_gen はタプルではありません。

```
In [ ]: squares_gen = (x**2 for x in range(6))
        for x in squares_gen:
            print(x)
        type(squares_gen)
```

3.3.8 練習の答え

以下のように書くことができます。

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

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")
```

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**（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、引数をカンマで区切ります。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

再帰は、対象のタスクを、容易に解ける小さな粒度まで分割していき、個々の小さなタスクを解いていくことで、タスク全体を解くための方法です。再帰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。通常、再帰は、関数がそれ自身の関数を呼び出すことで実現します。**再帰関数**は、一般的に以下のように書くことできます。

```
def recursive_function():
    if:
        非再帰的な基本処理
    else:
        再帰的な処理
```

以下で、再帰を使った処理の例をいくつか見ていきましょう。

3.5.1 再帰の例：リストの要素の和

```
In [ ]: #入力のリストの要素の和を計算する関数 list_sum
        def list_sum(l):
            if (len(l) == 0):
                #リストが空ならば 0 を返す
                return 0
            else:
        #         print(l[0])
                #入力リスト l の先頭 l[0] を除いた l[1:] を list_sum に渡し、再帰的にリストの要素の和を計算
                #(1+(2+(3+(4+(5+0)))))
                return l[0] + list_sum(l[1:])
```

```
print(list_sum([1,2,3,4,5]))
```

3.5.2 再帰の例：べき乗の計算

```
In [ ]: #入力の庭 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:
        #         print(expt)
                #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)

        print(power(2,10))
```

3.5.3 再帰の例：階乗の計算

```
In [ ]: #入力の整数 n の階乗を計算する関数 factorial
        def factorial(n):
            if (n < 2):
                #n が 2 未満なら 1 を返す
                return 1
            else:
        #       print(n)
                #n を 1 つずつ減らしながら factorial に渡し、再帰的に階乗を計算
                #(n*(n-1*(n-2*....*1)))
                return n*factorial(n-1)

        print(factorial(10))
```

一般に、再帰の処理は繰り返し処理でも書くことができます。単純な処理においては、再帰よりも繰り返し処理の方が、少ない計算資源（メモリ消費など）で早く計算できることが多いですが、複雑な処理や適用するタスクによっては、処理を再帰的に定義したほうがコードの可読性が高く、かつ効率的に計算できることもあります。

```
In [ ]: #階乗の計算を繰り返し処理で行った例
        def factorial(n):
            f = 1
            for i in range(2,n+1):
                f *= i
            return f

        print(factorial(10))
```

3.6 ▲関数型プログラミング

関数型プログラミングでは、処理（手続きの流れ）を記述するのに、関数を組み合わせることででプログラムを構成するようにします。関数プログラミングでは、主に、map、filter、reduce の高階関数を組み合わせて、データに様々な処理を行います。ここで、**高階関数**（こうかいかんすう）とは他の関数を引数として受け取る関数です。

3.6.1 関数オブジェクト

関数型プログラミングでは、関数をオブジェクト（**関数オブジェクト**）として利用します。オブジェクトについては 1-3 の説明を参考にしてください。

関数をオブジェクトと考えると、関数名の変数は、その関数のオブジェクトを参照（指し示）していることになります。

オブジェクトとしての関数は、関数の引数としても利用できます。map、filter、reduce などの高階関数は、関数を引数として受け取り、その関数にデータを渡し処理をさせることが特徴です。

map、filter、reduce について見る前に、まず、一般の関数について引数として関数を受け取ることを考えてみましょう。

```
In [ ]: #x の平方を計算する関数 square
        def square(x):
            return x*x

        #リスト args の各要素を関数 func の入力とし、その出力をリストにして返す関数 call_func
        def call_func(func, args):
            results =[func(i) for i in args]
            return results

        #関数 square を引数に渡した関数 call_func の呼び出し
        call_func(square,[1,2,3])
```

3.6.2 無名関数

上記では、平方を計算する関数 square を明示的に定義しましたが、lambda 式を使うことで、無名の（関数名のない）関数オブジェクト（**無名関数**）を作成できます。

lambda 引数:処理内容と戻り値

```
In [ ]: #x の平方を計算する無名関数を引数に渡した関数 call_func の呼び出し
        # lambda x:x*x により引数の平方を計算する無名関数を定義
        call_func(lambda x: x*x, [1,2,3])
```

無名関数の典型例として、タプルもしくはリストの二番目の要素を返す関数がよく使われます。

```
lambda x: x[1]
```

この関数は、辞書の要素をキーの値で並び替えるときに使うことができます。

mydict という変数に辞書が入っているとします。キーの値は数であるとします。

```
In [ ]: mydict = {'A': 3, 'T': 5, 'G' : 1, 'C' : 2}
```

mydict.items() によってキーと値のタプルのリストが得られます。（正確には、通常のリストではなくて特殊なリストになります。）

```
In [ ]: mydict.items()
```

以下のようにして、関数 sorted を用いてタプルのリストをソートした結果を得ることができます。この場合は、キーの順番でソートされます。

```
In [ ]: sorted(mydict.items())
```

キーの値で並び替えるのには key=lambda x:x[1] と指定します。すると、タプルを比較する際に、タプルの二番目の要素が用いられます。すなわち、以下のようにして sorted を呼び出します。

```
In [ ]: sorted(mydict.items(), key=lambda x:x[1])
```

さらに reverse=True と指定すると、キーの値の降順にソートされます。

```
In [ ]: sorted(mydict.items(), key=lambda x:x[1], reverse=True)
```

3.6.3 map 関数

map 関数は、関数とイテラブルオブジェクトを受け取り、リストなどのイテラブルオブジェクトの各要素にその関数を適用します。そして、その関数の戻り値を要素とするイテレータを返します。

イテラブルオブジェクトとイテレータに関しては、6-3 を参照してください。ここでは、イテラブルオブジェクトとしてリストを想定してください。

map は、リストの各要素に最初の引数である関数を適用します。そして、その関数の戻り値を要素とするイテレータと呼ばれるオブジェクトを返します。このイテレータに list という関数を適用すると、関数の戻り値を要素とするリストが返ります。

```
In [ ]: #x の平方を計算する関数 square
        def square(x):
            return x*x

        #map 関数に関数 square とリストを渡して、リストの各要素の平方を計算
        list(map(square, [1,2,3]))
```

ちなみに、list を適用しないと以下のようになります。

```
In [ ]: map(square, [1,2,3])
```

これは、イテレータと呼ばれているオブジェクトの一種です。

もちろん、map が返したリストに map を適用することができます。

```
In [ ]: list(map(square, list(map(square, [1,2,3]))))
```

実は、内側の list は省略することができます。

```
In [ ]: list(map(square, map(square, [1,2,3])))
```

内側の map が返したイテレータは、イテラブルオブジェクトでもあるので、そのまま外側の map に与えることができるからです。

map 関数に、イテラブルオブジェクト（たとえばリスト）を複数渡すと、各イテラブルオブジェクトの要素を並行に

参照しながら、それらの複数のイテラブルオブジェクトの各要素の組み合わせを、map 関数の引数で指定した関数へ順番に渡します。その際、関数は、イテラブルオブジェクトの数だけの引数を取らなければなりません。

```
In [ ]: #map 関数に複数のリストを渡して、それらのリストの各要素の積を無名関数で計算
        list(map(lambda x,y:x*y, [1,2,3], [1,2,3]))
```

3.6.3.1 zip 関数

zip 関数は、引数として渡すイテラブルオブジェクトの各要素の組み合わせを要素とするイテレータを返します。

```
In [ ]: #zip 関数に 3 つのリストを渡して、それらのリストの各要素の組み合わせを順番に出力
        list(zip([1,2,3], [4,5,6],[7,8,9]))
```

実は、map を用いても同様の計算ができます。

```
In [ ]: list(map(lambda *x: x, [1,2,3], [4,5,6],[7,8,9]))
```

3.6.4 filter 関数

filter 関数は、関数とイテラブルオブジェクトを受け取り、イテラブルオブジェクトの各要素にその関数を適用します。そして、その関数の戻り値が真 True を返す要素のみのイテレータを返します。

```
In [ ]: #無名関数 lambda x:x>0 により、関数の引数は正であることを条件式として指定
        #リストの要素の中で、正のもののみを出力
        list(filter(lambda x: x >0, [-1,0,1,2]))
```

3.6.5 reduce 関数

reduce 関数は、一般に 2 つの引数を受け取る関数とイテラブルオブジェクトを受け取ります。そして、イテラブルオブジェクトの各要素に対して、左から右に累積的に関数を適用し、要素を単一に縮約した値を返します。例えば、以下は (((((1+2)+3)+4)+5) を計算することになります。

```
reduce(lambda x, y: x+y, [1, 2, 3, 4, 5])
```

イテラブルオブジェクトが単一の要素しか持っていない場合、最初の要素が返されます。

```
In [ ]: #reduce 関数は functools モジュールからインポートします
        from functools import reduce

        # lambda x,y:x*y により 2 つの引数の積を計算する無名関数を定義
        #reduce 関数を用いて、リストの各要素を順番に以下のように繰り返し掛け算
        #(((1*2)*3)*4)
        reduce(lambda x,y:x*y,[1,2,3,4])
```

二進数の表現が 0 と 1 から成るリストとして与えられたとき、それが表す整数を求めるのに、以下のように reduce を用いることができます。

```
In [ ]: reduce(lambda x,y:2*x+y,[1,0,0,0])
```

```
In [ ]: reduce(lambda x,y:2*x+y,[1,0,1,0])
```

```
In [ ]: reduce(lambda x,y:2*x+y,[1,1,1,0])
```

```
In [ ]: reduce(lambda x,y:2*x+y,[1])
```

3.6.6 関数内の関数

関数の中で関数を定義することができます。これを**関数内の関数**と呼びます。関数内の関数はローカル変数となります。

関数内の関数は、再帰関数として定義して利用することがあります。

```
In [20]: #関数 outer_func 内に関数内の関数 innter_func を定義
         def outer_func():
             def inner_func():
                 print ("This is the inner function")
             return inner_func()

         #関数 outer_func を呼び出し
         outer_func()
```

```
This is the inner function
```

3.6.7 デコレータ

デコレータは関数を修飾して、新たな関数を生成します。複数の関数に共通する補助的な処理を、デコレータとしてまとめることができます。

デコレータは関数を受け取り、修飾した関数を返します。以下では、修飾する関数 f の出力に "This is the decorator" という表示を足して出力するデコレータを作成しています。

```
In [24]: #修飾する関数に"This is the decorator"という表示を足すというデコレータ deco の定義
         def deco(f):
         #デコレータでは、関数内の関数を定義し、処理を定義することが多いです
             def inner_func(*args, **kwds):
                     print ("This is the decorator")
                     return f(*args, **kwds)
         #修飾した関数を返す
             return inner_func
```

以下のように、修飾したい関数の定義の前に、デコレータを宣言することで、関数を修飾することできます。

```
@デコレータ
def 修飾したい関数
```

```
In [30]: # デコレータ deco による関数 func の修飾
         @deco
```

```
        def func(x):
            print ("Hello")
            return x
```

```
In [31]: func(100)
```

```
This is the decorator
Hello


Out[31]: 100
```

```
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' はファイル名の文字列で、一般にはファイルの絶対パス名か、そ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。 ノートブックの上のディレクトリの下にある 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' はファイルの読み書きのモードの一種で、**読み込みモード**を意味します。

書き込みについては後で説明します。

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)
```

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
```

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。

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

ファイル全体の内容が文字列として返されました。\n は改行文字です。
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、属性の値が関数であるとき、その属性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。

4.1.2.2 オブジェクトの比較

二つのオブジェクトが同じ参照値を持つかどうかは is という演算子（**is 演算子**）によって調べることができます。

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

```
In [ ]: f is ln
```

```
In [ ]: f2 = f
```

```
In [ ]: f is f2
```

変数 f にはファイルのオブジェクトが入っています。 変数 ln にはリストのオブジェクトが入っています。これらの参照値は異なります。

実は == で比較しても同じです。

```
In [ ]: f == ln
```

is の結果と == の結果は同じことが多いのですが、== はオブジェクトの型にしたがって中身まで調べることがあります。

```
In [ ]: ln2 = [1,2,3]
```

```
In [ ]: ln is ln2
```

```
In [ ]: ln == ln2
```

上の場合、変数 ln2 にもリストのオブジェクトが入っていますが、このオブジェクトは上の代入文が実行されたときに生成されたものなので、変数 ln に入っているオブジェクトとは異なります。 したがって ln is ln2 は偽になります。

一方、これらのリストの中身は同じです。したがって、ln == ln2 は真となります。

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。イテレータに対しては、next という関数を適用することができます。

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、ファイルがすべて読まれてしまっているので、もう一度 for 文を回しても for 文の中身は一度も実行されません。

（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。）

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が書かれます。
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。既にファイルが存在する場合、古い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 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。

```
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]
```

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

文字列を整数に変換するには、int を関数として用いればよいです。

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

ファイルの終わりまできましたので、上のようにエラーになります。

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

4.2.2 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()
```

4.2.3 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)
```

4.2.4 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、csv ファイルにデータが書き込まれます。

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, lineterminator='\n')
```

ここで、ファイルのオブジェクトに加えて、lineterminator というキーワード引数の値'\n' を lineterminator='\n' として指定することが必要ですので注意してください。

キーワード引数は、通常の引数に加えて指定するもので、

キーワード=値

という形式で指定します。通常の引数の後に、この形式をコンマで区切って指定します。

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

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

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

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

with 文を使うこともできます。

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

4.2.5 東京の 7 月の気温

tokyo-temps.csv には、気象庁のオープンデータからダウンロードした、東京の 7 月の平均気温のデータが入っています。

http://www.data.jma.go.jp/gmd/risk/obsdl/

48 行目の第 2 列に 1875 年 7 月の平均気温が入っており、以下、2016 年まで、12 行ごとに 7 月の平均気温が入っています。

以下は、これを取り出す Python の簡単なコードです。

```
In [ ]: import csv

        with open('tokyo-temps.csv', 'r', encoding='sjis') as f:
            dataReader = csv.reader(f)
            n=0
            year = 1875
            years = []
            july_temps = []
            for row in dataReader:
                n = n+1
                if n>=48 and (n-48)%12 == 0:
                    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 2016 までとします。
2. 作成した csv ファイルを Excel から読み込んで確認してください。

```
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== 142)
```

4.2.7 練習

整数データのみからなる csv ファイルの名前をもらって、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 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 [ ]:
```

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, lineterminator='\n')
            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 ファイルのダンプとロード

json 形式は、各種のデータを記録するための形式です。特に、辞書や辞書のリストを記録することができます。

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
```

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

```
In [ ]: with open("test.json", "w") as f:
            json.dump(d, f)
```

```
In [ ]: with open("test.json", "r") as f:
            d1 = json.load(f)
```

```
In [ ]: d1
```

```
In [ ]: d == d1
```

4.3.2 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 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 ▲ xml ファイルの入出力

4.4.1 xml ファイル

xml 形式は、木構造を格納するための汎用的な形式です。xml 形式によって表現された木構造が格納されたファイルが **xml ファイル**です。

木構造とは、生物の分類、会社の組織、ファイルシステム、住所などのように、階層的に構造化されたデータのことです。

以下では、大学の学部の組織を例としています。東京大学の下に学部があって、その下にコースや学科があります。学科の下にさらにコースがある場合もあります。このような階層構造を xml フォーマットで表現すると以下のようになります。これは `sample.xml` という名前の xml ファイルの内容です。

```
<?xml version="1.0" encoding="utf-8"?>
<university name="東京大学">
    東京大学は 1877 年に設立されました。
    <faculty name="法学部">
        <course name="私法">
        </course>
        <course name="公法">
        </course>
        <course name="政治">
        </course>
    </faculty>
    <faculty name="医学部">
        <department name="医学科">
        </department>
        <department name="健康総合科学科">
        </department>
    </faculty>
    <faculty name="工学部">
        <department name="社会基盤学科">
            <course name="設計・技術戦略">
            </course>
            <course name="政策・計画">
            </course>
            <course name="国際プロジェクト">
            </course>
        </department>
    </faculty>
    <faculty name="文学部">
    </faculty>
    <faculty name="理学部">
    </faculty>
    <faculty name="農学部">
```

```
    </faculty>
    <faculty name="経済学部">
    </faculty>
    <faculty name="教養学部">
    </faculty>
    <faculty name="教育学部">
    </faculty>
    <faculty name="薬学部">
    </faculty>
    <faculty name="芸術学部">
    </faculty>
</university>
```

このファイルは、東京大学の学部の組織構造を記述しかけたものです。(要するにまだ書きかけです。)

全体は木構造と考えられます。木構造はノードから成り立っています。ノードの間には親子関係があります。木構造の大本のノードを根と呼びます。この木構造では、東京大学のノードが根です。そして、その根の下に各学部のノードがあります。その下に学科やコースのノードがあります。

各ノードにはタグが付けられています。 根のノードのタグは university です。 学部のノードのタグは faculty です。xml ファイルの中で、各ノードは、

```
<タグ>
...
</タグ>
```

と表現されます。 .. のところには、そのノードのテキストに加えて、そのノードの子供のノード(さらのその子供のノード...)が記述されます。

ノードには属性を与えることができます。以下のように、属性とその値をタグとともに指定することができます。

```
<タグ 属性 1="値 1" 属性 2="値 2">
...
</タグ>
```

実際に、上の例では、各ノードに大学名や学部名が name という属性の値として設定されています。属性の値はすべて文字列と考えられます。

より詳しくは、以下のようなチュートリアルを参照してください。

https://docs.python.jp/3/library/xml.etree.elementtree.html

4.4.2 xml ファイルの読み込み

以下のようにして xml ファイルを読み込むことができます。

```
In [ ]: import xml.etree.ElementTree as ET

        tree = ET.parse('sample.xml')
```

ET.parse は、xml のオブジェクトを返します。(このオブジェクトのクラスは xml.etree.ElementTree.ElementTree です。)

```
In [ ]: tree
```

以下のようにして木構造のオブジェクトを得ることができます。

```
In [ ]: root = tree.getroot()
```

これは根のノードを表すオブジェクトです。（このオブジェクトのクラスは xml.etree.ElementTree.Element です。）

```
In [ ]: root
```

ノードのタグを取り出します。

```
In [ ]: root.tag
```

ノードの属性を取り出します。属性と値の辞書が得られます。

```
In [ ]: root.attrib
```

ノードのテキストを取り出します。

```
In [ ]: root.text
```

ノードはイテラブルオブジェクトと呼ばれるオブジェクトの一種です。イテラブルオブジェクトも for 文の in の後に指定することができます。

以下のように、ノードを for 文によって繰り返すことにより、子供のノードを順に得ることができます。

```
In [ ]: for child in root:
            print(child.tag, child.attrib)
```

ノードはイテラブルオブジェクトですが、イテレータではないので、もう一度 for 文を回すと再び子供のノードを順に得ることができます。

```
In [ ]: for child in root:
            print(child.tag, child.attrib)
```

以下では子供の子供まで表示させています。

```
In [ ]: for child in root:
            print(child.tag, child.attrib, child.text)
            for grandchild in child:
                print('    ', grandchild.tag, grandchild.attrib, grandchild.text)
```

4.4.3 再帰関数による辞書への変換

以下の関数はノードを辞書に変換して返します。

ノードの子供に対しても自分自身を適用して、各子供を変換した辞書から成るリストを求めます。そのリストを、自分を変換した辞書の children という属性の値とします。

```
In [ ]: def xmltodict(node):
            l = []
            for child in node:
                l.append(xmltodict(child))
```

```
        d = {}
        d['tag'] = node.tag
        d['text'] = node.text
        d['attrib'] = node.attrib
        d['children'] = l
        return d
```

```
In [ ]: xmltodict(root)
```

4.4.4 xml ファイルの更新

以下のようにすれば工学部のオブジェクトが得られます。

```
In [ ]: for child in root:
            if child.attrib['name'] == '工学部':
                engineering = child
```

```
In [ ]: engineering
```

以下のようにして、このノードの下に department をタグとする子供ノードを追加することができます。

```
In [ ]: d = ET.SubElement(engineering, 'department')
```

そして、追加した子供ノードの属性を設定しましょう。

```
In [ ]: d.attrib = {'name': '建築学科'}
```

ノードがどうなっているかは、以下のようにして、ノードとその子供（さらにその子供）を表示させることができます。

```
In [ ]: ET.dump(engineering)
```

東京大学には芸術学部はありませんので、これは削除しましょう。

```
In [ ]: for child in root:
            if child.attrib['name'] == '芸術学部':
                art = child
        root.remove(art)
```

根のノードを表示させてみます。

```
In [ ]: ET.dump(root)
```

4.4.5 xml ファイルの書き出し

以下のようにして、更新した内容をファイルに書き出すことができます。

```
In [ ]: tree.write('sample1.xml', encoding='utf-8',  xml_declaration=True)
```

```
In [ ]:
```

第 5 回

5.1 モジュールの使い方

5.1.1 モジュールの import

第 1 回でモジュールの import について説明しました。数学関係の各種の関数はモジュールの中で提供されており、これらの関数を使いたいときは、以下のようにして math というモジュールを import でインポートします。 そうしますと、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

        math.log(32,2)
```

($\log_2 32$ を計算しました。)

5.1.2 from

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 math import log
```

すると、以下のように math というモジュール名を付けずに関数 log を参照することができます。

```
In [ ]: log(32,2)
```

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。
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。

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。

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

```
In [ ]: np.ones((3,3))
```

個々の関数ごとに別の名前を付けることもできます。

```
In [ ]: from random import randint as rand
```

```
In [ ]: rand(1,10)
```

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。
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)
```

5.1.8 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。

5.1.9 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() 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() 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
```

5.1.10 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 [ ]: plt.plot([fibtime(i) for i in range(20,35)])
```

5.1.11 re

re は、**正規表現** (regular expression) に関係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/re.html

```
In [ ]: import re
```

5.1.11.1 re.split

正規表現について一般的に説明する前に、正規表現を用いる例として、正規表現を用いた文字列の分割について紹介しましょう。

そのためには、re.split という関数が用いることができます。 re.split は、正規表現と文字列を受け取ります。正規表現とは、文字列のパターンを表現する式です。 re.split は、受け取った文字列を、正規表現にマッチする部分文字列を区切り文字列（デリミタ）として分割して、文字列のリストを返します。

すなわち、re.split(e,s) は、正規表現 e にマッチする文字列を区切り文字列として、 文字列 s を単語に区切って、単語（文字列）のリストを返します。

以下が典型例です。

```
re.split(r'[^a-zA-Z]+', 'Hello, World! How are you?')
```

[^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]+' という構文を用います。 これについては、以下の説明を参照してください。

```
In [ ]: re.split(r'[^a-zA-Z]+', 'Hello, World! How are you?')
```

この例のように、返されたリストに空文字列が含まれる場合がありますので、注意してください。

以下の例は、英語の文書を 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求めます。空文字列は除きます。結果は変数 word_list に格納します。

```
In [ ]: import re
```

```
        word_list = []
        with open('text-sample.txt', 'r') as f:
            for word in re.split(r'[^a-zA-Z]+', f.read()):
                if word != '':
                    word_list.append(word.lower())
```

re.split(r'[^a-zA-Z]+', f.read()) は、ファイル全体の文字列を単語に区切ります。
word.lower() は、word の値である単語（文字列）の中の大文字を小文字に変換します。

```
In [ ]: word_list
```

以下の例は、text-sample.txt に出現する単語のリストを、単語の重複のないように求めます。辞書を使って重複がないようにしています。さらに、求めたリストをソートします。

```
In [ ]: word_dict = {}
        with open("text-sample.txt", "r") as f:
            for word in re.split(r'[^a-zA-Z]+', f.read()):
                if word != '':
                    word_dict[word.lower()] = 0
        word_list = []
        for word in word_dict:
            word_list.append(word)
        word_list.sort()
```

```
In [ ]: word_list
```

5.1.11.2 re.sub

もう一つ、正規表現を受け取る関数を紹介します。 re.sub は、文字列の置換を行う関数です。 re.sub(e,s,t) は、文字列 t の中で正規表現 e にマッチする部分文字列を文字列 s で置き換えて得られる文字列を（新たに作って）返します。（もちろん、もとの文字列 t は変化しません。）
たとえば、

```
re.sub(r'[ \t\n]+', ' ', t)
```

とすると、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]+' 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。

```
In [ ]: re.sub(r'[ \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。

5.1.11.3 正規表現

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。
o* という正規表現は、o という文字列の複数回の繰り返しを表すパターンです。繰り返しの数は 0 回でも構いませ

ん。したがって、空文字列 や o や oo や ooo などの文字列がこの正規表現にマッチします。

hello* という正規表現では、hell という文字列の後に o* というパターンが来ていますので、 hell や hello や helloo や hellooo などがマッチします。

hello 全体の繰り返し、たとえば、hellohello や hellohellohello をマッチさせたいならば、(hello)* という正規表現を用います。括弧で文字列（一般には正規表現でも構いません）をまとめて、その繰り返しを表現しています。

. は任意の 1 文字がマッチする正規表現です。 .* はその繰り返しですので、任意の文字列がマッチします。すると、たとえば、a.*z という正規表現には、a で始まって z で終わる文字列がマッチします。

[abc] という正規表現には、a という文字列か b という文字列か c という文字列がマッチします。[a-zA-Z] という正規表現には、任意の英文字 1 文字から成る文字列がマッチします。したがって、[a-zA-Z]* という正規表現には、英文字の 0 回以上の繰り返しがマッチし、[^a-zA-Z]* には、英文字以外の 0 回以上の繰り返しがマッチします。

[a-zA-Z0-9] という表現には、a から z までと A から Z までと 0 から 9 までのいずれかの一文字、 すなわち英数字一文字の文字列がマッチします。すると、[a-zA-Z][a-zA-Z0-9]* には、英文字で始まって英数字の 0 回以上の繰り返しが続く文字列がマッチします。

[^a-zA-Z] は、英文字以外の文字 1 文字から成る文字列がマッチする正規表現です。 ここで^ に注意してください。

[\t\n] は、半角の空白かタブか改行かを表す正規表現です。 正規表現の中に\ で始まるエスケープシーケンスを含めることができます。

* の代わりに + を使うと、1 回以上の繰り返しが表され、0 回の繰り返しは含まれません。たとえば、[\t\n]+ という正規表現には、半角の空白かタブか改行の 1 回以上の繰り返しがマッチします。

また、* の代わりに ? を用いると、0 回か 1 回の出現を意味します。 たとえば、ab?c には abc と ac がマッチします。

二つの正規表現を | を介して並べた正規表現には、どちらかの正規表現にマッチする文字列がマッチします。たとえば、morning|evening という正規表現には、morning という文字列と evening という文字列がマッチします。

文字列の連結（文字列を並べること）の方が | よりも強く結合しますので、連結の中で | を使いたいときは括弧が必要です。たとえば、Good (morning|evening), Hanako! という正規表現には、 Good morning, Hanako! と Good evening, Hanako! がマッチします。

* や . や ? など、正規表現の中で特別な意味を持つ文字そのものを表すには、\ を前に付ける必要があります。たとえば、OK? という文字列をマッチさせるには、OK\? としなければなりません。.^$*+?{}\[]|() のいずれかの文字には \ を付けなければなりません。

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*'
```

5.1.11.4 ▲ r を付ける理由

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。

5.1.11.5 re.search

re.search は正規表現と文字列を受け取り、文字列が正規表現にマッチする部分文字列を含かどうかを判定します。含む場合は、マッチした部分文字列に関する情報を返します。マッチしない場合は、None を返します。

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.search(正規表現, 文字列):
    ...
```

re.search が返す情報については先にも紹介した URL を参照してください。

https://docs.python.jp/3/library/re.html

```
In [ ]: re.search(r'hello*', 'Hi, hellooooo!')
```

```
In [ ]: print(re.search(r'hello*', 'Hi, good morning!'))
```

文字列の先頭からマッチさせたいときは、正規表現の先頭に ^ 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 $ を付けてください。

```
In [ ]: re.search(r'^hello*', 'Hi, hellooooo!')
```

```
In [ ]: re.search(r'^hello*', 'hellooooo!')
```

```
In [ ]: re.search(r'hello*$', 'Hi, hellooooo!')
```

```
In [ ]: re.search(r'hello*$', 'Hi, hellooooo')
```

```
In [ ]: re.search(r'^hello*$', 'hellooooo')
```

5.1.12 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 [ ]: def sumnumbers(s):
            ...
```

以下のセルによってテストしてください。

```
In [ ]: print(sumnumbers(" 2  33 45, 67.9") == 156)
```

5.1.13 練習の回答

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 [ ]: 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
```

```
In [ ]: def sumnumbers(s):
            numbers = re.split("[^0-9]+", s)
            numbers.remove('')
            n = 0
            for number in numbers:
                n += int(number)
            return n
```

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。

```
In [ ]: # リストから配列作成
        list_to_array = np.array([1,2,3,4,5])
        list_to_array
```

NumPy の配列は ndarray オブジェクトとなります。

```
In [ ]: # タプルからの配列作成
        tuple_to_array = np.array((1,2,3,4,5))

        # 配列の型
        type(tuple_to_array)
```

print() 関数を使って配列を出力すると、要素が空白で区切られて出力されます。

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

array() 関数では、第 2 引数 dtype で配列の要素の型を指定することができます。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 dtype 属性で調べることができます。

```
In [ ]: # 配列要素の型の指定
        list_to_array = np.array([1,2,3,4,5], dtype=float)

        # 配列要素の型の確認
        list_to_array.dtype
```

配列を array() 関数に渡し、第 2 引数 dtype で型を指定すると、既存の配列を別の型に変換した配列を作成できます。

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

        # 配列要素の型の変換
        float_array = np.array(int_array, dtype=float)
        float_array
```

5.2.2 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。

shape 属性で、配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 [ ]: # 多次元配列の作成
        mul_array = np.array([[1,2,3],[4,5,6]])
        print(mul_array)

        # 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)

        # 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())
```

5.2.3 様々な配列の作り方

5.2.3.1 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。arange() 関数の第 1 引数には開始値、第 2 引数には終了値、第 3 引数には刻み幅を指定します。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。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。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)

```
In [ ]: # 0 から 1 刻みで 5 つの要素を持つ配列
        a1=np.arange(5)
        print(a1)

        # 0 から 0.1 刻みで 1 未満の値の要素を持つ配列
        a2=np.arange(0,1,0.1)
        print(a2)

        # 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
        a3=np.arange(2*2).reshape(2,2)
```

```
print(a3)
```

5.2.3.2 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
        a=np.linspace(0,100,11)
        print(a)
```

5.2.3.3 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 3 列の 0 要素からなる多次元配列
        zero_array2=np.zeros((3,3), dtype=int)
        print(zero_array2)
```

5.2.3.4 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 [ ]: # 2 行 2 列の 1 要素からなる多次元配列
        one_array=np.ones((2,2), dtype=int)
        print(one_array)
```

5.2.3.5 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)
```

5.2.4 配列要素の操作

5.2.4.1 インデックス

NumPy の配列の要素には、リストと同様に 0 から始まるインデックスを使ってアクセスします。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 [ ]: a = np.array([1,3,5,7,9])
        print(a)

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

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

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

多次元配列では、**array[行, 列]** のように行と列で要素にアクセスできます。この時、行と列はインデックスと同じくそれぞれ 0 から始まります。また、多次元リストと同様に、**array[インデックス][インデックス]** のようにリストごとのインデックスを使っても要素にアクセスできます。

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

        # 0 行 1 列の要素
        print(a[1,2])

        # 0 行 1 列の要素に代入
        a[1,2]=0

        # 0 行 1 列の要素
        print(a[1][2])
```

5.2.4.2 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 [ ]: a = np.array([1,10,100,1000,10000])

        # 配列 a 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(a[1:4])

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

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

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

```
        # 配列 a の終端から先頭までの要素をスライス
        print(a[::-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 [ ]: a = np.array([1,10,100,1000,10000])

        # 配列 a 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
        a[1:4]=0
        print(a)
```

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カンマで区切って指定します。

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

        # 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(a[: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(a[1:,1:])
```

5.2.4.3 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。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が、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 [ ]: a1 = np.array([1,2,3,4,5,6])

        # 配列 a1 から要素の取り出し
        for num in a1:
            print(num)

        a2 = np.array([1,2,3,4,5,6]).reshape(2,3)
        # 多次元配列 a2 から行の取り出し
        for row in a2:
            # 多次元配列 a2 の行の要素の取り出し
            for num in row:
                print (num)

        # 多次元配列 a2 から行の要素と繰り返し回数の取り出し
        for i, num in enumerate(a2):
            print(i, num)
```

```
# 多次元配列 a2 から要素の取り出し、要素の位置を行と列のタプルで取得
for i, num in np.ndenumerate(a2):
    print(i, num)
```

5.2.4.4 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ますが、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。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 [ ]: a1 = np.array([5,3,1,4,2])
        # 配列 a1 の要素を並び替え
        a1.sort()
        print(a1)

        a2 = np.array([5,3,1,4,2])
        # 配列 a2 の要素を並び替えた結果から新たな配列 a3 を作成
        a3 = np.sort(a2)

        print(a2)
        print(a3)
```

5.2.5 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 [ ]: a = np.array([1,2,3,4])

        # 配列 a のすべての要素に 1 を加算
        b = a+1
        print(b)

        # 配列 b のすべての要素に 2 を乗算
        c=b*2
        print(c)

        # 配列 c のすべての要素に 2 を除算
        d=c/2
        print(d)

        # 配列 d のすべての要素を二乗
        e=d**2
        print(e)
```

この他に、NumPy には**ユニバーサル関数**と呼ばれる、配列を入力としての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 [ ]: a = np.array([1,2,3,4,5,6]).reshape(2,3)
        print(a)

        # 多次元配列 a のすべての要素の平均
        print(a.mean())

        # 多次元配列 a の各列の要素の平均
        print(a.mean(0))

        # 多次元配列 a の各行の要素の平均
        print(a.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

5.2.6 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 [ ]: A = np.array([1,2,3,4]).reshape(2,2)
        B = np.array([2,4,6,8]).reshape(2,2)

        # 配列の要素同士の足し算
        C = A+B
        print(C)

        # 配列の要素同士の引き算
        D = B-A
        print(D)

        # 配列の要素同士の掛け算
        E = A*B
        print(E)

        # 配列の要素同士の割り算
        F = B//A
```

```
print(F)
```

5.2.7 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
```

5.2.8 ▲配列要素の追加、挿入、削除

5.2.8.1 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 [43]: 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)
```

```
[[1 2 3]
 [4 5 6]]
[[1 2 3]
 [4 5 6]
 [7 8 9]]
[[1 2 3 0]
 [4 5 6 0]
 [7 8 9 0]]
```

5.2.8.2 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)
```

5.2.8.3 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)
```

5.2.9 ▲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)
```

5.2.10 ▲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)
```

5.2.11 ▲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

5.3 ▲ 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。

5.3.1 線グラフ

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、7-1 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。この時、セルの最後に評価されたオブジェクトの出力表示を抑制するために、以下ではセルの最後の行にセミコロンをつけています。

```
In [ ]: import numpy as np
        import matplotlib.pyplot as plt
        %matplotlib inline
```

以下では、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ています。y 軸の値に対応する x 軸の値は、リストの各要素のインデックスとなっています。

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

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

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。

```
In [ ]: # plot するデータ
        x =[0, 1, 2, 3, 4]
        y =[0, 1, 2, 3, 4]

        # plot 関数で描画
        plt.plot(x,y);
```

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。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

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、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();
```

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 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。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。この時、plot() 関数では、プロットする点のマーカーの形状を引数に指定しています。

```
In [ ]: x = np.random.rand(20)
        y = np.random.rand(20)
        plt.plot(x, y, 'o', color='blue');
```

5.3.3 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
        plt.bar(x,y);
```

5.3.4 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);
```

5.3.5 ヒートマップ

impshow() 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
        a = np.random.rand(100).reshape(10,10)

        # imshow 関数でヒートマップを描画
        im=plt.imshow(a)
        plt.colorbar(im);
```

5.3.6 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');
```

5.3.7 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、$y = e^x$ のグラフを描画してください。また、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。

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

この講義ではこれまで、プログラムを jupyter-notebook（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-notebook 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-notebook におけるコードセルの内容をファイルに記述します。オペレーティングシステム（実際にはシェル）から Python プログラムファイル、 例えば foo.py を実行するには、以下のようにします。（もちろん python は実行パスに入っている必要があります）

```
$ python foo.py
```

あるいは

```
$ python3 foo.py
```

5.4.1.1 Windows での実行方法（自身で Anaconda3 をインストールしたもの）

1. 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るはずです。 Start メニュー ⇒ Anaonda3(64-bit) ⇒ Anaconda Prompt

5.4.1.2 Windows での実行方法（ECCS の Windows 環境）

1. 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
2. python に代えて、/cygdrive/c/Anaconda3/python を実行します。

5.4.1.3 MacOS での実行方法（自身で Anaconda3 をインストールしたもの）

1. 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）

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 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 -*-
```

Unix ではプログラムスクリプトの先頭行には、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（shebang 行）。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 (.py) を開く
2. jupyter-notebook を Python プログラム (.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-notebook 形式を 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-notebook を .py としてセーブするには、

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

を選択します。

そうしますと、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、パラメータとして引数を与えることができます。

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。以下のプログラム argsprint.py はコマンドラインに与えられた引数を出力します。

```
from sys import argv
print(argv)
```

これを実行すると、以下のように出力されます。リストの最初の要素には、プログラム名そのものが格納されることに注意してください。

```
$ python argsprint.py foo bar baz
['argsprint.py', 'foo', 'bar', 'baz']
$
```

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ください。

5.4.6 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))
```

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))
```

5.4.7 練習

第 4 回の本課題のプログラムを、keyword.py というファイルに格納して、以下のようにして呼び出してください。

```
import keyword
keyword.count_keyword("理学部")
```

5.4.8 モジュールの検索パス

インポートされるモジュールは Python インタプリタの検索パスに置く必要があります。当面は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問題はありません。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
```

5.4.9 ▲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。このような場合は、プログラム実行部分に以下のコードを追加することで実現できます。

```
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。

5.4.10 ▲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
```

```
In [ ]:
```

第 6 回

6.1 オブジェクト指向

6.1.1 オブジェクト型

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**として実現されています。そして、**オブジェクト**はそれぞれに不変な**型**を持ちます。一般に型とはデータの種類のことですが、Python では、すべてのデータはオブジェクトなので、型は**オブジェクト型**ともいいます。オブジェクト型は以下のように分類されます。

- 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

オブジェクトの型は、`type()` という組み込み関数で調べることができます。

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

6.1.2 クラス定義

Python では**クラス**を定義することで、新しいオブジェクト型を作成します。**クラス定義**は次のように `class` で始まる構文によって行います。

```
class NewClass:
    クラスに関するプログラム文
```

6.1.2.1 オブジェクトの生成

定義されたクラスを型とするオブジェクトを**生成**する方法は以下のとおりです。

```
NewClass()
```

以下では、pass を用いて最も簡単なクラス（クラスに関するプログラム文が何もないクラス）を定義し、そのオブジェクトを生成します。

```
In [ ]: class EmptyClass:
            pass


        ec = EmptyClass()
```

ec という変数にはオブジェクトが入っています。

```
In [ ]: ec
```

関数 type で、このオブジェクトの型を確認してみます。

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

__main__.EmptyClass というオブジェクト型であることがわかります。 これは EmptyClass と同じです。ここで、__main__はこのプログラムのモジュールを示しています。（'4-4 モジュール' を参照してください。）

```
In [ ]: EmptyClass
```

```
In [ ]: type(ec) == EmptyClass
```

もう一つ、このクラスのオブジェクトを作ってみます。

```
In [ ]: ec1 = EmptyClass()
```

```
In [ ]: ec1
```

6.1.2.2 オブジェクトの比較

生成されたオブジェクト同士を == によって比較することができます。

```
In [ ]: ec == ec1
```

このように、EmptyClass() によって、それまでに存在しない新しいオブジェクトが生成されますので、二つのオブジェクトは等しくなりません。

EmptyClass のオブジェクトの比較は、単にオブジェクトの参照値が等しいかどうかで行われますので、is で比較しても同じ結果になります。（実は__eq__ というメソッドを定義すると == の結果を変更することができます。）

```
In [ ]: ec is ec1
```

```
In [ ]: ec is ec
```

```
In [ ]: ec == ec
```

```
In [ ]: ec != ec1
```

6.1.3 ▲クラス名について

クラス名に特に制約はありませんが、EmptyClass、MyClass、SimpleCourse のように、 単語の組み合わせではそれぞれの最初の文字を大文字にする（CapitalizedWords, CapWords, Camel case と呼ばれる）形式が一般的です。

6.1.4 属性の追加、変更、削除

生成されたオブジェクトに対して、属性の追加、変更、削除を行うことができます。

```
In [ ]: ec.name = 'bar'

In [ ]: ec.name

In [ ]: ec.name = 'boo'

In [ ]: ec.name

In [ ]: del ec.name

In [ ]: ec.name
```

6.1.5 メソッドの定義

クラス内で定義される関数はメソッドと呼ばれ、そのクラスのオブジェクトの属性となります。

メソッドが呼び出されると、その最初の引数にオブジェクト自身がわたされます。一般にこれに対応する引数の名前として self が使われます。ただし、この引数は、メソッドを呼び出す式の括弧の中には書かれません。

具体例として、MyClass を以下のように定義して、そのオブジェクトを生成して mc という変数に入れます。

```
In [ ]: class MyClass:
            def f(self):
                return type(self)
            def f2(self, param):
                print(self, ':', type(self))
                return param

        mc = MyClass()

In [ ]: mc
```

このオブジェクトの属性 f と f2 を確認してみます。関数 type を使うと、mc.f と mc.f2 がメソッドであることが確認できます。

```
In [ ]: mc.f

In [ ]: type(mc.f)

In [ ]: type(mc.f2)
```

メソッド f を呼び出してみます。

```
    def f(self):
        return type(self)
```

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

f の引数である self にこのオブジェクトがわたされて、type(self) が返されました。mc.f() には引数はありません。

メソッド f2 を呼び出すには引数が必要です。

```
    def f2(self, param):
        print(self, ':', type(self))
        return param
```

```
In [ ]: mc.f2(99)
```

オブジェクト自身 self とその型 type(self) が print されて、引数がそのまま返されました。

（メソッドでない）属性の設定は、メソッドの中でも行うことができます。このためには、メソッドの中で、

```
self.属性名 = 式
```

という代入を行えばよいわけです。

以下では二次元平面上の点を表すオブジェクトのクラス Point2D を定義します。 このクラスのオブジェクトは、x と y という属性を持ち、それぞれ x 座標と y 座標を表すとします。set_xy というメソッドによって、これらの属性が設定されます。

```
In [ ]: class Point2D:
            def set_xy(self,a,b):
                self.x = a
                self.y = b
            def dist_from_origin(self):
                return (self.x**2 + self.y**2)**0.5


        p = Point2D()
```

```
In [ ]: p
```

```
In [ ]: p.set_xy(3,4)
```

```
In [ ]: p.x, p.y
```

dist_from_origin() というメソッドは、原点からそのオブジェクトが表す点までの距離を返します。その中で、オブジェクトの属性 x と y を参照しています。

```
    def dist_from_origin(self):
```

```
        return (self.x**2 + self.y**2)**0.5
```

```
In [ ]: p.dist_from_origin()
```

6.1.6 オブジェクトの初期化

多くの場合、オブジェクトの生成にあたって、特定の初期状態が設定されることが望まれます。クラス定義に__init__() というメソッドが含まれていれば、オブジェクトの生成時にそれが自動的に呼ばれ、オブジェクトが**初期化**されます。

以下のクラス Point2DInitialized0 では、オブジェクトの生成時に属性 x と y が 0 に設定されます。繰り返しますが、メソッドの最初の引数にはオブジェクト自身がわたされ、これに対応する名前として self を使っています。

```
In [ ]: class Point2DInitialized0:
            def __init__(self):
                self.x = 0          # self は生成されたオブジェクト、ここでは self に x, y 属性を追加して
                self.y = 0          #
            def dist_from_origin(self):
                return (self.x**2 + self.y**2)**0.5


        p0 = Point2DInitialized0()
```

```
In [ ]: p0
```

属性 x と y は、両方とも 0 に初期化されています。

```
In [ ]: p0.x, p0.y
```

6.1.7 引数による初期化

初期化の際に、__init__() に引数をわたして、オブジェクト生成時の初期状態として設定することもできます。具体的には、生成時に与えられる引数が__init_() にわたされます。以下のクラス Point2DInitialized は生成時に x 座標と y 座標を与えるクラスです。

```
In [ ]: class Point2DInitialized:
            def __init__(self, x, y):
                self.x = x
                self.y = y
            def dist_from_origin(self):
                return (self.x**2 + self.y**2)**0.5


        p1 = Point2DInitialized(3, 4)
```

```
In [ ]: p1
```

```
In [ ]: p1.x, p1.y
```

```
In [ ]: p1.dist_from_origin()
```

__init__ の引数として self に加えて x と y が指定されています。

```
self.x = x
```

という代入文の右辺では x という引数が参照されています。引数 x の値が、オブジェクトの x という属性に設定されています。(Python のプログラムではこのように属性名と引数名を同じにすることが多いようです。)

6.1.8 練習

x 軸上を動く点のクラス MovingPoint1D を定義してください。このクラスのオブジェクトは、点の x 座標を保持する x という属性に加えて速度を保持する属性を持ちます。このクラスの初期化メソッドは、x 座標の初期値を引数とします。速度は 1 で初期化します。

forward() というメソッドは属性 x に速度の値を足し込みます。 reverse() というメソッドは速度の符号を反転します。(1 ならば -1 に、-1 ならば 1 にします。)

```
In [ ]: class MovingPoint1D:
            ...
```

以下のセルで動作を確認してください。

```
In [ ]: m = MovingPoint1D(0)
        print(m.x == 0)
        m.forward()
        m.forward()
        print(m.x == 2)
        m.reverse()
        m.forward()
        print(m.x == 1)
```

6.1.9 練習

極座標で二次元平面上の点を表すオブジェクトのクラス Polar を定義してください。 このクラスの初期化メソッドは、原点からの距離と x 軸から左回りの回転角(ラジアン)を引数とします。

さらにこのクラスは、x() と y() というメソッドを持っていて、 それぞれ、点の平面上の x 座標と y 座標を返します。

```
In [ ]: import math

        class Polar:
            ...
```

以下のセルで動作を確認してください。

```
In [ ]: p = Polar(1, math.pi/4)
        print(abs(p.x() - 2**0.5/2) < 0.0001)
        print(abs(p.y() - 2**0.5/2) < 0.0001)
        q = Polar(1, math.pi*(2/3))
```

```
print(abs(q.x() + 0.5) < 0.0001)
print(abs(q.y() - 3**0.5/2) < 0.0001)
```

6.1.10 練習

put(x) と get() というメソッドもつクラス Ring を定義してください。
このクラスのオブジェクトに対して put(x) を呼び出すと引数のデータ x を記録します。
get() を呼び出すと、それまでに put によって記録されたデータのうち、もっとも古いものを返します。そのデータは、最も新しいものとして記録し直されます。
このクラスのオブジェクトは、データのリストを値とする属性を持ちます。その属性は空リストで初期化されます。

```
In [ ]: class Ring:
            ...
```

以下のセルで動作を確認してください。

```
In [ ]: r = Ring()
        r.put(1)
        r.put(2)
        r.put(3)
        print(r.get() == 1)
        r.put(4)
        print(r.get() == 2)
        print(r.get() == 3)
        print(r.get() == 1)
        print(r.get() == 4)
        print(r.get() == 2)
```

6.1.11 ▲属性の検査、追加、変更、削除（組み込み関数による方法）

組み込み関数 hasattr() は属性の有無を検査する組み込み関数です。 属性が存在すれば True、存在しなければ False を返します。たとえば、以下では、属性 ec.name の有無を検査しています。

```
In [ ]: ec.name = 'foo'
```

```
In [ ]: hasattr(ec, 'name')
```

del を用いて属性を削除することができます。del ec.name によって属性を削除すると、以下のようになります。

```
In [ ]: del ec.name
```

```
In [ ]: hasattr(ec, 'name')
```

さらに、属性の操作を行うために、以下のような組み込み関数 setattr() と delattr() を使うことができます。

```
setattr(オブジェクト, 属性文字列, 式)     # 追加・変更
delattr(オブジェクト, 属性文字列)         # 削除
```

これらの組み込み関数により、属性の名前がプログラムを実行してから定まるときにも、属性の操作することができます。すなわち、上の属性文字列を書くべきところで、文字列に評価される式を書くことができます。

```
In [ ]: setattr(ec, 'na'+'me', 'bar')
```

```
In [ ]: ec.name
```

```
In [ ]: delattr(ec, 'NAME'.lower())
```

```
In [ ]: ec.name
```

6.1.12 ▲クラス変数

以下の例のように、クラス定義の直下で変数に代入を行うと、 その変数は

クラス名 . 変数名

によって参照することができます。このような変数を**クラス変数**と呼びます。

```
In [ ]: class StrangeClass:
            n = 1
            def foo(self):
                self.n = 3
```

```
In [ ]: StrangeClass.n
```

そのクラスのオブジェクトを作ると、クラス変数は、オブジェクトの属性としても参照できます。

```
In [ ]: sc = StrangeClass()
```

```
In [ ]: sc.n
```

ところが、その属性の値を変更すると

```
In [ ]: sc.n = 3
```

以下のように、もとのクラス変数とオブジェクトの属性は別ものになるのです。

```
In [ ]: StrangeClass.n
```

```
In [ ]: sc.n
```

メソッドの中で属性の値を変更しても同じです。

```
In [ ]: sc1 = StrangeClass()
```

```
In [ ]: sc1.n
```

```
In [ ]: sc1.foo()
```

```
In [ ]: StrangeClass.n
```

```
In [ ]: sc1.n
```

クラス変数の値を変更してみます。

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

```
In [ ]: sc2 = StrangeClass()
```

```
In [ ]: sc2.n
```

```
In [ ]: sc2.foo()
```

```
In [ ]: sc2.n
```

6.1.13 練習の解答

```
In [ ]: class MovingPoint1D:
            def __init__(self,x):
                self.x = x
                self.v = 1
            def forward(self):
                self.x += self.v
            def reverse(self):
                self.v = -self.v
```

```
In [ ]: import math

        class Polar:
            def __init__(self, r, theta):
                self.r = r
                self.theta = theta

            def x(self):
                return self.r*math.cos(self.theta)

            def y(self):
                return self.r*math.sin(self.theta)
```

```
In [ ]: class Ring:
            def __init__(self):
                self.contents = []
            def put(self,x):
                self.contents.append(x)
            def get(self):
                x = self.contents.pop(0)
                self.contents.append(x)
                return x
```

```
In [ ]:
```

6.2 クラスの継承

6.2.1 基底クラス・派生クラス

6-1 ではユーザが新しくクラスを定義する方法について説明しました。クラスを定義する際にすべてを新しく定義するのではなく、すでに存在する類似したクラスを拡張できれば便利です。そのような拡張はクラスの**継承**によって実現できます。ここで継承されるクラスを**基底クラス**（あるいは**親クラス**）、継承したものを**派生クラス**（あるいは**子クラス**）と呼びます。

実際、Python では多様な処理をおこなうための強力なソフトウェアフレームワークがいくつも用意されています。**フレームワーク**とは、お手本を集めたようなもので、ユーザはお手本と異なるところを手直ししたり追加したりする、すなわちカスタマイズする、ことによって、フレームワークを活用します。このカスタマイズにあたって、ユーザはフレームワーク側で用意された基底クラスを拡張した派生クラスをつくっていく方法が一般的です。

6.2.2 派生クラスの定義

基底クラス BaseClass を継承し、派生クラス DerivedClass を定義するには以下のように行います。

```
class DerivedClass(BaseClass):
    派生クラスに関するプログラム文
```

派生クラスでは、それの基底クラスの属性・メソッドをそのまま利用できるので、派生クラスに依存した属性・メソッドのみをプログラムすれば良いことになります。

以下の例では、2 次元平面上の点を表すオブジェクトのクラス Point2DInitialized を基底クラスとして、 それを継承する派生クラス Point2DPolar を定義しています。
Point2DPolar ではメソッド __init__, dist_from_origin や、属性 x, y は全く定義していませんが、基底クラスのものをそのまま使うことができます。
もちろん新しく定義したメソッド r, theta も使えます。

```
In [ ]: import math                                    # 逆 cos 関数に必要なため

        class Point2DInitialized:
            def __init__(self, x, y):
                self.x = x
                self.y = y
            def dist_from_origin(self):
                return (self.x**2 + self.y**2)**0.5

        class Point2DPolar(Point2DInitialized):
            def r(self):
                return(self.dist_from_origin())
            def theta(self):
                if self.dist_from_origin != 0:
                    return math.acos( self.x / self.dist_from_origin())
                else:
```

```
                return 0
```

```
In [ ]: p2p = Point2DPolar(3, 4)
        p2p
```

```
In [ ]: p2p.x, p2p.y           # Point2DPolar では x, y 属性は定義していないが、基底クラスの属性が使える
```

```
In [ ]: p2p.dist_from_origin() # メソッドも基底クラスのものが使える
```

```
In [ ]: p2p.r()               # 新しく定義したメソッド r()
```

```
In [ ]: p2p.theta()
```

6.2.3 属性・メソッドの上書き

派生クラスで基底クラスの属性・メソッドを上書き（**オーバーライド**）することもできます。以下の例では dist_from_origin を演算子 ** を利用したものから、 数学関数モジュールの平方根を求める関数 math.sqrt に置き換えています。

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

```
        class Point2DPolar2(Point2DPolar):
            def dist_from_origin(self):
                print("new dist_from_origin")
                return math.sqrt(self.x**2 + self.y**2)
```

```
In [ ]: p2p2 = Point2DPolar2(3, 4)
        p2p2.dist_from_origin()
```

6.2.4 基底クラスのメソッド呼び出し

派生クラスのメソッドから基底クラスのメソッド・属性を呼びだしたいことはよくあります。このような場合はメソッド super() を利用します。

以下は Point2DPolar を継承したクラス、Point2DCached の例です。 Point2DCached では x, y が与えられる初期化時に極座標も先に計算します。

派生クラス Point2DCached では 2 行目からのメソッド定義で、基底クラスの__init__ を置き換えています。 そして、メソッド定義の 3 行目で基底クラスの初期化メソッドを super().__init__(x,y) を使って呼び出しています。

```
In [ ]: class Point2DCached(Point2DPolar):
            def __init__(self, x, y):
                super().__init__(x, y)
                self.r_c = (x*x + y*y)**0.5
                self.theta_c = self.theta()
            def dist_from_origin(self):
                return self.r_cl
```

```
In [ ]: p2c = Point2DCached(3, 4)
        p2c.x, p2c.y
```

```
In [ ]: p2c.r_c, p2c.theta_c
```

6.2.5 練習

Point2DInitialized の派生クラスとして MovingPoint2D を定義してください。このクラスのオブジェクトは、Point2DInitialized の属性に加えて、速度ベクトルの絶対値と速度ベクトルの x 軸からの左回りの回転角を属性として持ちます。このクラスの初期化メソッドは、Point2DInitialized と同様に x 座標と y 座標の初期値を引数とします。 速度ベクトルの絶対値は 1 で、x 軸からの回転角は 0 で初期化します。

forward というメソッドは、x 座標と y 座標の位置ベクトルに速度ベクトルを足し込みます。 rotate(a) というメソッドは、速度を左回りに a（ラジアン）だけ回転します。

```
In [ ]: import math

        class Point2DInitialized:
            def __init__(self, x, y):
                self.x = x
                self.y = y
            def dist_from_origin(self):
                return (self.x**2 + self.y**2)**0.5

        class MovingPoint2D(Point2DInitialized):
            ...
```

以下のセルで動作を確認してください。

```
In [ ]: m = MovingPoint2D(0,0)
        print(m.x == 0 and m.y == 0)
        m.forward()
        m.forward()
        print(m.x == 2 and m.y == 0)
        m.rotate(math.pi/2)
        m.forward()
        print(abs(m.x - 2) < 0.0001 and abs(m.y - 1) < 0.0001)
```

6.2.6 ▲継承関係を検査する関数

issubclass(clas1, class2) は class1 が class2 の継承関係を検査する組み込み関数です:

```
In [ ]: issubclass(Point2DCached, Point2DPolar)        # Point2DCached は  Point2DPolar を継承している

In [ ]: issubclass(Point2DPolar, Point2DCached)        # 逆は正しくない

In [ ]: issubclass(Point2DCached, Point2DInitialized) # 継承関係が入れ子になっていても検査できます
```

6.2.7 練習の解答

```
In [ ]: import math

        class Point2DInitialized:
            def __init__(self, x, y):
                self.x = x
                self.y = y
            def dist_from_origin(self):
                return (self.x**2 + self.y**2)**0.5

        class MovingPoint2D(Point2DInitialized):
            def __init__(self, x, y):
                super().__init__(x,y)
                self.r = 1
                self.theta = 0
            def forward(self):
                self.x += self.r*math.cos(self.theta)
                self.y += self.r*math.sin(self.theta)
            def rotate(self, a):
                self.theta += a
```

6.3 ▲イテレータとジェネレータ

ここでの内容は全てが参考資料（▲）としての扱いです。

6.3.1 イテレータの定義

第 2 回で学んだリスト、タプル、文字列などは、それらの要素を一つずつ取り出す処理をサポートするオブジェクトです。このようなオブジェクトを**イテラブルオブジェクト** (iterable object) と呼びます。

イテラブルオブジェクトに対して、組み込みの `iter` 関数を適用すると、イテラブルオブジェクトからイテレータを作成することができます。**イテレータ**は、その要素を一つずつ取り出す`__next__` メソッドを持つオブジェクトで、データを逐次処理する際に用います。組み込みの `next` 関数を用いて、イテレータの `__next__` メソッドを呼び出すことで、`iter` 関数にわたしたもとのイテラブルオブジェクトの各要素を一つずつ取り出していくことができます。

```
In [ ]: #イテラブルオブジェクトであるリストから iter 関数によりイテレータを作成
        it = iter([1,2,3])

In [ ]: #データ型がイテレータとなっている
        type(it)

In [ ]: #イテレータの要素を 1 つ取り出し
        next(it)

In [ ]: #イテレータの要素を 1 つ取り出し
        next(it)
```

```
In [ ]: #イテレータの要素を 1 つ取り出し
        next(it)
```

イテレータで全ての要素を取り出した後は、StopIteration の例外（エラー）が発生します。

```
In [ ]: #取り出す要素がないのでイテレータは例外を発生
        next(it)
```

イテレータは list 関数を用いてリストにすることができます。

```
In [ ]: it = iter([1,2,3])
        list(it)
```

6.3.2 イテラブルオブジェクトとイテレータ

イテラブルオブジェクトもイテレータも、どちらも要素を順に取り出すことが可能なオブジェクトですが、イテラブルオブジェクトは要素が取り出された状態を保持しないのに対して、イテレータは要素の取り出しごとに、どこまで取り出されたかという状態を保持しています。これにより、イテレータでは、前回の取り出しの続きから要素を取り出すことや、別々の場所から要素を取り出すといったことが可能になります。

このようにイテレータを用いて、処理の対象を逐次処理することで、例えば大きなデータを全てロードして for 文などで処理するよりも、効率的な処理が可能になります。特に、3-6 で説明した関数型プログラミングでは、データをイテレータで扱うのが一般的です。なお、ここでは、イテラブルオブジェクトからイテレータを作成する例を示しましたが、より一般的にイテレータのように要素を一つずつ取り出すような任意の処理を記述するには、以下で説明するジェネレータを用います。

6.3.3 イテレータと for ループ

イテレータは、for 文に適用して要素を取り出していくこともできます。range や、関数型プログラミングの回 3-6 で説明した map,filter,zip などの組み込み関数は、イテレータを返すので、for 文で処理ができます。

for 文の in の後には、イテレータかイテラブルオブジェクトを指定します。リスト、タプル、文字列、range などのシーケンス型オブジェクトはイテラブルオブジェクトです。 in の後にイテラブルオブジェクトを指定した場合は、iter 関数が適用されてイテレータが作られます。 通常、イテレータに iter を適用すると自分自身が返ります。

for 文の繰り返しごとにそのイテレータの __next__ メソッドが呼ばれ要素が取り出されます。上記の StopIteration 例外が発生するまですべての要素の取り出しが行われます。

try と except についてはここでは説明しませんが、

```
for i in x:
    ...
```

という for 文は、以下のようにして実行されます。

```
try:
    it = iter(x)
    while True:
        i = next(it)
        ...
except StopIteration as e:
    pass
```

```
In [ ]: #イテレータの要素を for 文で 1 つずつ取り出し
        it = iter([1,2,3])
        for num in it:
            print(num)
```

enumerate 関数を用いることで、イテレータの要素の取り出しとともに、カウントすることができます。

```
In [ ]: #イテレータの要素を for 文で 1 つずつ取り出し、合わせて取り出し回数も i でカウント
        #カウントは 0 から始まることに注意
        it = iter([1,2,3])
        for i, num in enumerate(it):
            print(i, num)
```

6.3.4 イテレータと組み込み関数・演算子

組み込み関数や演算子を使って、イテレータの要素の取り出しや確認ができます。

```
In [ ]: #max 関数はイテレータの中で最大の要素を返します
        it = iter([1,2,3])
        max(it)
```

もちろん、わざわざイテレータにしなくても構いません。

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

```
In [ ]: #min 関数はイテレータの中で最大の要素を返します
        it = iter([1,2,3])
        min(it)
```

```
In [ ]: #all 関数はイテレータの全ての要素が真ならば True を返します。
        it = iter([True,False,True])
        all(it)
```

```
In [ ]: #any 関数はイテレータのいずれかの要素が真ならば True を返します。
        it = iter([True,False,True])
        any(it)
```

```
In [ ]: #in 演算子はイテレータに指定の要素が含まれれば True を返します。
        it = iter([1,2,3])
        0 in it
```

```
In [ ]: #not in 演算子はイテレータに指定の要素が含まれなければ True を返します。
        it = iter([1,2,3])
        0 not in it
```

6.3.5 クラスとイテレータ

これまで説明したように、for 文では、組み込み関数 iter() を呼び、イテラブルオブジェクトからイテレータを作成します。その後、イテレータに対して next() を繰り返し呼び、要素を一つずつ取り出しています。

ここでは、このような反復処理をサポートするクラスの定義方法を説明します。そのためには、二つのメソッド__iter__() と__next__() を定義します。

前者の__iter__() はオブジェクトに対応するイテレータを返します。__next__() を定義する場合は、オブジェクトそのもの、すなわち self を返すようにします。

後者の __next__() は、呼ばれるたびにデータを一つずつ返します。返すデータがなくなれば、raise StopIteration で例外（エラー）を送出します。例外を受けてプログラムは for ループの繰り返しから脱けます。

具体例として、以下の折れ線を扱うクラス Chain を考えます。Chain は座標のタプルのリストを値とする series 属性をもちます。

```
In [ ]: class Chain():
            def __init__(self, series):
                self.series = series
                self.index = 0
            def __iter__(self):
                return self
            def __next__(self):
                if self.index == len(self.series):
                    raise StopIteration
                self.index += 1
                return self.series[self.index - 1]
```

ここで、オブジェクトを生成し for ループでアクセスしてみます。

```
In [ ]: ch = Chain([(0,0), (0,1), (1,1), (1,0)])
        for p in ch:
            print(p)
```

ただし、一度繰り返しが終了すると、属性 index が最大値となっているためもう繰り返しは行われません。

```
In [ ]: for p in ch:
            print(p)
```

6.3.6 ジェネレータ

これまでは、リスト、タプル、文字列などのイテラブルオブジェクトからイテレータを作成する方法を見てきました。一方、イテレータのように要素を一つずつ取り出せるオブジェクトを自分で作成することも可能です。そのためには、**ジェネレータ関数**を定義します。通常の関数では return で戻り値を返しますが、ジェネレータ関数では、yield で戻り値を指定します。すると、yield はイテレータとなるオブジェクト（**ジェネレータイテレータ**）を返します。

yield 戻り値

イテレータと同じように、ジェネレータ関数で返されたジェネレータイテレータに対しては、組み込みの next 関数や for 文で、その要素を一つずつ取り出すことができます。全ての要素を取り出した後でジェネレータ関数が return を実行するか関数定義の最後に到達すると、StopIteration の例外が発生します。

```
In [ ]: #1,2,3 を順番に返すジェネレータ関数
        def gen():
            yield 1 #1 を返し、次に呼ばれるまで処理を中断
            yield 2 #2 を返し、次に呼ばれるまで処理を中断
```

```
        yield 3 #3を返し、処理を終了

    #ジェネレータ関数の返すジェネレータイテレータをfor文で処理
    for num in gen():
        print(num)
```

このように、yieldのたびに、ジェネレータは現在の実行状態を記憶して、処理を一時的に中断します。そして、再びジェネレータが呼ばれると、中断した箇所から処理を再開します。これは、通常の関数が実行の度に新たな状態から開始するのと異なることに注意してください。

6.3.7 ジェネレータと内包表記

内包表記を用いて、リスト内包表記のように、ジェネレータも内包表記で表現し、ジェネレータイテレータオブジェクトを生成することができます。その際、()で囲います。

```
In [ ]: #1から3のべき乗を返すジェネレータの内包表記
        gen = (i*i for i in range(1,4))

        type(gen)
```

```
In [ ]: gen = (i*i for i in range(1,4))

        #ジェネレータ式の返すジェネレータイテレータをfor文で処理
        for num in gen:
            print(num)
```

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 7-1 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 参考

7.1.12 ▲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)
```

7.1.13 ▲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
```

7.1.14 ▲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()
```

7.1.15 ▲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