目次

第2章

物価指数とインフレーション

2.1 概要

この講義では以下のことを学ぶ。

- 理論
  - ベクトルの復習
  - 価格指数の定義
  - 連鎖指数の計算方法
- プログラミング
  - リスト，タプル
  - **NumPy** の配列
  - **NumPy** の関数
  - **Matplotlib** を用いた可視化

実質 GDP の定義は次章を参照のこと。

2.2 理論：ベクトル

数の集合として実数全体の集合 $\mathbb{R}$ を考える。ひとまず，ベクトルとは一定の個数分だけ数字を並べたものと考えておこう。

$$\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_d \end{bmatrix}, \qquad x_1, x_2, \ldots, x_d \in \mathbb{R}$$

自然数 d は並べた数字の個数を表す。これは状況に応じて必要な数を固定する。縦に並べると余白が増えてしまうので，横に並べて省スペース化できる方が便利である。

$$\boldsymbol{x} = (x_1, x_2, \ldots, x_d).$$

縦に数字を並べると要素ごとの足し算が見やすくなったり，サイズが $n \times 1$ である行列と同一視できるので行列との積の計算に大変都合がよかったりといった利点がある。本書では，状況に応じて都合のよい方を使う。どうしても「数を横に並べたベクトル（サイズが $1 \times n$ の行列）」を書く必要があるときには，

$$\begin{bmatrix} x_1 & x_2 & \cdots & x_n \end{bmatrix}$$

のように，角括弧でコンマを付けずに書くことにしよう。なお，ベクトルを表す記号には太字を用いることが多いが，これも絶対に必要という訳ではない。

固定した d について，ベクトルの成分 $x_1, \ldots, x_d$ の組み合わせは無数に存在する。これらのベクトルを全部を集めた集合をベクトル空間と呼ぶ。$\mathbb{R}$ に含まれる数字が d 個並んでいるので，$\boldsymbol{x}$ を d 次元ベクトル，$\mathbb{R}^d$ を d 次元ベクトル空間という。d は次元（dimension）に由来する。

ベクトルの演算

ベクトル空間においては，和（足し算）とスカラー倍（定数倍）の 2 つの演算が基本的である。$\boldsymbol{x},\boldsymbol{y} \in \mathbb{R}^d$，$\alpha,\beta \in \mathbb{R}$ に対して，

$$\alpha\boldsymbol{x} + \beta\boldsymbol{y} = \alpha \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_d \end{bmatrix} + \beta \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_d \end{bmatrix} = \begin{bmatrix} \alpha x_1 + \beta y_1 \\ \alpha x_2 + \beta y_2 \\ \vdots \\ \alpha x_d + \beta y_d \end{bmatrix}$$

と定義する。最右辺も数字を d 個並べたベクトルであることに注意しよう。したがって，$\alpha\boldsymbol{x} + \beta\boldsymbol{y}$ も $\mathbb{R}^d$ の元である。

例 2.1. $d = 3$ としよう。

$$\boldsymbol{x} = \begin{bmatrix} 1 \\ 3 \\ -3 \end{bmatrix}, \quad \boldsymbol{y} = \begin{bmatrix} 2 \\ 0 \\ 1 \end{bmatrix}$$

に対して，

$$\boldsymbol{x} + \boldsymbol{y} = \begin{bmatrix} 3 \\ 3 \\ -2 \end{bmatrix}$$

$\alpha = -1$ であれば，

$$\alpha\boldsymbol{x} = \begin{bmatrix} -1 \\ -3 \\ 3 \end{bmatrix}$$

ベクトル同士の引き算は，和とスカラー倍を組み合わせて，

$$\boldsymbol{x} - \boldsymbol{y} = \boldsymbol{x} + (-1)\boldsymbol{y}$$

とできる。これは成分ごとに引き算をしているだけである。

内積

ベクトル $\boldsymbol{x},\boldsymbol{y} \in \mathbb{R}^d$ に対して，

$$\boldsymbol{x} \cdot \boldsymbol{y} = x_1 y_1 + x_2 y_2 + \cdots + x_d y_d = \sum_{i=1}^{d} x_i y_i$$

で定義される $\boldsymbol{x} \cdot \boldsymbol{y} \in \mathbb{R}$ を内積という[*1]。経済学の文脈では，価値を計算する場合に内積が使われる。

例 2.2. 経済に存在する財を 3 タイプに分類したとしよう。財 1，財 2，財 3 の価格を $\boldsymbol{p} = (p_1, p_2, p_3)$，取引量を $\boldsymbol{x} = (x_1, x_2, x_3)$ とする。このとき，取引総額は

$$\boldsymbol{p} \cdot \boldsymbol{x} = p_1 x_1 + p_2 x_2 + p_3 x_3$$

である。

内積はベクトル同士の角度を表すのに使うことを覚えているかもしれない。内積はベクトルの直交関係を規定する重要な概念だが，そのような解釈は必要になったら説明することにしよう[*2]。

注意

成分ごとの掛け算や割り算が使われることは少ないので，ここでも特別な記号は用意しない。しかし，**NumPy** のベクトル（後述）同士では乗算・除算が定義されていて，通常の乗算記号* や除算記号/ を用いる。これらは成分ごとの演算であることに注意しておこう。

[*1] 分野によって，$(\boldsymbol{x}, \boldsymbol{y})$,$\langle \boldsymbol{x}, \boldsymbol{y} \rangle$, あるいは $\langle \boldsymbol{x} | \boldsymbol{y} \rangle$ などと書く。

[*2] 今後，直交行列に関する議論が必要になるので，そのときに説明する。

2.3　理論：価格指数

2.3.1　パーシェ指数とラスパイレス指数

物価指数計算の基本は，各財の価格変化率の加重平均である。パーシェ式とラスパイレス式という計算方法が代表的である。

財・サービスの種類を $i = 1, 2, \ldots, N$ の N 個の財グループに分類する。個々の財グループについては代表的な価格が定まっていると考える[3]。t 期の財グループ i の価格を $p_{t,i}$ と書く。1 つのグループについて，$t = 0$ 時点からの価格の粗変化率は

$$\frac{p_{t,i}}{p_{0,i}}$$

と書ける。

複数の財が存在する経済で，マクロ経済の価格を代表する「物価」，あるいはその変化，をどのように定義すればよいだろうか。ベクトルとベクトルの割り算は定義していないので，

$$\frac{\boldsymbol{p}_t}{\boldsymbol{p}_0} \qquad \text{(誤り！)}$$

とする訳にはいかないし，単純に算術平均

$$\frac{1}{N}\sum_{i=1}^{N}\frac{p_{t,i}}{p_{0,i}} \qquad \text{(一般的には正しくない！)}$$

を取ることでは平均的な物価の変化を捉えることもできない。マーケットの小さい財の価格が大幅に変化することを想像すれば問題点が明確になるだろう。マクロの経済活動にほとんど影響していない（マーケットが小さいから）にも関わらず，算術平均に大きな影響を与えてしまうからだ。

[3] 実務上は各品目について平均価格を計算するというステップと，全品目を総合してマクロ経済全体の価格指数を計算するというステップが必要である。ここでは，2 つ目のステップについての議論であると考えればよい。

すべての財に共通の重み $1/N$ をかけていることが問題なので，マーケットの大きさに応じた重みを掛ければよい。つまり，

$$\sum_{i=1}^{N} w_i \frac{p_{t,i}}{p_{0,i}}$$

のように計算すればよい。なお，重み $w_i, i = 1, \ldots, N$, は次の性質を満たす。

$$\sum_{i=1}^{N} w_i = 1.$$

パーシェ式とラスパイレス式の指数には重みの定義において違いがある。

ラスパイレス指数

財 i の $t = 0$ 時点における取引量を $x_{0,i}$，$i = 1, \ldots, N$ についてベクトルにまとめて $\boldsymbol{x}_0 = (x_{0,1}, \ldots, x_{0,N})$ と書く。この当時の 財 i の市場価値（取引額）は $p_{0,i}x_{0,i}$ だった。すべての財について価値を合計したものは，内積を用いて次のように書くことができる。

$$\sum_{j=1}^{N} p_{0,j}x_{0,j} = \boldsymbol{p}_0 \cdot \boldsymbol{x}_0$$

これは，分析対象とするすべての財について $t = 0$ 期の総取引額を計算したものである。この総取引額に占める財 i のシェアを w_i^L としよう。すなわち，

$$w_i^L = \frac{p_{0,i}x_{0,i}}{\sum_{j=1}^{N} p_{0,j}x_{0,j}} = \frac{p_{0,i}x_{0,i}}{\boldsymbol{p}_0 \cdot \boldsymbol{x}_0}$$

この w_i^L は「重み」が持つべき性質を持っている。なぜなら，

$$\sum_{i=1}^{N} w_i^L = \sum_{i=1}^{N} \left(\frac{p_{0,i}x_{0,i}}{\boldsymbol{p}_0 \cdot \boldsymbol{x}_0} \right) = \frac{\boldsymbol{p}_0 \cdot \boldsymbol{x}_0}{\boldsymbol{p}_0 \cdot \boldsymbol{x}_0} = 1.$$

このように定義される重み $\boldsymbol{w}^L = (w_1^L, \ldots, w_N^L)$ を用いて計算する加重平均を**ラスパイ**

レス指数 (Laspeyres index) と呼ぶ。

$$\begin{aligned}(\text{ラスパイレス指数}) &= \sum_{i=1}^{N} w_i^L \frac{p_{t,i}}{p_{0,i}} \\ &= \sum_{i=1}^{N} \left(\frac{p_{0,i}x_{0,i}}{\boldsymbol{p}_0 \cdot \boldsymbol{x}_0} \right) \frac{p_{t,i}}{p_{0,i}} \\ &= \frac{\sum_{i=1}^{N} p_{t,i}x_{0,i}}{\boldsymbol{p}_0 \cdot \boldsymbol{x}_0} \\ &= \frac{\boldsymbol{p}_t \cdot \boldsymbol{x}_0}{\boldsymbol{p}_0 \cdot \boldsymbol{x}_0}. \end{aligned} \tag{2.1}$$

通常，このようにして得られた数に 100 を掛けて，基準化する。

パーシェ指数

財 i の t 期における取引量を $x_{t,i}$ と書くことにする。ベクトルにまとめて，$\boldsymbol{x}_t = (x_{t,1}, \ldots, x_{t,N})$ とする。次に考える重みは次のようなものである。

$$w_i^P = \frac{p_{0,i}x_{t,i}}{\sum_{j=1}^{N} p_{0,j}x_{t,j}} = \frac{p_{0,i}x_{t,i}}{\boldsymbol{p}_0 \cdot \boldsymbol{x}_t}.$$

これは t 期における市場の規模を，0 期の価格で評価したものになっている。この w_i^P は「重み」が持つべき性質を持っている。すなわち，

$$\sum_{i=1}^{N} w_i^P = \sum_{i=1}^{N} \left(\frac{p_{0,i}x_{t,i}}{\boldsymbol{p}_0 \cdot \boldsymbol{x}_t} \right) = \frac{\boldsymbol{p}_0 \cdot \boldsymbol{x}_t}{\boldsymbol{p}_0 \cdot \boldsymbol{x}_t} = 1.$$

このように定義される重み $\boldsymbol{w}^P = (w_1^P, \ldots, w_N^P)$ を用いて計算する加重平均を**パーシェ指数** (Paasche index) と呼ぶ。

$$
\begin{aligned}
(\text{パーシェ指数}) &= \sum_{i=1}^{N} w_i^P \frac{p_{t,i}}{p_{0,i}} \\
&= \sum_{i=1}^{N} \left(\frac{p_{0,i} x_{t,i}}{\boldsymbol{p}_0 \cdot \boldsymbol{x}_t} \right) \frac{p_{t,i}}{p_{0,i}} \\
&= \frac{\boldsymbol{p}_t \cdot \boldsymbol{x}_t}{\boldsymbol{p}_0 \cdot \boldsymbol{x}_t}
\end{aligned} \tag{2.2}
$$

通常，このようにして得られた数に 100 を掛けて，基準化する。

ラスパイレス指数とパーシェ指数の共通点と相違点

ラスパイレス指数とパーシェ指数ともに次のような形式をもつ物価変化の指標である。

$$
\text{物価変化率} = \frac{\boldsymbol{p}_t \cdot \square}{\boldsymbol{p}_0 \cdot \square}
$$

パーシェ式の物価指数は $\square$ の中に比較年（t）の数量を用いる。一方，ラスパイレス式の物価指数は $\square$ の中に基準年（0）の数量を用いる。パーシェとラスパイレスは次のようなニモニックで覚えるとよい[4]。

パーシェ式	Paasche	Present（今）の数量を使う
ラスパイレス式	Laspeyres	Long time ago（昔）の数量を使う

価格を評価するために補助的に用いる数量データの違いによって，2 つの指数には情報収集コストに違いが生じる。

問題 2.1. ラスパイレス式の物価指数はパーシェ式の物価指数よりも物価の変化を迅速に捉えることができる。これは，なぜか。情報収集のコストの観点から説明しなさい。

[4] E. Wayne Nafziger,*Economic Development*

Answer

2.3.2　バイアス

ラスパイレス式であってもパーシェ式であっても，特定の一時点の数量を用いて物価比較をする限り，価格の変化に対する消費の変化を完全に捉えることは容易ではない。なぜなら，価格が上昇した財の需要は減少して相対的に値下がりした別の財の需要が増加するのが通常の消費行動であるにも関わらず，上記の 2 つの物価指数はこのような代替効果を無視しているからだ。実際，ラスパイレス式の指数には物価の上昇を過大評価する傾向があり，パーシェ式の指数には過小評価する傾向があると言われている。

ある財 i が急激に値上がりしたとして，その財から別の財への代替が進んだとする。ラスパイレス指数の重み

$$w_i^L = \frac{p_{0,i}x_{0,i}}{\boldsymbol{p}_0 \cdot \boldsymbol{x}_0}$$

には代替による効果は反映されないので，$p_{t,i}/p_{0,i}$ の影響が大きく残る。一方，パーシェ指数の重み

$$w_i^P = \frac{p_{0,i}x_{t,i}}{\boldsymbol{p}_0 \cdot \boldsymbol{x}_t}$$

は，財 i の価格 $p_{t,i}$ が上昇したことで起こる需要減で $x_{t,i}$ が減少する効果は反映するものの，重みを計算する価格は $p_{0,i}$ という小さい数字のままである。したがって，$p_{t,i}/p_{0,i}$ に対する重みが小さくなってしまう。

フィッシャー指数

ラスパイレス指数の上方バイアス，パーシェ指数の下方バイアスという欠点を補うために，これらの平均を取った物価指数が考案されている。**フィッシャー指数**はパーシェ式とラスパイレス式の相乗平均を用いて，次のように定義される。

$$(\text{フィッシャー指数}) = \sqrt{\underbrace{\frac{\boldsymbol{p}_t \cdot \boldsymbol{x}_t}{\boldsymbol{p}_0 \cdot \boldsymbol{x}_t}}_{\text{パーシェ}} \times \underbrace{\frac{\boldsymbol{p}_t \cdot \boldsymbol{x}_0}{\boldsymbol{p}_0 \cdot \boldsymbol{x}_0}}_{\text{ラスパイレス}}}.$$

通常，このようにして得られた数に 100 を掛けて，基準化する。

2.3.3 インフレーションとデフレーション

ここまでに計算した物価指数，時点 0 を基準にして，時点 t の物価がどれくらい高いかを示す指標である。$\boldsymbol{x}_* = \boldsymbol{x}_0 \text{ or} \boldsymbol{x}_t$ として，価格指数を P_t と書こう。

$$P_t = \frac{\boldsymbol{p}_t \cdot \boldsymbol{x}_*}{\boldsymbol{p}_0 \cdot \boldsymbol{x}_*}$$

このようにすれば，平均物価の変遷を表す時系列を作ることが出来る。

$$P_0 = 1, \quad P_1, \quad P_2, \quad \cdots, \quad P_t, \quad \cdots$$

P_t が継続的に上昇し続ける状況を**インフレーション**，継続的に下落し続ける状況を**デフレーション**と呼んでいる。マクロ経済に多数ある財が総じて値上げする状況でインフレーションが起こり，総じて値下げされる状況でデフレーションが起こる。「物価の安定」といったときには，インフレーションのスピード（**インフレ率**）

$$\frac{\Delta P_t}{P_{t-1}} = \frac{P_t - P_{t-1}}{P_{t-1}} = \frac{P_t}{P_{t-1}} - 1$$

がゼロに近い正の値になるような政策的な取り組みを表すことが多い[*5]。

[*5] インフレ率がゼロになってしまうと外的なショックの影響で不況になったときに金融緩和余地がなく苦境に立たされてしまう。

2.3.4　固定基準年方式と連鎖方式

上のように定義した P_t は t が大きくなるにつれて，経済活動の実態に合わなくなってくる。例えば，ラスパイレス指数の場合，

$$P_t^{\text{Laspeyres}} = \frac{\boldsymbol{p}_t \cdot \boldsymbol{x}_0}{\boldsymbol{p}_0 \cdot \boldsymbol{x}_0}$$

の分子にある $\boldsymbol{p}_t \cdot \boldsymbol{x}_0$ は価格 $\boldsymbol{p}_t$ のもとでは全く現実味のない数量ベクトル $\boldsymbol{x}_0$ を使ってしまっているかもしれない。パーシェ指数の場合にも，

$$P_t^{\text{Paasche}} = \frac{\boldsymbol{p}_t \cdot \boldsymbol{x}_t}{\boldsymbol{p}_0 \cdot \boldsymbol{x}_t}$$

の分母にある $\boldsymbol{p}_0 \cdot \boldsymbol{x}_t$ が価格 $\boldsymbol{p}_0$ のもとでの経済活動では起こりそうにもない水準になるかもしれない。経済に新しい財がどんどん追加されたり，財の質的な向上と価格下落が著しい時代にあっては，このような問題が深刻になる。基準数量 $\boldsymbol{x}_0$ や基準価格 $\boldsymbol{p}_0$ を固定した前述のような計算方式を**固定基準年方式**と呼ぶ。このような問題に対処する 1 つの方法は，数量ベクトル $\boldsymbol{x}_0$ を頻繁に改定することである。例えば，代表的な物価指数である消費者物価指数では，マクロ経済における標準的な購入量を 5 年ごと見直して，ラスパイレス指数を計算している。基準年を変更することを，基準改定という。改定前と改定後の数値は直接接続しないことに注意をしなければならない。

固定基準年方式に代わる方式として，連鎖方式と呼ばれる算式が用いられることもある。連鎖方式の指数は，前年度の連鎖指数に前年度基準の価格指数を掛けることで計算される。つまり，

$$\text{ラスパイレス式連鎖指数}_t = \text{ラスパイレス式連鎖指数}_{t-1} \times \frac{\boldsymbol{p}_t \cdot \boldsymbol{x}_{t-1}}{\boldsymbol{p}_{t-1} \cdot \boldsymbol{x}_{t-1}}$$

$$\text{パーシェ式連鎖指数}_t = \text{パーシェ式連鎖指数}_{t-1} \times \frac{\boldsymbol{p}_t \cdot \boldsymbol{x}_t}{\boldsymbol{p}_{t-1} \cdot \boldsymbol{x}_t}$$

$$\text{フィッシャー式連鎖指数}_t = \text{フィッシャー式連鎖指数}_t \times \sqrt{\frac{\boldsymbol{p}_t \cdot \boldsymbol{x}_{t-1}}{\boldsymbol{p}_{t-1} \cdot \boldsymbol{x}_{t-1}} \times \frac{\boldsymbol{p}_t \cdot \boldsymbol{x}_t}{\boldsymbol{p}_{t-1} \cdot \boldsymbol{x}_t}}$$

GDP デフレーターはパーシェ式の連鎖指数である。なお，連鎖指数の初期値（$t = 0$ の

値）は 1 と設定する（指数を 100 で基準化する場合は 100）。

$$\text{連鎖指数}_0 = 1$$

関連する言葉の整理をしておこう。

比較年 物価の水準を観察したい年のこと。上の式では t に相当する。

参照年 指数を 1（または 100）に基準化する時点のこと。上の例では，0 期に相当する。

基準年 比較年の物価上昇率を計算するための基準となる年のこと。固定基準年方式の場合は参照年と同じで，0 期を指す。連鎖方式の場合は比較年の前年，つまり $t-1$ が基準年である。

GDP 統計には，さらに，体系基準年という概念がある。GDP 統計を作成するための基礎統計（産業連関表，国勢統計など）の更新に合わせて GDP 統計も基準数量が変更される。基礎統計が作成された年が体系基準年である。

2.4 プログラミング: NumPy 入門

この章では，複数の数字をまとめたオブジェクトである，リストとタプルを導入し，その後 **NumPy** の **ndarray** オブジェクトを説明する。数学的なベクトルや行列を **ndarray** で表現できるので，Python を用いた数値計算では必須の話題である。

2.4.1 リストとタプル

Python でベクトルや時系列を表現するための基礎となるデータ形式はリストやタプルと呼ばれるオブジェクトである。「基礎となる」と言ったのは，これらをそのまま使う訳ではないからだ。しかし，リストやタプルをすっ飛ばしてベクトルの表現を説明することはできないので，しばらく辛抱してほしい。

リスト

リストは複数の種類のデータをひとまとめにしたデータ形式である。コンマでつないで角括弧で括る。

```
x = [5, 8, 9, -1]
x
```

```
[5, 8, 9, -1]
```

和やスカラー倍は期待通りに動かないので，これはベクトルのようには使えない。

```
x + x
```

```
[5, 8, 9, -1, 5, 8, 9, -1]
```

```
3 * x
```

```
[5, 8, 9, -1, 5, 8, 9, -1, 5, 8, 9, -1]
```

問題 2.2. リスト同士の和，リストと自然数の積は Python ではどのように定義されているか。色々な入力を試して実験し，自分の言葉で説明しなさい。

問題 2.3. リストと小数の積を実行しようとするとどのような結果になるだろうか。結果を予想して，実行し，結果を記録しなさい[*6]。

```
3.5 * x
```

Answer

[*6] 面倒かもしれないが，「結果を予想」するというステップを飛ばさないでほしい。あなた自身が自分のためのコードを書くときに予想する能力が必要になる。

Python のリストを数学的なベクトルを表現するために使いにくい理由は，リストはベクトルよりも遥かに柔軟なデータ構造だからだ。数学的なベクトルは同種の数字しか並べることができないが[*7]，Python のリストは様々な種類のオブジェクトを並べることができる。

```
y = [10, 4.3, "Hello", [1, 2, 3]]
y
```

```
[10, 4.3, 'Hello', [1, 2, 3]]
```

上で定義したリスト `y` は次のような要素を持つ。

- 左から 1 つ目の要素は，整数 `10`
- 2 つ目の要素は小数 `4.3`
- 3 つ目の要素はテキスト `"Hello"`（1 重引用符や 2 重引用符で囲ったテキストは「文字列」と呼ばれる種類のオブジェクトになる）
- 4 つ目の要素はリスト `[1, 2, 3]`。そのリストには整数が 3 つ並んでいる。

要素に何が入るのかが分からないのであれば，要素ごとの加算や乗算を定義できないことは容易に想像が付くだろう。リストは意味のあるひとまとまりのデータを「記録」するために使うことができる。しかし，同種の数字だけ並べて「計算」したいならもっとよい方法がある。

個別のデータを取得する方法が気になることだろう。角括弧を使って，インデックス（データにおける要素の位置）を次のように指定すればよい。

```
x[0]
```

```
5
```

```
x[-1]
```

```
-1
```

[*7] 前節の解説では「同種」の部分は完全に割愛している。具体的には実数，複素数，有理数などである。気になった読者は線形代数学の教科書を参照せよ。

```
y[-2]
```

```
'Hello'
```

```
y[3][1]
```

```
2
```

次のことを覚えておこう。インデックスと中身を見比べてみよう。

- インデックスは 0 から始まる整数である。
- リストの最後の要素から数える場合には，-1,-2, ... という負のインデックスを使うことができる。
- リストの長さは len() で取得できる。

```
len(y)
```

```
4
```

リストの内容を変更するときには要素取得と同じ記法を用いる。

```
x[0] = 0
x
```

```
[0, 8, 9, -1]
```

オブジェクトに対応付けられた x や y という記号は，オブジェクトそのものではなくオブジェクトを呼びだすための名前に過ぎない。これは次のような実験からよく分かる。オリジナルの x には手を触れずにコピーした z だけを編集しようとしたのだけど，x にまで修正が及んでしまっている。このコードは失敗だ。

正しくは，次のように書かなければいけない。コロンは「スライス」と呼ばれる操作に関連している。いずれ詳しく説明することになるだろう。

なお，要素のないリストや，単一要素のリストも作ることができる。

```
[]
```

```
[]
```

```
[1]
```

```
  [1]
```

タプル

タプルとリストの違いとして次の 2 点を押さえておけばよい。

- 丸括弧で括る。
- あとから要素を変更できない。

```
u = (1, 2, 3)
u
```

```
  (1, 2, 3)
```

```
u[2]
```

```
  3
```

丸括弧は計算順序の変更という他の意味があるので，単一の要素を持つタプルを作成するときには注意が必要である。

```
()
```

```
  ()
```

```
(1,)
```

```
  (1,)
```

問題 2.4. タプルの要素を変更しようとするとどのような結果になるだろうか。下のコードを実行し，結果を記録しなさい。

```
u[0] = 0
```

Answer

数値計算だけに限定すれば，タプルを自分で作らなければならない状況というのは少ないかもしれない[8]。しかし，出力結果がタプルとなるケースは非常に多いので出力には慣れておこう。

2.4.2　NumPy の配列

さて，準備が整ったので本題に入ろう。

配列（array）というのは，同じ種類の要素（通常は，数字）を並べたリスト様のオブジェクトである。難しいことを特に意識する必要もないが，要素がすべて同じ種類でなければならないという制約があるおかげで要素の取得や要素に対する演算を効率的に実行できるということだけ覚えておこう。

Python で数値計算をするときには **NumPy** の **ndarray** という形式の配列を用いる。前章で紹介したとおり，**NumPy** を使えるようにするにはいつものインポート文を実行する必要がある。繰り返し書いているうちに無意識に打てるようになる。

```
import numpy as np
```

ベクトルを表現する配列を作るには，`np.array()` をという関数をリストに適用すればよい。すべての要素に小数点がついていることに注意しよう。配列は同じ種類の数字が並んだものなので，整数と小数が混ざっている場合には小数に変換される。

```
x = np.array([1, 2, 3.0])
x
```

[8] 多くの場合はリストで置き換えられる。リストでなく必ずタプルを使わないといけないケースには，辞書と呼ばれるデータ形式のキーにしたい場合などがある。本書では，このようなケースは（多分）出てこないと思う。

```
array([1., 2., 3.])
```

ndarray の nd は N-dimensional（N 次元）の略である。数学的なベクトル空間の次元と，配列の次元は意味合いが異なるので注意が必要である。例えば,

- 0 次元配列はスカラー
- 1 次元配列はベクトル
- 2 次元配列は行列

のようになる。数学的に N 次元ベクトルと呼ばれるものは，**NumPy** では要素数 N の 1 次元配列を用いて表現できる。配列の次元は `ndim`, 数学的な次元（サイズ）は `shape` で調べることができる。

```
x.ndim
```

```
1
```

```
x.shape
```

```
(3,)
```

ドットの後に `ndim` や `shape` と書いて呼び出している変数は**属性**（attribute）と呼ばれるオブジェクトの追加情報である。**ndarray** の **shape** 属性がタプルになっていることに注意しよう。

配列の演算

NumPy 配列同士の演算はおおむね普通の算術演算と同様に実行できるので，混乱はほとんどないだろう。原則的に要素ごとに計算される。

```
x = np.array([1, 2, 3.])
y = np.array([-1, 1, 4.])
2 * x + 0.5 * y
```

```
array([1.5, 4.5, 8. ])
```

スカラーとの演算もうまく定義されている。

```
np.array([1, 2, 3.]) + 10
```

```
array([11., 12., 13.])
```

低次元配列と高次元配列の演算にはブロードキャスティングという処理が実行されて shape が揃えられる。例えば，1 次元配列（ベクトル）と 2 次元配列（行列）の足し算は次のように振る舞う。

```
A = np.array([[0.1, 0.2, 0.3],
              [0.4, 0.5, 0.6],
              [0.7, 0.8, 0.9]])
A
```

```
array([[0.1, 0.2, 0.3],
       [0.4, 0.5, 0.6],
       [0.7, 0.8, 0.9]])
```

```
x + A
```

```
array([[1.1, 2.2, 3.3],
       [1.4, 2.5, 3.6],
       [1.7, 2.8, 3.9]])
```

低次元配列がスカラー以外の場合には最細の注意が必要であるが，ここで詳しく扱うには複雑すぎるので省略する（行列の定義の仕方を見ておけばよい。）。演算を行う前に十分テストをしたほうがよい。はじめから次元を合わせておくともっと安全だろう。

なお，ブロードキャスティングが適用できる場合を除いて，サイズの異なる配列同士の計算はできない。

問題 2.5. 次の計算を実行しようとするとどのような結果になるだろうか。下のコードを実行し，結果を記録しなさい。

```
np.array([1, 2, 3.]) + np.array([1, 2, 3., 4])
```

Answer

数学的には定義されていないベクトル間の演算も NumPy 配列に対して使用できる場合がある。例えば，乗算や除算などが要素ごとの演算として定義されている。

```
x * y
```

```
array([-1.,  2., 12.])
```

```
x / y
```

```
array([-1.  ,  2.  ,  0.75])
```

配列に作用する関数

配列を入力に取る数学関数には，次のようなタイプのものがある。

- 入力配列の要素ごとに関数値を計算し，入力と同じサイズの配列を出力するもの。
- 入力配列全体をひとまとめにして関数値を計算し，単一の数を出力するもの。
- 入力配列全体をひとまとめにして関数値を計算し，入力と同じサイズの配列を出力するもの。

1 つ目のタイプの関数には `np.log()` や `np.exp()` などがある。

```
np.log(x)
```

```
array([0.        , 0.69314718, 1.09861229])
```

```
np.exp(A)
```

```
array([[1.10517092, 1.22140276, 1.34985881],
       [1.4918247 , 1.64872127, 1.8221188 ],
       [2.01375271, 2.22554093, 2.45960311]])
```

2 つ目のタイプの関数の代表例は np.sum() や np.max() などである。

```
np.sum(x)
```

```
6.0
```

```
np.max(x)
```

```
3.0
```

このタイプの関数は，（すべてかどうかは分からないが）同名のメソッドを持っているので，次のように呼び出すこともできる。

```
x.max()
```

```
3.0
```

```
x.argmax()
```

```
2
```

2 次元以上の配列にこのタイプの関数を適用すれば，特定の次元にだけ適用して情報を集約するために使うことができる。例えば，次の 2 次元配列が 3 人の学生の英語と数学の試験の成績と解釈しよう。行方向（横方向）に見れば，各学生の英語・数学の 2 つの数字が並んでいる。列方向（縦方向）に見れば，各科目の 3 人の成績が並んでいる。

```
B = np.array([[90,  90],
              [100, 60],
              [80,  90]])
B
```

```
array([[ 90,  90],
       [100,  60],
       [ 80,  90]])
```

次のコードは各科目の平均得点（axis=0 と書くと，列（第 0 次元）の平均を計算する）と各学生の平均得点（axis=1 と書くと，行（第 1 次元）の平均を計算する）を計算している。

```
B.mean(axis=0)
```

```
array([90., 80.])
```

```
B.mean(axis=1)
```

```
array([90., 80., 85.])
```

3 つ目のタイプの関数には，累積和 cumsum()，累積積 cumprod() などがある。このタイプの関数も配列に付随するメソッドとして呼び出せる場合がある。

```
np.cumsum(x)
```

```
array([1., 3., 6.])
```

```
A.cumprod(axis=0)
```

```
array([[0.1  , 0.2  , 0.3  ],
       [0.04 , 0.1  , 0.18 ],
       [0.028, 0.08 , 0.162]])
```

2 つの配列に作用する数学関数

ここまでは単一の配列に対する操作を紹介した。ベクトルの内積やデータの共分散など，複数の配列（ベクトル）に対して関数を適用したいケースもある。出力は単一の数字，入力と同じサイズのベクトル，1 つ次元の高い配列（行列）になる場合がある。

```
np.dot(x, y)
```

```
13.0
```

```
np.maximum(x, y)
```

```
array([1., 2., 4.])
```

```
np.minimum(x, y)
```

```
array([-1.,  1.,  3.])
```

```
np.cov(x, y)
```

```
array([[1.        , 2.5       ],
```

```
       [2.5        , 6.33333333]])
```

なお，dot() を用いた計算はここではうまく内積を計算できたが，これはどちらも 1 次元の配列だからだ。x,y が同じシェイプを持つ 2 次元配列（ベクトル）になっている場合には，行列の積を計算しようとして失敗する。私はこのような振る舞いに使いにくさを感じているし，行列積を計算するだけなら@ 演算子を使う方が美しく書ける。今後本書で dot() に出会うことはもうないだろう。さようなら。

問題 2.6. dot() を使わずに内積を計算する方法を説明しなさい。

Answer

配列を生成する関数

■基本の関数　配列を作成するために毎回リストを書かなければいけないのは手間が大きいので，よく使う配列を生成する関数が用意されている。以下の関数はタプルまたは単一の数を入力として受け取って配列のサイズとして用いる。それぞれどのような配列が出力されるかを確認しよう。

```
np.ones(2)
```

```
array([1., 1.])
```

```
np.zeros((2, 3))
```

```
array([[0., 0., 0.],
       [0., 0., 0.]])
```

シミュレーション分析では配列の値を特に設定せずに，サイズだけ指定したい場合が

ある。`empty()` という関数を使う。値は適当に決まる[*9]。

```
np.empty((2, 3))
```

```
array([[0., 0., 0.],
       [0., 0., 0.]])
```

問題 2.7. `np.ones_like()`,`np.zeros_like()`,`np.empty_like()` はどのように使用するか。使用方法を調べて使ってみなさい。そして，それぞれの用途，使用方法を自分の言葉で説明しなさい。ヒント：IPython では関数名の前か後に? をつければ関数のドキュメントを読むことができる。例えば，`np.empty_like()` は次のようにして表示できる。ドキュメントから抜けるにはキーボードの「q」を押下する。

```
np.empty_like?
```

Answer

■乱数の生成 ランダムに数字を生成して配列を生成したい場合がある。**numpy.random** というモジュールで定義された関数を使うことができる。

$[0, 1)$ の一様乱数を生成するには，`np.random.random()` を用いる[*10]。

[*9] このような関数はシミュレーション結果を保存する配列を作るために用いる。結局はすべての値が上書きされることになるので，最初に初期化する（例えばゼロを代入する）作業を省いても問題にならない。

[*10] 「乱数」というのは，コンピュータ上で確率変数の実現値を擬似的に計算したものである。あたかも

```
np.random.random((3, 2, 2))
```

```
array([[[0.54340494, 0.27836939],
        [0.42451759, 0.84477613]],

       [[0.00471886, 0.12156912],
        [0.67074908, 0.82585276]],

       [[0.13670659, 0.57509333],
        [0.89132195, 0.20920212]]])
```

標準正規分布に従う乱数を生成する方法はいくつかある。上で紹介した関数は，入力としてタプルを渡す使い方をしているので，これと同じ使用法である np.random.standard_normal() を紹介しよう。

```
np.random.standard_normal((3, 2))
```

```
array([[-0.45802699,  0.43516349],
       [-0.58359505,  0.81684707],
       [ 0.67272081, -0.10441114]])
```

■特別な行列の生成　特別な行列を生成するときも，いくつかの便利な関数がある。対角成分に 1 が並ぶ単位行列を作るときは eye() を使う。

```
np.eye(3)
```

```
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

対角成分だけを指定して対角行列を作るには diag() を使う。

```
np.diag([1, -1, 3.])
```

ランダムに見える数列というくらいに考えておけばよい。なお，プログラムの開発中にはランダムさが邪魔になる場合がある。np.random.seed() という関数を呼び出せば乱数の種（数列の初期値，パラメータ）が固定される。本書では，np.random.seed(1000) を実行した後に，続く乱数生成のコードが順次実行されている。

```
array([[ 1.,  0.,  0.],
       [ 0., -1.,  0.],
       [ 0.,  0.,  3.]])
```

■連続する数ベクトルの生成 `linspace()` は連続的な区間を出力サイズを指定して等間隔に離散化する際に用いる。非常によく使うので覚えておこう。入力に与えた数字は，左端・右端・出力サイズの順に並んでいる。

```
np.linspace(0, 1, 5)
```

```
array([0.  , 0.25, 0.5 , 0.75, 1.  ])
```

出力サイズではなく，隣り合う数の間の距離を指定したい場合には，`arange()` を用いる。右端の点が含まれないことには十分注意しよう。

```
np.arange(5.0, 7.0, 0.2)
```

```
array([5. , 5.2, 5.4, 5.6, 5.8, 6. , 6.2, 6.4, 6.6, 6.8])
```

```
np.arange(6)
```

```
array([0, 1, 2, 3, 4, 5])
```

■配列の連結 2 つ以上の配列を連結して 1 つの配列を生成する方法を紹介しよう。

- `np.c_[]` は列（column）の方向に配列を拡張する。
- `np.r_[]` は行（row）方向に配列を拡張する。

丸括弧ではなく各括弧で呼び出すことに注意しよう[*11]。

```
u = np.array([1, 2, 3, 4]).reshape((2, 2))
u
```

```
array([[1, 2],
       [3, 4]])
```

[*11] 関数でなければ気持ち悪いという場合は，`hstack()`,`vstack()`,`concatenate()` の使い方を調べてみよう。

```
v = np.array([10, 11])
v
```

```
  array([10, 11])
```

```
np.c_[u, v]
```

```
  array([[ 1,  2, 10],
         [ 3,  4, 11]])
```

```
w = v.reshape((1, 2))
np.r_[u, w]
```

```
  array([[ 1,  2],
         [ 3,  4],
         [10, 11]])
```

シェイプの変更

配列の変形を自由自在に行えると，後々都合がよいので，ここで説明しておこう。次のベクトル（1 次元配列）a を 2×4 行列（2 次元配列）に変換したいとする。

```
a = np.arange(8)
a
```

```
  array([0, 1, 2, 3, 4, 5, 6, 7])
```

2 つの方法がある。

- reshape() メソッドを使って変換後のオブジェクトを生成する。この場合，元のオブジェクトは変更されない。
- resize() メソッドを使って変換前のオブジェクトを書き換える。このように，元のオブジェクトが変更される操作を「in-place」な操作とか，破壊的な操作と呼ぶ。

```
a.reshape((2, 4))
```

```
array([[0, 1, 2, 3],
       [4, 5, 6, 7]])
```

```
a
```

```
array([0, 1, 2, 3, 4, 5, 6, 7])
```

```
a.resize((2, 4))
a
```

```
array([[0, 1, 2, 3],
       [4, 5, 6, 7]])
```

行列に変換されるときには行方向に数字が埋められていくことに注意しよう[*12]。

転置行列を作るには，`transpose()` あるいは `T` 属性を用いる。

```
a.transpose()
```

```
array([[0, 4],
       [1, 5],
       [2, 6],
       [3, 7]])
```

```
a.T
```

```
array([[0, 4],
       [1, 5],
       [2, 6],
       [3, 7]])
```

サイズが 1 の次元を除去するメソッド `squeeze()` は多用される（私が多用する）ので，覚えておいてほしい。以下の例は，3 つのベクトル時系列

$$b_1 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \quad b_2 = \begin{bmatrix} 2 \\ 3 \end{bmatrix}, \quad b_3 = \begin{bmatrix} 4 \\ 5 \end{bmatrix}$$

[*12] Python と同じくデータ分析でよく用いられる R 言語は列方向に数字を埋めていく。すでに R に慣れている人にとってはしばらく混乱するかもしれない。3 次元配列の作り方にも微妙が違いがあるので，色々と比較をしてみてほしい。

をシェイプが $(3, 2, 1)$ であるような 3 次元配列として表現した b と，それを表形式で表現し直すために最後の次元を落としたものである。ベクトル時系列は 3 次元配列として表現する方がシミュレーション・コードを簡潔に保ちやすい。しかし，2 次元配列（行列）として表現しておく方が可視化や統計処理には便利である。一長一短があるので，どちらの表現も使えるようにしておこう[13]。

```
b = np.arange(6).reshape((3, 2, 1))
b
```

```
array([[[0],
        [1]],

       [[2],
        [3]],

       [[4],
        [5]]])
```

```
b.squeeze()
```

```
array([[0, 1],
       [2, 3],
       [4, 5]])
```

2.4.3　可視化

ここまで分かれば，簡単な可視化を実行することができる。ちょっと脱線して，作図の方法を紹介しよう。

可視化にも様々な方法があるが，もっともよく用いられるのは **Matplotlib** というライブラリだろう。次のインポート文はイディオム的に覚えてしまおう。

```
import matplotlib.pyplot as plt
```

[13] 表形式から 3 次元配列にするには，`np.expand_dims(a, 2)` のようにすればよい。

関数（$y = f(x)$）を描くには，次の 3 ステップが基本である。

1. 関数値を計算したい横軸の値を決める。
2. 指定した横軸すべてについて関数値を計算する。
3. プロット関数を呼び出す。

Matplotlib のもっとも初歩的な使い方を用いれば，次のようになる（図 2.1）。なお，井桁記号（#）に続くテキストは人間のためのコメントであり，Python は無視する。ここでのコメントは私が読者のために書いたものなので，読者は入力する必要がない。

```
x = np.linspace(0, 5, 200)   # Step 1
y = np.exp(x)                # Step 2
plt.plot(x, y)               # Step 3
plt.show()
```

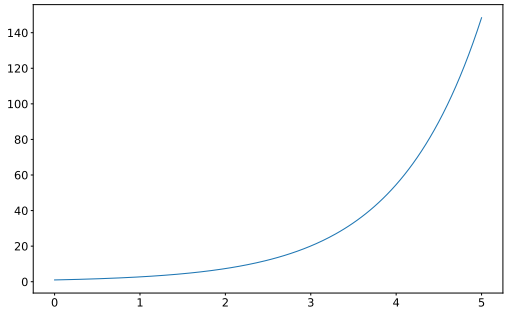

図 2.1: **Maplotlib** の使用例

最後のコマンド `plt.show()` は描画したグラフを画面に表示するためのものである[14]。

[14] IPython で `%matplotlib` マジックコマンドを実行している場合には `plt.show()` は不要になる。Jupyter Notebook を用いている場合も，`plt.show()` を忘れても図が表示されるかもしれない。表示されない場合は，`%matplotlib inline` というマジックコマンドを実行して，作図のコードを再実行してみよう。

ランダム・ウォーク[*15]

$$x_{t+1} = x_t + \varepsilon_{t+1}, \qquad \varepsilon_{t+1} \overset{\text{i.i.d}}{\sim} \mathcal{N}(0,1), \quad t = 0, 1, \ldots$$
$$x_0 = 0$$

をシミュレーションするには次のコードを実行するとよい（図 2.2）。1 つだけだと簡単過ぎて拍子抜けしてしまうかもしれないので，2 つの経路を描いておこう。

```
# Solid line
eps1 = np.r_[0, np.random.standard_normal(100)]
x1 = np.cumsum(eps1)
# Dashed line
eps2 = np.r_[0, np.random.standard_normal(100)]
x2 = np.cumsum(eps2)

plt.plot(x1, 'k-')
plt.plot(x2, 'k--')
plt.show()
```

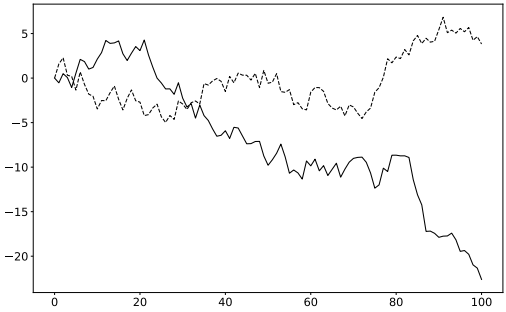

図 2.2: ランダム・ウォークのシミュレーション

[*15] $\varepsilon_{t+1} \overset{\text{i.i.d}}{\sim} \mathcal{N}(0,1)$ は，ランダムな撹乱項 $\varepsilon_1, \varepsilon_2, \ldots$ が
- 互いに独立で同じ確率分布に従っている（i.i.d = independent and identically distributed）
- 確率分布は標準正規分布 $\mathcal{N}(0,1)$ である

という 2 つの性質を表現している。なお，「ランダム・ウォーク」と言った場合には普通は $\mathcal{N}(0,1)$ に限定する必要はないので，ここでは通常の定義よりも強い仮定を置いている。

cumsum() の使い方も分かっていただけただろうか。

2.5 プログラミング: 価格指数の計算

乱暴な言い方をしてしまえば，価格指数の計算は

- ベクトルの内積
- 2 つの数の割算

の 2 つの計算をしているだけである。**NumPy** の内積を計算する方法は分かったので（問題 2.6），指数計算の準備は整っている。

公務員試験などでも出題される問題に次のようなものがある。

問題 (市役所・平成 21 年度)**.** 下の表はある国における A 財と B 財の価格と数量について，2000 年と 2001 年とで比較したものである。（以下略）

	A 財		B 財	
	価格	数量	価格	数量
2000 年	40	3	80	6
2001 年	80	5	30	7

次のようなコードを書けばラスパイレス指数，パーシェ指数（の 100 を掛ける前の数字）が得られる。式 (2.1), (2.2) とコードをよく見比べてほしい。

```
price00 = np.array([40, 80])
price01 = np.array([80, 30])
quantity00 = np.array([3, 6])
quantity01 = np.array([5, 7])

# Laspeyres
sum(price01* quantity00) / sum(price00 * quantity00)
```

```
0.7
```

```
# Paasche
sum(price01 * quantity01) / sum(price00 * quantity01)
```

```
0.8026315789473685
```

データ保存形式を工夫する

上のようなコードを書くときに，頭の中で元の表を次の形式に置き換えるという操作を実行している。太字にした部分が quantity00 に対応する。

価格	A 財	B 財
2000 年	40	80
2001 年	80	30

数量	A 財	B 財
2000 年	**3**	**6**
2001 年	5	7

上のコードを書いたときの気持ちは，各表の行（年）ごとに変数を作れば内積計算ができるだろうというものだった。非常に愚直に考えてコードを書いた訳だが，これではデータの年数が増えたときに大変困ったことになる。10 年分のデータがあると変数が 20 個になる。これでは管理が大変だ。そこで，この 2 つの表をそのままコードに変換してみよう。表形式のデータを行列と見なして，次のように書ける。

```
price = np.array([[40, 80],
                  [80, 30]])
price
```

```
array([[40, 80],
       [80, 30]])
```

```
quantity = np.array([[3, 6],
                     [5, 7]])
quantity
```

```
array([[3, 6],
       [5, 7]])
```

NumPy の 2 次元配列から一部分を取り出すには,

配列名［行インデックス，列インデックス］

の形式を用いる[*16]。特定の次元からすべての要素を抜き出すためにはコロン（:）をインデックスに指定する。したがって，quantity00 に相当する部分を配列 quantity から抜き出すためには，「0 行目のすべての列」を指定すればよい。

```
quantity[0, :]
```

```
array([3, 6])
```

このようにデータを保存した場合には，ラスパイレス指数の計算は次のようになる。

```
sum(price[1, :] * quantity[0, :]) / sum(price[0, :] * quantity[0, :])
```

```
0.7
```

さらに言えば，比較年を表す 1 という数字は新しいデータが来れば変わるので，これは変数に保存しておくだけの価値がある。次のコードと式 (2.2) がかなり近い表現になっていることを確認してほしい。

```
t = 1
sum(price[t, :] * quantity[t, :]) / sum(price[0, :] * quantity[t, :])
```

```
0.8026315789473685
```

データの保存形式を工夫することで，次のようなご利益がある。

- 数式とコードの対応関係をわかりやすくできるので，他の人に説明しやすくなる。
- 結果的にコードの保守が容易になる。

1 ヶ月後の自分の理解力を決して過信してはいけない。読みやすいコードを書くように心がけよう。

[*16] 多次元配列の場合は，第 1 次元のインデックス，第 2 次元のインデックス，第 3 次元のインデックス・・・とコンマでつないでいけばよい。なお，このような記法は **ndarray** でないただのリストのリストには使えないことに注意しよう。

連鎖指数の計算

連鎖指数が意味を持つには 3 年以上の期間が必要なので，さきほどのデータを次のように拡張する。

価格	A 財	B 財
2000 年	40	80
2001 年	80	30
2002 年	70	40
2003 年	60	55

数量	A 財	B 財
2000 年	3	6
2001 年	5	7
2002 年	6	8
2003 年	8	10

ラスパイレス式の連鎖指数（参照年 = 2000 年）を計算するには次のようにすればよい。

```
price = np.array([[40, 80],
                  [80, 30],
                  [70, 40],
                  [60, 55]])
quantity = np.array([[3, 6],
                     [5, 7],
                     [6, 8],
                     [8, 10]])

laspeyres = np.empty(price.shape[0])   # 価格指数の保存場所を確保

laspeyres[0] = 1.
t = 1
laspeyres[t] = laspeyres[t-1] * (sum(price[t, :] * quantity[t-1, :])
                                 / sum(price[t-1, :] * quantity[t-1, :]))
t = 2
laspeyres[t] = laspeyres[t-1] * (sum(price[t, :] * quantity[t-1, :])
                                 / sum(price[t-1, :] * quantity[t-1, :]))
```

```
t = 3
laspeyres[t] = laspeyres[t-1] * (sum(price[t, :] * quantity[t-1, :])
                                 / sum(price[t-1, :] * quantity[t-1, :]))
laspeyres
```

```
array([1.        , 0.7       , 0.72295082, 0.78156845])
```

上のコードには

```
laspeyres[t] = laspeyres[t-1] * (sum(price[t, :] * quantity[t-1, :])
                                 / sum(price[t-1, :] * quantity[t-
1, :]))
```

という同じコードが繰り返されていることに注目しよう。このような冗長なコードは本来は避けるべきだが，連鎖指数が同じ計算の繰り返しで計算されている感じを十分に理解していただきたい。さらに，1 つ前のステップの計算結果を使って次の計算が実行されるような構造になっている点も重要である。これは，「漸化式」とか「再帰方程式」と呼ばれるものの一種である。動学モデルでは基本的なパターンなので，何度も練習して自力で書けるようになってほしい。おおよそ次のようなステップに分解される。

1. 結果を保存する配列を用意する。
2. 初期値を設定する。
3. 再帰方程式に基づいて順番に値を埋めていく。

繰り返し処理のための `for` ループは次章で学ぶ。

問題 2.8. 上の表に対して，パーシェ式の連鎖指数とフィッシャーの連鎖指数を計算しなさい。（ヒント: 平方根は `x ** 0.5` または `np.sqrt(x)` で計算できる。）

索引