

アルゴリズム入門 #6

地引 昌弘

2021.11.11

はじめに

今回は、大量のデータをコンピュータ上で効率良く扱うためのデータ構造である配列について説明します。配列は、非常によく使われるデータ構造なので、その特徴・処理方法について、ここでしっかりと習得しておいて下さい。

1 前回の演習問題の解説

1.1 演習 5-1 — テイラー級数で sin と cos を計算

参考のため、計算式を再掲しておきます:

$$\begin{aligned}\cos x &= \frac{x^0}{0!} - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \dots \\ \sin x &= \frac{x^1}{1!} - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots\end{aligned}$$

この計算は、「階乗や x^n を計算しながら」加えて行くのでちょっと面倒です。しかも、交互に+/-が変わることに注意する必要があります。sin, cos のテイラー級数を計算する Python プログラムを、以下に示します:

```
def sincos(x, n):
    sign = 1; pow = 1.0; fact = 1; sin = 0.0; cos = 0.0
    for i in range(n):
        cos = cos + sign * pow / fact
        pow = pow * x
        fact = fact * (2*i + 1)      # sin の計算では奇数までの階乗を使う
        sin = sin + sign * pow / fact
        pow = pow * x
        fact = fact * (2*i + 2)     # cos の計算では偶数までの階乗を使う
        sign = -sign
    return(sin, cos)
```

このプログラムでは、べき乗 (pow) および階乗 (fact) の数を 2 ずつ増やしながら、cos と sin のテイラー展開を並行して計算しています (pow と fact の値を変更する部分に注意して下さい)。実際のプログラムでは、項を無限に計算することはできず、どこかで計算を終了する必要があります。そのため、「何項まで計算するか」によって精度が変わり、終了した先の項は値が無視されるため誤差となります。これは、打ち切り誤差 (Cutoff Error) と呼ばれ、既に学んだ丸め誤差、情報落ち、桁落ちと並び、数値計算において誤差を生じる要因の一つになります。

¹次ページより: テイラー級数の各項は、“指数÷階乗” という形になっています。一般に、 $\frac{a^n}{n!} = \frac{a \cdot a \cdots a}{n \cdot (n-1) \cdots 1} = \frac{a}{n} \cdot \frac{a}{n-1} \cdots \frac{a}{a+1} \cdot \frac{a}{a} \cdot \frac{a}{a-1} \cdots \frac{a}{1} < \frac{a}{a+1} \cdots \frac{a}{a+1} \cdot \frac{a^a}{a!} = \left(\frac{a}{a+1}\right)^{n-a} \cdot \frac{a^a}{a!}$ より、 $\lim_{n \rightarrow \infty} \left(\frac{a}{a+1}\right)^{n-a} = 0$ なので、指数より階乗の方が大きくなります。しかし、数値計算では、特に実数の場合には扱える数の範囲が限られているため、往々にして、“指数 \ll 階乗” となる前に指数計算の方が先に実用限度を超えてしまいます。

では、実際に計算してみましょう:

```
>>> import math                                ← プロンプト上で math.pi を使いたいで
>>> sincos(math.pi/3, 5)                       ← π/3 の sin, cos は?
(0.8660254450997811, 0.500000433432915)       ← まあこんなもの
>>> sincos(math.pi/3, 10)                      ← 項を増やすと
(0.8660254037844385, 0.5000000000000001)      ← 精度も向上している
>>> sincos(math.pi, 10)                        ← π だと
(-5.289187244469688e-10, -1.00000000352908)  ← sin の値は e-10 であることに注意
>>> sincos(math.pi*10, 10)                     ← 10 π にしてみよう
(-167876715321.24396, -104528895953.88087)  ← 破綻した!!
>>>
```

$\pi/3$ や π の計算で生じる誤差は、打ち切り誤差の影響が大きいのだろうと予想は付きますが、 10π で破綻した原因は何だったのでしょうか? それは、 x が大きくなると、高次の項を計算する際に x^n の数値が実用限度を超えてしまうため、誤差が想定より大きくなってしまいますからです (前ページの脚注¹ も参照)。この問題に対処するには、 \sin と \cos が周期関数であることを利用して、計算範囲を絶対値の小さい $0 \leq x \leq \frac{\pi}{4}$ ぐらいに限定します (この範囲の \sin と \cos があれば、残りの範囲は全部これらをもとに計算できますね)。また、 x の範囲を上のように限定するならば、テイラー級数の項は 8 項くらいあれば十分と分かります (その先の項は、分子の絶対値が 1 より小さく、分母は 10^{10} 以上になるので、そこで計算を終了しても精度は十分と言えます)。さらにその場合、テイラー級数の計算は後ろの項から順に加えて行く方が良いです。何故ならば、後ろの項ほど絶対値が小さいので、前から順に足して行くと情報落ちが生じ易くなるからです。項の数を決めておけば、後ろから足すように書くのも簡単です。

この例のように、数式通りに素直に書いたプログラムがうまく動かないというのは困りますね。以前の演習でも取り上げましたが、アルゴリズムや数式が分かっている、それを精度良く計算するプログラムを作成することは、一筋縄では行かない面があります (これを研究する分野がソフトウェア科学です)。

1.2 演習 5-2b — ロジスティック方程式の差分

教科書 (練習問題 6.4) にもあるように、ロジスティック方程式は、生物における個体数の変化を表す数理モデルです。18 世紀末、トマス・マルサス (Thomas Malthus) はその著書「人口論」にて、“幾何級数的に増加する人口と算術級数的に増加する食糧の差により、貧困が発生する。これは必然であり、社会制度の改良では回避されない。” とする意見を発表し、大きな反響を呼びました。ロジスティック方程式は、マルサスの主張における不自然な点を解消するために、ピエール・フェルフルスト (Pierre Verhulst) が考案しました。これは、以下の考えに基いています。

- ある時刻 t における生物の個体数を $x(t)$ とする。
- 生物は、現在の個体数と経過時間に比例して増える。

$$\frac{d}{dt}x(t) = \gamma x(t) \quad (\gamma > 0)$$

- 但し、個体数が多くなり過ぎると、縄張り争いや食料難により、増加率は頭打ちになるはず ($x(t)$ がその最大値 X_{max} に近付くにつれ、係数 γ は 0 に近付く)。「人口論」では、この部分が考慮がされていないとして、議論が交わされたわけです。

$$\gamma = \gamma_0 \left\{ 1 - \frac{x(t)}{X_{max}} \right\} \quad (\gamma_0, X_{max} > 0)$$

- これらより、

$$\frac{d}{dt}x(t) = \gamma_0 \left\{ 1 - \frac{x(t)}{X_{max}} \right\} x(t)$$

$$\frac{d}{dt}x(t) = ax(t) - bx(t)^2 \quad (\text{但し、} a = \gamma_0 > 0, b = \frac{\gamma_0}{X_{max}} > 0)$$

ロジスティック方程式を、差分法 (オイラー法) により数値的に解くプログラムは、次のようになります:

```
!pip install ita # Google Colaboratory へのログイン毎に 1 度実行して下さい。
import ita

def logi_func(x0, y0, xmax, a, b, count):
    h = (xmax-x0) / count
    x = x0; y = y0
    s = ita.array.make1d(count)
    for i in range(count):
        x = x + h
        y = y + h * (a*y - b*(y**2)) # f(x, y) = a*y - b*y**2
        s[i] = y
    ita.plot.plotdata(s, line=True)
```

$a = 1.0, b = 1.0$ とし、 $(0.0, 0.1)$ から x 座標が 10.0 になるまで、刻み幅 0.01 で数値計算したグラフは下記の通りです (以下、ロジスティック方程式の形状を示すグラフは全て、あくまで傾向を見るためだけに示しているの、グラフにある x 座標の値は実際の x 座標の値と合っておりません)。 $y = 1$ に漸近していますね。

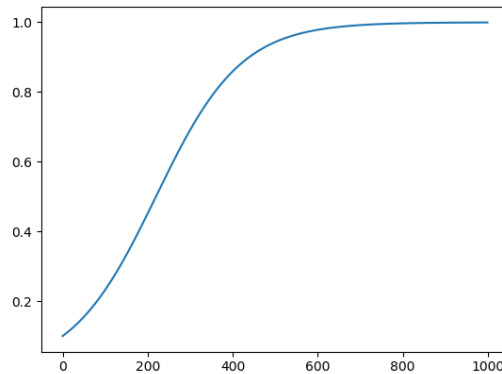


図 1: ロジスティック方程式の形状 (その 1)

また、ここでは詳細に触れませんが、実はロジスティック方程式は解析的に解くことが可能で、その解は

$$x(t) = \frac{X_{max}}{1 + e^{-\gamma_0 t} \{X_{max}/x(0) - 1\}} \quad (\text{但し、}\gamma_0 = a, X_{max} = \frac{\gamma_0}{b})$$

となります。これより、ロジスティック方程式は、 $\gamma_0 > 0, X_{max}/x(0) > 1$ の時、図 1 のように $y = 1$ に漸近します。また、 $X_{max}/x(0) < 1$ とした時のグラフも示しておきます (図 2)。グラフの形は大きく変わりますね。

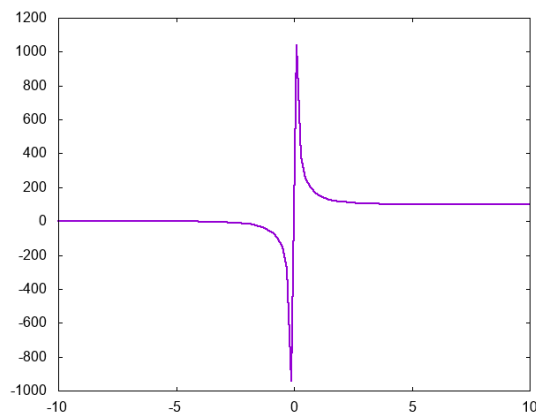


図 2: ロジスティック方程式の形状 (その 2)

次に、安定的な解が得られる刻み幅について考えてみましょう。ロジスティック方程式から得られる差分方程式は、次のようになります。

$$\frac{x(t + \Delta t) - x(t)}{\Delta t} = ax(t) - bx(t)^2$$

$$x(t + \Delta t) = (a\Delta t + 1)x(t) - b\Delta t \cdot x(t)^2$$

前回の付録2でも述べましたが、ロジスティック方程式は非線形微分方程式なので、差分法による近似解が安定する条件を調べることは少々面倒です。但し、上の式をよく見てみると、右辺は $x(t)$ の2次式なので、 $x(t)$ が大きい or 小さい場合は、どんな Δt を考えても $x(t + \Delta t)$ の絶対値が必ず $x(t)$ の絶対値より非常に大きくなってしまい、得られた近似解が安定しているとは言えません (ロジスティック方程式が表す数理モデルを考えると、 $|x(t)| \rightarrow \infty$ はあまり意味がありません)。しかし、これを上に凸の2次関数だと考えると、 $(a\Delta t + 1)x(t) - b\Delta t \cdot x(t)^2 = 0$ より、 $x(t)$ が区間 $[0, \frac{1}{b}(a + \frac{1}{\Delta t})]$ にある間は、 $x(t + \Delta t)$ の絶対値もある程度の範囲に収まるので²、近似解は安定する「可能性」があります (つまり今回は、少なくとも $x(0)$ の値が上記の区間に入るように Δt を決める必要があるというわけです)。

では実際に、刻み幅 Δt はそのまま、 $x(0)$ の値を大きくしたらどうなるか、試してみましょう (ロジスティック方程式の $x(t)$ は、プログラムでは y に該当します)。先ほどは、(0.0, 0.1) から刻み幅 0.01 で計算しましたが、今回は、(0.0, 102) から計算を始めてみます ($a = 1.0, b = 1.0$, 終点の x 座標 10.0: 全て前回と同じ)。

```
>>> logi_func(0.0, 102.0, 10, 1.0, 1.0, 1000)
Traceback (most recent call last):
  File "<pyshell#19>", line 1, in <module>
    logi_func(0.0, 102.0, 10, 1.0, 1.0, 1000)
  File "logi.py", line 11, in logi_func
    y = y + h * (a*y - b*(y**2))          # f(x, y) = a*y - b*y**2
OverflowError: (34, 'Result too large')
```

確かに、 $x(t + \Delta t)$ の値 (つまり、 y の値) が、システムで扱える限界を超えてしまったというエラーが表示されています。刻み幅は 0.01 なので、上で述べた近似解が安定する可能性のある区間は $[0, (1/1)(1 + 1/0.01)] = [0, 101]$ です。 $x(0) = 102.0$ は、この区間に入っていない。次に、(0.0, 99.0) から計算を始めてみると³、今回は数値計算がうまく進み、次のグラフが得られました。これは、ロジスティック方程式で $X_{max}/x(0) < 1$ とした場合のグラフに該当します。

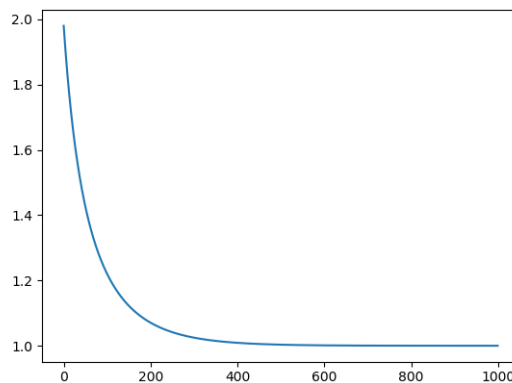


図 3: ロジスティック方程式の形状 (その 3)

²もう少し正確に言うと、ロジスティック方程式は拡散方程式と違って変数が一つ (t だけ) なので、その差分法とは、漸化式 $x_{n+1} = (a\Delta t + 1)x_n - b\Delta t \cdot x_n^2$ より得られる数列 x_n を求めることだと言えます。この漸化式の意味、つまり x_{n+1} と x_n との関係は、関数 $y = (a\Delta t + 1)x - b\Delta t \cdot x^2$ と $y = x$ より得られる、図 12 (7 ページ) の a と $b = f(a)$ (あるいは b と $c = f(c)$, c と $d = f(c)$) のような関係にあります。この時、もし $x(t)$ が区間 $[0, \frac{1}{b}(a + \frac{1}{\Delta t})]$ から出てしまうと (図 12 で言えば、区間 $[0, 1]$ を出てしまうと)、 $x(t)$ が $a \sim d$ のように一定の区間で折り返し合うことなく、一方的に $-\infty$ あるいは $+\infty$ 方向へ飛んでしまうというわけです。

³ $x(0) = 0, 101$ は、図 12 (7 ページ) の $x = 0, 1$ に該当するので、(折り返しが生じず) $x(t)$ は常に 0 になります。

さて、前ページでは、 $x(0)$ の値が区間 $[0, \frac{1}{b}(a + \frac{1}{\Delta t})]$ に入っていれば近似解は安定する可能性があると言いましたが、 $a, b, \Delta t$ の値を変えてもう少し様子を見てみましょう。今度は、 $a = 0.75, b = 1.0$ とし、刻み幅を $\Delta t = 4.0$ にします。初期値は、 $[0, (1/1)(0.75 + 1/4.0)] = [0, 1]$ より、 $x(0) = 0.1$ にしてみます。これを実行してみると、図4のグラフが得られました。これは、ちょっと驚きですね。因みに、刻み幅および $x(0)$ の初期は変えず、 a, b を $a = 0.1, b = 0.1$ にしたグラフは図5になります。こちらは、綺麗に収束していますね。

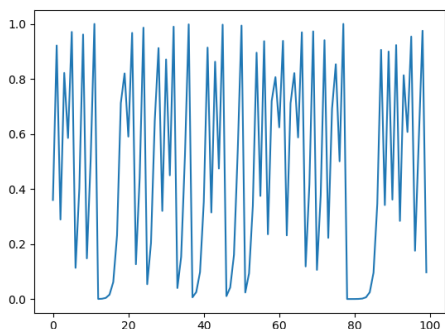


図 4: ロジスティック方程式の形状 (その 4)

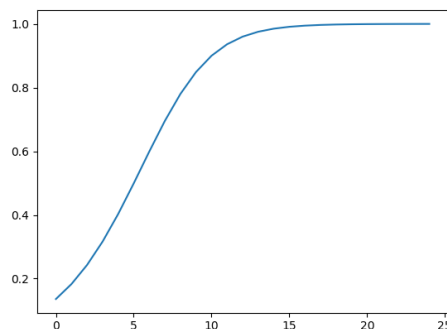


図 5: ロジスティック方程式の形状 (その 5)

図4は、これまで見て来た不安定状態が示す振動や発散のようには見えません。 $a = 0.75, b = 1.0$ の場合は内部で何が起きているのか、もう少し掘り下げてみることにしましょう。

ロジスティック方程式をそのまま扱うのは少々複雑そうなので、以下では簡略化のため (パラメータを減らすため)、 $r = a\Delta t + 1$, $x_n = \{(b\Delta t)/(a\Delta t + 1)\}x(t)$ とおき、ロジスティック方程式を $x_{n+1} = rx_n(1 - x_n)$ の形に変えます (これをロジスティック写像と呼びます)。数値計算用のプログラムは、以下の通り:

```
import ita

def logi_map(v0, count, r):
    v = v0
    s = ita.array.makeid(int(count))
    for i in range(int(count)):
        v = r*v*(1 - v)
        s[i] = v
    ita.plot.plotdata(s, line=True)
```

まずは、状況を整理するため、先ほどと同じパラメータ ($a = 0.75, b = 1.0, \Delta t = 4.0$) に該当する $r = 4.0$ および、 r の値をほんの少し変えた $r = 3.99999996$ を用いて、40 ステップまで計算したグラフを比べてみることにします (ステップ数を減らした理由は、グラフを拡大して見るためです)。

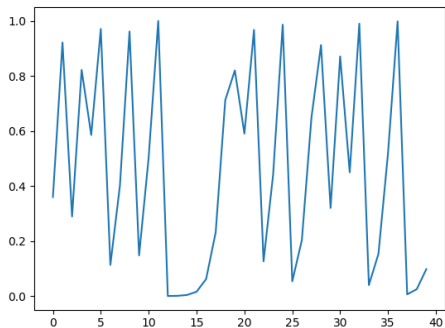


図 6: $r = 4.0$ のグラフ

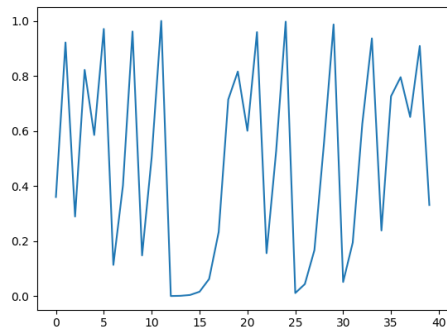


図 7: $r = 3.99999996$ のグラフ

これを見る限り、25 ステップを過ぎた辺りから両者に違いが見えて来ます。念のため、別の観点からも調べてみましょう。今度は、初期値 ($x(0) = 0.1$) をほんの少しだけ変えながら、計算したグラフを見てみることにします。

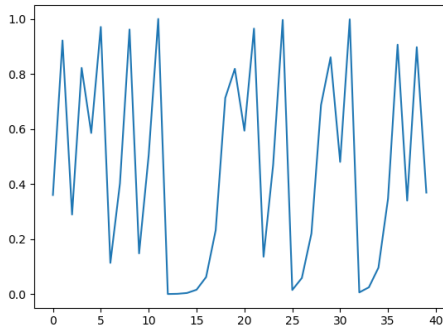


図 8: 初期値 0.099999999 のグラフ

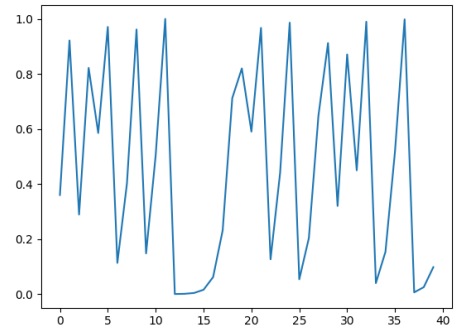


図 9: 初期値 0.1 のグラフ

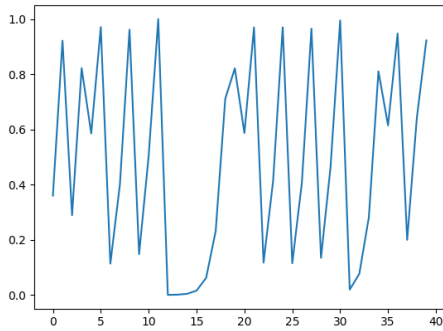


図 10: 初期値 0.100000001 のグラフ

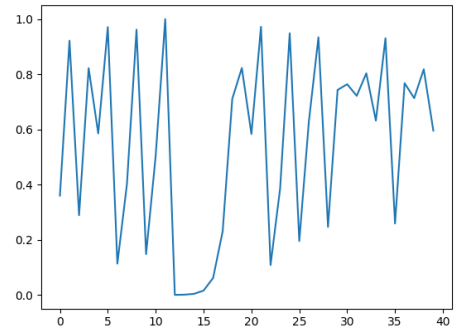


図 11: 初期値 0.100000002 のグラフ

これらも同様に、25 ステップを過ぎた辺りから違いが大きくなっています。 r にせよ $x(0)$ にせよ、それぞれ 1 億～10 億分の 1 程度しか変えていないにもかかわらず、ステップ数が増えると急激に違いが出て来ます。不思議ですね。

このような不安定性 (今回の場合は、初期値鋭敏性 (Sensitivity to Initial Conditions) と呼びます) は、19 世紀末にアンリ・ポアンカレ (Henri Poincaré) による 3 体問題⁴の研究において報告されました。その後、1960 年代にエドワード・ローレンツ (Edward Lorenz) が気象モデルの数値計算において、また上田 ヨシ亮 (ヨシは目偏に完)(Ueda Yoshisuke) が非線形常微分方程式の数値的計算において初期値鋭敏性を報告しています。しかし当時では、これらの現象は数値計算における一時的な過渡状態であると考えられ (つまり、長時間計算させればいつかは収束する)、ほとんど注目を集めませんでした。この現象が研究者の注目を集めるようになったのは、1970 年代に入ってからです。まずは 1975 年に、李天岩 (Tien-Yien Li) とジェイムズ・ヨーク (James Yorke) が、(例えば、ロジスティック写像のような) 数値的な対応関係が初期値鋭敏性を示す条件について証明しました⁵。また同時に、このような初期値鋭敏性をカオス (Chaos) と名付けました。以下では、この条件について簡単に説明しておきます。

まずは、 $x_{n+1} = f(x_n)$ という関係があったとします (これを分かり易く関数 f と呼びましょう)。さらに、関数 f に関連して、次の定義をします。

不動点: $f(p) = p$ となる点 p を不動点と呼ぶ。

k 周期点: 関数 f を k 回適用した $f^k = f(f(f \dots))$ について、「初めて $f^k(p) = p$ となる点 p 」を k 周期点と呼ぶ (つまり、点 p は f^k の不動点であるが、 f^i ($i \leq k-1$) の不動点ではない)。

この時、関数 f を区間 $[0, 1]$ 上の連続関数であるとし、かつ以下の条件を満たしていれば、

区間 $[0, 1]$ に属する $d \leq a < b < c$ に対して、 $f(a) = b$, $f(b) = c$, $f(c) = d$ が成り立つ場合がある⁶。

⁴これは、互いに相互作用する 3 体以上からなる位置関係を扱う問題です。例えば、地球と太陽のような 2 体問題は解析的に解くことができますが、地球・月・小惑星のような 3 体になると解析的にはうまく解けません。よって、数値的に解くこととなりますが、これが初期値鋭敏性を持つため、計算する度にその結果がバラバラになってしまうというわけです。怖い話をすれば、地球に近づく小惑星がどんな軌道を描くのか (つまり、衝突するのかそうでないのか) を正確に予測するのは難しい (別の言い方をすれば、近付いてみないと分からない) というわけです。

⁵これにより、コンピュータを用いて計算する際に生じる単なる誤差ではなく、数学的に厳密な計算においても (例えば、数を拡張しても) 発生する本質的な現象であることが分かったわけです。

⁶この関係は拡大 3 周期と呼ばれますが、 $d = a$ 、即ち $f^3(a) = a$ の場合とは 3 周期点を持つことを意味し、これを背景に李-ヨークは “Period three implies chaos.” という論文タイトルで上記を発表しました。

関数 f は次の特徴を示すというわけです。

- 全ての自然数 n について、 n 周期点が存在するような関数 f の初期値が、区間 $[0, 1]$ 内に存在する。
- 区間 $[0, 1]$ にある二つの値 x, y に対して関数 f を繰り返し適用して行くと、両者の描く軌道⁷は、1) 任意の距離を保って同じ軌道になる or 2) 全く異なる起動になることができる⁸。

これらは、まさに初期値鋭敏性の特徴と一致していますね。

$r = 4.0$ としたロジスティック写像は、まさにこの条件を満たしてしまい、カオスの特徴 (脚注⁸) を備えてしまったわけですね。この写像を例に、李-ヨークが示した条件のイメージを図 12 に示します。この図より、 r を小さくすると 2 次曲線の頂点も低くなり、上の条件を満たさなくなる (つまり、収束する) ことが分かります。

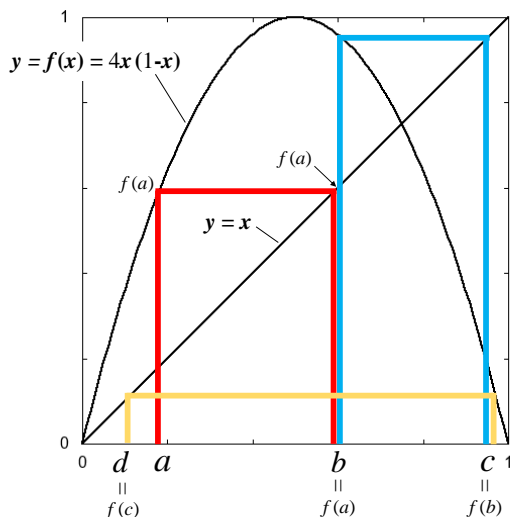


図 12: 李-ヨークの条件

ロジスティック写像はまだ少々複雑なので、これよりももう少し分かり易い例 (つまり、“ n 回適用する” ことの意味をもう少し直感的に捉え易い例) を用いて、初期値鋭敏性が生じる理由について考えてみましょう。まずは以下のような対応関係 (写像) Z を考えます。写像 Z は、初期値鋭敏性を持っています。

$$Z_{n+1} = \begin{cases} 2Z_n & (0 \leq Z_n < \frac{1}{2}) \\ 2 - 2Z_n & (\frac{1}{2} \leq Z_n \leq 1) \end{cases}$$

また、写像 Z は、図 13 左端に示したテント型の関数 $f(x)$ で表されるため、テント写像と呼ばれています。

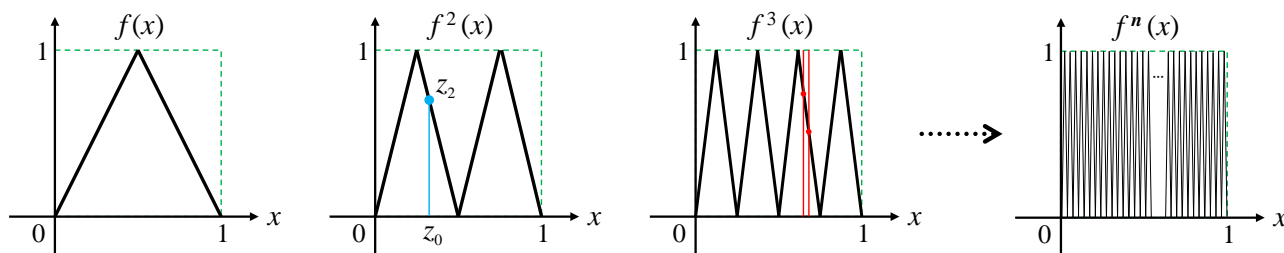


図 13: テント写像

Z_n は、初期値 Z_0 に対して、写像 Z を n 回繰り返し適用することにより得られる値です。これは、 x に対して $f(x)$ を n 回適用することと同じです (つまり、 $f(x)$ を n 回適用した $f^n = f(f(\dots f(x)))$ に初期値 x_0 を入力すると、 $x_n = f^n(x_0)$ より x_n が直接得られるというわけです)。まずは、 $f(x)$ を 2 回適用する場合 ($f^2(x) = f(f(x))$) を考えてみましょう。 $f(x)$ は、 $0 \leq x < \frac{1}{4}$ の時に $0 \leq f(x) < \frac{1}{2}$ となり、 $\frac{1}{4} \leq x < \frac{1}{2}$ の時に $\frac{1}{2} \leq f(x) < 1$ となるので、 $f^2(x)$ は区間 $[0, \frac{1}{2}]$ で一山で

⁷ n を増やして行った時の x_{n+1} や y_{n+1} の変化です。具体的には、前ページのグラフのことです。

⁸特に、 y が周期点の初期値である場合 (最初の特徴より、どんな周期点の初期値も対象にできます)、 x は全く周期を持たない軌道になります。

きます。また、区間 $[\frac{1}{2}, 1]$ の場合も、同様に考えることで一山できます。よって、 $f^2(x)$ は、図 13 中左に示した 2 山関数となります。これより、帰納的に考えると、 $f^n(x)$ は 2^{n-1} 山関数になることが分かります (図 13 右端)。

ではここで、写像 Z において初期値鋭敏性が生じる理由について考えてみましょう。まずは、初期値 x_0 と、この値を少しだけ変えた $x_0 + \Delta$ を考えます。そして、両者に $f(x)$ を n 回適用した $f^n(x_0)$ と $f^n(x_0 + \Delta)$ の値を比べてみます。 $f^n(x)$ は 2^{n-1} 山関数なので、 n を増やして行った場合、図 13 中右に示したように、 x の値を少し変えるだけで $f^n(x)$ の値は予測不可能なぐらい変わってしまいます。ここで言う“予測不可能”とは、 $f^n(x_0)$ と $f^{n+1}(x_0)$ の大小関係が増加傾向でも減少傾向でもなく、毎回 (不規則に) 変わるため次の値を予測できないという意味です⁹。同様に、 $f^n(x_0)$ と $f^n(x_0 + \Delta)$ の値も、予測不可能なほど変わってしまいます。これより、初期値鋭敏性が生じているというわけです。

李-ヨークの定理が発表された後、カオスという現象が認知されて多くの研究者の関心を集め、カオス理論という研究分野が生まれました。これまで無作為だと考えられていた現象 (つまり、確率を用いてモデル化されていた現象) のうち、カオス状態、即ち実は何らかの法則に従いながら、見た目上では法則性を示さないものがあるのではないか、と考えられるようになりました。これより現在では、もしそのような何らかの隠れた法則に従っている現象があるならば、条件を追加・削除しながらその現象を観察することで、支配的な条件 (カオスを引き起こしている条件) を発見できるのでは (つまり、カオスを緩和することで、数値計算による将来的な状態の予測精度を向上できるのでは) という期待のもと、様々な分野への応用が試みられています¹⁰。

さて皆さん、逐次細分最適化法による数値計算は、いかがだったでしょうか。観測されたデータから何らかのモデルや法則を導こうとする場合、通常は、基準となるデータの変化と、他のデータの変化とを比較することから始めます。その際、互いの変化について、数理的かつ合理的な因果関係のモデルを考えられる場合は、微分方程式を作成することができます¹¹。逐次細分最適化法に基づく数値計算のアルゴリズム自体は、それほど難しいものではありません。しかし、そこから少し外れてみると、深みのあるまだ見ぬ世界が潜んでいました (例えば、ロジスティック方程式では、安定的な解を得るために、前回検討した振動や発散以外にも、カオス性といった様々な配慮が必要でした)。このようにして、新しい発見がなされて行くのです。

1.3 演習 5-3 — 素数発見と実行時間の計測

素数の判定を行なう手順を示す擬似コードは次の通りです:

- isprime1: N が素数か否かを返す
- prime ← 「真」。
- i を 2 から $N - 1$ まで変化させながら繰り返し:
- もし N が i で割り切れるならば、prime ← 「偽」
- (繰り返し終わり。)
- prime を返す。

変数 prime はフラグとして利用します。先に素数である場合を表す True を入れておき、素数でない場合は False に入れ替えて、最後に結果を見ます。では Python コードを見てみましょう:

```
def isprime1(n):
    prime = True
    for i in range(2, n):    # 終値 = n-1
        if n % i == 0:
            prime = False
    return(prime)
```

これまで利用した range 関数は引数の一つ (カウンタの終値) でしたが、状況によっては、カウンタの始値やその増分を変えたい場合があります。range 関数では、次ページのように、カウンタの始値 (start)、終値 (stop)、増分 (step) を指

⁹別の言い方をすれば、 $n \rightarrow \infty$ とした場合、 $f^n(x)$ は頂上が無数ある山脈 (と言うか、無限個の絶壁?) なので (図 13 右端)、 n や x が変化した時にどの山の何合目へ移るか分からない、というわけです。

¹⁰最近では、金融の予測や都市交通の制御、セキュリティなどの分野に応用されています。例えば、金融の予測では、カオスを緩和できないことが分かれば、予測不能を前提とした金融オプションを用意する or 都市交通の制御では、信号の時間間隔がカオスを引き起こしていると分かれば、カオスを引き起こさない時間間隔を検討するというわけです。

¹¹数理的かつ合理的な因果関係を見つけにくい場合は、この後で取り上げる、回帰モデルなどを構築することになります。

定することができます。また、二つしか指定しなかった場合は、“始値と終値を指定/増分は1”という扱いになります。カウンタ i は、 $start$ から始まり、 $stop$ の直前までの値を、 $step$ ずつ取ります ($start$ は含みますが、 $stop$ は含みません)。 $start > stop$ の場合や $step < 0$ の場合もこの原則は変わりません。但し、例えば $start > stop$ かつ $step > 0$ など、明らかに暴走することが事前に分かった場合は、ブロック部分を実行しません。(range 関数は 1 ~ 3 個の引数を指定可能で、引数の個数により振る舞いが異なります/ 値の大小や正負なども混乱し易いので慎重に考えましょう):

```
for i in range(start, stop, step):
    ...
```

さて、上の素数判定を利用した最も素朴な素数発見プログラムは、下記のようにになります:

```
def primes1(n):
    for i in range(2, n+1):          # 終値 = n
        if isprime1(i):
            print(i)
```

これを、「発見した素数の個数だけを表示する」かつ「処理に要した時間も表示する」ように拡張したコードは、下記のようにになります:

```
import time

def primes1a(n):
    start = time.process_time()      # measurement of start time
    cnt = 0
    for i in range(2, n+1):          # 終値 = n
        if isprime1(i):
            cnt += 1                 # counting the number of prime numbers
    print(cnt)
    finish = time.process_time()     # measurement of finish time
    print("%g" % (finish - start))    # display of required time
```

「発見した素数の個数」は、変数 cnt に記録されます。また、「所要時間の表示」部分については、前回の資料を参照して下さい。これをあるマシンで動かしてみたところ、2 ~ 150,000 までの間に存在する素数の個数を調べるのに、693.1 秒掛かりました。これは遅いです。どうすれば速くなるか、アルゴリズムの改良だけではなく、今回のテーマである様々なデータ型を利用しながら工夫を試みてみましょう。

2 様々なデータ型

様々なプログラム言語で使われる“一般的”なデータ型を、2.1 および 2.2 節で説明します。Python に関しては、2.3 節以降で説明します。

2.1 基本データ型

コンピュータでは、様々なデータを 0・1 の列として表現し、扱っています。よって、元のデータが何であるかによって、どのような表現を使うかを適宜選択する必要があります。実際には、プログラムはプログラミング言語で記述するので、プログラミング言語が提供する表現方法をそのまま利用したり、あるいは組み合わせたりしてデータを表現します。この、「表現の種類」ないし「データの種類」のことをデータ型 (Data Types) と呼びます。

多くの言語では、内部に構造 (例えば、複数のデータを組み合わせるなど/詳細は後述) を持たないデータ型である基本データ型 (Primitive Data Types) を別扱いしています:

- 整数型 — 二進表現で整数を表す。「123」など。Python では小さい値には固定ビット数の表現が使われ、それで表現できなくなると複数のデータを使う表現に切り換えられる。一般に、後者の場合 (ある基準より大きい整数) は、内部構造を持つ型なので基本データ型とは言えない。
- 実数型 — 二進浮動小数点表現。「3.14」など。Python では 64 ビット IEEE 754 形式の浮動小数点が使われる。

- 文字列型 — 文字の並び。文字列を「" ” (ダブルクォーテーション) あるいは「' ’ (シングルクォーテーション) で囲んだもの。「abcd"」 or 「'abcd'」など。文字列は中に文字が複数入るという構造を持つので、基本データ型ではない。
- 論理型 — 「はい・いいえ」 (これを真偽値 (Truth Value) と呼ぶ) のどちらかの値を取る。Python では、「はい or 真」を値 “True” で、「いいえ or 偽」を値 “False” で表す。
- None — 「値がない」ことを表す値。

各基本型が備える演算子や処理の優先順位 (例えば、四則演算にも優先順位がありますよね) については、“Pythonによるプログラミングの初歩”にある“データ型・演算子”を参照して下さい。

2.2 構造データ型・複合データ型

構造データ型 (Structured Data Type) ないし複合データ型 (Compound Data Type) とは、その内部に複数個の基本データ型を含む (要は構造を持つ) データ型を言います (図 14)。

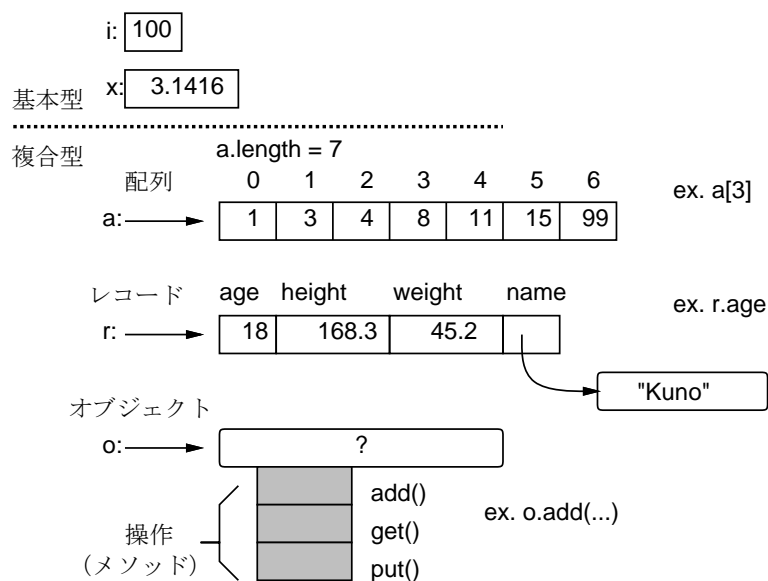


図 14: 様々なデータ型

- 配列型 — (多くの言語では) 同種類の値が並んだもの¹²。
- レコード (Record) 型 — 複数の型の値を組にしたもの。各値をフィールド (Field) と呼び、その名前で参照できる (成分名の付いたベクトルのようなイメージ)。フィールド参照の前に「.」を置く言語が多い¹³。
- オブジェクト (Object) 型 — レコード型と同様なデータを保持しているが、データへのアクセス (読み取り・書き込みなど) は、基本データ型 (or レコード型) のように直接行なうことはできず、操作 (Operation) ないしメソッド呼び出しという特別な処理を用いて行う¹⁴。

ところで、図 14 を見て不思議に思ったことはないでしょうか。具体的には、基本型 (整数など) では変数 (`i`, `x`) の位置に「箱」が書かれており、そこに値が入っていますが、複合型 (配列など) では少し離れたところにデータを入れる場所があり、変数 (`a`, `r`, `o`) からはそこへ矢印が出ています。実は、この矢印はデータの場所を示す参照 (reference — 場所を指す値で、実体はメモリ上の番地だと思ってよい) です。レコードのフィールド `r.name` に文字列を入れると、実際には

¹²ここでは、数列 x_i (添字つき変数) みたいなものと考えて下さい。

¹³例えば、`h` という変数が人のデータを表すレコード型なら、`h.name` には名前 (文字列)、`h.age` には年齢 (整数) が入っている、という感じで使います。

¹⁴その意義ですが、内部データ (例えば、レコード型におけるフィールドなど) に対するアクセス方法を限定することで、誤用によるデータの書き潰しを減らすといったプログラムの堅牢性向上が挙げられます。オブジェクト型を利用できるプログラミング言語を、オブジェクト指向言語 (Object-Oriented Language) と呼びます。オブジェクト型とレコード型は、「内部に値を保持する」という点で似ているため、多くのオブジェクト指向言語では、オブジェクト型とレコード型を統合しています。

文字列は別の場所に入り、フィールドにはその場所への参照が入るわけです。プログラミングにおいて、このような「値と参照の区分」を理解しておくことは重要です。例えば、「g = h」のように変数間で代入をする場合、基本型では値(箱の中身)がコピーされますが、複合型では参照(矢印)がコピーされるだけで、本体は一つのまま(単に二つの変数が同じ場所を指すようになるだけ)です。このような複合型変数の間で代入を行ない、二つの変数が同じ場所を指している状態で、片方の変数が指す複合型のデータを書き換えると、(もちろん複合型データは一つだけなので)もう片方の変数から見えるデータも同じように変化して見えます。この辺りの挙動は勘違いし易い(バグを生じ易い)ので注意が必要です。

2.3 Pythonにおける複合データ型

Pythonでは、複数のデータをまとめて管理する複合データ型に属する主なデータ型として、予め以下の四つが用意されています。またPythonでは、`type`関数を用いることで、各変数のデータ型を調べることができます。

	リスト型 [v,...]	タプル型 (v,...)	集合型 {v,...}	辞書型 {k:v,...}
異なるデータ型の格納	○	○	×	○
要素数を変更	○	○	○	○
要素の書き換え	○	×	○	○
要素へのアクセス方法	添字番号	添字番号	×	キー (文字列・数値)

前節で説明した配列型は、リスト型として定義されています(とは言え、配列型の方が一般的な用語なので、この講義では今後も配列型と呼ぶことにします)。上の表からも分かるように、配列型は様々な用途に適用できるため、今後は主に配列型を利用します。配列型については、以降の節で詳しく説明します。

タプル型については、関数が複数の戻り値を返す場合の説明(第2回)にて出て来ました。その詳細については、第2回の資料を参照して下さい。

集合型は、主に数学的な集合演算を行なうことを想定して用意されたデータ型です。集合型には、同じ値は一つしか格納できず、また、個々の要素へ個別にアクセスすることはできません。こんな感じです:

```
>>> s = {1, 1, 2, 2.0, 3, 4, 4.0, 4.00, 'a'}    ← 集合型のデータを定義
>>> print(s)
{1, 2, 3, 4, 'a'}
>>> s[0]                                         ← 添字番号によるアクセスは不可
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    s[0]
TypeError: 'set' object is not subscriptable
>>> s['a']                                       ← 辞書型のようなアクセスも不可
Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    s['a']
TypeError: 'set' object is not subscriptable
```

辞書型は、値(Value)を格納する際に、値だけではなく、その値にアクセスするためのキー(Key)も一緒に格納します。また、個々の要素へのアクセスでは、キーを指定することで、そのキーに該当する値が返されます。扱うアルゴリズムやデータの特徴によっては、配列型より辞書型を使う方が、プログラムの見通しや効率が良くなる場合もあるので、状況に応じた使い分けを考えましょう(教科書の8.5節「さまざまなデータ構造」および8.6節「文章の分析を少しだけ」は、必ず目を通して置いて下さい)。

```
>>> j = {"A": "dog", "B": "cat"}                ← 辞書型のデータを定義
>>> print(j)
{'A': 'dog', 'B': 'cat'}
>>> j['A']                                       ← キーによるアクセス
'dog'                                           ← 値が返される。
```

最後に、文字列型の扱いについて少し触れておきます。文字列は文字が並んだものなので、文字列型は上記 4 種類のどれかに該当すると予想されますが、Python では、これら四つとは異なるデータ型として定義されています。文字列型と配列型はよく似ていますが、要素を指定してデータ (文字) を書き換えることができません。こんな感じです:

```
>>> s = "aiueo"                ← 文字列型のデータを定義
>>> print(s)
aiueo
>>> s[1]                        ← 添字番号によるアクセスは OK
'i'
>>> s[1] = 'x'                 ← 添字番号を指定した要素の書き換えは不可
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    s[1] = 'x'
TypeError: 'str' object does not support item assignment
```

2.4 配列の生成とその処理

「配列」とは、「同種のデータを沢山並べたもの」という意味です¹⁵。配列を使うには、まず配列を作り出す必要があります。Python には配列の作成方法が幾つかありますが、この講義で利用する方法は主に以下の二つです:

```
a = [1, 2, 3]                    # 要素数と値を直接指定
a = ita.array.make1d(size)       # 要素数 (size) だけを指定 (各要素の値 = 0)
```

1 番目の方法は、要素数と要素を直接指定する方法で、比較的少数の要素を用意する場合に使います。2 番目は、要素数だけを指定する方法で、要素数の多い配列を用意する時には、この方法が一番単純です。各要素の値は全て 0 が設定されます。但し、その使用に際しては、事前に `import ita` を用いて、アルゴリズム入門の演習用ライブラリを読み込んでおく必要があります¹⁶。

配列を一度用意してしまえば、個々の要素は一つの変数と同様に扱えます。「どの要素か」を指定するには、`[...]` の中に式を書いて指定します。これが添字番号 (Index) です。添字番号の範囲は、0 ~ “作成時に指定した要素数 - 1” です (0 番目から数えることは、慣れないと忘れやすいので注意しましょう)。添字番号の範囲を超えてアクセスした場合は、エラーとなります:

```
>>> a = [1, 2, 3]                ← 要素数 3 の配列を作成
>>> a[0]
1
>>> a[2]
3
>>> a[3]
Traceback (most recent call last):    ← 添字番号 3 は "要素数-1" を超えているのでエラーとなる。
  File "<pyshell#12>", line 1, in <module>
    a[3]
IndexError: list index out of range
>>> a = []                       ← 要素数 0 の配列を作成してしまうと...
>>> a[0]
Traceback (most recent call last):    ← 一つもアクセスできない。
  File "<pyshell#14>", line 1, in <module>
    a[0]
IndexError: list index out of range
```

¹⁵参考までに、Python では値の種類に制約がないので、同種でなくてもよいのですが、一般に配列は x_i のような添字付き変数として使うため (添字を順繰りに変えながら、各変数へ同じ処理を行なうなど)、添字によって値の種類 (データ型) が異なると扱いづらく、通常は同種のものを入れるのが普通です。もし、異なる種類のデータ型をまとめて扱う必要が生じた場合は、配列型ではなく辞書型を利用し、キーの部分を同種のデータで統一する、という方法があります。

¹⁶さらに、Google Colaboratory を利用する場合は、`import ita` の前に `!pip install ita` を実行して下さい。これは、Google Colaboratory への毎ログイン時に 1 度実行すればよいです。

このような場合は、必要な数だけ配列に要素を追加する必要があります。Python には配列へ要素を追加する関数として、`insert` 関数が用意されています。`insert` 関数では下記のように、事前に作成した配列 `a` に対して、追加箇所の添字番号 (`index`), 追加する要素の値 (`val`) を指定します¹⁷。

```
a.insert(index, val)
```

配列 `a` の先頭に追加する場合は `index` に `0` を、同、末尾に追加する場合は `index` に `len(a)` を指定します (`len` 関数は、配列の要素数を返してくれます)。こんな感じに使います:

```
>>> a=[1, 2, 3]
>>> a[3]
Traceback (most recent call last):      ← 添字番号 3 は "要素数-1" を超えているのでエラーとなる。
  File "<pyshell#1>", line 1, in <module>
    a[3]
IndexError: list index out of range
>>> a.insert(len(a), 4)                 ← 添字番号 len(a)(= 3) は、現末尾の一つ先
>>> a[3]
4
```

参考として、`index` に `0` や `len(a)` 以外の値を指定したらどうなるかについて述べておきます。`insert` 関数では、`index` に `0` 以上の値 `i` が指定された場合は、配列の先頭から末尾方向へ向かって `i` 番目の要素へ、同じく `-1` 以下の値 `-j` が指定された場合は、配列の末尾の「一つ前」から (末尾が“-0” というイメージです) 先頭方向へ向かって `j` 番目の要素へ値が入ります。指定された `i` や `j` の値が要素数より大きかった場合は、先頭 or 末尾に追加されます。この規則は少々複雑な上、より簡単な代替方法 (つまり、直接 `a[index] = val` と代入) が存在するので、本講義では、`insert` 関数は `index` に `0` や `len(a)` を指定することで (特に後者)、配列の先頭 or 末尾を伸ばすもの、と考えておけば十分です。

最後に、配列から要素を削除する方法について説明します。ここで取り上げる要素の削除とは、要素の値に例えば `NaN` といった評価不能 (非数) の値を代入するということではなく、要素そのものを削除することを意味します。Python では配列から要素を削除する関数として、`pop` 関数が用意されています。

```
a.pop(index)
```

この関数は、配列 `a` から添字番号 `index` の要素を削除し、その前後を繋いでくれます。使い方は、こんな感じ (添字番号は `0` から始まることに注意しましょう):

```
>>> a = [0, 1, 2, 3, 4, 5, 6, 7]
>>> print(a)
[0, 1, 2, 3, 4, 5, 6, 7]
>>> a.pop(3)
3
>>> print(a)
[0, 1, 2, 4, 5, 6, 7]
```

`pop` 関数では、添字番号 `index` に負数を指定した場合は、`insert` 関数と同じ扱いですが、配列の要素数を超える値を指定した場合は、`insert` 関数とは異なりエラーとなります:

```
>>> a = [0, 1, 2, 3]
>>> print(a)
[0, 1, 2, 3]
>>> a.pop(100)
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    a.pop(100)
IndexError: pop index out of range
```

少々複雑ですね (まあ、要素数を超える添字番号の使用といった変則的なことはしない方が無難と言うことです)。

¹⁷この書き方は、これまで出て来た関数の書き方と少し違い、`○○` (この場合は配列 `a`) に「対して」この関数 (この場合は `insert`) を実行するという意味で、`○○` と関数名を「.」で繋げています。これは、前に説明したオブジェクト指向の考えに基づいた書き方です。このような書き方の関数は、これから数多く出て来ます。

2.5 配列の利用

次に、配列を利用したデータ処理を見てみましょう。まずは、配列を与えてその合計を求める処理を考えてみます：

- `arraysum` : 配列 `a` の数値の合計を求める (`a` は引数として渡される)
- `sum ← 0`。
- `i` を 0 から配列要素数の手前まで変えながら繰り返し、
- `sum ← sum + a[i]`。
- (繰り返し終わり。)
- `sum` を返す。

Python のコードは次の通り：

```
def arraysum(a):
    sum = 0
    for i in range(0, len(a)):
        sum = sum + a[i]
    return(sum)
```

一応、動かした様子も示しておきます (引数として直接、配列を渡しています)：

```
>>> arraysum([1,2,3,4,5])
15
>>>
```

演習 6-1 `ita.array.make1d` 関数を用いて 10 要素の配列を生成し、`for` ループを用いて (a)~(d) の値を設定しなさい。

(a)

10	9	8	7	6	5	4	3	2	1
----	---	---	---	---	---	---	---	---	---

(b)

0	1	0	1	0	1	0	1	0	1
---	---	---	---	---	---	---	---	---	---

(c)

4	3	2	1	0	1	2	3	4	5
---	---	---	---	---	---	---	---	---	---

(d)

1	1	1	1	1	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---

演習 6-2 まずは、前ページで示した配列合計プログラムを作成しなさい。動作を確認した後、これを参考に下記の処理を行なう Python プログラムを作りなさい¹⁸。

- a. 数の配列を受け取り、その最大値を返す。
- b. 数の配列を受け取り、最大値が何番目かを返す。なお先頭を 0 番目とし、最大値が複数あればその最初の番号が答えであるとする。
- c. 数の配列を受け取り、最大値が何番目かを出力する。なお先頭を 0 番目とし、最大値が複数あればそれらを全て出力する。
- d. 数の配列を受け取り、その平均より小さい要素を出力する (例: 1、4、5、11 → 1、4、5)。

演習 6-3 前回作成した、与えられた正の整数 N までの素数を見つけるために $2 \sim N - 1$ まで逐一調べるアルゴリズムは、大変に遅かった。単位時間当たりに処理できる N を大きくする工夫を考え、プログラムに組み入れて速度を比較せよ (処理を速くするためには、除算の回数をできるだけ少なくする工夫が効果的です)。

¹⁸ 「返す」の場合は、上の例と同様に `return` 関数を使い、「出力する」の場合は、`print` 関数を使って画面に直接 (その場で) 出力させて下さい。念のため確認ですが、`return` 関数は、使った瞬間にその関数呼び出し (実行) が終わってしまうため、`return` 関数を複数回使うことはできません。

演習 6-4 素数の発見については、配列を使うことでさらに高速な実行が期待できる。次の二つの方針を用いたプログラムを作成し、以前の演習で作成したものと速度を比較せよ。

- a. 発見した素数を配列に覚えておき、新たな素数の候補をチェックする際は、これまでに発見した素数で除算を試みて、割り切れなければ素数と判断する。
- b. 以下の手順により、 N 未満の素数を求める¹⁹。
 - 要素数 $N + 1$ の配列を作り (“添字番号 = 調べる数” にしたいため)、全て 0 にしておく。
 - 2: 4, 6, 8, …と、2 以外で 2 の倍数番目となる要素を 1 に変更。
 - 3: 6, 9, 12, …と、3 以外で 3 の倍数番目となる要素を 1 に変更。
 - 以後、 i 番目の要素が 0 であれば、 i 番目以外で i の倍数番目となる要素を 1 に変更して行く。
 - 最後に、0 が残っている要素 (の添字番号) を素数と判断する。

¹⁹この方法を考案したのは、ギリシャの哲学者エラトステネス (Eratosthenes) であり、この方法を彼の名前を冠してエラトステネスのふるい (Sieve of Eratosthenes) と呼びます。素数でないもの (各数の倍数) をふるい落としてしまえば、残ったものは素数だ、という考え方です。