アルゴリズム入門　＃4

地引　昌弘

2018.10.18

はじめに

一般に、アルゴリズムをコンピュータ上で実行する場合、コンピュータで処理し易い手順に修正したプログラムを作成し、これを実行します。「コンピュータで処理し易い」手順の一つとして、「“少しだけ or 一つだけ実行する” を何度も繰り返し、その結果を集計する」手順があります。今回は、この考えに沿って、前回学んだ繰り返しの制御構造を利用する手法について説明します。

1　前回の演習問題の解説

1.1　演習 3-1a — 枝分かれの復習

演習 3-1a は、前回の資料にある例題とほとんど同じです (0 と比較するか、b と比較するかの違い)。まずは擬似コードを見てみましょう:

- max: 数 a、b の大きい方を返す
- 　もし a > b であれば、
- 　　$result \leftarrow a$。
- 　そうでなければ、
- 　　$result \leftarrow b$。
- 　(枝分かれ終わり。)
- 　$result$ を返す。

Python では次の通り:

```
def max(a, b):
  if a > b:
    result = a
  else:
    result = b
  return(result)
```

これも、次のような「別解」があり得ます:

- max2: 数 a、b の大きい方を返す
- 　$result \leftarrow a$。
- 　もし $b > result$ であれば、
- 　　$result \leftarrow b$。
- 　(枝分かれ終わり。)
- 　$result$ を返す。

これの Python 版は次の通り:

```
def max2(a, b):
  result = a
  if b > result:
    result = b
  return(result)
```

これらはどちらが正解ということはありません。皆さんは、どちらが好みですか?

ところで、「2 数が等しい場合はどうするのか」について、皆さんの中には迷った人がいると思います。問題には「異なる数」と書いてあるので、今回は考えなくてもよいのですが、仮にそれが書いていなかったとします。そうなると、等しい場合について何らかの指示が本来あるべきですよね。例えば、次のような方針が考えられます。

- 「等しい場合はその等しい数を返す」
- 「等しい場合は何が返るかは分からない」
- 「等しい数を渡してはならない」

上から二つ目までの場合は、それほど問題にはならないでしょう (2 番目では何が返ってもよいので、等しい数でもよい)。最後の場合はどうでしょう。これも次の考え方があり得ます。

(a) 「渡してはならない」以上、渡されることはないのだから、特別な処理は必要ない

(b) 「渡してはならない」値が渡されたのだから、エラーを表示するなどして警告するべき

どちらにも (互いに裏返しの) 利点と弱点があります。(a) の方は、簡潔で短いプログラムを作ることができます。(b) の方は、起きるべきでないことが起きていることが分かるので、対処が必要な場合には有用です。現実世界の多くでは、発注者 (例えば地引) の注文を受けてプログラムを作成することになります。よって、解釈が曖昧な場合は、発注者に仕様 (プログラムの振舞い) を確認することになります。特に大きなプログラムでは、複数人が開発したプログラムを結合して全体を構成するため、互いに仕様の解釈が異なった場合は正しい動作をしないだけでなく、不良個所の特定が大変難しくなります (各自、自分の解釈が正しいと考えているため)。

1.2 演習 3-1b — 枝分かれの入れ子

演習 3-1b はもう少し複雑です。まず考えつくのは、a と b の大きい方はどちらかを判断し、それぞれの場合について、それを c と比べるというものでしょう:

- max3: 数 a、b、c で最大のものを返す
- 　もし $a > b$ であれば、
- 　　もし $a > c$ であれば、
- 　　$result \leftarrow a$。
- 　　そうでなければ、
- 　　$result \leftarrow c$。
- 　　(枝分かれ終わり。)
- 　そうでなければ、
- 　　もし $b > c$ であれば、
- 　　$result \leftarrow b$。
- 　　そうでなければ、
- 　　$result \leftarrow c$。
- 　　(枝分かれ終わり。)
- 　(枝分かれ終わり。)
- 　$result$ を返す。

かなり大変ですね。これを Python にしたものは次の通り:

```
def max3(a, b, c):
  if a > b:
    if a > c:
      result = a
    else:
      result = c
  else:
    if b > c:
      result = b
    else:
      result = c
  return(result)
```

このプログラムを見ると、インデント (字下げ) の効果は一目瞭然ですね。これがないと、どの `if` とどの `elif` や `else` が対応しているかを見極めるのは、少々面倒です。これまで何度も述べて来たように、多くの言語では “{” と “}” や “begin” と “end” によりブロックの境界を示しますが、これだけでは、どのブロックがどのヘッダに属するかを理解するのは無理なので、インデントも併用します。Python のデザイン哲学は、文法を極力単純化することで、プログラムの可読性/作業性/信頼性を高めることを目指しており、begin/end とインデントを両方使うなら、どちらか一つ必須な方で十分という判断より、このような仕様になっています。ところで、先の別解から発展させるとどうなるでしょう?

- max3a: 数 a、b、c で最大のものを返す
- result ← a
- もし $b > result$ であれば、
- $result \leftarrow b$。
- (枝分かれ終わり。)
- もし $c > result$ であれば、
- $result \leftarrow c$。
- (枝分かれ終わり。)
- $result$ を返す。

Python では次の通り (今度はどちらが好みですか?):

```
def max3a(a, b, c):
  result = a
  if b > result:
      result = b
  if c > result:
      result = c
  return(result)
```

一般に、枝分かれ (if) の中に枝分かれを入れるより、枝分かれを並べるだけで済ませられるならば、その方が制御構造を理解し易い場合が多いです。因みに、上の方法では、入力の数 N が何個あってもプログラム自体は簡単に拡張できますね。

実は、さらなる別解もあります。それは、前に作成した `max2` を利用するというものです。

```
def max3b(a, b, c):
  return(max2(a, max2(b, c))
```

このように、一度作って完成したプログラムを、後から別のプログラムを作る時の「部品」として使う、という考え方は大変重要です。是非、覚えておいて下さい。

1.3 演習 3-1c — 多方向の枝分かれ

演習 3-1c では、3 種類の結果を表示しなければならないため、必ず三つに枝分かれします。よって、if の中にまた if が入るのはやむを得ないでしょう。Python コードを見てみましょう:

```
def sign1(x):
  if x > 0:
    return("positive.")
  else:
    if x < 0:
      return("negative.")
    else:
      return("zero.")
```

このような「複数の条件判断」はよく使うので、if の入れ子にしなくても書けるような構文が用意されています。具体的には、if 文の後に、「elif 条件: 動作」という分岐を何回でも入れられるようになっています[1]。それを使うと次のようになります:

```
def sign2(x):
  if x > 0:
    return("positive.")
  elif x < 0:
    return("negative.")
  else:
    return("zero.")
```

擬似コードだと次のようになります:

- 実数 x を入力する。
- もし $x > 0$ ならば、
- 　　「positive.」を返す。
- そうでなくて $x < 0$ ならば、
- 　　「negative.」を返す。
- そうでなければ、
- 　　「zero.」を返す。
- (枝分かれ終わり。)

「そうでなくて〜ならば、」は何回現われても構いません。また、そのどれもが成り立たない場合は、最終的に「そうでなければ」に至りますが、この部分は不要なら無くても構いません[2]。これを、先ほどの最大値の問題に適用してみましょう。複合条件を使えば「a > b and a > c」なら a が最大だと分かりますから、elif を用いることにより、次のような 3 方向枝分かれで書くこともできます (変数を使わず、その場で値を返すスタイルにしてみました):

```
def max3c(a, b, c):
  if (a > b) and (a > c):
    return(a)
  elif b > c:
    return(b)
  else:
    return(c)
```

[1] “elseif” ではなく “elif” なので、注意して下さい。

[2] とは言え、あらゆる条件を事前に明記することは、現実的には難しい (誤りの温床になりやすい) ので、多くの場合は「そうでなければ」/else を追記します

このプログラムでは、最初の枝分かれにある条件が二つの条件式より構成されています。複数の条件式がある場合、どんな条件式が書かれているか一見するだけでは分かりにくい、条件式間に優先度を付けたい[3]といった理由より、条件式を括弧で括る書き方がよく用いられます。上のプログラムも、そのようか書き方に沿って書いてあります。

最後に、例えば今回取り上げたような、与えられた N 個の数値から最大値を求める問題において、単純に枝分かれを使うだけでは、N が 4、5 と増えてくると条件内の比較演算が大幅に増えてしまいます (一般に最大値の決定では N^2 に比例して増えます)。N の個数が多くなれば、別の比較方法 (アルゴリズム) を検討する必要がありそうです (このテーマは、後で取り上げます)。

1.4 演習 3-2 — 誤差の補正

この演習では、これまで学んだ誤差の知識を全て活用すると共に、枝分かれの制御構造を用いて数値計算における誤差の影響を可能な限り排除し、計算精度を向上させる具体的な事例を取り上げました。数学的には、2 次方程式 $ax^2 + bx + c = 0$ の解は、$x = (-b \pm \sqrt{D})/2a,\ D = b^2 - 4ac$ より求められますが、これをそのままプログラムにしただけでは、2 次方程式の種類によっては、得られた解に誤差の影響が見られます (参考までに、2 次方程式の解を求める素直なプログラムを下記に再掲しておきます)。

```
def fa(a, b, c):
  D = b**2 - 4.0*a*c
  x1 = (-b + math.sqrt(D)) / (2.0*a)
  x2 = (-b - math.sqrt(D)) / (2.0*a)
  return(x1, x2)
```

これは、“数学による計算 $\rightarrow$ プログラムによる計算” のあらゆる場合に現れます。そこで、プログラムによる計算では、誤差の影響を可能な限り排除して、計算精度を向上させる手段がないかどうかを**常に考える**必要があります。一般に、丸め誤差はある意味、実数そのものが持ってしまう誤差とも考えられるので、これを減らすことは難しいです。これに対し、桁落ち誤差や情報落ち誤差は計算に伴い発生する誤差なので、計算の順番を変えるなどの工夫により、減らせる可能性があります。その考え方は前回のヒントに述べてあるので、再度確認して身に付けて下さい (これから先も使う基本的な考え方です)。

さて、2 次方程式の解を求めるプログラムにおいて、どこに着眼し、どのように修正するかの方針については、これも前回のヒントで述べました。具体的には、解の公式にある $-b$ と $\sqrt{D}$ の足し算/引き算のうち、「足し算」により得られる解を利用し、もう一方は解と係数の関係を用いて求めようというものでした。2 解を α, β とすると、解と係数の関係のうち引き算の可能性がない方は $\alpha\beta = \frac{c}{a}$ です。もう一方は、a, b, β の値により、引き算となる可能性があります (おっと、これは大ヒントですね)。

最後は、fa 関数内にある x1, x2 のうち、どちらが「引き算」であるかを考えることになります。一見すると、x1 の方が「引き算」になっているように見えます。x2 については、`-(b + math.sqrt(D))` より、「足し算」+「符号反転」となっているように見えます。しかし、これは `b` と `math.sqrt(D)` が正数であることが前提です。`math.sqrt(D)` が正数であることは明らかですが、(上でも述べましたが)`b` の正負は 2 次方程式の種類によって変わります。つまり、x1, x2 のどちらも、`b` の正負に応じて「引き算」を使うわけです。以上の考察をもとに、fa 関数を拡張して誤差を可能な限り取り除いた fb 関数のプログラムを示します (次ページ):

[3]例えば、条件式 1 が成り立ちかつ、条件式 2 あるいは条件式 3 が成り立つ場合は、“条件式 1 and 条件式 2 or 条件式 3” ではなく、“条件式 1 and (条件式 2 or 条件式 3)” と書く必要があります。

```
import math

def fb(a, b, c):
  D = b**2 - 4.0*a*c
  x1 = (-b + math.sqrt(D)) / (2.0*a)
  x2 = (-b - math.sqrt(D)) / (2.0*a)
  if b > 0:
      x1 = c / (a*x2)
  else:
      x2 = c / (a*x1)
  return(x1, x2)
```

この fb 関数より 2 次方程式 $x^2 - 100x + 1 = 0$ の解を求め、check 関数により検査した結果は、下記の通りです。

```
>>> x1, x2 = fb(1, -100, 1)
>>> x1
99.98999899979995
>>> x2
0.010001000200050014
>>> check(1, -100, 1, x1, x2)
(0.0, 0.0)
```

x1 については、fa 関数と fb 関数で値に変化はありませんが、x2 については変化があります (前回の資料にある値と比べてみて下さい)。この結果を見る限り、fb 関数で得られた解については、元の式に入れて丸めるときっちり 0 になるため (つまり、check 関数では誤差を検出できなかったため)、システムの限界まで精度を向上できていると言えます。

1.5 演習 3-3 — 計算の繰り返し

演習 3-3 は、testdiv2 関数を参考に計算の繰り返しを使って、このシステムで扱える最大数を調べるプログラムを作ってみようというものでした。testdiv2 関数は、最小数を調べるため、与えられた値を繰り返し 2 で割って行きましたが、今回は最大値を求めるため、与えられた値を繰り返し 2 倍して行きましょう。具体的なプログラムは、下記の通りです:

```
import math

def testdouble(x):
  while x != math.inf:
    print(x)
    x = x * 2.0
```

testdouble 関数に 1.0 を渡して実行した結果は、こんな感じになります:

```
>>> testdouble(1.0)
...
1.1235582092889474e+307
2.247116418577895e+307
4.49423283715579e+307
8.98846567431158e+307
>>>
```

この結果より、ここで使われている浮動小数点表現では、最も大きい数というのはおよそ「10^{307}」くらいであることが分かります。

2　制御構造の組み合わせ

簡単なプログラムでは、制御構造として「枝分かれ」「繰り返し」のどちらか一つだけを使えば済みますが、もう少し込み入ったプログラムになると、ある制御構造 (枝分かれ、繰り返し) の内側に、さらに別の制御構造を入れる必要が出て来ます (下記の疑似コードは一例です):

- もし～であれば、
- 　条件～が成り立つ間繰り返し:
- 　　○○をする
- 　以上を繰り返し。
- (枝分かれ終わり。)

このような例の一つとして、「0～与えられた数までを順に打ち出せ。但し、3 の倍数だけは fizz と打ち出すこと。」というプログラムを考えてみます[4]:

- fizz1: 3 の倍数の時だけ fizz
- 　変数 i を 0 から与えられた数の手前まで変えながら繰り返し、
- 　　もし i が 3 の倍数ならば、
- 　　「fizz」と出力。
- 　　そうでなければ、
- 　　i を出力。
- 　　(枝分かれ終わり。)
- 　以上を繰り返し。

これを Python に直したものは次のようになります (少し複雑ですね):

```
def fizz1(n):
  i = 0
  while i <= n - 1:
    if i % 3 == 0:
      print("fizz")
    else:
      print(i)
    i = i + 1
```

では動かしてみましょう:

```
>>> fizz1(20)
fizz
1
2
fizz
(途中略)
16
17
fizz
19
```

このように、基本的な制御構造を組み合わせて行けば、どんなに複雑なプログラムでも作成できます。これはちょうど、簡単な規則と単語から、どんなに複雑な文章でも (日本語や英語で) 作れるのと同じだと考えて下さい。

[4]海外で古くからある言葉遊びに、**fizzbuzz** というものがあります。これは、輪になって順に「$1, 2, \ldots$」と数を唱えて行く遊びです。但し、数が 3 の倍数なら「fizz」、5 の倍数なら「buzz」、3 と 5 の公倍数なら「fizzbuzz」と (数の代わりに) 言わなければならず、間違えた人は輪から抜けて行きます。

演習 4-1 まずは、上の fizz1 プログラムを打ち込んでそのまま動かせ。動いたら、繰り返しと枝分かれを組み合わせて次の動作をする Python プログラムを作成せよ。

a. 0 から与えられた数までのうち、2 の倍数でも 3 の倍数でもないものだけを順に打ち出す。

b. 0 から与えられた数までを順に打ち出すが、3 の倍数の時は fizz、5 の倍数の時は buzz、3 の倍数かつ 5 の倍数の時は fizzbuzz と (いずれも数値の代わりに) 打ち出す (fizzbuzz 問題)[5]

c. 0 から与えられた数までを順に打ち出すが、3 が付く数字の時は数値の代わりに hoge と打ち出す。

ヒント: 要は、各桁の数字が 3 かどうかを調べるわけですが、33 が与えられても hoge の出力は 1 回だけ、3 が入っていない場合は数値を出力、といった対応が必要です。これを比較演算子による論理演算の組み合わせだけで行なうことは、少々面倒です。このような場合は、フラグ (詳細は第 3 回資料の 8-9 ページを参照) を用意し、例えば、各桁の数字に一つでも 3 があればフラグを True にして以後の制御に利用する、という方針が良さそうです。

3 数値積分

これまでの知識をもとに、いよいよアルゴリズムをコンピュータ上で実行する代表的な手法の一つを取り上げてみます。具体的には本資料の冒頭でも述べましたが、「“少しだけ or 一つだけ実行する” を何度も繰り返し、その結果を集計する」という手法です[6]。以下では、**数値積分** (numerical integration — 定積分の値を数値計算により求めること) を題材に、この手法について少し検討してみましょう。

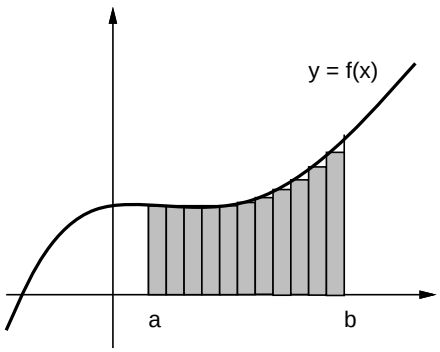

図 1: 数値積分の原理

参考: 関数 $y = f(x)$ の $x = a$ から $x = b$ までの定積分というのは、図 1 のように関数のグラフを描いたとして、区間 $[a, b]$ の範囲における関数の下側の面積です。そこで、図 1 にあるように、その部分を多数の細長い長方形に区分けしてその面積を合計すれば、知りたい面積の値、つまり定積分の値が求まることになります。各長方形の幅は区間 $[a, b]$ を n 等分した値であり (これを dx とする)、高さは $f(x)$ の値なので、その面積を計算するのは簡単です。これが皆さんよく御存じの高校数学で習うリーマン和ですが、なぜこのような話をしたかと言うと、コンピュータ上にリーマン和を求めるプログラムを作成することで、$f(x)$ の不定積分が簡単に求められないような場合であっても、定積分を近似計算できるようになるからです (数式の形で一般的に問題の解を求めることを**解析的** (analytical) に解くと言い、数値計算により特定の問題の近似解を求めることを**数値的** (numerical) に解くと言います)。不定積分のみならず微分方程式などでも、簡単に解を求められないが、その現象については具体的に捉えたい事例が数多く存在します。コンピュータによる数値的な解法の進化は、工学を始めとした様々な分野の革新的な発展を促しました。

[5] 米国では、fizzbuzz 問題のプログラムを書けないプログラマが多いので、プログラマ募集の応募者に対するふるい分けに使っている、という噂があります。本当かなぁ。

[6] これは基本的な手法なので、今後も様々な場面で出て来ます。その度に、「少しだけ or 一つだけ…」と言うのは面倒なので、取りあえず本講義では、これを “逐次細分最適化法” と呼ぶことにします (これは一般的な名称ではありません/本講義内だけの名称です)。

今回は「正しい」値が求まるかどうかをチェックしたいので、簡単な関数 $y = x^2$ で試してみます。不定積分は $\frac{1}{3}x^3$ なので、区間 $[a, b]$ の定積分は $[\frac{1}{3}x^3]_a^b$ となります。例えば、$[1, 10]$ ならば $\frac{1000}{3} - \frac{1}{3} = \frac{999}{3} = 333$ です。では、アルゴリズムを作ってみましょう:

- integ1: 関数 x^2 の区間 $[a, b]$ の定積分を区間数 n で計算
- $dx \leftarrow \frac{b-a}{n}$。
- $s \leftarrow 0$。
- $x \leftarrow a$。
- $x < b$ が成り立つ間、繰り返し:
- $y \leftarrow x^2$。 # 関数 $f(x)$ の計算
- $s \leftarrow s + y \times dx$。
- $x \leftarrow x + dx$。
- (繰り返し終わり。)
- s を返す。

まずは、x に a を格納しておき、繰り返しの中で $x \leftarrow x + dx$、つまり x に dx を足した値を作ります。それを x に入れ直すことで、x を徐々に (dx 刻みで) 動かしていき、b まで来たら繰り返しを終わります。このように、繰り返しでは「こういう条件で変数を動かしていき、こうなったら終わる」という考え方が必要なのです。面積の方は、s を最初 0 にしておき、繰り返しの中で細長い長方形の面積を繰り返し加えていくことで、合計を求めます。

では、Python のプログラムを示しましょう。「#」の右側に書かれている部分は**注記**ないし**コメント** (comment) と呼ばれ、Python ではこの書き方でプログラム中に覚え書きを入れておくことができます。コードの意味が分かりづらい (何のためにこのような計算をしているのか読み取りにくい) 箇所には、必ずその意図を注記しておくようにして下さい (**重要:** “コメント” のないプログラムは素人が書いたプログラムです)。また、一時的に命令を実行しないようにするために (言い換えれば、その命令が存在しないようにするために)、コメントを使う場合もあります。これを**コメント アウト** (comment out) と呼びます。この例でも後で使うコードをコメント アウトしてあります:

```
def integ1(a, b, n):
  dx = (b - a) / n
  s = 0.0
  x = a
# count = 0
  while x < b:
    y = x**2        # 関数 f(x) の計算
    s = s + y * dx
    x = x + dx
#   count = count + 1
#   print("count=%d x=%.20f" % ((count), (x)))
  return(s)
```

やっていることは、先の擬似コードそのままだと分かるはずです。さて、333 が求まるでしょうか? 実行させてみます:

```
>>> integ1(1.0, 10.0, 100)
337.5571499999994                ← ふーん?
>>> integ1(1.0, 10.0, 1000)
332.55462150000733               ← 小さい
>>> integ1(1.0, 10.0, 10000)
333.04545121491196               ← 大きい…
```

何だか変ですね。そこで、繰り返しの回数が幾つになっているかをチェックすることにして、上の行頭の「#」を削って動かし直してみました[7]:

```
>>> integ1(1.0, 10.0, 100)
...
count=98 x=9.819999999999990       ← x の値に誤差がある
count=99 x=9.909999999999989
count=100 x=9.999999999999989
count=101 x=10.089999999999989     ← えっ、101 回目…
337.5571499999994                  ← 101 回目の影響で大きくなっている
>>>
```

理由が分かりました。区間数が 100 個なのに、長方形を 1 個余計に加えてしまい、値が大き過ぎたわけです。何故こんなことが起きるのでしょうか? それは「$x \leftarrow x + dx$ で x を増やして行き、b になったら止める」というアルゴリズムに問題があるのです。そもそもコンピュータでの浮動小数点計算は近似値の計算なので、dx を区間長の $\frac{1}{100}$ にしたとしても、そこに (丸め) 誤差があります。このため 100 回足しても僅かに b より小さい場合があり、その時は余分に繰り返しを実行してしまいます (2 次方程式の解を求める演習でも見た通り、このような部分に、アルゴリズムに素直に従って “実際に計算するプログラム作成の難しさ” が潜んでいたりします)。

4 計数ループ

では、どうすれば良いでしょうか。繰り返し回数を 100 回と決めているので、回数を数える際は整数型で行ない[8]、それをもとに各回の x を計算するのが良さそうです。つまり、次のようなループを書くことになります (**カウンタ** (counter) とは「数を数える」ために使う変数のことを言います):

```
i = 0              # i はカウンタ
while i < n:       # 「n 未満の間」繰り返し
  ...              # ここはループ内側（ブロック）の動作
  i = i + 1        # カウンタを 1 増やす
```

このように、指定した上限まで数を数えながら繰り返して行くような繰り返しを、**計数ループ** (counting loop) と呼びます。計数ループはプログラムで頻繁に使われるため、ほとんどのプログラミング言語は計数ループのための専用機能や構文を持っています (while 文でも計数ループは書けますが、専用の構文の方が書き易くて読み易いからです)。

Python では、計数ループ用の構文として **for 文** (for statement) を用意しています[9]。これを使って上の while 文による計数ループと同等のものを書くと、次のようになります:

```
for i in range(n):
  ...
```

これは、カウンタ変数 i を 0 から始めて一つずつ増やしながら n-1 まで繰り返していくループとなります。ここで、カウンタ変数 i の範囲は、$1 \sim n$ ではなく、$0 \sim n-1$ である点に注意して下さい。十進法における各桁の数字は、1 から 10 ではなく 0 から 9 なので、大半のプログラミング言語では、カウンタの始まりを 1 ではなく 0 にしています。インデントと同様、これも是非慣れて下さい。

以後、擬似コードでは計数ループを次のように記します (“繰り返し終わり。” については、これまでの慣例通りです):

[7] つまり、変数 count で回数を数えつつ、x を表示するようにするわけです。このように、コメント アウトしてあったコードを活かして (コメントの記号を削って) 動かすことを、「コメント アウトを外す」と言います。また、ここでは、x の値を細かく見たいので、print 関数の書式に “%.20f” を指定し、十進法浮動小数点数として 20 桁を強制的に表示させている点にも注意して下さい。

[8] 整数ならば、溢れない限り誤差はありません。

[9] 多くのプログラミング言語では、計数ループを表すのに for というキーワードを使うので、計数ループのことを **for ループ** (for loop) と呼ぶこともあります。

- 変数 i を 0 から n の手前まで変えながら繰り返し、
- ... # ループ内の動作
- (繰り返し終わり。)

5 数値積分 (続き)

では、先の積分プログラムを、計数ループにより書き直してみましょう:

- integ1: 関数 x^2 の区間 $[a, b]$ の定積分を区間数 n で計算
- $dx \leftarrow \frac{b-a}{n}$。
- $s \leftarrow 0$。
- 変数 i を 0 から n の手前まで変えながら繰り返し、
- $x \leftarrow a + i \times dx$。
- $y \leftarrow x^2$。 # 関数 $f(x)$ の計算
- $s \leftarrow s + y \times dx$。
- (繰り返し終わり。)
- s を返す。

先のプログラムと違うのは、毎回 x を i から計算している部分です。では、これの Python プログラムを示します:

```
def integ2(a, b, n):
  dx = (b - a) / n
  s = 0.0
  for i in range(n):
    x = a + i * dx
    y = x**2        # 関数 f(x) の計算
    s = s + y * dx
  return(s)
```

これを動かしてみましょう:

```
>>> integ2(1.0, 10.0, 100)
328.5571499999999
>>> integ2(1.0, 10.0, 1000)
332.5546214999998
>>> integ2(1.0, 10.0, 10000)
332.9554512149995
>>>
```

今度は、刻みを小さくすると順当に誤差が減少して行きます。しかし、常に正しい面積である 333 より小さいのは、何故でしょうか? それは、長方形の面積を計算する際に、微小区間の左端にある x を使って高さを求めているため、増加関数では、図 2 (次ページ) のように微小な三角形の分だけ面積が小さめに計算されてしまうからです (逆に、減少関数だと大きめに計算されます)。

演習 4-2 上の演習問題にあるプログラムを打ち込んで動かせ。動いたら「減少する関数だと値が大きめに出る」ことを確認せよ。また、左端ではなく右端を用いた計算もしてみよ。最後に、次のような考え方で誤差が減少できるかどうか、実際にプログラムを書いて試してみよ。

a. 左端の $f(x)$ だけでも右端の $f(x)$ だけでも欠点があるので、両方で計算して平均を取る。
b. 左端や右端だから良くないので、区間の中央の $f(x)$ を使う。
c. 上記 a と b をうまく組み合わせてみる (どうすれば “うまい” のか、各自で考えよ)。

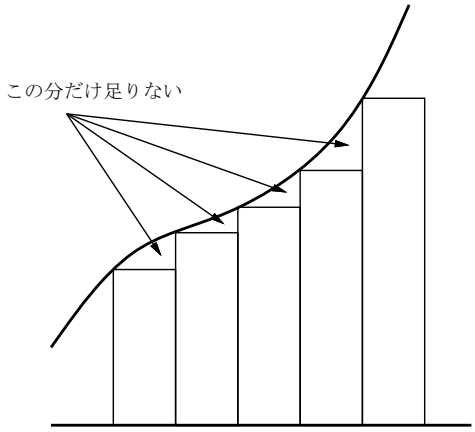

図 2: 区間の左端を使う場合の誤差

演習 4-3 次のような、繰り返しを使ったプログラムを作成せよ。

a. 整数 n を受け取り、2^n を計算する。

b. 整数 n を受け取り、$n! = n \times (n-1) \times \cdots \times 2 \times 1$ を計算する。

c. 整数 n と整数 $r\,(\leq n)$ を受け取り、${}_nC_r = \dfrac{n \times (n-1) \times \cdots \times (n-r+1)}{r \times (r-1) \times \cdots \times 1}$ を計算する。

d. x と計算する項の数 n を与えて、次の**テイラー展開** (Taylor expansion) を計算する。

$$\cos x = \frac{x^0}{0!} - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \cdots$$

$$\sin x = \frac{x^1}{1!} - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots$$

実際に値の分かる x を入れて精度を確認してみること。$\pm 10\pi$ ではどうか? n は幾つくらいが適切か?