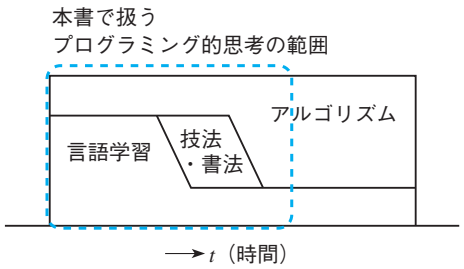
はじめに

「プログラミング的思考」とは、ある問題を解決するための方法や手順をプログラミングの概念に基づいて考えることである。この抽象的な概念は実際にプログラミングをしてみなければわからない。本書はPythonというプログラミング言語を用いて「プログラミング的思考」とは何かを解説する。

狭い意味でのプログラミング学習（デバッギング、OS関連、システム設計などを除く）は、下図のように、言語学習、技法・書法、アルゴリズム学習の組み合わせにより上達していくと考えられる。本書で扱うプログラミング的思考の範囲は、言語仕様、技法・書法、簡単なアルゴリズムである。より詳細なアルゴリズムは『Pythonによるはじめてのアルゴリズム入門』（河西朝雄著、技術評論社）を参考にされたい。



プログラミング的思考を支える5本柱として以下が考えられる。

①流れ制御構造（組み合わせ）

連接、分岐（判断）、反復などの基本制御構造を組み合わせてプログラムの骨格ができる。

②データ化

プログラムではいろいろなデータを扱う。実社会で扱うデータには様々なものがある。こうした各種データをプログラムで扱う場合にどのようにデータ化するかは重要である。

③抽象化と一般化

たとえば三角形、四角形、五角形を描く問題を「*n*角形を描く」という問題に一般化する。

④分解とモジュール化

複雑な問題の場合には、解決できる小さな問題に分解して、問題を解決しやすくする。

⑤データ構造とアルゴリズム

コンピュータを使った処理では多量のデータを扱うことが多い。この場合、取り扱うデータをどのようなデータ構造（data structure）にするかで、問題解決のアルゴリズムが異なってくる。

本書はこうした考えを基本にしながらも、プログラミング初心者がPythonを使ってモチベーションを持ちながら学習できるように簡単でも興味が持てる例題を用意した。例題を理解したうえで、練習問題を解くことにより、より理解が定着することを期待する。細かなPython文法は付録にまとめたので、必要に応じて参考にしてもらえばよい。

2024年3月
河西朝雄

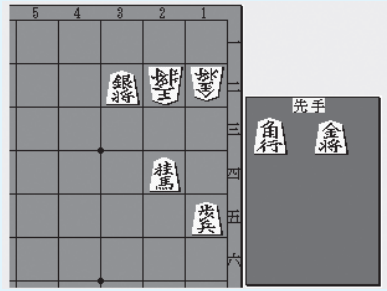
Chapter
0

プログラミングというものが存在しない古い時代から「論理的思考」という考え方はあった。

論理学では以下のようなアリストテレスの三段論法がある。三段論法とは演繹的推論を定式化したものであって、大前提、小前提、結論から成り立っている論証法である。

すべての人間は死すべきものである（大前提）
ソクラテスは人間である（小前提）
ゆえにソクラテスは死すべきものである（結論）

一般に、論理的思考とは、物事を体系的に整理し、矛盾や飛躍のない筋道を立てる思考法である。世の中には三段論法のような定式化できない複雑な論理的思考がある。たとえば、詰将棋である。以下は「1一角・同金・2三金」または「1一角・同玉・2一金」または「1一角・1三玉・1四金」の3手詰である。



この詰将棋の問題を解決するための論理的思考は人間の経験的知識を元にしていて、数学の証明のように画一的にできない。

この画一的にできない論理的思考をコンピュータが理解できる形式やコンピュータが得意とするアルゴリズムに置き換えることがプログラミング的思考である。

2010年代に将棋AIソフトが登場し、AIと人間のどちらの方が、将棋が強いのかを決めるべく、将棋電王戦が開催された。近年の将棋AIソフトはディープラーニング系のアルゴリズムの進歩でプロ棋士をもしのぐ実力を持っている。また将棋の対局中継では先手・後手の形勢判断を数値（パーセント）で表示し、最善手の候補を示す。

本書ではAIアルゴリズムのような高度なものは扱わないが「21を言ったら負けゲーム」のような比較的かんたんな問題をプログラミング的思考で解く方法を解説する。

序章

0-1 プログラミング的思考とは

文部科学省の学習指導要領では

文部科学省では学習指導要領改訂において、小・中・高等学校を通じてプログラミング教育を充実することとし、2020年度から小学校においてもプログラミング教育を導入することになった。プログラミング教育の中で重要なことの1つとして「プログラミング的思考」を掲げている。小学校プログラミング教育の手引（第三版：令和2年2月文部科学省）では、「プログラミング的思考」とは、「自分が意図する一連の活動を実現するために、どのような動きの組み合わせが必要であり、一つ一つの動きに対応した記号を、どのように組み合わせたらいいのか、記号の組み合わせをどのように改善していけば、より意図した活動に近づくのか、といったことを論理的に考えていく力」としている。

論理的思考とプログラミング的思考の違い

論理的思考とプログラミング的思考はいくつかの共通点があるものの、以下のような相違がある。

- 論理的思考
  三段論法、ロジックツリー、証明など論理的思考を実現する方法がある。論理的思考では問題を分析し、関連する事実や前提条件を考慮して論証を構築する。仮説を立て、証拠を提示し、それに基づいて結論を導き出す。

- プログラミング的思考
  論理的思考をコンピュータが理解できる形式やコンピュータが得意とするアルゴリズムに置き換えること。プログラミングでは、具体的な手順を記述し、コンピュータに命令を与えることで、目標を達成する。プログラミング的思考では、流れ制御構造、データ化、抽象化と一般化、分解とモジュール化、データ構造とアルゴリズムなどのプログラミングリテラシーが必要となる。

1-7 変数の値の更新

「n = n + 1」を数学の等式と考えれば「0=1」となり、意味不明である。プログラミングでは「=」は等号ではなく、値の代入を意味している。「=」を挟んで左辺と右辺に同じ変数がある場合は変数の値の更新が行われる。

代入演算子

変数nに対して

```
n = 1
n = n + 1
```

を行うと、nの値は「1」から「2」に更新される。「=」を代入演算子と呼ぶ。

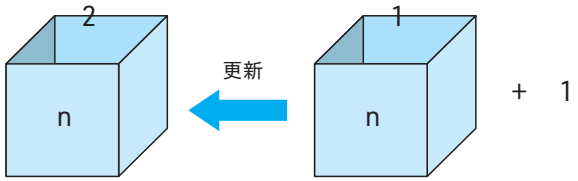


図 1.14 変数の値の更新

複合代入（累積代入）演算子

「n = n + 1」のように左辺と右辺に同じ変数がある場合は「n += 1」と書くことができる。この「+=」を複合代入演算子と呼ぶ。「+」以外にもすべての算術演算子とビット演算子が指定できる(-=、*=、/=など)。本書では変数の更新は、「+=」のような複合代入演算子を用いて行う。

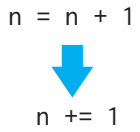


図 1.15 複合代入演算子

例題 1-7 「1+2+3+・・・+99+100」の合計を求める。

```
s = 0
for n in range(1, 101):
    s += n
print(f'合計={s:d}')
```

実行結果

```
合計=5050
```

練習問題 1-7 2^nを求めなさい。

```
n = 4
p = 1
for i in range(n):
    ①
print(f'2^{n:d}={p:d}')
```

実行結果

```
2^4=16
```

第 2 章

2-2 プログラミング技法

良いプログラムの条件

スタイル（書法）という外見でなくプログラムの内容を考えた場合に、良いプログラムの条件として以下のようなものが考えられる。

①信頼性が高いこと：Reliability

予想しない出来事に対してもできるだけエラーがなく正常に動作する

②保守容易性が高いこと：Serviceability

不具合が出たときに、他人でも修正が行えるような一般的な手法を使う

③拡張性が高いこと：Extensibility

ある機能を追加したり削除したりする場合に、プログラム全体を作り直すのではなく、その機能単位で修正できる。

④処理効率が高いこと：Processing Efficiency

同じ結果が得られても、処理効率が悪ければ、使用に耐えないことがある。

このような観点に立ち、多くの人が見ても分かりやすく、一般的で効率のよいプログラムを書くテクニックを技法と呼ぶ。技法はアルゴリズムとまではいかないが、プログラムを作成する上でのテクニック的なものである。一般にアルゴリズムは言語に依存しない抽象化されたものであるが、技法は各言語に共通するものとプログラム言語に依存するものがある。

2-3以後に各カテゴリごとに説明する。

2-3	言語仕様上の注意点
2-4	ちょっとしたテクニック
2-5	ビット演算子
2-6	文字列処理
2-7	リスト操作

2-8	クラスの活用
2-9	辞書の活用
2-10	ファイル処理
2-11	ライブラリの活用

第 2 章

2-3 言語仕様上の注意点

for in文の使い方

Pythonのfor in文はC系言語のfor文と使い方が異なるので、他の制御構造に比べ注意が必要である。Python流のfor in文の使い方は**付録**の**6.3.1　for in文**参照。

rangeの範囲

rangeの範囲をリストにして表示すると以下のようになる。指定した最終値までではなく、最終値未満までが範囲であることに注意すること。最終値まで含めたければ、「+1」した値を使う。listはリストを作成するための関数。

```
a = list(range(10))
b = list(range(1, 10))
c = list(range(-10, 10, 2))
print(a)
print(b)
print(c)
```

実行結果

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
[-10, -8, -6, -4, -2, 0, 2, 4, 6, 8]
```

整数型以外の型で範囲を作る場合

rangeには整数型しか指定できないので、その他の型を指定したいときは工夫が必要である。

- 実数型

  「-1.0、-0.8、-0.6、…、0.6、0.8、1.0」の繰り返しを作る場合、range(-1.0, 1.1,0.2)とすることはできないので、以下のようにする。

```
for i in range(-10, 11, 2):
    x = i / 10
    print(x, ',', end='')
```

4-2 データ化

プログラムでは、いろいろなデータを扱う。実社会で扱うデータには様々なものがある。こうした各種データをプログラムで扱う場合に、どのようにデータ化するかは重要である。

主に使うデータ型

格納するデータが1つなら変数を使えば良いが、大量のデータは変数に納めることはできないのでリストを使う。型の違う（数値型や文字列型）データはクラスを使う。

①変数

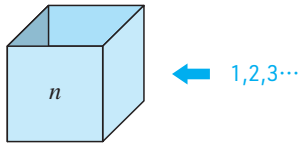


図 4.2 変数

②リスト

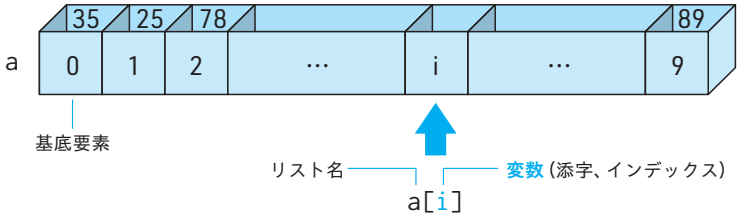


図 4.3 リスト

③クラス

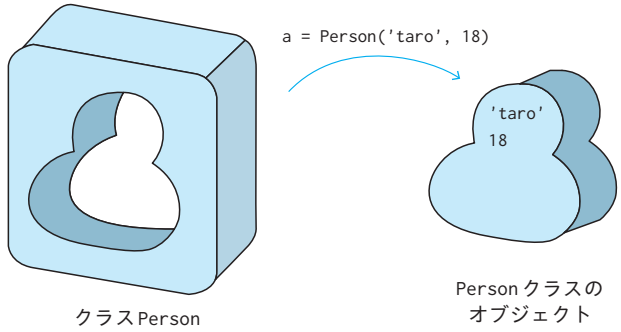


図 4.4 クラス

フラグの導入

3-4 文字の描画で、「A」のように一筆書きできない場合に負値を用いる方法を示した。ここでは、一筆書きできる直線群ごとに区別するために、新たに始点フラグ*f*を導入する。直線群の開始なら「1」、そうでないなら「0」を格納する。フラグ（flag）は旗という意味で、旗の上げ（1）、下げ（0）で状態を示す。

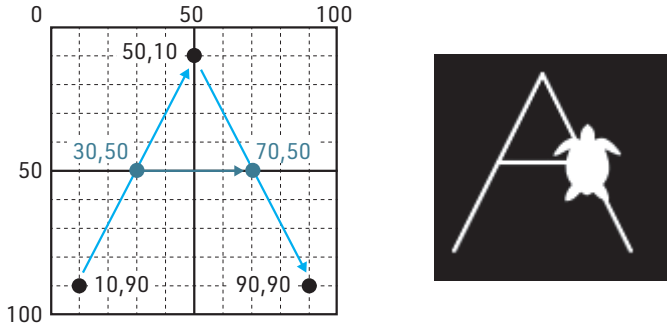


図 4.5 フラグを使った文字の描画

例題 4-2 始点フラグ*f*を用いて「A」を描く。

```
!pip3 install ColabTurtle
from ColabTurtle.Turtle import *
initializeTurtle(initial_speed=8)
```

第５章

5-6 フィボナッチ数列

イタリアの数学者フィボナッチ（Fibonacci）が1202年に著した「算盤の書」の中で、ウサギの増殖問題に関する次のような記述がある。

「1つがいの子ウサギがいる。この子ウサギは1か月後に親ウサギとなりその1か月後に1つがいの子ウサギを産む。どのつがいも死なずにこの増殖を繰り返していくと12か月後には233つがいになる。」

これがフィボナッチ数列である。フィボナッチ数列のn項は「n-2項＋n-1項」の関係がある。たとえば以下の数列中の「34」はその前の「21」とさらに前の「13」を足した数になっている。

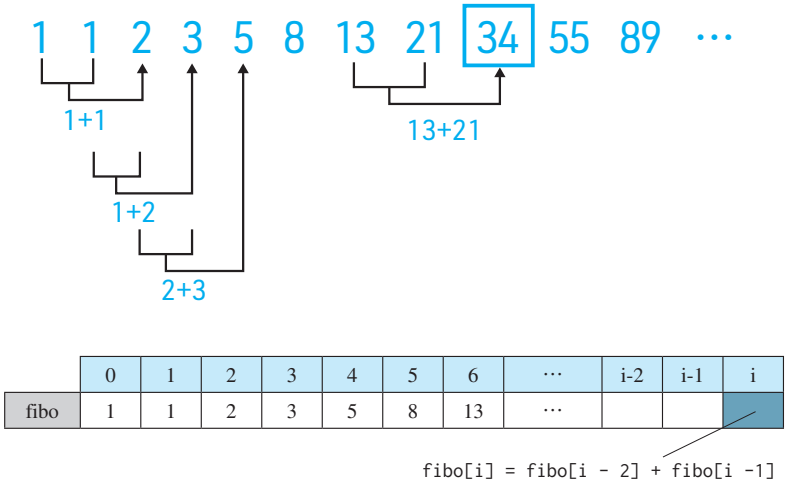


図 5.9 フィボナッチ数列

例 題 5-6 フィボナッチ数列をリストfiboに求める。


```
N = 20
fibo = [0 for i in range(N)]
fibo[0] = fibo[1] = 1
for i in range(2, N):
    fibo[i] = fibo[i - 2] + fibo[i - 1]
print(fibo)
```

実行結果

```
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, ↩
2584, 4181, 6765]
```

練習問題 5-6 フィボナッチ数列を求める関数fibを作りなさい。fib(n)で第n項のフィボナッチ数を求める。リストは使わず、変数a、bでn項の値と$n+1$項の値を表すものとする。

```
def fib(n):
    a, b = 1, 1
    for k in range(3, n + 1):
        dummy = b
        [ ① ]
        a = dummy
    return b

for n in range(1, 21):
    print(f'{n:3d}:{ [ ② ] :5d}')
```

実行結果

```
  1:    1
  2:    1
  3:    2
  4:    3
  5:    5
  6:    8
  7:   13
  8:   21
  9:   34
 10:   55
 11:   89
 12:  144
 13:  233
 14:  377
 15:  610
 16:  987
 17: 1597
 18: 2584
 19: 4181
 20: 6765
```

フィボナッチ数列は、貝の殻の巻き方や植物における枝や花の配置などの自然界におけるいろいろな並びに関係している。長さ1の正方形から初めて、フィボナッチ数列の正方形を並べて行く。長方形の中にあるそれぞれの正方形の角を滑らかに

第6章

6-3 ハノイの塔

ハノイの塔とは、以下のようなパズルである。

「3本の棒a、b、cがある。棒aに、中央に穴の空いたn枚の円盤が大きい順に積まれている。これを1枚ずつ移動させて棒bに移す。ただし、移動の途中で円盤の大小が逆に積まれてはならない。また、棒cは作業用に使用するものとする。」

ハノイの塔は再帰の典型的な例である。

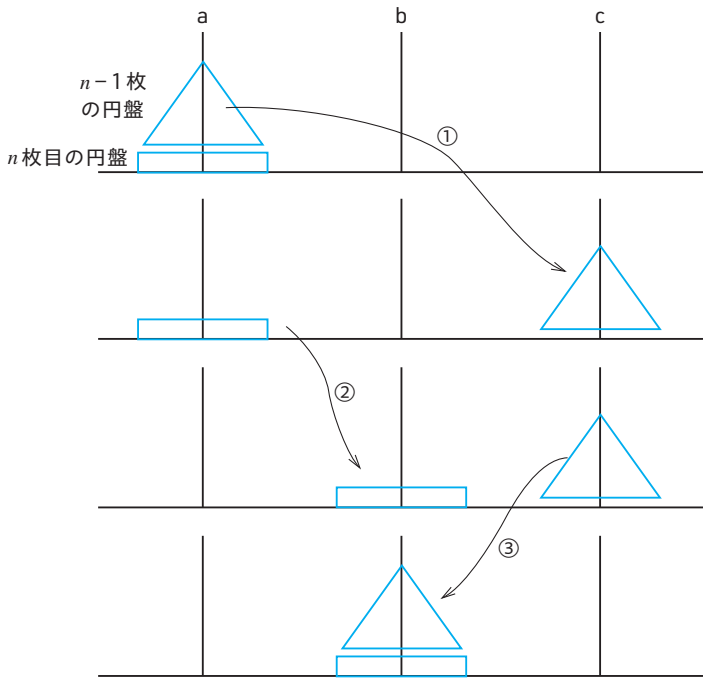


図6.4 ハノイの塔の再帰的作業

n枚の円盤をa⇒bに移す作業は、次のような作業に分解できる。①と③の作業が再帰的な作業（再帰呼び出し）となる。

①aのn-1枚の円盤をa⇒cに移す（再帰呼び出し）
②n枚目の円盤をa⇒bに移す
③cのn-1枚の円盤をc⇒bに移す（再帰呼び出し）

n枚の円盤のa→bの移動は次のように表現できる。

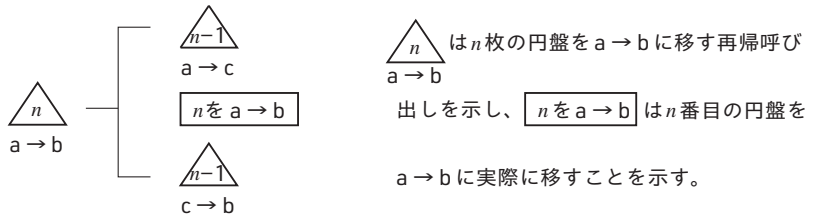


図6.5 n枚の円盤をa→bに移す再帰的な作業

4枚の場合について、hanoiがどのように再帰呼び出しされるかを**図6.6**に示す。

第7章

7-3 素数を探す

素数

素数とは、1と自分自身以外には約数を持たない数のことで、以下のようなものである。1は素数には含めない。

2、3、5、7、11、13、17、19、23、29、31、37、41、43、47、53、59、61、67、71…

例題 7-3 nが素数か、素数でないか判定する。

nが素数であるか否かは、nがn未満の整数で割り切れるか否かを2まで繰り返し、割り切れるものがあった場合は、素数でないとしてループから抜ける。ループの最後までいっても割り切れる数がなかったら、その数は素数であるとする。

なお、nを$n/2$以上の整数で割っても割り切れることはないので、調べる開始の値はnでなく$n/2$からでよいことは直感的にわかるが、数学的には$\sqrt{n}$からでよいことがわかっている。

```
import math
data = [2, 5, 21, 991]
for n in data:
    if n >= 2:
        Limit = int(math.sqrt(n))
        for i in range(Limit, 0, -1):
            if n % i == 0:
                break
        if i == 1:
            print(f'{n:d}は素数')
        else:
            print(f'{n:d}は素数でない')
```

実行結果

```
2は素数
5は素数
21は素数でない
991は素数
```


エラトステネスのふるい

素数を効率よく求める方法に、「エラトステネスのふるい」がある。

練習問題 7-3 「エラトステネスのふるい」を使って2～Nまでの素数をすべて見つけ出しなさい。

「エラトステネスのふるい」のアルゴリズムは以下のようになる。

① 2～Nの数をすべて「ふるい」に入れる。リスト`prime[i]`に「1」を格納。
② 「ふるい」の中で最小数を素数とする。下図の▼。
③ 今求めた素数の倍数をすべて「ふるい」からはずす。リスト`prime[i]`を「0」にする。下図で斜線を引いた数。
④ ②～③を$\sqrt{N}$まで繰り返し「ふるい」に残った（斜線が引かれなかった）数が素数である。

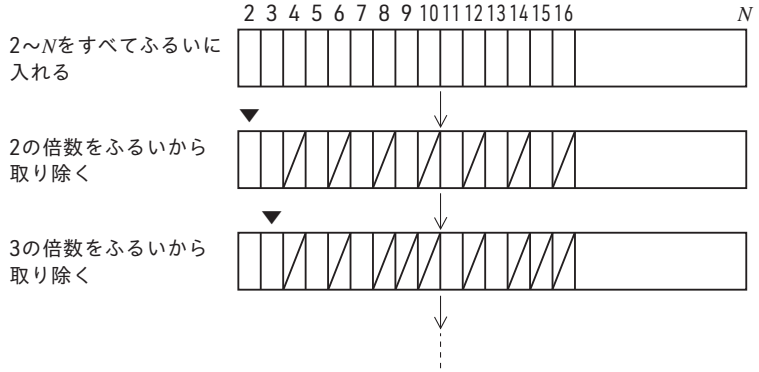


図7.7 エラトステネスのふるい

```
import math
N = 1000
prime=[1 for i in range(N + 1)]
Limit = int(math.sqrt(N))
for i in range(2, Limit + 1):
    if [    ①    ]:
        for j in range(2 * i, N + 1, i):
            if [  ②  ] == 0:
```

第8章

8-4 3D棒グラフの表示

3D棒グラフ

3D棒グラフを描くにはbar3dメソッドを使う。書式は以下である。

```
bar3d(x座標，y座標，棒グラフの底の値，棒グラフの幅，棒グラフの奥行き，z座標)
```

x、y座標からmeshgridを使って格子点(*mx*,*my*)を作る。bar3dに渡すx、y、z座標データはravelを使って1次元リストに変換したものを使う。

meshgridメソッドとravelメソッド

np.meshgridは、2組の1次元リストを受け取って格子点を生成する。

```
x = np.array([0, 1, 2])
y = np.array([0, 1, 2])
mx, my = np.meshgrid(x, y)
```

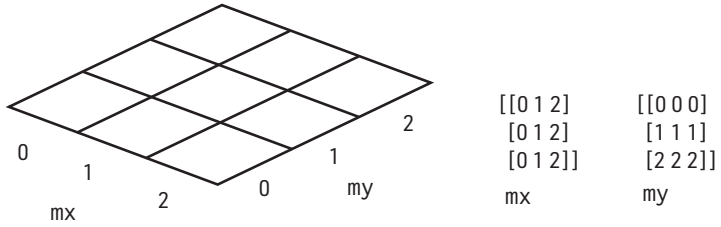


図8.5 meshgridメソッドよる格子点の生成

np.ravelは、多次元のリストを1次元のリストにして返す。

```
rx = np.ravel(mx)
ry = np.ravel(my)
```

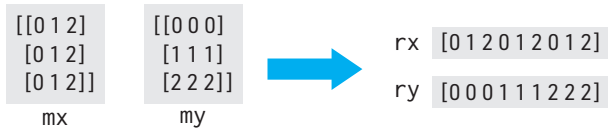
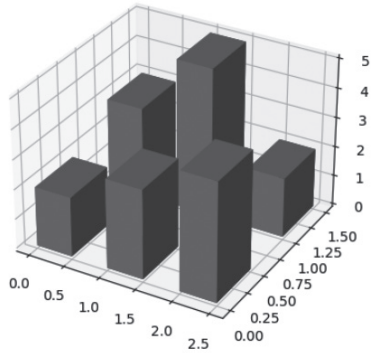



図8.6 ravelメソッドよる多次元リストの1次元化

例題 8-4 *x*軸に3項目、*y*軸に2項目の3D棒グラフを表示する。棒の高さはzに格納されている。

```
import matplotlib.pyplot as plt
import numpy as np

fig = plt.figure()
ax = fig.add_subplot(projection='3d')

x = np.array([0, 1, 2])   # np.arange(3)でも良い
y = np.array([0, 1])
z = np.array([[2, 3, 4],  # 棒の高さ
              [3, 5, 2]])

mx, my = np.meshgrid(x, y)
rx = mx.ravel()
ry = my.ravel()
rz = z.ravel()

ax.bar3d(rx, ry, 0, 0.5, 0.5, rz)
plt.show()
```

実行結果



1 バージョン

使用しているPythonのバージョンを調べておくことは重要である。バージョンによって使える機能と使えない機能がある。バージョンとプラットフォームは、以下のプログラムで調べることができる。

```
import sys
print(sys.version)
print(sys.platform)
```

実行結果

```
3.10.12 (main, Nov 20 2023, 15:14:05) [GCC 11.4.0]
linux
```

2 予約語と識別子

2.1 予約語（キーワード）

予約語はコンピュータ言語の根幹をなすもので、絶対不可侵である。Pythonの予約語は、以下のプログラムで調べることができる。

```
import keyword
for key in keyword.kwlist:
    print(key)
```

実行結果

```
False
None
True
and
as
assert
async
await
break
class
continue
def
del
elif
else
except
finally
for
from
global
if
import
in
is
lambda
nonlocal
not
or
pass
raise
return
try
while
with
yield
```

2.1.1 ソフトキーワード

通常のキーワードは語句解析で使用される。語句解析の時点では識別子として働き、構文解析のときにキーワードとして働くものをソフトキーワードと呼ぶ。「match」、「case」、「_」が該当する。ソフトキーワードは、上の予約語（キーワード）を調べた実行結果には含まれない。

2.2 識別子

変数、リスト、辞書、集合、関数、クラスなどに付ける名前を識別子という。識別子には英字（大小）と数字と「_」が使用できる。ただし、次のような名前付け規則がある。

- 先頭は数字ではいけない。
- 英字の大小は区別される。
- ifやforなどの予約語は使用できない。しかしそれらを含むものは良い。たとえば、forceなどは使える。

命名規則ほど厳格でないが、Pythonで識別子を付けるときの慣例として以下がある。

- 変数名、リスト名、関数名は小文字ベースで付ける。必要に応じて「_」を使う。たとえばmax_value。
- 定数は大文字ベースで付ける。たとえば、N、PI。
- クラス名の先頭は大文字。たとえばPerson。
- リスト名は複数形を使うという慣習があるが、本章では単数形とした。
- 'l'（小文字のエル）、'O'（大文字のオー）、'I'（大文字のアイ）を単独の識別子にしない。これらは数字の0や1と見間違いやすい。
- 組み込み関数の名前を変数として使用してもエラーにはならないが、名前の衝突を起こすので、使わないようにする。一般の言語ではsum、max、min、listなどの変数名をよく使う。Pythonではこれらは組み込み関数としてあるので、変数として使う場合は、たとえばsumならsums、sum_valueとするか、同様な意味の別の単語のtotalなどにする。Pythonは識別子にスネークケースを使う例が多いのでsum_valueが妥当かもしれないが、初心者が長い変数名を使うとタイピング量が増え煩雑になるので、本書の短いプログラムでは「sum」は単に「s」とした。

2.2.1 スネークケースとキャメルケース

識別子の命名規則には各言語で慣例がある。Pythonでは単語をアンダーバー（_）でつなぐスネークケース（SnakeCase）が使われる。単語の先頭を大文字にするキャメルケース（CamelCase）を使う言語もある。ところどころに大文字が出ることからラクダ（camel）の「こぶ」に例えた表現。

- スネークケース
  単語をアンダーバーで繋ぐ方式。break_flagなど。Pythonなどが採用。
- アッパーキャメルケース
  各単語を大文字で始める方式。BreakFlagなど。Pascal、C#などが採用。
- ローワーキャメルケース
  最初の単語だけ小文字で始める方式。breakFlagなど。Java、JavaScriptなどが採用。

2.2.2 厄介な問題

以下のような場合、変数sumではエラーにならず、関数sumでエラーとなる。

```
sum = 0
print(sum([1, 2, 3]))
```

TypeError: 'int' object is not callable

これは、「sum = 0」により識別子sumがintオブジェクトとして定義されてしまうため、sumを