# Python ドリル

自分で書いて確かめながら身につける

水谷正大

2017 年度版 ver.1.0

## ドリルの利用方法

以下の囲み内の python スクリプトは “全て” を実際に検証してください。つまり、入力し（適切なファイル名を付与して）保存・実行し、その動作結果を完全に説明できるようにするのです。もちろん、演習問題はすべて試みるべきです（ほぼ自明であるように配置してあります)。

もし理解をこえるような局面に至った場合には、当たり前であると理解している場所に戻って、再度改めて自ら手を動かして学習し直す必要があります。

これ以外に、たとえいくら丁寧に説明されたとしても、自ら取り組んで膨大な作業経験を経ない限りは学習（理解したことを自らの力に変換できること）には至らないでしょう。プログラミングに限らず全ての学習には、大量な時間の投入と集中が必要です。“涙無し” で済むような学習形態はありません。

掲載しているスクリプトやその断片は、実行させた結果との関係を検討し、さらに変数に代入した文字列や数字を変更するなどしてその実行結果がどうなるか（エラーになる場合もあるだろう）を自ら調査・確認するためのものであり、Python の完全な紹介を意図したものではありません。その気で探せば優れた参考書は沢山あります。

Web ページの記事 http://www.ic.daito.ac.jp/~mizutani/python/ も適宜参考にしてください。

# 目次


1 変数と代入 4
1.1 文字列 (string) . . . 4
1.1.1 文字列同士の演算 . . . 4
1.1.2 キーボード入力を返す . . . 5
1.2 整数 (integer) . . . 5
1.2.1 整数同士の演算 . . . 5

2 range 関数を使った for 文 6

3 条件文 7
3.1 論理式 . . . 7
3.1.1 比較演算 . . . 8
3.1.2 辞書式順序 . . . 8
3.1.3 論理演算 . . . 8
3.2 if 構文 . . . 9
3.3 if-else 構文 . . . 9
3.4 if-elif-else 構文 . . . 9
3.5 キーボード入力を使う . . . 10
3.5.1 数字文字列の整数への変換 . . . 10

4 while 文 11
4.1 無限ループ . . . 12
4.2 文字列の長さ . . . 12
4.3 平方根を求める . . . 12

5 Python スクリプトの構造 13

6 制御構造のまとめ 14
6.1 分岐構文 . . . 15
6.1.1 if 構文 . . . 15
6.1.2 if-else 構文 . . . 15
6.1.3 if-elif-else 構文 . . . 16
6.2 while 構文 . . . 16
6.3 for 構文 . . . 16

7 Python の関数 17
7.1 関数の定義 . . . 18
7.2 キーワード引数とデフォルト値 . . . 19
7.3 相互関数呼び出し . . . 19
7.4 モジュール関数の利用 . . . 20
7.4.1 モジュール time . . . 20
7.4.2 モジュール sys . . . 20

8 順序型（列型）オブジェクト 22
8.1 文字列 . . . 22
8.2 リスト . . . 22
8.2.1 リスト化関数 `list ()` . . . 23

8.3 タプル . . . 23
8.3.1 タプル化関数 tuple() . . . 23
8.4 重要: 順序型（列型）オブジェクト要素のスライス . . . 24
8.5 イテレータを使った for 繰り返し構文 . . . 25

9 文字列操作 27
9.1 文字列から replace() で不要文字を置き換える . . . 29
9.2 文字列を split() でリストに分割 . . . 32

10 リスト操作 33
10.1 リストに要素を追加する . . . 33
10.2 リストにリストを追加する . . . 35
10.3 リストを並べ替える . . . 36
10.3.1 リストメソッド sort() を使う . . . 36
10.3.2 関数 sorted() を使う . . . 37

11 集合型 38
11.1 集合オブジェクトの定義 . . . 38
11.2 集合をイテレータとして使う . . . 39
11.3 要素の集合帰属判定 . . . 40
11.4 集合同士の演算 . . . 40
11.5 集合のメソッド . . . 42

12 ディクショナリ型 43
12.1 ディクショナリの生成、値の参照 . . . 43
12.2 ディクショナリの更新と追加 . . . 44
12.3 単語リストから単語の出現頻度ディクショナリを作成する . . . 46
12.4 テキストファイルの単語の出現頻度ディクショナリを作成する . . . 47

13 複合構造を持つオブジェクトを並べ替える 48
13.1 ラムダ関数 . . . 48
13.2 並べ替えのキー . . . 49
13.2.1 タプルの集まりを並べ替える . . . 50
13.2.2 ディクショナリを並べ替える . . . 50

14 Zipf の法則を再発見する 52
14.1 登場頻度の規格化 . . . 52
14.2 標準出力のリダイレクト . . . 53
14.2.1 csv ファイル . . . 53
14.3 Zipf の法則 . . . 54
14.4 言語による差異 . . . 57

# 1 変数と代入

プログラム内で扱われるデータ（これを抽象化して**オブジェクト** (object) という[*1] ）を名前で紐付けるために**変数** (variable) を使う。変数はオブジェクトを格納しておく容器の名札 (ラベル) と見なすのである。変数にオブジェクトに割り当てることを**代入** (assign) といい、この操作を次のように**代入文** (assignment) で、次のように記号 + の左に変数名を、右にオブジェクトを書いて表す（数学における等号 = は左辺と右辺が等しいという論理的主張を意味しており、プログラミングの代入記号 + とはまったく異なる)。

**代入文**

**変数名 = オブジェクト**

次のように既にオブジェクトが代入されている変数を代入文の右辺に置いて、別の変数に代入することもできる。

**変数名** 1 **= オブジェクト**
**変数名** 2 **= 変数名** 1

Python のオブジェクトには**型** (type) がある、整数 (integer)、浮動小数点 (float)、論理値 (bool) などの内部構造を持たない**スカラー型** (scalar) と、文字列等のような内部構造を持つ非スカラー型がある。

## 1.1 文字列 (string)

**エラー有り！**

```
st1 = 'Hello, world"
print(st1)
```

```
st1 = 'Hello, world'
st2 = "I'm learning Python programming"
print(st1)
print(st2)
```

```
st1 = 'Hello, world'
st2 = "I'm learning Python programming"
st2 = "Enjoy Python program"
print(st1)
print(st2)
```

```
st1 = 'Hello, world'
st2 = "Enjoy Python program"
print(st1, st)
```

### 1.1.1 文字列同士の演算

演算子 +

```
st1 = 'Hello, world'
st2 = "Enjoy Python program"
print(st1 + st2)
```

[*1] Python におけるすべてのデータは、一貫してオブジェクトまたはオブジェクト間の関係として表されるようになっている。このようなプログラミングを**オブジェクト指向プログラム** (Object Oriented Program) といい、Python, Ruby, Java, C++ などがある。

### 1.1.2 キーボード入力を返す

input( ) [*2] 内の文字列は、‘入力を促すヒントとなるよう’ に適宜工夫する。

```
st = input("input from keyboard =")
print(st)
```

キーボード入力された値は文字列として扱われる

## 1.2 整数 (integer)

```
i = 23
print(i)
```

```
i = 23
print('i = ', i)
```

```
i = 23
st = 'Hello'
print(i, st)
```

### 1.2.1 整数同士の演算

演算子 +, -, *, /, //, % の意味を探れ。

```
i = 23
j = 9
print(i + j)
print(i * j)
print(i / j)
print(i // j)
print(i % j)
```

ユーザ（自分自身）に分かりやすいような優しい表示を心がける。

```
i = 23
j = 9
print('i =', i, 'j = ', j)
print('i + j = ', i + j)
print('i * j = ', i * j)
print('i / j = ', i / j)
```

```
i = 23
j = 9
print('i = ', i, 'j = ', j)
next = i + j
print('next = ', next)
next = next + j
```

[*2] Python2 では raw_input() を使います。

```
print('next = ', next)
next = next + j
print('next = ', next)
```

## 2 range 関数を使った for 文

**ジェネレータ** (generator) は Python の**シークエンス**（sequence: 要素の並び）を作り出すオブジェクトであり、繰り返し文の 1 つである for 文における**イテレータ** (iterator) として使うことができる（節 6.3 参照）。

その典型の 1 つである関数 `range()` は、連なった整数シークエンスを作るジェネレータです[*3]。ジェネレータはイテレータの一種で、1 つの要素を取り出そうとする度に処理を行いながら要素をジェネレートしてくれます（Pythom がやってくれる）。シーケンスが長大になりあらかじめ要素すべてを計算していおくこと（それらをみなオンメモリで読み込むなど）で計算コストが高くなる場合に、イテレータとしてジェネレータを使うようにプログラミングすることがあります。

**for 構文**

```
for  カウンタ変数 in range(m, n):
    ブロック文
```

ブロックをなす文は同じ数だけ字下げ (indent) されています。

1. range 関数 `range(m, n)` は m から n-1 までの 1 ずつ増加する整数**リスト**を生成
2. リストの先頭から順に値をカウント変数に代入
3. ブロック文を実行
4. リスト要素がなくなるまで繰り返す

`range(m, n)` が返すリストを確認する。

```
print(range(3, 10))
print(range(10, 3))
```

```
st = 'Sleping now'
for i in range(0,4):
    print(i)
    print(st)
```

```
st = 'Sleping now'
for i in range(-2,5):
    print(i, i * i, st)
```

次のスクリプトで、for 文の終了時に変数 sum にはどんな値が格納されているかを考える。

```
n = 10
sum = 0
for i in range(0, n):
    sum = sum + i
    print(sum)
```

[*3] Python2 では、連なった整数シークエンスを作るジェネレータは `xrange()` で、`range()` は連なった整数リストを返した。Python3 では `range()` に統一されてジェネレータとして使います。`range()` で生成されるベネレータをリストとするにはリスト化関数 `list()` を使って、たとえば `list(range(10))` とします（節 8.2.1 ）。

```
n = 10
sum = 0
for i in range(0, n):
    sum = sum + i
print(sum)
```

**演習** 2.1 （値の入れ替え）たとえば、次のようなスクリプト書く（一部が伏せられている)。

```
col1 = 'blue'
col2 = 'red'
print('col1 = ', col1, 'col2 = ', col2)
.... この間のスクリプトを考える
print('col1 = ', col1, 'col2 = ', col2)
```

これを実行させてみると

```
col1 = blue col2 = red
col1 = red col2 = blue
```

となって、変数内の値が入れ替わっているように、先のスクリプトを完成させ、改めてその動作を説明しなさい。

**演習** 2.2**（数の総和）** 正整数 $n$ を与えて（← 変数 n に適切な正整数値を代入するということ） $0, 1, .., n$ まで足していった総和 $0+1+2+\cdots+n$ を表示するスクリプトを考える。計算途中の総和も表示するスクリプトと、求める総和だけを表示するスクリプトの 2 通りを書く（適切なファイル名を付ける)。結果が正しいかは $n = 10$ 程度として実際に検算してみること。

**演習** 2.3 正整数 $n$ を与えて $n \times (n-1) \times \cdots \times 2 \times 1$ の計算（$n$ の階乗 $n!$ の計算）結果を表示する（適切なファイル名を付ける)。スクリプトが正しいかを $n = 10$ 程度にして実際に検算してみること。

**演習** 2.4 正整数 $n$ を与えて $0, 1, .., n$ のぞれぞれを 2 乗して足していった総和 $0^2 + 1^2 + 2^2 + ... + n^2$ を表示するスクリプトを考える。計算途中の総和も表示するスクリプトと、求める総和だけを表示するスクリプトの 2 通りを書（適切なファイル名を付ける）く。スクリプトが正しいかを $n = 10$ 程度にして実際に検算してみること。

**演習** 2.5 上記のスクリプトにおいて、$n$ としてすこし大きな正整数を代入して、その結果を求めてみなさい。

# 3 条件文

## 3.1 論理式

**論理式**とは 2 値 True と False のいずれかの値だけ返す。

**論理式**

```
print(3 > 5)
print(3 < 5)
print(3 == 5)
print(3 != 5)
print('Apple' > 'apple')
print('Apple' < 'apple')
print('Apple' == 'apple')
print('Apple' != 'apple')
```

### 3.1.1 比較演算

Python の比較演算子を表 1 に掲げる。比較演算の結果、論理値 True または False を返す

表 1　Python の比較演算子

| 意味 | 記号 |
|---|---|
| より小さい (未満 less than) | < |
| 以下 (less than or equal to) | <= |
| より大きい (greater than) | > |
| 以上 (greater than or equal to) | >= |
| 不一致 (not equal) | != |

```
x = 3
y = 5
print(x != y)
```

### 3.1.2 辞書式順序

```
print('a'< 'A')
print('0'< '9')
print('0'< 'a')
print('0'< 'A')
```

```
x = 'Apple'
y = 'Apple1'
z = 'apple'
print(x < y, x < z, y < z)
```

### 3.1.3 論理演算

かつ and または or、否定 not。比較演算と論理演算および括弧'( )' の優先順序に注意する。

```
x = 3
y = 5
z = 7
print(not x > y)
print(not (x > y))
print(x > y and x < z)
print((x > y) and (x < z))
print(x > y or x < z)
print((x > y) or (x < z))
```

```
print(not (x > y or x < z))
print(not ((x > y) or (x < z)))
print(not (x > y) or (x < z))
print((x > y or x < z) and (x > z))
```

```
x = 3
```

```
y = '3'
print(x == y)
```

## 3.2 if 構文

if 文

```
if 論理式:
    ブロック文
```

ブロック文は同じ字数だけ字下げされている。

1. 論理式を評価する。False ならば if 文を終了（何もしない）。
2. True ならブロック文を実行。

```
x = 13
if x > 7:
    print('Do you see this?')
    print('Congratulation!')
```

## 3.3 if-else 構文

if-else 構文（2 分岐）

```
if 論理式:
    ブロック文(T)
else:
    ブロック文(F)
```

ブロック文は同じ字数だけ字下げされている。

1. 論理式を評価する。
2. True ならブロック文 (T) を実行。
3. False ならばブロック文 (F) を実行。

```
x = 5
if x > 7:
    print('Do you see this')
    print('Congratulation!')
else:
    print('Oops :-(')
    print('See you next time')
```

## 3.4 if-elif-else 構文

論理式を上から評価してどこかで True になった段階でブロック文を実行し、その後の評価は行わないことに注意。

if-elif-else 構文（$n$ 分岐）

```
if 論理式 P(1):
    文(1)
elif 論理式 P(2):
    文(2)
```

```
.......
elif 論理式 P(N):
    文(N)
else:
    文(e)
```

ブロック文は同じ字数だけ字下げされている。

1. 論理式 $P_1$ を評価して、True なら**文**(1) を実行して、if-elif-else 構文を終了。
2. 論理式 $P_1$ が False のとき論理式 $P_2$ を評価して、True ならば**文**(2) を実行して if-elif-else 構文を終了。
   ⋮
3. すべての論理式の値が False 場合、**文**(e) を実行。

```
x = 5
if x > 7:
    print('Do you see this')
    print('Congratulation!')
elif x == 3:
    print('Great!')
    print('You are lucky!')
elif x == 4 or x == 5:
    print('Fine!')
else:
    print('Oops :-(')
    print('See you next time')
```

## 3.5 キーボード入力を使う

キーボードから入力した文字列をスクリプト内に渡すために、Python3 では input( ) 、Python2 では raw_input( ) を使う。

```
x = 3
y = input('Press a key with keytop 3 and Return key: ')
print('Your input: ', y)
print(x == y)
```

### 3.5.1 数字文字列の整数への変換

キーボードからの数字 (とピリオド) 入力であっても文字列として渡される (**数字文字列**)。スクリプト内で整数 (integer) あるいは実数 (浮動小数点: real) のような数値として使うためには数字文字列を**型変換** (cast) する。

```
x = 3
y = input('Press a key with keytop 3(and return) =  ')
y = int(y)
print('Your input: ', y)
print(x == y)
```

```
x = int(input('Input integer x = '))
y = int(input('Input integer y = '))
z = float(input('Input a real number z = '))
print('x = ', x, 'y = ', y)
print('x // y = ', x // y)
print('x / y = ', x / y)
print('x * z = ', z * z)
```

**演習** 3.1　キーボードから入力した 2 つの数 $m, n$ (ただし、$m < n$ とする）について、$m + (m+1) + \cdots + (n-1) + n$ を計算するスクリプトを書いて、実行しなさい。実行結果の表示においては、たとえば

```
Input m = 10
Input n = 100
Sum from  10  until 100 = 5005
```

となるようになど、何を行っているか使う者（自分）が分かるように書きなさい（常に「ユーザに優しい」配慮を工夫、努力することが実用性を向上させる)。

**演習** 3.2　演習問題 3.1 のように入力した 2 つの整数 $m, n$ について、暗黙の了解を仮定せず、入力した 2 数のうちの小さい数から大きな数までの総和を計算するようにスクリプトを書いて実行しなさい。2 整数が等しいときは総和が $m$ または $n$ になることは使ってもよい。

**演習** 3.3　if-else 文（2 分岐）だけを使って、キーボードから入力した任意の 2 整数 $m, n$ の大小関係（$m = n$, $m < n$, $m > n$ のいずれか）を判定するスクリプトを書きなさい。

**演習** 3.4　if-else 文（2 分岐）だけを使って、入力した任意の 3 整数 $i, j, k$ を小さい順に書き出すスクリプトを書きなさい。

## 4　while 文

**while 構文**

```
while 論理式:
    ブロック文
```

1. 論理式を評価する。False ならば while 文を終了（何もしない)。
2. True ならブロック文を実行。
3. 論理式の評価に戻る、を繰り返す。

```
x = 15
while x < 20:
    print('Now, inside while...')
    x = x + 1
```

```
# Assume min < max
max = 10
min = 3
sum = 0
i = min
while i < max:
    sum = sum + i
```

```
    i = i + 2
print('Sum  = ', sum)
```

## 4.1 無限ループ

```
while True:
    n = int(input('Input an integer = '))
    print('Square of ', n, ' = ', n * n)
```

このままではいつまで経っても終了しない。Ctrl + C（control キーを押しながら c キーを押すこと）で「割り込み」をかけて、コマンドを**強制終了**させる。

## 4.2 文字列の長さ

文字列を構成する文字の数を文字列の長さといい、組み込み関数 len( ) で取得できる。

```
s = input('Input an arbitrary string(and hit [Return]) = ')
while len(s) != 0:
    print('Length of word: ' + s + '= ', len(s))
    s = input('Another string = ')
print('You input empty word(length 0)!')
```

## 4.3 平方根を求める

与えた 1 以上の正実数 $x > 1$ の平方根 $\sqrt{x}$ を指定した精度で見つけることは、'数学' を使わずに算数（加減乗除）だけでしかも素早く見つけることができる。

```
x = 5.0
accuracy = 0.01
print('Find square root of ', x, ' winthin accuracy = ',accuracy)

high = x
low = 0.0
num_guess = 0
guess = (high + low) / 2.0
while abs(guess ** 2 - x) >= accuracy:
    print('guess = ', guess, ': low = ', low, ' high = ', high)
    if guess ** 2 < x:
        low = guess
    else:
        high = guess
    guess = (high + low) / 2.0
    num_guess = num_guess + 1
print('Done!')
print('It takes a number of guess = ', num_guess)
print(guess, ' is close to square root of ', x)
```

**演習** 4.1　キーボードから入力した 1 以上の数字文字列（人は '数字' だと見る文字列）を変数 x に実数（浮動

小数点）として代入するスクリプトを考える。このためには関数 `float()` を使って、たとえば

```
x = float(input('Input positive real number = '))
```

と書く。

キー入力した 1 以上の正実数の平方根を指定した精度で求めるスクリプト `square_root.py` を書いて、実行結果を報告しなさい。

**演習 4.2（単語当てゲーム）** 変数 target に適当な英単語をセットしておく。キーボードから推測した英単語を変数 guess にセットし、target と guess との辞書式順序関係を評価し、目的の単語 target が guess の後または前に登場するというヒントを表示して、一致するまで推測を何度も繰り返すスクリプト `guess_word.py` を書きなさい。ただし、次のような実行結果となるようにスクリプトを工夫する。

ヒント）while 文 `while guess != target:` を使って入力を繰り返し、繰り返し文内で if else 文で順序関係を比較する。

```
Guess a word:
Input guess word = orange
Input more previous word = lemon
Input more following word = maple
Input more following word = muscat
Input more previous word = mother
Congraturation! You hit the target =  mother
```

# 5 Python スクリプトの構造

Python の制御構文 (control statements) の基本は次の 3 つである（この他に**例外処理**を行うための構文があるが省略)。

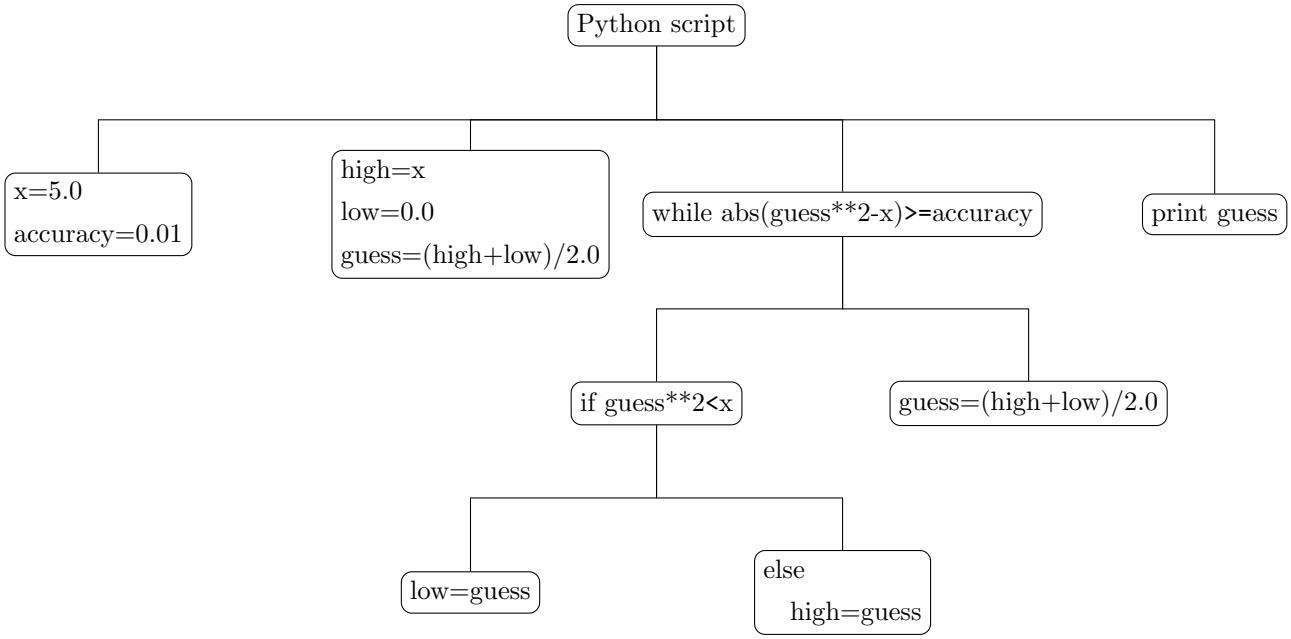
- for 文
- if 文ファミリ（if, if-else, if-elif, if-elif-else）
- while 文

Python における**文** (statement) は以下のように再帰的に定義される。

**定義 5.1（Python の文）**

1. 同じだけ字下げされた連続した文は 1 つの (ブロック）文である。
2. for 文は 1 つの文である
3. if 文の各ファミリーは 1 つの文である。
4. while 文は 1 つの文である。
5. 以上の手続きによって得られるものだけが文である。
6. 空文：0 個以上の空白と改行文字からなる行は文ではない。

たとえば、プリント (2) の [平方根を求める] のスクリプトは次のような文構造を持っている（明かな print は省略した)。




Python 文の定義から、Python スクリプトが 1 つのブロック文であることがわかる。ブロックを構成する各文は同じ文字数だけ字下げしたブロック文を持つことができる、というようにブロック文は階層構造を持つことができる。

Python の文法は、ブロック文の要素となる文として、Python 構文をなす文をいくらでも挿入することができる**文脈自由文法** (context free grammar:CFG) である[*4]。

数や文字列、論理値などの基本的な型と代入、基本制御文さえあれば、計算可能な問題はすべて記述することができる。この性質を**チューリング完全** (Turing complete) という。

記述されたプログラムの動作は**構文解析** (parsing) によって知ることができる（構文解析に失敗するとエラーとなる)。プログラミングとは、問題解決のためのアルゴリズムを考え、目的とする処理を行うように文をデザインする文芸的行為 (The Art) の総体である。

## 6 制御構造のまとめ

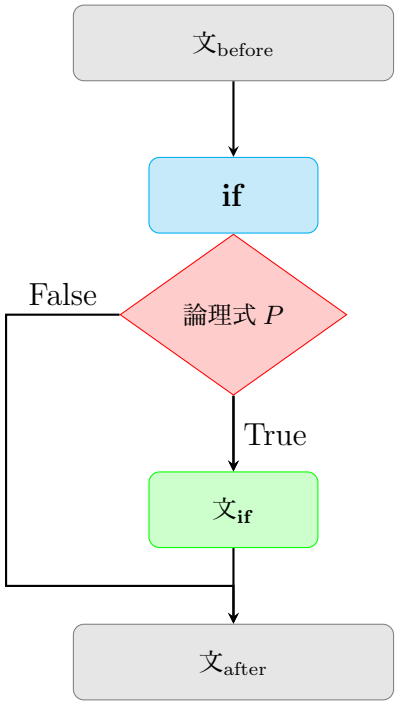
プログラムによって記述される文の並び (statements) が、プログラミングにおける**基本制御構文** for, if ファミリ（if, if-else, if-elif, if-elif-else）, while によってどのように制御されるかをまとめておく。

以下で文とあるのはブロック文として再帰的に定義されるもので、巨大になり得るものだ。

[*4] 文脈自由文法は 1956 年からの Noam Chomsky の研究に起源を持つ。

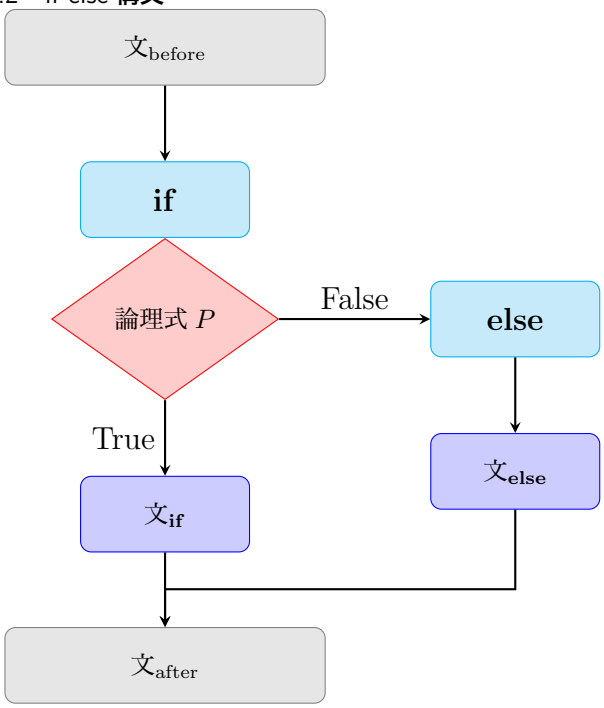
## 6.1 分岐構文

### 6.1.1 if 構文



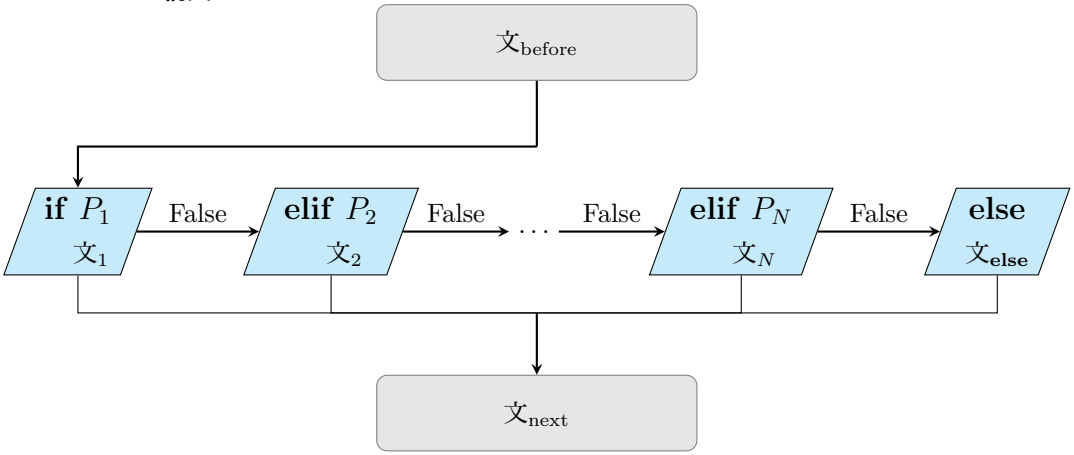

論理式 $P$ の値が True であるとき 文$_{\text{if}}$ を実行し、$P$ が False のときには何もしない（文$_{\text{if}}$ は実行されない）。

### 6.1.2 if-else 構文



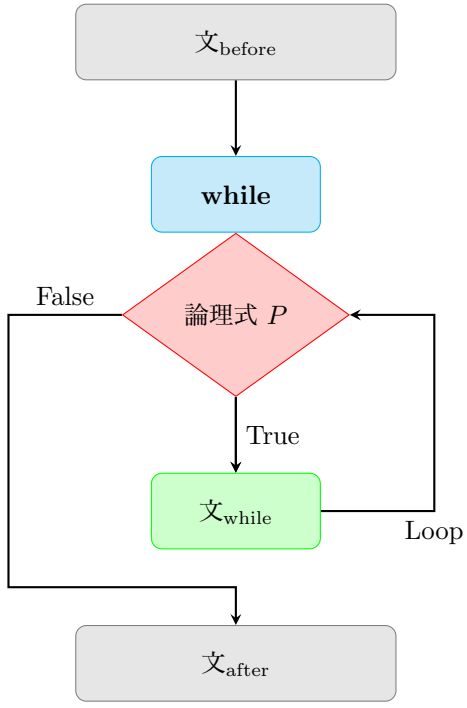

論理式 $P$ の値が True であるとき 文$_{\text{if}}$ を実行し、$P$ が False のときには 文$_{\text{else}}$ を実行する。

### 6.1.3 if-elif-else 構文




論理式 $P_1$ の値が True であるとき 文$_1$ を実行して if-elif-else 構文を終える。$P_1$ が False のとき、$P_2$ が評価され True のとき 文$_2$ を実行して if-elif-else 構文を終える。$P_1, \ldots, P_{N-1}$ がどれも False のとき、$P_N$ が評価され True のとき 文$_N$ を実行して if-elif-else 構文を終える。以上 $P_1, \ldots, P_N$ のどれもが False のときには 文**else** を実行する。

## 6.2 while 構文




論理式 $P$ を評価し True であるとき文 文$_{\text{while}}$ を実行し、再び $P$ を評価することを繰り返す。論理式 $P$ が False のとき 文$_{\text{while}}$ はスキップされ、while 文の次の 文$_{\text{after}}$ が実行される。

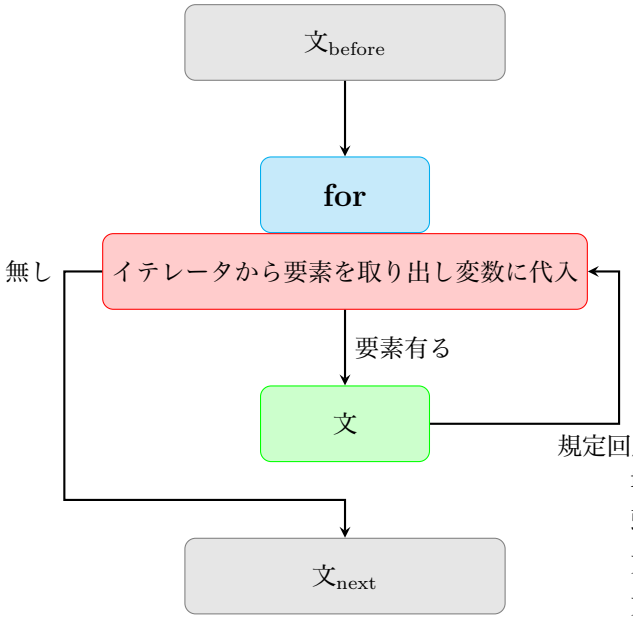
## 6.3 for 構文

for 構文は、要素を 1 つづつ取り出すことのできる**イテレータ** (iterator) を使って、取り出した要素を変数 `var` に代入しながら文を繰り返すためのもので、次のように書く。

**for 構文**

```
for var in イテレータ:
    文
```

イテレータとなり得るのは、要素間に順番がある文字列やタプルやリストなどのシークエンス型や、要素間に十分のない集合型（節 11）である。




for 行で指定された既定の長さを持つシークエンスの先頭から順に要素を取り出しながら変数に代入しながら $文_{after}$ を繰り返す。取り出す要素が無くなると for 構文の次の $文_{next}$ を実行する。

**演習 6.1**　キーボードからは正または負の数字文字列（人には '数字' だとみえる）だけが入力されると仮定し、これを浮動小数点 (float) に型変換したとする。

次の実行例のように、正実数が入力されるまでキー入力が促され続け、その平方根を指定した精度で求めるスクリプト square_root.py を数の加減乗除と while 文と if 文だけを使って書き、実行結果を報告しなさい。(ヒント): 正数プリント (3) にある 1 よりも大きい数の平方根を求める方法を 1 未満の正数についても計算できるように改良する。

```
 python square_root.py
Find square root of your input within accuracy = 0.01
Input positive real number = -5
    Oops! Your input is negative :-(
Input positive real number = -2
    Oops! Your input is negative :-(
Input positive real number = 0.7
guess = 0.5 : low = 0.0  high = 1
guess = 0.75 : low = 0.5  high = 1
guess = 0.875 : low = 0.75  high = 1
guess = 0.8125 : low = 0.75  high = 0.875
guess = 0.84375 : low = 0.8125  high = 0.875
guess = 0.828125 : low = 0.8125  high = 0.84375

Done!
It takes 6 guesses to get this value.
0.8359375  is close to square root of 0.7
```

## 7　Python の関数

プログラムにおける**関数** (function) または**手続き** (procedure) は、これを文の構成要素として呼び出して利用することができるため、スクリプトを劇的に減少させ、その結果、可読性も向上する。

関数の機能を知った利用者が関数を使うには、**関数呼び出し** (function call) の方法（とその戻り値のあり方）を知るだけでよく、関数がどのように実現されているかの詳細は知る必要はない。大抵のプログラム言語では基本的な関数が**組み込み関数** (built in function) として用意されており、さらに様々な用途のために**モジュール** (module) あるいは**ライブラリ** (libary) として提供されたものを読み込めばユーザは拡張された機能

を自由に利用することができる。

## 7.1 関数の定義

関数を定義するには、関数名と**仮引数** (arguments) の並びを括弧 ( ) 内に書く。引数をとらない関数でも括弧 ( ) は省略できない。

**関数の定義**

```
def 関数名（引数の並び）:
    関数本体の記述
```

次の例のように、関数定義では関数の働きや意味などをコメントにしておくこと、関数を定義しそれらを利用するスクリプト本体 (main script) が始まる箇所もコメントで明記するよう習慣づける。

**関数を使う例 1**

```
# return sum calculated from 0 until n>0
def total(n):
    sum = 0
    for i in range(0, n + 1):
        sum = sum + i
    return(sum)

# main script
x = int(input('Input positive integer = '))
print('sum =', total(x))
print('Done!')
```

**関数を使う例 2**

```
# return bigger number
def max_number(x, y):
    if x > y:
        return(x)
    else:
        return(y)

# main script
m = int(input('Input integer = '))
n = int(input('Input integer = '))
print('Bigger = ', max_number(m, n))
```

引数を取る必要のない関数であっても、関数定義や呼び出しの際には括弧 '( )' を省略できない

**関数を使う例 3**

```
# print greeting messages
def greeting():
    print('Hello!')
    print('How are you?')

# main script
greeting()
```

関数呼び出しは式であり、全ての式のように値を持ち、その値を関数の**戻り値**という。予約語 **return** は関数定義でのみ使うことができ、関数の**返り値**を呼び出し側に**戻り値**として返す特別な関数である。

スクリプト内での関数呼び出しは次のようになる。

1. スクリプト内の呼び出し側で**実引数**をセットして関数呼び出しを行うと、該当する関数定義の仮引数とが対応付けられる（実引数が仮引数に**渡される**という）。上の例 1 では、呼び出し `total(x)` では実引数 `x` と仮引数 `n` とが対応され、例 2 では呼び出し `max(m,n)` では実引数 `m` と仮引数 `x`、実引数 `n` と仮引数 `y` とが対応付けられる。
2. スクリプトの**実行位置**（point of execution）が、関数を呼び出した場所から関数定義の本体の先頭に移動する。
3. `return` 関数を見つけるまで、あるいは実行する文が無くなるまで関数本体の文が実行される。前者の場合には `return` 値が関数の返り値に、後者の場合の関数の返り値は `None` になる（`retun` に式がないときにも返り値は `None` になる）。例 1 や例 2 ではそれぞれ目的の計算値が返り値である。例 3 では特に返り値が指定されていないために、返り値は `None` である。
4. 関数呼び出しの戻り値は関数の返り値である。
5. 実行位置は関数呼び出しした直後のコード位置になり、スクリプトの実行は次に進んでいく。

## 7.2 キーワード引数とデフォルト値

Python では、関数引数の柔軟な活用が可能である。しばしば利用される機構が**キーワード引数** (keyword argument) とキーワード引数の方法と、引数の**デフォルト値** (default vaue) だ。

キーワード引数とは、関数の呼び出しの際に **仮引数=値**として実引数の値を渡す方法がキーワード引数である。キーワード引数に実引数が渡してあれば、その並び順は問わない。ただし、キーワード引数の方法によって実引数の並びを渡した後で、さらにキーワード引数を使わずに直接に実引数を渡して関数呼び出しするとエラーになる。

関数を定義する際に、仮引数の並びに**仮引数=値**として値を渡す方法が引数のデフォルト値である。デフォルト値が設定してあれば、関数呼び出しで、該当する仮引数への実引数渡しを省略することでき、関数本体ではそのデフォルト値が使われる。

次の例は、初項 `low` から上限 `high` まで等差 `step` の整数等差列の総和

$$\text{low} + (\text{low} + \text{step}) + \cdots + (m - \text{step}) + m, \qquad \text{high} - \text{step} < m \leqq \text{high}$$

を返す関数 `total` を定義している。2 つのキーワード変数にデフォルト値として `low = 0`, `step = 1` を設定している。

**関数を使う例 3**

```
# return sum calculated from low to high with setep
def total(high, low = 0, step = 1):
    sum = 0
    for i in range(low, high + 1, step):
        sum = sum + i
    return(sum)
# main script
print('sum = ', total(10, step = 2, low = 5))
print('Done!')
```

## 7.3 相互関数呼び出し

関数内から別の関数を呼び出すことができる。

```
# sum of two numbers
def plus(x, y):
    return(x + y)

# multipy of two numbers
```

```
def mul(x, y):
    return(x * y)

# main
n = int(input('Input integer = '))
w = float(input('Input real number = '))
print('sum = ', plus(n, w))
print('(n + 1) * w = ', plus(w, mul(n, w)))
```

## 7.4 モジュール関数の利用

Python を標準インストールした状態でも多数のモジュールが付属しており、多彩な処理が可能だ。Python では世界中で膨大なモジュール（ライブラリ）が開発されており、さらにそれらをインストールすればきわめて高度な処理を利用することができる。

インストールされているモジュールの読み込みは予約語 **import** を使う。

**関数を使う例 3**

```
import モジュール名
```

モジュール内で定義されている関数の呼び出しは **モジュール. 関数名（引数）**である。

### 7.4.1 モジュール time

Python の標準モジュール `time` は時刻に関するさまざまな関数を提供している。関数 `time.time()` は**エポック時間**[*5] からの経過秒数を浮動小数点で返す。

```
import time

# return sum calculated from low to high with setep
def total(high, low = 0, step = 1):
    sum = 0
    for i in range(low, high + 1, step):
        sum = sum + i
    return(sum)
# main script
n = int(input("input positive rather large number = "))
start = time.time()
print('sum = ', total(n))
finish = time.time()
print('Done!')
print('Calculation time[sec] = ', finish - start, '[sec]')
```

### 7.4.2 モジュール sys

ファイルの取り扱いなど気の利いた実用的プログラミングには欠かせないモジュールで、'同じ' スクリプトでも OS に違いを吸収して実行してくれる。

次の短いスクリプト `cat.py` はコマンドラインで指定したテキストファイル `targetfile` の内容を表示す

[*5] Python のエポック時間はインストールされている C ライブラリに依存するが、通常は UTC（協定世界時）での 1970/1/1 00:00:00 からの経過秒数である。

る。コマンドラインから

```
$ python cat.py targetfile
```

のように利用する。

cat.py

```
import sys

fh = open(sys.argv[1], 'r')
line = fh.readline()
while line:
    print(line.strip())
    line = fh.readline()
fh.close()
```

ここでは、文字列の先頭と末尾から指定した文字列 chars を除去するメソッド strip(chars) を引数を省略する形で使って、空白文字だけでなく' 改行文字' も剥ぎ取っている（以下の print 文で、試しに print(line) として実行してみると、この意味が理解できる)。

**演習** 7.1　実数 $x$ と非負整数 $n \geqq 0$ について、関数 power(x, n) として

$$x^n = \begin{cases} 1 & \text{if } n = 0, \\ \overbrace{x \times \cdots \times x}^{n \text{ 回}} & \text{if } n > 0. \end{cases}$$

を計算するように定義する。キー入力された実数（float(input('...')) を使う）と正整数（int(input('...')) を使う）について、その値を計算するスクリプト calc_power.py を書いて、その実行結果を報告せよ。

**演習** 7.2　$n \geq 0$ 番目の Fibonacci 数を次のように定める。

$$\begin{aligned} f_0 &= 1 \\ f_1 &= 1 \\ f_n &= f_{n-1} + f_{n-2}, \qquad n \geq 2. \end{aligned}$$

たとえば 0 番目から 7 番目の Fibonacci 数は 1, 1, 2, 3, 5, 8, 13, 21 である。次の Fibonacci 数 $f_{n+1}$ は前 2 つの数 $f_n, f_{n-1}$ の和になっている。

$$f_{n+1} = f_n + f_{n-1}, \quad n \geqq 1$$

$n \geqq 0$ 番目の Fibonacci 数の値を返す関数 fibonacci(n) を定義する。

キー入力された非負整数（int(input('...')) を使う）についての Fibonacci 数を表示するスクリプト fib.py を書いて、その実行結果を報告せよ。

**演習** 7.3　先のプログラムファイル cat.py を修正して、指定したテキストファイルをその行番号とつけて、たとえば次のように実行・表示されるスクリプト cat2.py を書いて実行しなさい。

```
$ python cat2.py cat.py
1: import sys
2:
3: fh = open(sys.argv[1], 'r')
4: line = fh.readline()
5: while line:
6:     print(line.strip())
7:     line = fh.readline()
8: fh.close()
```

## 準備

### Python shell の起動と終了

Python shell は Python で書かれたスクリプトを 1 行 1 行を直ちに評価して結果を返すので、python 語の説明には都合がよい。その代わりに、スクリプトを一括して動作処理させるプログラミングには向いていない。

コマンドプロンプト $（プロンプト記号$は自分の環境で適宜置き換えて解釈する）にコマンド python を入力すると、プロンプト>>> が現れて **python shell** に入ったことがわかる。

以下は macOS で Python3 を起動した様子である（Windows であっても動作は変わらない）。

```
$ python
Python 3.5.2 (default, Aug 16 2016, 00:56:27)
[GCC 4.2.1 Compatible Apple LLVM 7.3.0 (clang-703.0.29)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Python shell を**終了する**には、次のようにプロンプト>>> に quit() と入力するとプロンプト $に戻る。

```
>>> quit()
```

Python の働きを知るために Python shell で説明する場合があるが、そのときには Python shell プロンプト>>> を併記して混乱を避けるようにする。このドリルでは全てのスクリプトは python shell にではなくテキストファイルとして入力保存されて Python コマンドに渡して実行するという立場を再確認しておく。

# 8 順序型（列型）オブジェクト

**順序型** (oredered type) あるいは**列型** (sequence type) のオブジェクトとは、そのオブジェクトが順序関係がついた要素（それもオブジェクトだ）から構成されているもので、単に**シークエンス**とも呼ぶ。順序型（列型）には、**文字列** (string)、**リスト** (list) および**タプル** (tuple) がある。

シークエンスを構成する各要素には**順番**つまり前後関係 (order) があり、たとえば ‘シークエンスの先頭’ とか ‘シーケンスの末尾’ などという言い方には字義通り意味がある。

## 8.1 文字列

たとえば、文字列’ apple’ は長さ 5 の文字列であり、文字 (character) ’a’, ’p’,’p’,’l’,’e’ を順に並べて構成されている（文字は長さ 1 の文字列である）。より正確にいうと、先頭の文字として’a’、その次として’p’、その次として’p’、というになって、最後（末尾）の文字として’e’ が順に並んで文字列を構成している。

## 8.2 リスト

リストは Python の強力な機構を支える重要なオブジェクトだ。**リスト** (list) とは、任意のオブジェクトをカンマ (,) で区切って順番に並べて角括弧 ’[’ と’]’ で囲んだオブジェクトで、**イミュータブル**（mutable、要素の変更やや要素並び変更可能）である。

たとえば次のようにリストを扱うことができる。並べるリスト要素は同じ型である必要はないことにも注意しよう。

```
>>> a = [’apple’, 3, ’orange’, 3.1456, [1,2,3], ’T’]
>>> print(a)
[’apple’, 3, ’orange’, 3.1456, [1, 2, 3], ’T’]
```

上の例の変数 a に代入されたリストのように、リストを構成する要素としてリストオブジェクトを含んでもよい。

### 8.2.1 リスト化関数 list ()

リスト化関数 list() とは、順序型（列型）オブジェクトやタプル（要素変更不可）から要素変更可能なリストを生成する関数である。文字列をリスト化すると、文字からなるリストを返すが、リストをリスト化しても、同じリストのままだ。

```
>>> list('apple')
['a', 'p', 'p', 'l', 'e']
>>> list([3,1,4, 'apple', [4,2,7]])
[3, 1, 4, 'apple', [4, 2, 7]]
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> s = list(range(3, 10))
[3, 4, 5, 6, 7, 8, 9]
```

for を使う繰り返し構文でしばしば使われることの多い関数 range( ) が生成するリストの要素並び（特に左端と右端）を確認しておこう。

リストはミュータブルなので、リスト操作メソッド（節 10）によってリスト内容を変更すると、文字列操作 (9) とは異なりその結果を代入する必要はなく、リスト変数の内容はその段階で変わっている。

```
>>> s
[3, 4, 5, 6, 7, 8, 9]
>>> s.pop()
9
>>> s
[3, 4, 5, 6, 7, 8]
```

## 8.3 タプル

リストと同じくさまざまなオブジェクトをカンマ (,) で区切って順番に並べて記号（ と）で囲んだオブジェクトを**タプル** (taple) という。タプルはリストと似たような取り扱いもできるが、次の点でリストとは違う。

- 角括弧 [ ] でなく、(常に必要ではないが）丸括弧（ ）の中に要素をカンマ , で区切ったオブジェクトを列挙する。
- 要素の追加や削除が可能なリストとは異なり、タプルは**イミュータブル**（immutable、変更不可能）な要素並びである。

### 8.3.1 タプル化関数 tuple()

タプルは文字列やリストからタプル化関数 tuple() を使って生成できる。

```
>>> s = 'apple'
>>> tuple(s)
('a', 'p', 'p', 'l', 'e')
>>> lst = [3, 1, 'apple', [7, 5, 1]]
>>> tuple(lst)
(3, 1, 'apple', [7, 5, 1])
```

## 8.4 重要: 順序型（列型）オブジェクト要素のスライス

要素が順番に並んでいるシークエンス（順序型）オブジェクト onj に対して、その $n$ 番目の要素にアクセスして取り出すには、四角括弧 [と] を使って次のように書く。

Python がそうであるように多くのプログラム言語では**先頭を 0 番目**として順番を数える（その方が都合が良いためだ)。文字列'apple' の 0 番目は'a'、1 番目は'p'、2 番目は'p'、3 番目は'l'、4 番目は'e' である。

**$n$ 番目要素を取り出す**

```
obj[n]
```

```
>>> 'Apple'[0]
'A'
>>> 'Apple'[4]
'e'
>>> 'Apple'[5]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

指定した番号要素が目的オブジェクトになければ、添字エラー (IndexError: string index out of range) が生じる。

```
>>> t = [0, 1, 2, 3, 4]
>>> t[2:4]
[2, 3]
>>> t[5]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

**演習** 8.1　Python shell で次を確かめなさい。
リスト変数 a = ['apple', 3, 'orange', 3.1456, [1,2,3], 'T'] では、変数 a の 2 番目は文字列 'orange' で、4 番目の要素はリスト [1,2,3] である。

一般に、オブジェクト obj を構成する指定した添字番号 $start$ から $end-1$ までの要素を取り出すことを**スライス** (slice) といい、次のように書く。

***start* から *end* − 1 番目をスライス**

```
obj[start, end]
```

文字列について、スライスの実際を注意深く確認してみよう（その結果をじっくり観察することに大変意味がある)。

**演習** 8.2　Python shell で、文字列オブジェクトの要素にアクセスできることを実際に確かめよ。以下の例のようにスライス指定を省略することが可能であり、柔軟なスライスが可能になる。

```
>>> alphabet = 'abcdefghijklmnopqrstuvwxyz'
>>> len(alphabet)
26
>>> alphabet[0:3]
'abc'
>>> alphabet[3:10]
```

```
'defghij'
>>> alphabet[3:]
'defghijklmnopqrstuvwxyz'
>>> alphabet[:10]
'abcdefghij'
>>> alphabet[:]
'abcdefghijklmnopqrstuvwxyz'
'Application'
```

**演習** 8.3　スライス指定に負数を使うことができる。さらに、次の Python shell が返す例で確かめて、負数のスライス指定の意味を正確に説明しなさい。

```
>>> alphabet[-4:]
>>> alphabet[10:-4]
>>> alphabet[-10:-4]
```

実は、スライスはさらに一般的に次のように指定して、部分列を取り出すことができる。

```
obj[start, end, step]
```

**演習** 8.4　3 つ目のパラメータ指定によって、連続した部分列でなく、飛び飛びの要素をスライスすることができる。次の次の Python shell 例を、正確に説明しなさい。

```
>>> alphabet[::2]
'acegikmoqsuwy'
>>> alphabet[::3]
'adgjmpsvy'
>>> alphabet[10::3]
'knqtwz'
>>> alphabet[:20:3]
'adgjmps'
>>> alphabet[::-1]
'zyxwvutsrqponmlkjihgfedcba'
```

特に、最後の例は大変著しい結果をもたらすことに注意しよう。

## 8.5　イテレータを使った for 繰り返し構文

文字列をなす各要素を表示する for 文スクリプトをさまざまなに書くことができる。

文字列がシークエンスであり、その長さを持つ。長さを返す関数 `len( )` を使って、1 文字ずつスライスするやり方がある。

```
s = 'apple'
for k in range(0, len(s)):
    print(s[k])
```

for 構文は、次のように取出し可能オブジェクトオブジェクトを**イテレータ** (iterator) として、次々に要素を取り出すようして定義できる。**取出し可能**とはオブジェクトを構成する要素を何らかの仕方で全て 1 つずつ取り出せる仕方を Python が知っているということだ。取出し可能オブジェクトには、**順序型**（列型）や**集合型**がある。

Python の for 構文

```
for 変数 in イテレータ:
    文
```

1. 取り出し可能オブジェクトのコレクションであるイテレータを用意。
2. コレクションから要素が取り出されて（コレクションから要素が 1 つ減る）変数に代入され
3. 文を実行する
4. コレクションから取り出される要素が無くなるまで繰り返す。

次は文字列を構成する各文字をイテレータから取り出す for 文である。

```
s = 'apple'
for ch in s:
    print(ch)
```

**演習** 8.5　上の結果を確かめなさい。

イテレータは、角括弧'[' と']' とで要素のカンマ (,) で区切られているリストであってもよい。

```
lst = [2.3, 3.5, 4, 9.0, -6.2, 1.5, -3.9, 2]
total = 0
for k in lst:
    print(k)
    total = total + k
print('Sum = ', total)
```

**演習** 8.6　上の結果を確かめなさい。

イテレータは、集合型、中括弧'{' と'}' とで要素のカンマ (,) で区切られていてもよい。ただし、集合オブジェクトには要素間には順番は存在しないため、集合を定義している要素の順番通りに取り出されるわけではないことに注意する。

集合型を使う for 文

```
s = {1, 'apple', 3, 2, 'apple', 9, 1, 2, 'orange', 2}
for el in s:
    print(el)
```

**演習** 8.7　上の結果を確かめなさい（重要)。

次のプログラム rand_sum.py は、乱数モジュール random をインポートしている。空リストを用意して、0.0 と 1.0 間の一様乱数を指定された回数 $n$ 生成しながらリスト末尾に追加する。こうして得られたリストを for 文のイテレータとして、それらの合計を計算している（合計するだけなら、わざわざリスト作成する必要はない)。

rand_sum.py

```
import random

n = 1000
lst = []
for i in range(0, n):
    lst.append(random.random())
total = 0
for rnd in lst:
```

```
    total = total + rnd
print('Sum of the generated random numbers = ', total)
```

**演習** 8.8　上のプログラム rand_sum.py を実行しなさい。繰り返し回数を増やしていくと、どのような結果となるかを論じなさい。

**演習** 8.9　**回文** (palindrome) とは、"Madam, I'm Adam"のように先頭から読んでも末尾から読んでも文字の出現順序が同じかつ言語として有意味な文字列である。古来から言葉遊びの 1 つとして収集されている（文字列ではなく音節とした場合に回文を構成することは大変難しい！）[*6]。回文からなる文字列集合は文脈自由言語をなしている。

ローマ字（と空白）からなる回文を収集せよ。

**演習** 8.10　引数で指定した文字列が回文かどうかを判定する関数 ispalindrome(値 True または False を返す）を書いて、次のスクリプト judge_palindrome.py を完成し、実行結果を報告せよ。

judge_palindrome.py

```
def ispalindrome(str):
    ....

str = input('Input a candidate for palindrome word = ')
print('Your input is ', ispalindrome(str)"
```

# 9　文字列操作

文字処理はたいていの場合、ソフトウエア開発の核心部分をなしている。表 2 に文字列操作のメソットとその意味を紹介した（すべて括弧'( )' が付いていることに注意)。

表 2 の文字列操作メソッドは、Python が提供する有用な機構で、今後盛んに利用ために完全な理解を要するため、表 2 の文字列操作メソッドのそれぞれについて、実際にプログラム動作を確認し、その働きが正確に説明できるようにしておく必要がある。文字列操作の上手く利用すると、プログラムの長さを劇的に短く、かつ洗練したコード記述を可能とする。

**演習** 9.1　Python Shell で以下のように、文字列リストにメソッド join() を使った動作を確認し、正確に説明してみなさい。

```
>>> str = 'apple'
>>> clist = list(str)
>>> clist
['a', 'p', 'p', 'l', 'e']
>>> '-'.join(clist)
'a-p-p-l-e'
>>> ')('.join(clist)
'a)(p)(p)(l)(e'
>>> ''.join(clist)
'apple'
```

**演習** 9.2　Python Shell で以下のように、文字列リストにメソッド lower(), upper(), title() を使った動作を確認し、正確に説明してみなさい。

```
>>> str = 'I have a pen'
```

[*6] 楽譜記号とした場合には、Joseph Haydn の交響曲第 47 番 (1772) が有名だ。第 3 楽章のメヌエットの第二部 (Minuetto al Roverso) が楽譜回文列になっている。

```
>>> str = str.lower()
>>> str
'i have a pen'
>>> str = str.upper()
>>> str
'I HAVE A PEN'
>>> str = str.title()
>>> str
'I Have A Pen'
```

表 2: 文字列のメソッド

| method 名 | 使用例 | 意味 |
|---|---|---|
| 大文字/小文字の変換 | | |
| **lower()** | `str.lower()` | 文字列 `str` の全ての文字を小文字にして返す。`str` の内容は変わらない。 |
| upper() | `str.upper()` | 文字列 `str` の全ての文字を大文字にして返す。`str` の内容は変わらない。 |
| swapcase() | `str.swapcase()` | 文字列 `str` の大文字を小文字に, 小文字を大文字に変換して返す。`str` の内容は変わらない。 |
| capitalize() | `str.capitalize()` | 文字列 `str` の最初の文字のみを大文字にしてして返す。`str` の内容は変わらない。 |
| title() | `str.title()` | 文字列 `str` 内の各単語の最初の文字のみを大文字にしてして返す。`str` の内容は変わらない。 |
| 文字列の置換 | | |
| **replace()** | `str.replace(find, put)` | 文字列 `str` 内の文字列 `find` を検索し、文字列 `put` にすべて置き換えた結果を返す。`str` の内容は変わらない。 |
| 文字文字列の位置の判定 | | |
| count() | `str.count(item)` | 文字列 `str` 内の文字列（文字）`item` の登場回数を返す。 |
| find() | `str.find(item)` | 文字列 `str` 内の文字列（文字）`item` が初めて登場する位置を返す。 |
| rfind() | `str.rfind(item)` | 文字列 `str` 内の文字列（文字）`item` が最後に見つかった位置を返す |
| index() | `str.index(str)` | 文字列 `str` 内の文字列（文字）文字列 `str` を左から探して最小（最左端）のインデックスを返す。見つからねばエラーを返す。 |
| rindex() | `str.rindex(s)` | 文字列 `str` 内の文字列（文字）文字列 `s` を左から探して最大（最右端）のインデックスを返す。見つからねばエラーを返す。 |
| 文字列をリストに変換 | | |
| **split()** | `str.split(st)` | 文字列 `str` を指定した文字列 `st` を区切りとして部分文字列に分割し、部分文字列のリストを返す (文字列 `st` は部分文字列には現れない)。文字列 `st` を明示せずに省略したときは、空白 1 文字 ␣ で分割される。 |
| join() | `sep.join(list)` | 文字列を要素とするリスト `list` の要素を文字列 `sep` 挟み込んで連接した文字列を返す。`sep` が ’’（空文字）のときはリスト `list` の要素をそのまま並べた文字列を、`sep` が 空白 (’␣’) としたときは空白を挟んで並べた文字列が返る。 |

表 2: 文字列のメソッド

| | | |
|---|---|---|
| splitlines() | `str.splitlines()` | 文字列 `str` を改行部分で分解し、各行からなるリストを返す。引数を指定しない（`splitlines()` 限り、返されるリスト要素に改行は含まない。 |
| 文字列を含むかを判定 | | |
| startswith() | `str.startswith(item)` | 文字列 `str` が文字列（文字）`item` で始めるかどうかで論理値 True または False を返す |
| endswith() | `str.endswith(item)` | 文字列 `str` が文字列（文字）`item` で終わるかどうかで論理値 True または False を返す。 |
| 文字列の構成要素の判定 | | |
| isalnum() | `str.isalnum()` | 文字列 `str` がすべて英数文字かどうかかどうかで論理値 True または False を返す。 |
| isalpha() | `str.isalpha()` | 文字列 `str` がすべてがすべて英字かどうかで論理値 True または False を返す。 |
| isdigit() | `str.isdigit()` | 文字列 `str` がすべてがすべて数字かどうかで論理値 True または False を返す。 |
| islower() | `str.islower()` | 大小文字区別のある文字列 `str` が全て小文字かどうかで論理値 True または False を返す。 |
| isupper() | `str.isupper()` | 大小文字区別のある文字列 `str` が全て大文字かどうかで論理値 True または False を返す。 |
| isspace() | `str.isspace()` | 文字列 `str` が全て空白文字かどうかで論理値 True または False を返す。 |
| istitle() | `str.istitle()` | 文字列 `str` がタイトルケースかどうかで論理値 True または False を返す。 |

**演習** 9.3 「Python メソッド名」で検索して、その利用法を Python shell などで確かめなさい。中でも、メソッド lower(), replace() および split() はこの小冊子では多用するため、使えるように完全に理解しておくことが必要だ。

## 9.1 文字列から `replace()` で不要文字を置き換える

メソッド `replace(find, put)` は目的の文字列において、左端から文字列 find を探して文字列 `put` に置き換えしながら右端まで置き換え操作した結果を返す。

**文字列操作メソッド** `replace(find, put)`

**文字列**`.replace(find, put)`

目的の文字列に探している文字列 `put` が見つからないときにには、目的の文字列をそのまま返す。

**演習** 9.4 以下の例を実際に確かめなさい。

```
>>> st ='apapapaa'
>>> st.replace('ap','')
'aa'
>>> st.replace('pa','')
'aa'
>>> st.replace('app', 'xx')
```

```
'apapapaa'
>>> st
'apapapaa'
```

実用的な文字列の置き換え操作を確かめて見るために、次のプログラム replace_string00.py を考えてみる。長〜い（表示上では 4 行にわたっているが、改行なく 1 行と考える）文字列を変数 str に代入しておく[*7]。文字列中で 1 重引用符 (') を使っているために、ここでは文字列全体を 2 重引用符 (") ではさんでいる。表示上は複数行にまたがっているが、文字列として途中で改行はなく一行である。

replace_string00.py

```
# -*- coding: utf-8 -*-

str = "Alice was beginning to get very tired of sitting by her sister on the bank,
and of having nothing to do: once or twice she had peeped into the book her sister
was reading, but it had no pictures or conversations in it, 'and what is the use of
a book, 'thought Alice 'without pictures or conversations?'"

modified = str
modified = modified.lower()
modified = modified.replace(',', ' ')
modified = modified.replace('.', ' ')
modified = modified.replace("'", ' ')
modified = modified.replace('?', ' ')

print(str)
print('-----')
print(modified)
```

プログラム replace_string00.py の実行結果は次のようになる。

```
$ python replace_string00.py
Alice was beginning to get very tired of sitting by her sister on the bank, and of having
nothing to do: once or twice she had peeped into the book her sister was reading, but it
had no pictures or conversations in it, 'and what is the use of a book,' thought Alice
'without pictures or conversations?'
-----
alice was beginning to get very tired of sitting by her sister on the bank  and of having
nothing to do: once or twice she had peeped into the book her sister was reading  but it
had no pictures or conversations in it   and what is the use of a book   thought alice
  without pictures or conversations
```

文字列 str の内容は、文字列 modified にそのまま代入された後に、文字列操作の結果が modified に次ぐ次と上書きされていく。lower() で小文字化され、文字列カンマ (',') を空白文字 ('␣') に、次いでピリオド ('.')、1 重引用符 ("'")、疑問符 ('?') が空白文字 ('␣') 置き換えられて modified に上書きされている。

このプログラム replace_string00.py を修正して、文字列 string を不要文字を空白文字に置き換えして返す関数 repalce_string_with_spaces(string) を使って次のようにスッキリと replace_string0.py 書くことができる。

[*7] Lewis Carroll の Alice's Adventures in Wonderland(1916) の一文である（リンク先は UTF-8 文字符号化されたテキストファイル）。

replace_string0.py

```
# -*- coding: utf-8 -*-
def repalce_string_with_spaces(string):
    modified = string
    modified = modified.lower()
    modified = modified.replace(',', ' ')
    modified = modified.replace('.', ' ')
    modified = modified.replace("'", ' ')
    modified = modified.replace('?', ' ')
    return(modified)

str = "Alice was beginning to get very tired of sitting by her sister on the bank,
and of having nothing to do: once or twice she had peeped into the book her sister
was reading, but it had no pictures or conversations in it, 'and what is the use of
a book, 'thought Alice 'without pictures or conversations?'"

print(str)
print('-----')
print(repalce_string_with_spaces(str))
```

さらに単語に属さないような文字記号（たとえば 2 重引用符 ('"')、コロン (':')、セミコロン (';') などなど）を考えて、これらを順次、空白文字 ('␣') に置き換えた後に空白文字列で区切られた英語単語からなる文字列を表示するプログラムに修正することは容易である。

**演習** 9.5　上の関数 `repalce_string_with_spaces(str)` を以下のように改良することを考えよう。関数内において、単語に属すとは思われないような記号からなる文字列、たとえば `',.:;/><[](){}=!?@#$%^&*+'` を文字列 `removing_character_string` として与える。for 文でこの文字列をイテレータとして使い、置き換えるべき文字列を 1 文字ずつ取り出して、文字列内を検索して空白文字 ('␣') に置き換えることを繰り返す。必要なら、さらに文字列に処理を加える（たとえば、1 重引用符 (') を空白文字に置き換える)。このように関数 `repalce_string_with_spaces(str)` を書き直す。

先で作成したプログラム `replacing_string0.py` を修正し、完成した関数 `repalce_string_with_spaces(str)` を使ったプログラム `replacing_string.py` として保存する。この関数が上手く働くかどうか、与える文字列をさまざまに工夫して、'意地の悪い' 文字列をいくつか与えてその表示結果を検討しなさい。

**改行文字** (文字記号 '\n') や**タブ** (文字記号 '\t') もそれぞれ不可視空白文字の一種である。先の関数 `repalce_string_with_spaces(str)` のさらなる改良を考えよう。**改行文字**や**タブ**文字からなるような文字列[*8]には改行が含まれている. 表示上は 2 行にまたがっているが、一行の文字列である。

```
str = '\tApril is the cruellest month, breeding\n\tLilacs out of the dead land,
 mixing\n\tMemory and desire, stirring\n\tDull roots with spring rain.'
```

**演習** 9.6　この文字列 `str` をプリントしてみなさい。先の関数 `repalce_string_with_spaces(str)` で、改行やタブ文字も空白文字 ('␣') に置き換えるように修正しなさい。実際に、この関数が上手く働くかを実行し確認してみなさい。

[*8] The Waste Land(T.S. Eliot, 1922) の一文である（リンク先は UTF-8 文字符号化されたテキストファイル)。

## 9.2 文字列を split() でリストに分割

テキスト処理の基本は、何行もの文字列からなるテキストを 1 行ずつ読み込んで、その文字列にさまざまな処理を施すことである。節 7.4.2 では、自明な処理として読み込んだ 1 行をそのままプリントしてファイル内容を閲覧する短いプログラムを紹介した。

非自明な処理として、読み込んだ 1 行をそれを構成している単語に分割することが考えられる（テキスト処理—たとえば英語読解意の難しさは、意味を確定するためにそれらの単語の構文上の役割を決定する構文解析にある）。

テキストの構文解析はこのテキストのレベルを越えてしまうが、登場頻度の高い単語を調べることでテキスト上での役割を推測することは可能である。実際、長大なテキストを与えて、そこで登場した単語の頻度表を作成することは実用上の意味がある。

Python では、目的とする文字列に対して指定した文字列 sep を単語境界（分割子）としてその文字列を文字列（単語）に分割し、分割された単語を要素として並べてリストとして返すメソッド .split() が用意されていてたいへん重宝する。

**文字列操作メソッド .split(sep)**

**文字列**.split(sep)

区切り文字 sep を省略して .split() とした場合、連続する空白文字（空白[*9] ␣、改行文字 '\n'、タブ文字 '\t'）を一つの分割子と見なして文字列（単語）に分割して要素の並びからなるリストを返す。

```
>>> str = 'I have an apple.  You have a pineapple.'
>>> str.split()
['I', 'have', 'an', 'apple.', 'You', 'have', 'a', 'pineapple.']
```

**演習 9.7**　次は、Alice's Adventures in Wonderland（Lewis Carroll, 1865）の冒頭の一節を文字列 str に代入して、区切り文字としてカンマ（,）および空白を指定した場合のスクリプトである。これを実行して、返される文字列リストを検討せよ

string_split0.py

```
str = "Alice was beginning to get very tired of sitting by her sister on the bank,
and of having nothing to do: once or twice she had peeped into the book her sister
was reading, but it had no pictures or conversations in it, 'and what is the use of
a book,' thought Alice 'without pictures or conversations?'"
print(str.split(','))
print('------')
print(str.split(' '))
```

**演習 9.8**　演習 9.7 を、演習 9.5 や演習 9.6 で行ったように、文字列において英単語を構成しないような各種記号を空白文字␣で置き換えた文字列を返す関数 repalce_string_with_spaces(str) を使って、与えた文字列を空白文字列で切り出した英単語リストを出力するプログラム string_split.py を書いて実行しなさい.

**演習 9.9**　与えたリストに対して（たとえば、以下）、リスト要素とその重複回数（単語の頻度）を求めるにはどうすればよいだろうか、考えてみなさい。

```
['I', 'have', 'an', 'apple.', 'You', 'have', 'an', 'apple.']
```

[*9] 空白 1 文字を' 'とするだけでは紛らわしいので、ここでは特殊記号を使って'␣' と表記している。

# 10　リスト操作

Python プログラムではリストはたいへん柔軟で強力な処理手段を提供する。表 3 に代表的なリスト操作メソッドを挙げた。

節 8.2 でも注意したように、リストオブジェクトは**イミュータブル** (mutable) で要素変更可能で、リスト操作メソッドを使ってリスト内容を変更すると、リスト内容は直ちにその内容が反映される。ただし、表 3 に併記したリストを並べ替えた結果を返す関数 `sorted()` は指定したリストそれ自体は変更せず、並べ替え結果を返す。

## 10.1　リストに要素を追加する

メソッド `append()` を使って、**リストの末尾に要素を追加**するには、次のように追加する要素 `item` を引数にセットする。

**リストに要素を追加する**

**リスト**`.append(item)`

```
>>> t = ['apple', True]
>>> t.append(4)
>>> t
['apple', True, 4]
>>> t.append(False)
>>> t
['apple', True, 4, False]
>>> t.pop()
False
>>> t
['apple', True, 4]
>>> t.append('orange')
>>> t
['apple', True, 4, 'orange']
>>> t.pop()
'orange'
>>> t
['apple', True, 4]
```

この例のように、リスト要素の追加 `append()` と**リスト要素の取り出し** `pop()` は原則、リスト末尾に対して行われることに注意する。

**演習** 10.1　空リスト `charlist` を用意しておく。キーボードから文字列 `str` を入力して、リスト化関数 `list()` を利用せず、for 文でイテレータとして textttstr から 1 文字ずつ取り出し、リスト要素として `charlist` に追加していく `string2charlist.py` を考える。このプログラムを実行しなさい。

たとえば、次のような実行結果となる。

```
$ python string2charlist.py
Input a string = Thank you.
List of the input string =  ['T', 'h', 'a', 'n', 'k', ' ', 'y', 'o', 'u', '.']
```

次のプログラム `append_list.py` は、まず空リスト `randlist = []` を用意しておき、これに乱数モジュール random を使って平均 $\mu$、分散 $\sigma$ の正規分布 $N(\mu, \sigma)$ に従う乱数を発生させて、リストに次々と追加して

いく。確率論的には、有限回の繰り返した後にプログラムは終了する。

append_list.py

```
# -*- coding: utf-8 -*-
import random

randlist = []
# 生成した乱数の数
sample_number = 1

# 平均 50、分散 20 の正規分布に従う乱数
rand = random.gauss(50,20)
while rand < 90:
    randlist.append(rand)
    rand = random.gauss(50,20)
    sample_number += 1

print(randlist)
print('Sample Number = ', sample_number)
```

**演習** 10.2　上のプログラム append_list.py を実行しなさい。正規乱数値が平均値 + 分散値より大きく離れるほど、リストに追加される乱数の数が多数になることを確かめなさい。

表 3: 文字列のメソッド。リストソート関数 sorted() も併記。

| method 名 | 使用例 | 意味 |
|---|---|---|
| append() | list.append(item) | リスト list の末尾に要素 item を追加する。リスト list の内容は変化する。 |
| insert() | list.insert(k, item) | リスト list の $k$ 番目位置に末尾に要素 item を挿入する．リスト list の内容は変化する。 |
| extend() | list.extend(applist) | リスト list に別に用意したリスト applist を末尾に追加する。リスト list の内容は変化する。 |
| pop() | list.pop() | リスト list 末尾の要素を削除し、その要素を返す。リスト list の内容は変化する。 |
| pop() | list.pop(k) | リスト list の $k$ 位置の要素を削除し、その要素を返す。リスト list の内容は変化する。 |
| remove() | list.remove(item) | リスト list 内で値 item を持つ最初の要素を削除する。リスト list の内容は変化する。該当する要素が無ければエラー。 |
| reverse() | list.reverse() | リスト list の要素の並びを逆にする。リスト list の内容は変化する。 |
| index() | list.index(item) | リスト list 内で item が最初に登場する位置を返す。リストに item がなければエラー。 |
| count() | list.count(item) | リスト list 内にある item の数を返す。リストに item がなければ 0 を返す。 |

表 3: 文字列のメソッド。リストソート関数 sorted() も併記。

| | | |
|---|---|---|
| sort() | list.sort() | リスト list 内の要素を並べ替える。リスト list の内容は変化する。デフォルトで sort(key=None, reverse=None) とオプション key と reverse をもつ（オプションを省略したときはデフォルト値 None が指定されたとする）。reverse=True で降順並べ替え。key を使って、比較を行う前にリストの各要素に対して呼び出される関数を指定。 |
| リストをソートする関数（指定したリスト内容は変更しない） | | |
| sorted() | sort(list) | リスト list そのものは変更せずに、ソート済みの結果を返す。リストのソートメソッドと同様に、オプション key と reverse をもつ（オプションを省略したときはデフォルト値 None が指定されたとする）。 |

## 10.2 リストにリストを追加する

メソッド extend() を使って、**リストの末尾にリストを追加**するには、追加するリスト applist を引数にセットして次のように使う。

**リストにリストを追加する**

```
リスト.extend(applist)
```

```
>>> alist = ['apple', 3, True]
>>> blist = ['orange', False, 4]
>>> clist = ['melon', 5, 1]
>>> alist.extend(blist)
>>> alist
['apple', 3, True, 'orange', False, 4]
>>> clist.append(blist)
>>> clist
['melon', 5, 1, ['orange', False, 4]]
```

リストメソッド append() と extend() と区別することは重要である。上の例のように、リスト clist にリスト blist を append() すると、もとのリスト clist の末尾に '1 つのリスト要素' としてリスト blist が追加されてしまうことに注意する。

**演習 10.3** キーボードから文字列 str1, str2 を入力する。2 つの入力文字列に対応した文字リスト charlist1 およびに charlist1 から、charlist1 の末尾にリスト charlist1 を追加して、次のような結果を出力するプログラム extend2charlists.py を書いて実行しなさい。

```
$ python extend2charlists.py
Input a string = apple
Input another string = orange
A merged list of the two input strings =  ['a', 'p', 'p', 'l', 'e', 'o', 'r', 'a', 'n', 'g', 'e']
```

**演習 10.4（重要）** 演習 9.8（英単語を構成しないような各種記号が混じった文字列から英単語リストを得る）、及び節 7.4.2 で取り上げた cat.py（コマンドラインで指定したテキストファイルの内容を書き出す）を組み合わせて、次のようなプログラム wordlist_from_file.py を書いて、実行しなさい。

まず、空白リスト wordlist を用意しておく。指定したテキストファイルを開いて、1 行ずつ読み込み込んだ文字列 line において、英単語を構成しない文字列を空白文字に置き換えた後に切り出される英単語リスト wlist を毎回リスト wordlist に合併（リストメソッド extend() を使う）することを繰り返して（読み終

わったファイルは閉じる)、多数の英単語が並んだ 1 つのリスト wordlist を求める。このリスト内容を表示し、さらにリストの長さ len(wordlist) も表示する。

## 10.3 リストを並べ替える

### 10.3.1 リストメソッド sort() を使う

リスト要素を並べ替えに成功するためには、次のように各要素互いにすべての組の間で比較可能であることが前提である。

```
>>> a = [6, 2, 4, 7, 1, 4]
>>> a.sort()
>>> a
[1, 2, 4, 4, 6, 7]
>>> a.sort(reverse=True)
>>> a
[7, 6, 4, 4, 2, 1]
>>> b = ['apply', 'apple', 'zoo', 'yatch']
>>> b.sort()
>>> b
['apple', 'apply', 'yatch', 'zoo']
>>> b.sort(reverse=True)
>>> b
['zoo', 'yatch', 'apply', 'apple']
```

**演習** 10.5　上の結果を確かめなさい。

リスト要素に比較不可能な組があるとき、並べ替えを試みると型エラー'<' not supported between instances が発生する。

```
>>> c = [3, 'apple', 4]
>>> c.sort()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between instances of 'str' and 'int'
```

**演習** 10.6　上の結果を確かめなさい。

また、集合型のように要素間に順番を持たないデータやジェネレータに対しては、そもそも並べ替え対象でない旨の属性エラー'set' object has no attribute 'sort' が発生する。

```
>>> d = {2, 4, 5, 1}
>>> d.sort
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'set' object has no attribute 'sort'
>>> range(10).sort()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'range' object has no attribute 'sort'
>>> range10 = list(range(10))
>>> range10.sort(reverse=True)
```

```
>>> range10
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

### 10.3.2 関数 sorted() を使う

関数 sorted() は、イテラブルを並べ替えたリストを作成する組み込みの関数で、リストをそのメソッドとして並べ替える sort() よりも汎用的利用できます。ただし、.sort() とは違って、指定したリスト要素を並べ替えたリスト結果のコピーを返し、元のリストの内容は変更しません。

```
>>> b = ['apply', 'apple', 'zoo', 'yatch']
>>> sb = sorted(b)
>>> sb
['apple', 'apply', 'yatch', 'zoo']
>>> b
['apply', 'apple', 'zoo', 'yatch']
>>> rsb = sorted(b, reverse=True)
>>> rsb
['zoo', 'yatch', 'apply', 'apple']
>>> b
['apply', 'apple', 'zoo', 'yatch']
```

キーとその値を 1 つのエントリとするディクショナリは集合型であり、上の例のようにエントリ間には順序がなく、並べ替えることはできません。それでも、ディクショナリのキーあるいは値について並べ替えたエントリからなるリストが欲しい場合には、次のように関数 sorted() を使ってエントリを並べ替えることができます（節 13 参照)。

```
>>> fruits = {'apple': 4, 'orange':10, 'banana':3, 'melon':5}
>>> sorted(fruits, key=lambda dic: dic[0])
['apple', 'banana', 'melon', 'orange']
>>> sorted(fruits, key=lambda dic: dic[1])
['banana', 'melon', 'apple', 'orange']
```

ただし、この結果はディクショナリのキーのリストを返すだけで、ディクショナリのエントリが並べ替えられリストされているわけでありません。このことについては、節 13 で改めて取り上げます。

**演習 10.7**　乱数モジュールをインポートして、0.0 から 1.0 までの一様乱数を 10 個発生させて、リスト numberlist とする。次のような結果を表示するプログラム sort_numbers.py を書いて実行しなさい。

```
$ python sort_numbers.py
==> Generated numbers:
[0.6978751357884905, 0.6219696872972246, 0.07224451078637728, 0.8190343135964878,
0.8440693125564255, 0.4164415070669988, 0.5107176865662775, 0.0773422652157435,
0.5108757442771982, 0.2334510160318105]

==> Sorted numbers in reverse order: using the function sorted():
[0.8440693125564255, 0.8190343135964878, 0.6978751357884905, 0.6219696872972246,
0.5108757442771982, 0.5107176865662775, 0.4164415070669988, 0.2334510160318105,
0.0773422652157435, 0.07224451078637728]

==> Sorted numbers in reverse order: using the method sort():
[0.8440693125564255, 0.8190343135964878, 0.6978751357884905, 0.6219696872972246,
```

```
0.5108757442771982, 0.5107176865662775, 0.4164415070669988, 0.2334510160318105,
0.0773422652157435, 0.07224451078637728]
```

# 11 集合型

## 11.1 集合オブジェクトの定義

Python では要素の集まりを**コレクション** (collection) と呼ぶ。リスト、タプルや文字列はコレクションである。この他にも、'重複する要素を持たず'、'順序づけられていない' コレクションを**集合型** (set) として取り扱うことができる。

集合型は、関数 `set()` または `frozenset()` 関数を使って、要素が並んでいる文字列やリスト・タプルから生成できる（重複した要素は検出されて集合から除かれる）[*10]。あるいは、要素をカンマ (,) で区切って並べて記号 '{' と'}] で囲んでもよい。

```
>>> charset = set('application')
>>> len(charset)
8
>>> print(charset)
{'o', 'l', 'p', 'i', 'n', 't', 'a', 'c'}
>>> lst = ['apple', True, 3, 'Apple', 'apple', 3, 'Apple']
>>> len(lst)
7
>>> s = set(lst)
>>> len(s)
4
>>> print(s)
{True, 'apple', 3, 'Apple'}
>>> t = set(['apple',True, 3.14, ['apple',2]])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

上の例では、文字列に関数 `set()` を適用すると、重複しない文字からなる文字集合が返る。リストについても、重複のないリスト要素からなる集合を返す。集合型の要素数（**濃度**という）は関数 `len()` で知ることができる。ただし、集合型の要素は**イミュータブル** (immutable、変更不能) でなければならず、ミュータブル（変更可能）なリストを集合型の要素とすることはできず、最後の例が示すように、ミュータブルなリスト要素（たとえばリスト）の場合には `set()` による集合の生成に失敗する。

**演習** 11.1　上の例を実際に確かめなさい。

要素を持たない**空集合** (empty set) は、'{ }' でなく、`set()` によって生成する。次でわかるように、'{ }' によって生成されるオブジェクトは集合でなくディクショナリ型（節 12）である。

```
>>> a = set()
>>> type(a)
<class 'set'>
>>> b ={}
>>> type(b)
<class 'dict'>
```

[*10] `set()` はミュータブル（変更可能）な集合を、`forzenset()` はイミュータブル（変更不能）な集合を生成する

## 11.2 集合をイテレータとして使う

集合型は一意な順序の保証がないオブジェクトの集まり（コレクション）であり、したがって、コレクション内のインデックス（位置番号）によるアクセスやスライス（連続するインデックスによる一括取りだし）などのシークエンス的な振舞いは有しない。

それでも、集合型は for 構文のイテレータとして使うことができる。ただし、取り出される要素の順番は保証さておらず集合を定義した順序とは異なため、要素順を期待した繰り返し処理はできない。

```
>>> for elm in s:
...     print(elm)
...
Apple
orange
3.14
apple
```

**演習** 11.2　集合型の特徴である以下の 2 点を確認できる例を作って、その様子を示して見せなさい。

- 集合要素の並び順は保持されない。要素間の順番が大切となる場合には、集合データとして取り扱うべきではない。
- 要素を重複させて並べても、集合型を構成する各要素はコレクションにおいて唯一つだけである。要素が重複している集合をイテレータとした or 構文を使って、取り出された要素を書き出して確かめることもできる。

**演習** 11.3　演習 9.8 では（演習 9.5 の英単語を構成しないような各種記号を空白文字␣で置き換えた文字列を返すように改良された関数 `repalce_string_with_spaces(str)` を使って)、文字列から英単語のリストを得るプログラム `string_split.py` を書いた。

これを次のように修正してプログラム `string2set.py` を考える。同様に与えた文字列から英単語からなるリスト `wlist` を求めて、さらに `set(wlist)` によって集合データ `wlist2set` も求める。リスト要素数 `len(wlist)` とリスト内容、および、集合の要素数（濃度）とその内容を書き出す。

[実行結果]

```
$ python string2set.py
list length =  57
['alice', 'was', 'beginning', 'to', 'get', 'very', 'tired', 'of', 'sitting', 'by', 'her',
 'sister', 'on', 'the', 'bank', 'and', 'of', 'having', 'nothing', 'to', 'do', 'once',
 'or', 'twice', 'she', 'had', 'peeped', 'into', 'the', 'book', 'her', 'sister', 'was',
 'reading', 'but', 'it', 'had', 'no', 'pictures', 'or', 'conversations', 'in', 'it',
 'and', 'what', 'is', 'the', 'use', 'of', 'a', 'book', 'thought', 'alice', 'without',
 'pictures', 'or', 'conversations']
number of set =  40
{'her', 'but', 'conversations', 'she', 'without', 'nothing', 'get', 'is', 'twice', 'was',
 'alice', 'or', 'of', 'to', 'use', 'book', 'and', 'pictures', 'sitting', 'what', 'by',
 'in', 'sister', 'beginning', 'had', 'on', 'into', 'very', 'tired', 'a', 'thought',
 'do', 'no', 'having', 'bank', 'once', 'peeped', 'the', 'it', 'reading'}
```

この実行結果のように、リストから集合型に型変換すると、リスト内の重複単語は削除され、登場要素はすべて異なった文字列が並んでしまうことに注意せよ（しかも、その並び順はもとのリストとの並び準と同じになるとは限らない)。

## 11.3 要素の集合帰属判定

要素 element が集合 s の要素であるかどうかの帰属判定演算は **in**（element $\in$ s のとき True を返す）または **not in**（element $\notin$ s）とき True を返す）によって行うことができる。

```
element in s
element not in s
```

```
>>> s = {4, 2, 'apple', 'a', 2}
>>> 'aha' in s
False
>>> 'Apple' in s
False
>>> 'Apple' not in s
True
```

## 11.4 集合同士の演算

2 つの集合 s と t の間の集合演算は、それぞれの集合が小さい（要素数が少ない）ときには自明なように思われるが、大きな集合同士ではその結果を求めること自体たいへん手間がかかる。Python では集合演算結果を（巨大集合の場合には時間はかかるとしても）簡単に実行してくれる。

表 4 にあるように、2 つの集合 s と t の間の集合演算には演算子を使うよりも集合メソッドを使う方が分かりやすい。メソッド s.union(t) は、集合 s と t のどちらかに属する要素からなる集合（**和集合**）

$$s \cup t = \{e \,|\, e \in s \text{ または } e \in t\}$$

を返す。メソッド s.intersection(t) は、集合 s と t の両方に属する要素からなる集合（**共通集合**）

$$s \cap t = \{e \,|\, e \in s \text{ かつ } e \in t\}$$

を返す。メソッド s.difference(t) は、集合 s の要素から t の要素を取り除いた集合（**差集合**）

$$s \backslash t = s - t = \{e \,|\, e \in s \text{ かつ } e \notin t\}$$

を返す。メソッド s.symmetric_difference(t) は、集合 s と b の和集合 $s \cup t$ の要素から共通集合 $s \cap t$t を取り除いて、両方の集合に同時に含まれない要素からなる集合（**対称差集合**）

$$\begin{aligned} s \triangle t &= (s \cup b) \backslash (s \cap b) \\ &= (s \backslash b) \cup (b \backslash s) \end{aligned}$$

を返す。

次の様子を確認してみよう。集合演算メソットの結果は、元の集合内容には影響は与えないことに注意する（リスト操作メソッドは直ちにリスト内容に影響を与えた）。

```
>>> s = {3, 2, 5, 9}
>>> t = {2, 5, 1, 4}
>>> s.union(t)
{1, 2, 3, 4, 5, 9}
>>> s.intersection(t)
{2, 5}
>>> s.intersection(t)
```

```
{2, 5}
>>> s.difference(t)
{9, 3}
>>> s.symmetric_difference(t)
{1, 3, 4, 9}
>>> s
{9, 2, 3, 5}
>>> t
{1, 2, 4, 5}
```

**演習** 11.4　次の乱数を使うプログラム set_operation.py を実行して、その結果を詳しく確認してみなさい。

set_operation.py

```
# -*- coding: utf-8 -*-
import random
def random_number_set(n):
    '''n 個の整数乱数を発生させて集合を返す'''
    a = 0
    b = 10
    numbers= set()
    for k in range(n):
        numbers.add(random.randint(a, b))
    return(numbers)

anumbers =  random_number_set(15)
bnumbers =  random_number_set(15)

print('Size of set A = {}: \n\tA = {}'.format(len(anumbers), anumbers))
print('Size of set B = {}: \n\tB = {}'.format(len(bnumbers), bnumbers))
print('-----')
print('Union A and B\n\t= {}'.format(anumbers.union(bnumbers)))
print('Intersection A and B\n\t= {}'.format(anumbers.intersection(bnumbers)))
print('Difference A - B\n\t= {}'.format(anumbers.difference(bnumbers)))
print('Symmetric Difference A - B\n\t= {}'.format(anumbers.symmetric_difference(bnumbers)))
```

[実行結果]

```
$ python set_operation.py
Size of set A = 8:
    A = {0, 1, 3, 6, 7, 8, 9, 10}
Size of set B = 7:
    B = {1, 4, 5, 6, 7, 8, 9}
-----
Union A and B
    = {0, 1, 3, 4, 5, 6, 7, 8, 9, 10}
Intersection A and B
    = {8, 1, 9, 6, 7}
Difference A - B
    = {0, 10, 3}
Symmetric Difference A - B
```

```
= {0, 3, 4, 5, 10}
```

## 11.5 集合のメソッド

表 4 に集合に関わる関数や操作メソットの一部および演算子による演算をまとめた。

表 4: 集合型のメソッド：集合関数や集合帰属判定、演算子利用も追加

| method 名 | 使用例 | 意味 |
|---|---|---|
| 集合の生成 | | |
| | `set(collection)` | コレクション collect から集合を生成 |
| 集合の濃度 | | |
| | `len(s)` | 集合 s の濃度（要素数）を返す |
| 要素の集合帰属 | | |
| `in` | `ele in s` | 要素 ele が集合 s のメンバであるかを判定し、論理値 True または False を返す |
| `not in` | `ele not in s` | 要素 ele が集合 s のメンバで「ない」かを判定し、論理値 True または False を返す |
| 要素の追加・削除 | | |
| `.add()` | `s.add(elem)` | 集合 s に要素 elem を追加 |
| `.discard()`<br>`.remove()` | `s.discard(elem)`<br>`s.remove(elem)` | 集合 s から要素 elem を削除 |
| 集合間の関係 | | |
| `.isdisjoint()` | `s.isdisjoint(t)` | 集合 s が集合 t と共通の要素を持たない $s \cap t = \phi$ とき True を返す |
| `.issubset()`<br>演算子 <= | `s.issubset(t)`<br>`s <= t` | 集合 s の全ての要素が集合 t に含まれる $s \subseteqq t$ とき True を返す |
| 演算子 < | `s < t` | 集合 s が集合 t の真部分集合（s <= t and s != t）である $s \subsetneqq t$ とき True を返す |
| `.issuperset()`<br>演算子 >= | `s.issuperset(t)`<br>`s >= t` | 集合 s に集合 t の全ての要素が含まれる $s \supseteqq t$ ときに True を返す |
| 演算子 > | `s > t` | 集合 s が集合 t の真上位集合 $s \supsetneqq t$ であるとき True を返す |
| 集合演算 | | |
| `.union()`<br>演算子 \| | `s.union(t)`<br>`s \| t` | 集合 s と集合 t に含まれるすべての要素を持つ集合 $s \cup t$ を返す。結果は、もとの集合には影響を及ぼさない。 |
| `.intersection()`<br>演算子 & | `s.intersection(other)`<br>`s & t` | 集合 s と集合 t に共通に含まれる要素を持つ集合 $s \cap t$ を返す。結果は、もとの集合には影響を及ぼさない。 |

表 4: 集合型のメソッド：集合関数や集合帰属判定、演算子利用も追加

| | | |
|---|---|---|
| `.difference()`<br><br>演算子 - | `s.difference(t)`<br><br>`s - t` | 集合 s には含まれるが集合 t には含まれない要素を持つ集合 s\t を返す。結果は、もとの集合には影響を及ぼさない。 |
| `.symmetric_difference()`<br><br>演算子 ∧ | `s.symmetric_difference(t)`<br><br>`s ∧ t` | 集合 s と集合 t のうち、両者には含まれない要素を持つ集合 $s \triangle t$ を返す。結果は、もとの集合には影響を及ぼさない。 |

**演習** 11.5　演習 10.4 のプログラム `wordlist_from_file.py` を次のように書き直して `wordset_from_file.py` とする。

空の集合 `wordset` を用意する。コマンドラインで指定した英語テキストファイルを 1 行ずつ読み込んで、英単語からなる集合 `wset` を求めて、`wordset = wordset.union(wset)` と繰り返し和集合を取る。こうして得られた英文テキストの英単語集合 `wordset` を出力し、合わせて集合のサイズを表示する。
[実行結果] https://www.gutenberg.org/ebooks/730 にある Plain Text(UTF-8, 914kB) を oliver.txt として利用。

```
$ python wordset_from_file.py oliver.txt
{.....
.....
....., 'wildly', 'susceptible', 'date', 'alone!', 'wigs'}
Size of the word set =  10591
```

集合型では重複要素を持たないために、登場単語数は（複数形など語尾変化があるとしても）随分と少ない（分厚い「Oliver Twist」を読むには 1 万語の英語を知っているだけでよい）。

# 12　ディクショナリ型

Python には、集合型に属する特別な構造を持つ**ディクショナリ** (辞書、dictionary) というオブジェクト構造がある。他のプログラム言語で連想配列 (associative array) あるいは連想記憶 (associated memory) といわれているものである。

## 12.1　ディクショナリの生成、値の参照

ディクショナリは、記号'{' と'}' とで、**キー** (key) とその**値** (value) がコロン (:) を使って `key:value` の形で組になったものをカンマ (,) で区切って並べた集合型データである。

$$\{\text{キー}_1 : \text{値}_1,\ \text{キー}_2 : \text{値}_2,\ \ldots,\ \text{キー}_n : \text{値}_n,\}$$

あるいは、**コンストラクタ** (constructor) `dict()` を使って、キーと値のペアのタプル（**キー:値**）からなるリストからディクショナリを生成できる。

$$\texttt{dict([}(\text{キー}_1 : \text{値}_1),\ (\text{キー}_2 : \text{値}_2),\ \ldots,\ (\text{キー}_n : \text{値}_n)\texttt{])}$$

**空ディクショナリ**は {␣} というように要素を持たない集合形で生成する。

```
>>> type({})
<class 'dict'>
```

ディクショナリのキーとなり得るのは変更不能な型（文字列、数値やタプル）だけであり、リストはディクショナリのキーにはできない。

では、次にディクショナリ操作の様子をみてみよう。

```
>>> phone = {'Thom': 3456, 'Susan': 6789, 'Chloe': 1234}
>>> phone
{'Thom': 3456, 'Chloe': 1234, 'Susan': 6789}
>>> print(phone['Chloe'])
1234
>>> phone['Thom']=9876
>>> phone
{'Thom': 9876, 'Susan': 6789, 'Chloe': 1234}
>>> phone['Cecil'] = 4567
>>> phone
{'Thom': 9876, 'Cecil': 4567, 'Chloe': 1234, 'Susan': 6789}
```

文字列をキーに整数を値としてコロン (:) を挟んだものを組みとして、3 つの組からなるディクショナリ phone を生成している。phone['Chloe'] によって、ディクショナリにあるキー'Chloe' に関連付けられた**値を参照**している。

次いで、phone['Thom']=9876 によって、ディクショナリ内にあるキー'Thom' に関連付けられた**値を更新**している。さらに、ディクショナリ内には存在しないキー'Cecil' に対して、phone['Cecil'] = 4567 によって'Cecil' と関連付けられた値の組みをディクショナリに**登録**している。

**演習** 12.1　上の例を実際に確かめなさい。

## 12.2　ディクショナリの更新と追加

先の例で見たように、ディクショナリはキーを指定することで**インデックス化可能**で、次の 2 つの用途に使う。

**ディクショナリの更新/追加**

**辞書 [キー] = 値**

- 辞書に指定した 'キー' を持つエントリがあるとき、関連付けられた値を**更新**する。
- 辞書に指定した 'キー' を持つエントリが「なければ」、キーと値の組みを辞書に**追加**する。

キーが辞書のエントリにあるかどうかを論理値（True または False）で調べるには次の 2 通がある。**in** はキーが辞書のエントリに「ある」とき True を、**not in** はキーが辞書のエントリに「ない」とき True を返す。

**キー in 辞書**
**キー not in 辞書**

```
>>> 'Susan' in phone
True
>>> 'Bill' not in phone
True
```

辞書から 'キー' とそれに関連付けられた値の組みを**辞書から削除** (delete) するには次のようにする。

**del 辞書名 [キー]**

```
>>> del phone['Susan']
>>> 'Susan' in phone
```

```
False
>>> phone['Bob'] = 5637
>>> phone
{'Bob': 5637, 'Cecil': 4567, 'Thom': 9876, 'Chloe': 1234}
```

**演習** 12.2　上の Python Shell 挙動を実際に確かめてみよ。

**演習** 12.3　果物辞書 fruitdictionary を用意しておく。キーボードから値として果物を入力すると、関連付けられた値内容を表示する fruit_dictionary.py を考える。 辞書 fruitsdict の組みは、

```
{..., 'lemon': ('yellow', 'sour', ' spindle'), ....}
```

のように、キーはフルーツ文字列、キーは 3 つの要素 color(色）, taste（味）, shape（形） からなるタプル（括弧'(' と ')' でか囲まれている）とする。

color として、文字列 'green', 'yellow', 'red'... のように、taste として、文字列 'sweet', 'bitter', 'sour', 'hot'... のように、shape として文字列 'triangle', 'round', 'spindle', square'... のように考えて、以下のようにプログラム内に 10 個の組からなる辞書 fruitdictionary を用意しておく。

プログラム fruit_dictionary.py では、辞書 dic とキー key を与えると関連付けられた値を書き出す関数 show_values(dic, key) を定義してある。次を実行し、その実行結果を報告せよ。

fruit_dictionary.py

```
# -*- coding: utf-8 -*-

def show_values(dic, key):
    values = dic[key]
    print('color = ', values[0])
    print('taste = ', values[1])
    print('shape = ', values[2])

fruitsdict = {
    'apple': ('red', 'sweet', 'sphere'),
    'lemon': ('yellow', 'sour', 'spindle'),
     ......
}

fruit = input('Input a name of fruits = ')
if fruit in fruitsdict:
    show_values(fruitsdict, fruit)
else:
    print('Your input has Not been registered in the dictionary')
```

たとえば次のような実行結果になる。

```
$ python fruit_dictionary.py
Input a name of fruits = melon
color =  green
taste =  sweet
shape =  sphere
$ python fruit_dictionary.py
Input a name of fruits = strawberry
Your input has Not been registered in the dictionary
```

ディクショナリに対する基本メソッドを表 5 に示した。

表 5　ディクショナリに対する代表的メソッド

| メソッド | 操作例 | 意味 |
|---|---|---|
| items() | dict.items() | ディクショナリ dict の要素であるキー key とその値 value をタプル (key, value) とするイテレータ（dict_items 型）を返す。dict 内容は変化しない。 |
| keys() | dict.keys() | ディクショナリ dict の要素の全てのキーを要素とするイテレータ（dict_values 型）を返す。dict 内容は変化しない。 |
| values() | dict.values() | ディクショナリ dict の要素の全ての値を要素とするイテレータ（dict_values 型）を返す。dict 内容は変化しない。 |

## 12.3　単語リストから単語の出現頻度ディクショナリを作成する

節 12.2 のディクショナリの更新/追加を使うと単語リスト wlist が与えられていると、リスト内の各単語をキーとそ、その出現頻度を値とする辞書を作成することができる。

次のプログラム word_frequency.py は、与えられた単語リスト wlist 内の全ての単語をキーとし、その頻度を関連付けられた値として持つディクショナリ word_frequency_dict を作成する。ディクショナリ word_frequency_dict を空集合として準備しておき、与えた単語リスト wlist の先頭から順番に単語（文字列）word を取り出して、word がディクショナリ word_frequency_dict にキーとして含まれているかどうかを判定し、含まれていれば関連付けられた値　word_frequency_dict[word] を演算子 = によって +1 だけ更新、含まれていなければ関連付けられた値を 1 として改めてディクショナリに登録する。下のプログラムでは単語リストの wlist への代入において、1 行とみなされる Python の**改行記法**—適当な箇所で記号 '\' (Windows では円記号'¥') を入力して改行しても 1 行とみなされる—を使っている。

word_frequency.py

```
# -*- coding: utf-8 -*-
'''
単語リスト \verb+wlist+ 内の全ての単語をキーとし、その頻度を
関連付けられた値として持つディクショナリ word_frequency_dict を作成
'''

wlist = ['banana', 'apple', 'orange', 'melon', \
   'Apple', 'apple', 'apple', 'orange', 'melon']
word_frequency_dict = {}
for word in wlist:
    if word in word_frequency_dict:
        word_frequency_dict[word] += 1
    else:
        word_frequency_dict[word] = 1

print(wlist)
print('----')
print(word_frequency_dict)
```

[実行結果]

```
$ python word_frequency.py
['banana', 'apple', 'orange', 'melon', 'Apple', 'apple', 'apple', 'orange', 'melon']
```

```
----
{'banana': 1, 'apple': 3, 'orange': 2, 'melon': 2, 'Apple': 1}
```

**演習** 12.4　上のプログラム word_frequency.py において、単語リスト wlist を追加し、実行してみなさい。

プログラム word_frequency.py 内の手続きを関数化し、単語リスト word_list を与えると、単語頻度辞書を返す関数 make_word_frequency_dictionary(word_list) を次のよう定義しよう。

**単語リスト word_list から単語頻度辞書を作成**

```
def make_word_frequency_dictionary(word_list):
    word_frequency_dict = {}
    for word in word_list:
        if word in word_frequency_dict:
            word_frequency_dict[word] += 1
        else:
            word_frequency_dict[word] = 1
    return(word_frequency_dict)
```

**演習** 12.5　演習 12.4 を関数 make_word_frequency_dictionary を使って、プログラム word_frequency.py を書き直し、実行してみなさい。

## 12.4　テキストファイルの単語の出現頻度ディクショナリを作成する

こうして、コマンドラインで指定したテキストファイルに登場する単語の出現頻度ディクショナリ word_frequency_dict を求めるプログラム word_frequency_from_file.py の基本形は以下のようになる。演習 10.4 で考えたように、指定したファイルを 1 行ずつ読み込んで単語リストを次々と extend メソッドで追加して、テキストに登場した全ての単語リストを求めるプログラム wordlist_from_file.py を若干修正して得られていることに注意しよう。

関数 repalce_string_with_spaces(str) は、演習 9.5 や演習 9.6 で考えたように、単語に属すとは思われないような記号群を文字列 str 内を検索して空白文字 ('␣') に置き換えた文字列を返す関数である。

**word_frequency_from_file.py の基本形**

```
# -*- coding: utf-8 -*-
import sys

def repalce_string_with_spaces(str):
    ...
    省略

def make_word_frequency_dictionary(word_list):
    ...
    省略

#------  main script  -------
fh = open(sys.argv[1], 'r')
word_list = []
line = fh.readline()
while line:
    wlist = repalce_string_with_spaces(line).split()
    word_list.extend(wlist)
    line = fh.readline()
```

```
fh.close()

word_frequency_dict = make_word_frequency_dictionary(word_list)
print(word_frequency_dict)
```

**演習 12.6（重要）** このプログラム word_frequency_from_file.py を完成しなさい。コマンドラインで指定するテキストファイルとして、1863 年 11 月 19 日に行った Lincoln の The Gettysburg Address http://www.abrahamlincolnonline.org/lincoln/speeches/gettysburg.htm をテキストファイル gettysburg_address.txt に保存したもの（Four score からはじまり、from the earth. で終わる）を使い、演説の単語の出現頻度ディクショナリを出力しなさい。このとき、重複して数えた全単語数 total_num_words の値も出力しなさい。
[実行結果] リンカーンの Gettysburg 演説の場合：ディクショナリの並び順所は毎回同じである保証はないことに注意。

```
$ python word_frequency_from_file.py gettysburg_address.txt
{'and': 6, 'great': 3, 'fathers': 1, 'final': 1, 'now': 1, 'ground': 1,
...
...
 'but': 2, 'advanced': 1,'take': 1, 'so': 3, 'will': 1, 'those': 1, 'who': 3}
Total number of words =  272
```

# 13 複合構造を持つオブジェクトを並べ替える

ここでは一般のイテラブル（イテレータとして利用できるオブジェクト）の要素を並べ替えてリストとして取り出すことを考えてみます。

既に節 10.3（36 ページ）では、簡単な場合の並べ替えについて、リストメソッド sort()（リスト内容を直接変化させる）または汎用のイテラブルを並べ替える組み込み関数 sorted()（そのコピーを返し、元の内容は変化しない）を使う方法を説明しました。ディクショナリでは各エントリ間に順序がなく、そのままでは並べ替えることができませんでした。

節 10.3.2 では、キーとそれに関連付けられた値をエントリとするディクショナリを、組み込み関数 sorted() を使ってそのキーまたは値によって並べ替えてリストとして返す様子を紹介しました。ただし、その場合、以下のようにディクショナリのキーのリストを返すだけで、元のエントリが並べ替えられリストされているわけでありません。

```
>>> fruits = {'apple': 4, 'orange':10, 'banana':3, 'melon':5}
>>> sorted(fruits, key=lambda dic: dic[0])
['apple', 'banana', 'melon', 'orange']
>>> sorted(fruits, key=lambda dic: dic[1])
['banana', 'melon', 'apple', 'orange']
```

**演習 13.1** 上の Python シェルを実行してみなさい。

## 13.1 ラムダ関数

ラムダ関数は無名関数 (anonymous function) または純関数 (pure function) と称され、計算理論ではラムダ計算（λ-calculus）として知られています。ラムダ計算は計算可能性理論の研究において A. Church や S. Kleene によって導入された概念で、1936 年に Church が一階述語論理の決定可能性問題を否定的に解いたときに使われました。

ラムダ関数は Scheme、Haskell や Mathematica などのプログラム言語での中心的記述に使われますが、

Java や JavaScript などでもその考え方を取り入れています。Python では、関数を通常は予約語 def に続けて関数名を定めて定義しますが、ラムダ関数の記述においては関数名を定めることなく「無名」で次の書式に従って記述します。

```
lambda 引数 1, 引数 2,..., 引数 n:  引数を使った式
```

Python シェルでラムダ関数を使ってみましょう。

```
>>> (lambda x: x * x)(9)
81
>>> (lambda x, y: x * y)(5, 7)
35
>>> wa = lambda x, y: x * y
>>> print(wa(3, 5))
15
```

括弧 () 内に 2 乗するラムダ関数を定義して、それに 9 を渡してみると 81 が返ります。同様に、引数を 2 つにして 5 と 7 を渡すとその積を返すラムダ関数を定義して、5 と 7 を渡すと 35 が返ります。最後の例は、そのラムダ関数を変数名 wa に代入してやると、関数 wa を定義したように使うことができることに注意してください。いずれにせよ、この程度でわざわざラムダ関数を定義する必要はないように思われます。

組み込み関数 map() は、定義された関数 $f(x)$ を使ってイテラブル、たとえばリスト $[x_1, x_2, \ldots, x_n]$ に対して map$[f, [x_1, x_2, \ldots, x_n]]$ と書くと、それぞれの要素に適用した結果を返します。

$$[f(x_1), f(x_2), \ldots, f(x_n)]$$

したがって、map() とラムダ関数を使うと、次のようにしてリスト要素の 2 乗を一気に求めることができます。

```
>>> list(map(lambda x: x * x, range(10)))
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

**演習** 13.2　上の Python シェルの結果を実際に確かめなさい。

**演習** 13.3　次のプログラム lambda_func.py を実行し、どのように動作しているかを説明しなさい。

lambda_func.py

```
# -*- coding: utf-8 -*-

students = {('Hanako', 'C', 15), ('Taro', 'B', 16), ('Jiro', 'A', 15), \
            ('Sakura', 'C', 14)}
age = lambda child: child[2]

for child in students:
    print(age(child))
```

## 13.2　並べ替えのキー

ディクショナリやリスト要素が再びリストやタプルであるときのように、各要素の「何について着目するか」を比較を行う前に決めておく必要があります。この行為を比較の際に各要素に対して呼び出す並べ替えの**キー関数** (sort key) といい、リストメソッド sort() または組み込み関数 sorted() 内でオプションとして次のように指定します。

```
key = キー関数
```

次の例は、文字列を空白で単語に切り出すためにメソッド split() で区切った結果をリスト pico に代入して、sorted() で並び替えた結果です。

```
>>> pico = 'I have a Pen I have a Pineapple Ah Pineapple pen'.split()
>>> sorted(pico)
['Ah', 'I', 'I', 'Pen', 'Pineapple', 'Pineapple', 'a', 'a', 'have', 'have', 'pen']
>>> sorted(pico, key = str.lower)
['a', 'a', 'Ah', 'have', 'have', 'I', 'I', 'Pen', 'pen', 'Pineapple', 'Pineapple']
```

前者は各文字列をそのまま辞書式順序で並べ替えた結果で、Python では大文字が小文字より小さいことを反映しています。後者は比較に先立って、key = str.lower でキー関数を指定して（この場合、メソッド lower() の括弧 () は不要になっていることに注意)、各文字列を小文字化した上で辞書式順序で並べ替るという結果を元の文字列として並べています。

### 13.2.1 タプルの集まりを並べ替える

次のようなタプル（名前, クラス, 年齢）からなる集合を考えよう。

```
>>> students = {('Hanako', 'C', 15), ('Taro', 'B', 16), ('Jiro', 'A', 15), ('Sakura', 'C', 14)}
```

この集合要素はどれも長さ 3 のタプルであり、その何番目に着目するか（0,1,2 番目の選択がある）を**ラムダ関数** (lambda function) の書式キーワード 'lambda' を使ってキー関数をその場で定義し、これによって以下のように要素を並べ替えます。

```
>>> sorted(students)
[('Hanako', 'C', 15), ('Jiro', 'A', 15), ('Sakura', 'C', 14), ('Taro', 'B', 16)]
>>> sorted(students, key = lambda child : child[1])
[('Jiro', 'A', 15), ('Taro', 'B', 16), ('Sakura', 'C', 14), ('Hanako', 'C', 15)]
>>> sorted(students, key = lambda child : child[2])
[('Sakura', 'C', 14), ('Jiro', 'A', 15), ('Hanako', 'C', 15), ('Taro', 'B', 16)]
```

パラメータ key の指定 key = lambda child : child[1] において、引数をここでは child とセットし、ラムダ関数の実体として引数の 1 番目 child[1] を返すものとしてラムダ関数を定義している。

**演習 13.4** 上のように、集合 students の要素の 0,1,2 番目について逆順に並べ替えたリスト（35 ページの表 3 参照）を返す Python シェルを実行してみなさい。
(ヒント)：sorted() において、並べ替えオプション reverse = True をラムダ関数と合わせて指定する。

### 13.2.2 ディクショナリを並べ替える

表 5 にあるディクショナリのメソッド items() を使うと、ディクショナリのエントリであるキーとそれに関連付けられた値の組がタプルとなったイテレータ（dict_values 型）を得ることができます。そうすれば、節 13.2.1 でタプルの集まりを並べ替えたように、ディクショナリも注目するキーについて並べ替えができます。

```
>>> fruits = {'apple': 4, 'orange':10, 'banana':3, 'melon':5}
>>> fruits_items = fruits.items()
>>> fruits_items
dict_items([('apple', 4), ('orange', 10), ('banana', 3), ('melon', 5)])
>>> sorted(fruits_items, key = lambda item: item[1])
[('banana', 3), ('apple', 4), ('melon', 5), ('orange', 10)]
```

**演習 13.5（重要）** 演習 12.6 で作成したプログラム word_frequency_from_file.py において、単語の出現頻度ディクショナリ word_frequency_dict をディクショナリのメソッド items() を使って要素をタプル化したうえで、sorted() を使って出現回数を逆順（大きい順）で並べ替えたリスト word_frequency_list を

プリントするように書き加えて、実行してみなさい。最後に、重複して数えた全単語数 total_num_words の値も出力しておこう。
[実行結果] リンカーンの Gettysburg 演説の場合：元となるディクショナリの並び順所は毎回同じである保証はなく、同じ登場頻度をもつエントリの並び替え結果も同じとはならないことに注意。

```
$ python word_frequency_from_file.py gettysburg_address.txt
[('that', 13), ('the', 11), ('we', 10), ('to', 8), ('here', 8),
 ('a', 7), ('and', 6), ('for', 5), ('it', 5), ('of', 5), ('have', 5), ('can', 5),
.....
.....
 ('under', 1),('government', 1), ('world', 1), ('engaged', 1), ('their', 1),
 ('task', 1), ('full', 1)]
Total number of words =  272
```

さて、演習 13.5 で、並べ替えたリスト word_frequency_list の各要素であるタプル (word, freq) の entry を for 文で取り出すときに

```
for word, freq in word_frequency_list:
    print('{1},{0}'.format(word, freq))
```

と書いて、タプルでの登場順を変えて、変数 freq と word の内容をカンマで区切って 1 行づつ書き出すことができます。.format() 内に登場する変数を順番に $0, 1, 2, \ldots$ と番号付けられます。ここでは変数内容{}とカンマ, を出力していますが、{}内に変数の順番を明示して、freq、word の順で出力されるようにしています。.format() を使って print をすることを**書式指定** (format) といい、思うような書き出し方が可能になります。

**演習** 13.6　プログラム word_frequency_from_file.py を次のように order_frequency_word.py 書き直そう。
word_frequency_list の各要素に頻度の大きな順に順位を付けて（同じ頻度でも、リストの並び順に大きなものから $1, 2, 3, \ldots$ と順位をつけることにする)、順位、頻度、単語をカンマで区切ってを一行に書き出すプログラム order_frequency_word.py を書いて、実行する。
[実行結果] リンカーンの Gettysburg 演説の場合：先頭行の『1,13,that』は第 1 位が 13 回登場した単語'that'だという行になっています。ただし、元となるディクショナリの並び順所は毎回同じである保証はなく、同じ登場頻度をもつエントリの並び替え結果も同じとはならないことに注意。重複して数えた全単語数 total_num_words の値も出力しておこう。

```
$ python order_frequency_word.py gettysburg_address.txt
that,1,13
the,2,11
we,3,10
here,4,8
to,5,8
a,6,7
....
....
now,135,1
nobly,136,1
forth,137,1
civil,138,1
Total number of words =  272
```

# 14　Zipf の法則を再発見する

Zip はテキストに登場する単語の出現頻度とその出現順位に着目したときに成立する関係を生涯にわたって追求しました。Zipf が到達した法則性—Zipf の法則を、再発見してみよう[*11]。

## 14.1　登場頻度の規格化

将来においてテキスト同士の比較において、単語の出現頻度の直接比較は好ましくありません。ある単語の登場頻度は対象とするテキストの分量に応じて当然大きくなると期待されるため、テキストの比較においては頻度数の多寡には意味がないからです。

比較したい値をある基準を設けて同じ尺度で比較できるように調整する作業を**規格化** (normalization) または正規化といいます。

今の場合、テキスト $T$ 内の単語 $w_i$ に着目し、“重複を許して数えた全単語数” `total_num_words` で単語頻度を除して

$$\text{相対頻度} = \frac{\text{単語頻度}}{\text{全単語数}}$$

と、頻度を規格化しておきます。この規格化によって $0 \leqq$ 相対頻度 $\leqq 1$ となるのでテキストに出現する単語頻度をテキスト同士で比較することができます。

相対出現頻度（規格化頻度）$f_i$ を降順に並べた相対頻度順位 $k_i$ に対して、 データ集合 $\{(k_i, f_i) \,|\, i = 1, 2, \ldots, N, N$ は単語種類数 $\}$ にどんな関係が成り立つのかを調べてみよう。

次は演習 13.6 で書いたプログラム `order_frequency_word.py` を順位、単語相対出現頻度（降順）とその単語を書すように修正した最後の部分です。まず、最初に 1 行ずつに指定したテキストファイル名、そこに登場した全単語数、そして文字列’order’、’rfreq’、’word’ をカンマ区切りで書いていることに注意してください。出力結果を別ファイルに書く出す際に、どのファイルから何を書き出したかど重要情報を記録するらめです。

**修正した** `order_frequency_word.py`

```
# -*- coding: utf-8 -*-
import sys
import math
...
...
total_num_words = len(word_list)
print(sys.argv[1])
print('Total number of words = ', total_num_words)
print('{0},{1},{2}'.format('order', 'rfreq', 'word'))
order = 1
for word, freq in word_frequency_list:
    rfreq = freq / total_num_words
    print('{0},{1},{2}'.format(order, rfreq, word))
    order += 1
```

**演習 14.1**　上のように、1 行ずつに指定したテキストファイル名、そこに登場した全単語数、そして文字列’order’、’rfreq’、’word’ をカンマ区切りで書いて、順位、単語相対出現頻度（降順）とその単語をき出すように修正したプログラム `order_frequency_word.py` を実行しなさい。

[*11] Zipf の法則についての信頼できる記述は「言語情報処理」（岩波口座 言語の科学 9, 1998）や「計量情報学」景浦峡（丸善、2000）にあります。

## 14.2 標準出力のリダイレクト

コマンドラインから実行されるプログラムの出力を textbf 標準出力 (standard output) とよび、今イチでは標準出力装置はモニタに設定されています。Python で `print()` を実行すると、その結果は標準出力装置にプリントされていたわけです。

プログラムを実行しているオペレーティング・システム (O.S) は、その出力先としてテキストファイルに書き出すことができます。これを出力の**リダイレクト** (redirect) といい、コマンドラインから記号 '>' を使って次のように実行します。ただす、この書き方ではエラー出力はリダイレクトされません。

**出力のリダイレクション**

**出力があるコマンド > ファイル名**

この場合、既に存在しているファイルを指定するとファイル内容は上書き（消去）されてしまうので注意してください。記号'>>' を使うと、指定したファイルがなければ新しく書き出されますが、存在しているときには、書き出される内容が既存ファイルの末尾に追記されます。

**演習 14.2（重要）** 演習 14.1 のプログラム `order_frequency_word.py` を使って指定したテキストファイル `alice_adventures.txt` [*12] を読み込んで、順位、単語相対出現頻度（降順）、単語をカンマ区切りで標準出力にプリントした結果をリダイレクトして、テキストファイル `alice_adventures.csv`（拡張子`.cvs` が大事）に次のようにして書き出しなさい。

```
$ python order_frequency_word.py alice_adventures.txt > alice_adventures.csv
```

### 14.2.1 csv ファイル

テキストファイルにおいて、その 1 行ずつが**区切り記号** (delimiter) を使ってデータが並んでいるようなファイルがしばしば利用され、慣用的に拡張子 **.csv** 付けます。その区切り記号をカンマ (',') 区切ったファイルを **CSV** ファイル (comma separated variables file) と呼びます[*13]。

テキストの各 1 行が定められた区切り記号でデータが並んでいると一括処理がしやすく、特に拡張子 `.csv` のついたファイルは表計算ソフトに関連付けられており（インストールされていればファイルアイコンをクリックするだけで開く）、表計算ソフトを使ったさまざまな処理やグラフ化が可能になります。

このファイル `alice_adventures.csv` を表計算ソフトウエア Excel で開いた様子が図 1 です。

図**??**は、表計算ソフトウエア Excel を使って Alice's Adventures in Wonderland に登場した単語の相対頻度（縦軸）とその登場順位（横軸）をプロットした散布図です。

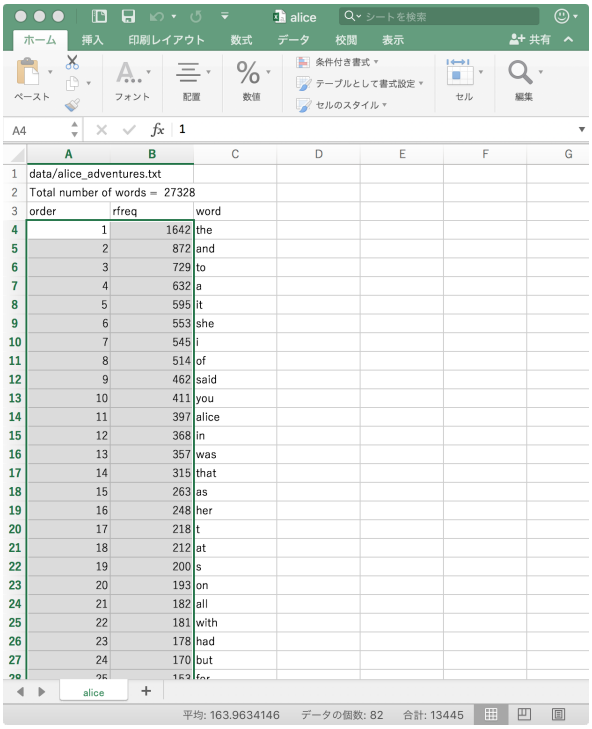
**演習 14.3（重要）** 演習 14.2 でリダイレクトしてファイルに出力した CSV ファイルから、表計算ソフトを使って図 2 のような登場する英単語の順位（横軸）とその相対出現頻度（縦軸）との関係をプロットした図を作成しなさい。ただし、グラフのタイトルや軸数値や軸ラベルの説明は必須です。「クラフの要素の追加」において、『軸』『軸ラベル』などを追加して整えなさい（どれか一つでも欠けると、グラフにはなんの意味も価値もなくなります)。

得られた図は表データシートとは別のシートとして全体をファイル保存しなさい。ただし、もはや CSV 形式ではなくなり、Excel の場合には、拡張子`.xlsx` を持つ違った書式に従う別ファイルとして保存します。

---

[*12] ここでのテキストファイル `alice_adventures.txt` は、Alice's Adventures in Wonderland (Lewis Carroll, 1865) http://www.gutenberg.org/ebooks/11 にある Plain Text(UTF-8, 170KB) をダウンロードして使います。ただし、パソコンの環境や設定によっては全角左右の 1 重引用符 (' 、') や全角左右の 2 重引用符 ( “、") など大変紛らわしい 2 バイト文字が混入し、プログラムがうまく動作しないことがあります。非本質的なことですが、適切なテキストエディタなどを使って全角文字を完全に除去し去ることが大変重要になります。

[*13] CSV ファイルとして、1 行に並んだデータの区切り記号としてカンマ (',') 以外に、しばしば空白 ('␣')、セミコロン (';')、タブ ('\t') も使うことがあり、これらも CSV ファイルと呼んでいます。



| | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| 1 | data/alice_adventures.txt | | | | | | |
| 2 | Total number of words = 27328 | | | | | | |
| 3 | order | rfreq | word | | | | |
| 4 | 1 | 1642 | the | | | | |
| 5 | 2 | 872 | and | | | | |
| 6 | 3 | 729 | to | | | | |
| 7 | 4 | 632 | a | | | | |
| 8 | 5 | 595 | it | | | | |
| 9 | 6 | 553 | she | | | | |
| 10 | 7 | 545 | i | | | | |
| 11 | 8 | 514 | of | | | | |
| 12 | 9 | 462 | said | | | | |
| 13 | 10 | 411 | you | | | | |
| 14 | 11 | 397 | alice | | | | |
| 15 | 12 | 368 | in | | | | |
| 16 | 13 | 357 | was | | | | |
| 17 | 14 | 315 | that | | | | |
| 18 | 15 | 263 | as | | | | |
| 19 | 16 | 248 | her | | | | |
| 20 | 17 | 218 | t | | | | |
| 21 | 18 | 212 | at | | | | |
| 22 | 19 | 200 | s | | | | |
| 23 | 20 | 193 | on | | | | |
| 24 | 21 | 182 | all | | | | |
| 25 | 22 | 181 | with | | | | |
| 26 | 23 | 178 | had | | | | |
| 27 | 24 | 170 | but | | | | |

図 1　テキスト `alice_adventures.csv'` を Excel で開いて第 1,2 列を選択した様子。この例ではグラフデータとして 4 行目以降を選択している。

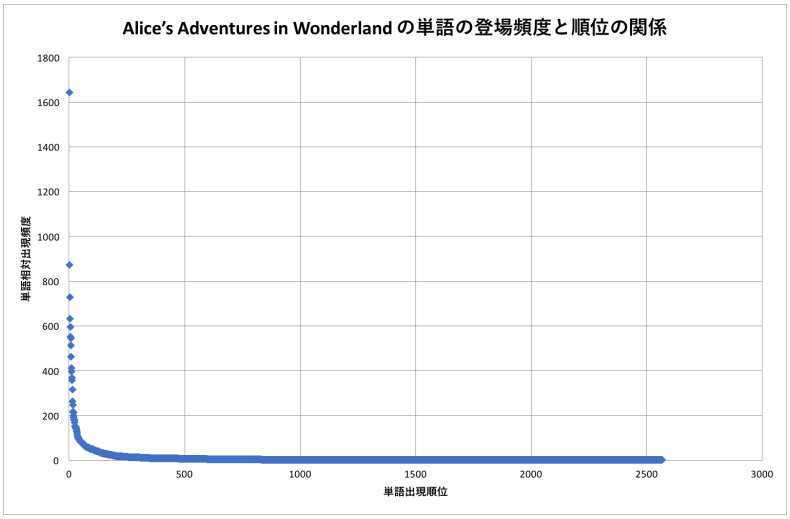



図 2　Alice Adventure in Wonderlandhttp://www.gutenberg.org/ebooks/11 に登場する英単語の順位（横軸）とその相対出現頻度（縦軸）との関係をプロット。

## 14.3　Zipf の法則

以上で、ある程度の分量（∼150KB 程度以上）の英文テキスト $T$ について、その単語の出現回数の順位（降順）$y$ と出現順位 $x$ とがある指数 $\gamma_T > 0$ を持つ**べキ法則** (powerlaw) に準じていることが予想される足がかりを得た (**Zipf の弱法則**)。

$$y \sim a_T x^{-\gamma_T}, \quad \gamma_T > 0, x > 0$$

定数 $a_T$ はテキスト $T$ で決まる定数である。これの意味することは次のようである：もし、出現回数の順位（降順）$y$ と出現順位 $x$ とはべき法則に従っているとすると、両辺の対数を取って

$$\log y \sim -\gamma_T \log x + b_T$$

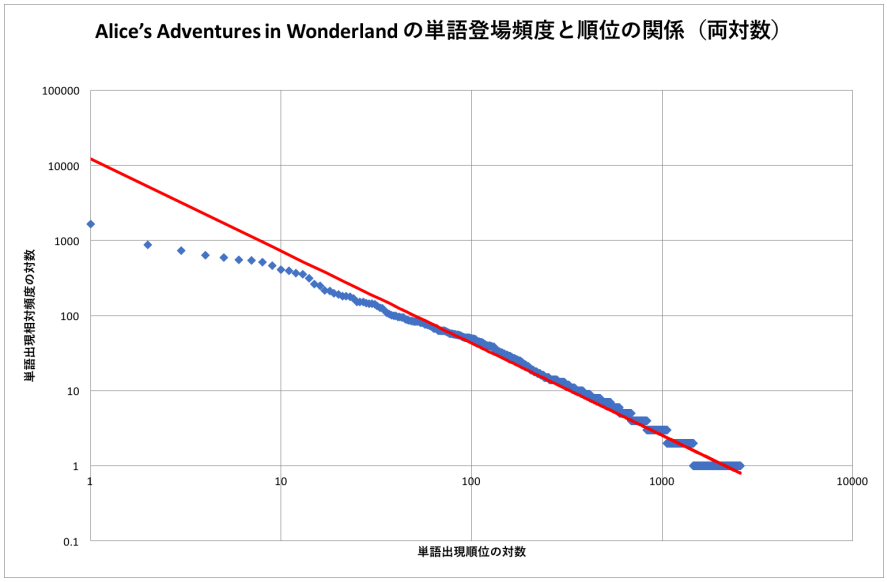



図 3　Alice Adventure in Wonderlandhttp://www.gutenberg.org/ebooks/11 に登場する英単語の順位（横軸）とその相対出現頻度（縦軸）との両対数プロット。両端の点群を除くと、直線（赤色）で示した直線上にプロットが乗っているように観測できる。

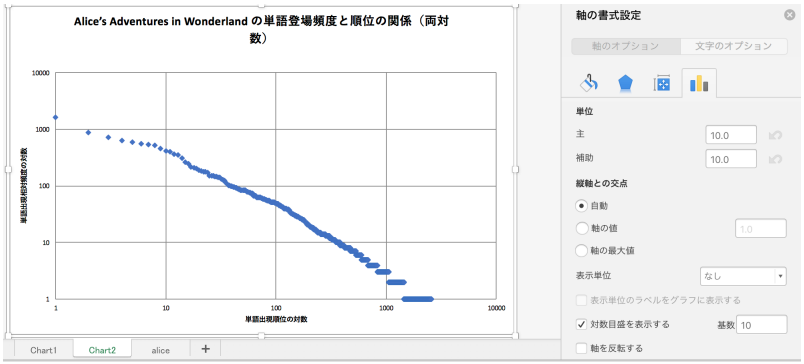



図 4　対数目盛を振るための設定。Excel では軸数値を選択して、対数目盛（基底は 10）でプロットすることができます。両対数目盛りとしるために、それぞれに軸ごとに設定する。

というように、量 $\log y$ と $\log x$ は直線の関係にあることになる。図 3 は、Alice Adventure in Wonderland に登場する英単語の順位（横軸）とその相対出現頻度（縦軸）を両対数をとってプロットしたものである。両端の点群を除くと、直線（赤色）で示した直線上にプロットが乗っているように観測できる。もしべき法則が成立しているならば、この直線の傾きがべき法則の指数 $\gamma_T$ である。

**演習 14.4（重要）**　演習 14.3 で描いた図（図 2 参照）とは別に、英単語の順位（横軸）とその相対出現頻度（縦軸）との関係を両対数を取ってプロットした図を作成しなさい。

この Zipf の弱法則という現象は他のテキストではどうなっているだりうか。

Zipf の弱法則とは、（一定分量の）複数のテキスト作品 $T_1, T_2, \ldots, T_m$ について、それぞれテキストに固有なべキ指数 $\gamma_1, \gamma_2, \ldots, \gamma_m$ が存在して、それぞれが同じようにべき法則

$$y \sim b_{T_1} x^{-\gamma_{T_1}}, \quad y \sim b_{T_2} x^{-\gamma_{T_3}}, \ldots, \qquad y \sim b_{T_m} x^{-\gamma_{T_m}},$$

に従っていることを主張します。特に、べき法則におけるべき指数（両対数をとったときには直線の傾き）$\gamma_k, k = 1, \ldots, m$ はテキストごとに固有な異なった値 $\{\gamma_k\}$ を持つのでしょうか。それとも一定の時代区分という範囲内で、英文テキスト $T_k$ の作者や内容・文体によらずに概ね同じ（普遍な）定数 $\gamma = \gamma_k, k = 1, \ldots, m$ なのでしょうか。テキストによらずにべき指数が普遍定数 $\gamma$ であるような現象があるとき、**Zipf の強法則**といいます。作家やテキスト毎に登場単語とその出現の様子は当然それぞれに異なっているため、この問いは大変興味深いのです。

このことを確かめるために、Charles Dickens の http://www.gutenberg.org/ebooks/730 にある “*Oliver Twist*” の Plain Text UFT-8(900KB) について本文だけを慎重に取り出して保存したテキストに同様な処理を行ってみた。そのプロットを重ねたものが図 5 である。

それぞれの作品で得られる単語の出現順位とその相対出現頻度直線の関係を両対数目盛りでプロットしてみると直線をなすように集まり、それぞれの作品がべき分布にしたがっていることから Zipf の弱法則が成立し、さらにその傾きが 2 作品の概ね同じ傾きを持つことが観測される。したがって、2 作品での比較であるが Zipf の強法則の成立を示唆している。

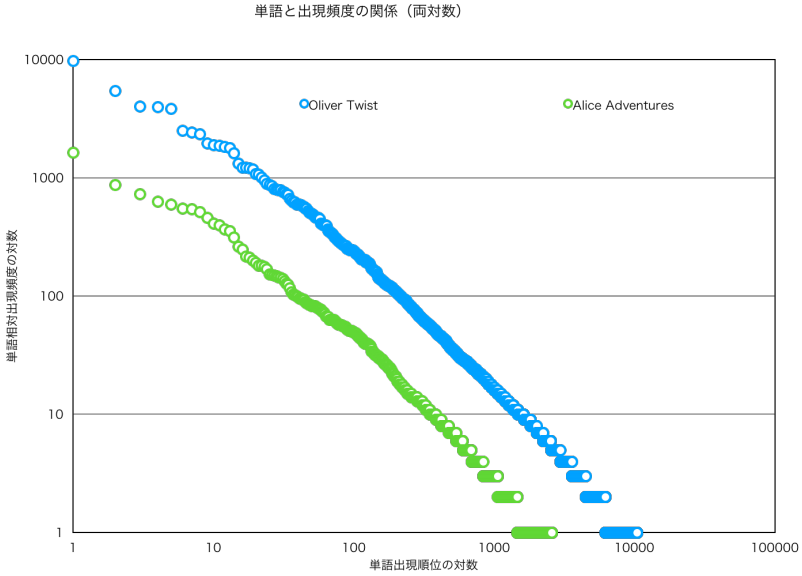



図 5　“*Alice Adventure in Wonderland*”(L.Carroll) と “*Oliver Twist*”(C. Dickens) に登場する単語の順位とその相対出現頻度の関係を両対数でプロット。青色と緑色の点群はそれぞれ直線をなすように集まり、その傾きが 2 作品の概ね同じ傾きを持つことが観測される。

**演習 14.5（重要）**　図 5 のように、2 つの作品のそれぞれについて単語の順位とその相対出現頻度の関係を両対数でプロットして得られる図を重ねて 1 つの散布図として描いてみなさい。

調査対象の英文テキストは Free ebooks by Project Gutenberg(`https://www.gutenberg.org`) から同じ時代区分に属する作品で Plain text として 100KB 以上ある（それなりの分量がある）ものを選ぶ。たたし、ダウンロードしてテキストには、前後に Project Gutenberg のクレジット情報を含んでいるので、事前にテキストエディタで取り除いておく（他に不要な文なども削除しておく）。こうした事前作業を丁寧に行うことは大変重要である。

このために次の手順さ作業するとよい。

1. テキスト 1 とテキスト 2 についてテキストファイルとして準備して、各テキストについて、たとえば、演習 14.2 でやったようにして、出現順位、相対出現頻度、単語などをリダイレクトして書き出した CSV ファイルを 2 つ `text1.csv`, `text2.csv` と作成する。
2. これらの 2 つの CSV ファイルを表計算ソフトウエアで開いて、新しいファイルに 1 列目を順位（登場する単語数が大きい CSV ファイル `text1.csv` からコピーするとよい)、2 列目に相対出現頻度として `text1.csv` の相対出現頻度を、3 列目に `text2.csv` の相対出現頻度をコピーし、次のような内容をもつ CSV ファイル `test12.csv` を作成する。

| | テキスト 1 | テキスト 2 |
|---|---|---|
| 順位 | 頻度 1 | 頻度 2 |
| 1 | 9809 | 1642 |
| 2 | 5498 | 872 |
| 3 | 4042 | 729 |
| 4 | 3996 | 632 |
| 5 | 3843 | 595 |
| 6 | 2507 | 553 |
| 7 | 2443 | 545 |
| 8 | 2357 | 514 |
| 9 | 1975 | 462 |
| 10 | 1898 | 411 |
| ⋮ | ⋮ | ⋮ |

3. `test12.csv` から、横軸を順位、縦軸に相対出現頻度を取って（対数目盛りとする）2 つのテキスト結果を重ねた散布図を描く。グラフタイトル、それぞれの軸ラベル、2 種類にプロット点を凡例で明示など、図として必要最小限の加工をする。

登場する単語との頻度は報告する必要はないが、以下のように利用したテキストの完全な書誌情報に加えて、上位 20 位までの最頻出単語とその登場回数の一覧を添えることとする。

- 正確な作品名称と作者、発表年
- テキスト内容だけのバイト数
- テキストで使われた総単語数
- 上位 20 位までの最頻出単語とその登場回数の一覧

## 14.4 言語による差異

考えてきた Python プログラムは、ローマ字とコンピュータキーボード入力可能な記号からなるテキストを対象としてものです。日本語や中国語などの他バイト文字や、ヨーロッパ言語であっても Göthe や Poincaré などのようなアクセント記号が交じるテキストに対しては、さらに工夫した処理が必要になります。

それでも、ラテン語作品は正真正銘のローマ字で書かれているために、Zipf の法則の探求対象です。Free ebooks by Project Gutenberg には、Saint Augustine の「告白:Confessiones」や Julius Caesar の「ガリア戦記:De Bello Gallico」などのラテン語著作[*14] ももちろん多数収録されています。言語の違いによって Zipf の法則の指数は違ってくるのでしょうか、すぐにでも調査してみる意義があります。石川啄木の Romaji Diary のような日本語はローマ字表記可能です。このとき、ローマ字表記した日本作品で Zipf の法則は成りたつのでしょうか。

Zipf の法則は言語における単語出現頻度の研究によって発見された経験法則です。今日では Zipf の法則に当てはまる現象がたくさん報告されています。言語がそうであるように、コンピュータネットワークや社会ネットワークといったモノとモノや人間同士のつながりなど関係が描けるものを対象として 2000 年ごろから盛んに研究されるようになりました。今日では textbf ネットワーク科学と呼ばれ、従来の研究の枠を越えて学際的な視点を数多く提供しています。

[*14] ハーバード大学の The Digital Loeb Classical Library は古代ギリシャとローマ時代のラテン文献がほぼ網羅されている。