Think Python：コンピュータサイエンティストのように考えてみよう
第二版

アレン・B・ダウニー

相川利樹（日本語訳）

（原題）Think Python: How to Think Like a Computer Scientist
2nd Edition
（著者）Allen B. Downey

Copyright©2018 相川利樹
「ThinkPython: コンピュータサイエンティストのように考えてみよう」（第二版）
by 相川利樹 is licensed under a Creative Commons 表示 3.0 非移植 License

目 次

はじめに **11**

第 1 章　プログラムが動くまで **15**
1.1　プログラムとは何か 15
1.2　Python を走らせる 16
1.3　初めてのプログラム 17
1.4　算術演算子 18
1.5　値と型 18
1.6　形式言語と自然言語 19
1.7　デバッギング 21
1.8　語句 22
1.9　練習問題 23

第 2 章　変数、表式、文 **25**
2.1　代入文 25
2.2　変数名 25
2.3　表式と文 26
2.4　スクリプト・モード 27
2.5　演算子の順位 29
2.6　文字列処理 29
2.7　コメント 30
2.8　デバッギング 31
2.9　語句 31
2.10　練習問題 33

第 3 章　関数 **35**
3.1　関数呼び出し 35
3.2　数学関数 36
3.3　混合計算 37
3.4　新規関数の追加 38

3.5 関数定義とその利用法 39
3.6 実行の流れ 40
3.7 仮引数と実引数 41
3.8 変数や仮引数はローカルである 42
3.9 スタック図 43
3.10 結果を生む関数とボイド関数 44
3.11 なぜ関数？ 45
3.12 デバッギング 46
3.13 語句 46
3.14 練習問題 48

第 4 章 事例研究：インタフェース設計 51
4.1 `turtle` モジュール 51
4.2 簡単な繰り返し 53
4.3 練習問題 54
4.4 カプセル化 55
4.5 一般化 55
4.6 インタフェース設計 56
4.7 再因子分解 57
4.8 開発計画 59
4.9 ドキュメント文字列 59
4.10 デバッギング 60
4.11 語句 61
4.12 練習問題 61

第 5 章 条件文と再帰 65
5.1 打ち切り除算とモジュラ演算子 65
5.2 ブール代数表現 66
5.3 論理演算子 67
5.4 条件処理 67
5.5 二者選択処理 68
5.6 条件文の連鎖 68
5.7 入れ子の条件処理 69
5.8 再帰 70
5.9 再帰関数のスタック図 71
5.10 無制限な再帰 72
5.11 キーボード入力 72

5.12 デバッギング 74
5.13 語句 75
5.14 練習問題 76

第 6 章 結果を生む関数 **81**
6.1 戻り値 81
6.2 段階的な改良法 82
6.3 合成関数 84
6.4 ブール代数関数 85
6.5 再帰関数の拡張 86
6.6 信用して跳び越える 88
6.7 もう 1 つの例題 89
6.8 型の検証 90
6.9 デバッギング 91
6.10 語句 92
6.11 練習問題 93

第 7 章 繰り返し処理 **97**
7.1 多重代入 97
7.2 変数更新 98
7.3 while 文 99
7.4 ブレイク 100
7.5 平方根 101
7.6 アルゴリズム 103
7.7 デバッギング 103
7.8 語句 104
7.9 練習問題 104

第 8 章 文字列 **107**
8.1 文字列は文字の配列 107
8.2 `len` 108
8.3 `for` ループによる横断処理 109
8.4 文字列のスライス 110
8.5 文字列は変更不可 111
8.6 探索 111
8.7 ループ処理とカウンタ変数 112
8.8 文字列メソッド 112

8.9 in 演算子 114
8.10 文字列の比較 114
8.11 デバッギング 115
8.12 語句 117
8.13 練習問題 118

第 9 章 事例研究：単語あそび **121**
9.1 単語リストの読み込み 121
9.2 練習問題 122
9.3 探索 123
9.4 インデックス付きループ 125
9.5 デバッギング 126
9.6 語句 127
9.7 練習問題 127

第 10 章 リスト **131**
10.1 リストは配列である 131
10.2 リストは変更可能 132
10.3 リストの横断的処理 133
10.4 リストに対する演算 134
10.5 リストのスライス 134
10.6 リストメソッド 135
10.7 写像・フィルタ・還元 136
10.8 要素の削除 137
10.9 リストと文字列 138
10.10オブジェクトと値 139
10.11別名参照 140
10.12リストを引数に使う 141
10.13デバッギング 143
10.14語句 145
10.15練習問題 146

第 11 章 辞書 **149**
11.1 辞書は写像 149
11.2 カウンタの集合として辞書を使う 151
11.3 ループ処理と辞書 152
11.4 逆ルックアップ 153

11.5 辞書とリスト 155
11.6 メモ 156
11.7 大域変数 158
11.8 デバッギング 160
11.9 語句 161
11.10練習問題 162

第 12 章 タプル **165**
12.1 タプルは変更不可 165
12.2 タプルの代入 167
12.3 タプルを戻り値 168
12.4 可変長引数タプル 168
12.5 リストとタプル 169
12.6 辞書とタプル 171
12.7 配列の配列 173
12.8 デバッギング 174
12.9 語句 175
12.10練習問題 176

第 13 章 事例研究：データ構造・選択 **179**
13.1 単語頻度分布解析 179
13.2 乱数 180
13.3 単語ヒストグラム 181
13.4 頻度の高い単語 183
13.5 選択的な仮引数 184
13.6 辞書の差し引き 184
13.7 乱雑な単語選択 185
13.8 マルコフ解析 186
13.9 データ構造 188
13.10デバッギング 189
13.11語句 191
13.12練習問題 191

第 14 章 ファイル **193**
14.1 永続性 193
14.2 読み込み・書き込み 193
14.3 記述演算子 194

14.4 ファイル名とパス 195
14.5 例外捕捉 197
14.6 データベース 198
14.7 削ぎ落とし 199
14.8 パイプ 200
14.9 モジュールを書く 202
14.10デバッギング 203
14.11語句 203
14.12練習問題 204

第 15 章 クラスとオブジェクト **207**
15.1 ユーザ定義型 207
15.2 属性 208
15.3 長方形 210
15.4 戻り値としてのインスタンス 211
15.5 オブジェクトは変更可能 211
15.6 コピー 212
15.7 デバッギング 214
15.8 語句 215
15.9 練習問題 215

第 16 章 クラスと関数 **217**
16.1 時刻 217
16.2 純関数 218
16.3 修正関数 219
16.4 原型と開発計画 220
16.5 デバッギング 222
16.6 語句 223
16.7 練習問題 223

第 17 章 クラスとメソッド **225**
17.1 オブジェクト指向の特徴 225
17.2 オブジェクトの `print` 226
17.3 別な例 228
17.4 もっと複雑な例 229
17.5 `init` メソッド 229
17.6 `__str__`メソッド 230

17.7 演算子の多重定義 231
17.8 型別処理 231
17.9 多態性 233
17.10デバッギング 234
17.11インタフェースと実装 235
17.12語句 235
17.13練習問題 236

第 18 章 継承 **239**
18.1 カードオブジェクト 239
18.2 クラスの属性 240
18.3 カードの比較 241
18.4 積み札 243
18.5 積み札のプリント 243
18.6 追加・移送・シャッフル・ソート 244
18.7 継承 245
18.8 クラス図 247
18.9 デバッギング 248
18.10データカプセル化 249
18.11語句 251
18.12練習問題 252

第 19 章 便利グッズあれこれ **255**
19.1 条件付き表式 255
19.2 リスト内包 256
19.3 ジェネレータ表式 258
19.4 `any` と `all` 259
19.5 セット 259
19.6 カウンター 261
19.7 デフォルト辞書 262
19.8 名前付きタプル 264
19.9 キーワード付き引数を纏める 266
19.10語句 267
19.11練習問題 267

付 録 A デバッギング **269**
A.1 構文エラー 269
A.2 実行時エラー 271
A.3 意味的エラー 275

付 録 B アルゴリズムの解析 **279**
B.1 増加の次数 280
B.2 Python の基本操作の解析 282
B.3 探索アルゴリズムの解析 285
B.4 ハッシュ表 285
B.5 語句 290

付 録 C 日本語の処理（Python2 版） **291**
C.1 ユニコード文字列の生成 291
C.2 エンコード方式の指定 292
C.3 ユニコード文字列のエンコード変換 293
C.4 辞書やタプルで日本語 294
C.5 日本語を含むファイル 295

付 録 D 日本語の処理（Python3 版） **299**
D.1 辞書やタプルで日本語 300
D.2 日本語を含むファイル 300
D.3 クラスで使う 302

訳者あとがき **304**

はじめに

本書の来歴

１９９９年１月 Java を使ったプログラミング入門の講義の準備をしていた。これまで三回もこの講義を行っていたが、段々と嫌気が募ってきた。この講義の落第者の割合は高く、及第した学生の全体的な到達レベルもそんなに高くなかった。

問題の一つは書籍だ。それらの書籍は余りにも大きく、必要以上に Java の文法の細部にページを割き、如何にプログラムを組むかといった点に充分な組織的な配慮がなされてないものであった。それらの書籍の全てが「落とし穴」効果に犯されていた。つまり、最初は易しく、徐々に先に進むスタイルになっているが、第五章あたりから急に難しくなって床が抜けてしまう。学生たちは余りにも急に多くの新しいことにぶつかり、これ以降私は系統だった講義ができなくなり、切れ切れの講義をせざるをえないという状況に陥る訳だった。

講義開始の二週間前私は自分自身で本を書くことに決めた。目標は

- でき得る限り短く。同じ内容ならば五十ページを割くより、十ページで済ます方が学生の負担は少ない。
- 語彙に注意する。呪文は最小限にし、語句は最初にキチンと定義して使う。
- 徐々に組み立てる。「落とし穴」を避けるため、難しいテーマはそれを細分化したステップで構成する。
- プログラム言語ではなく、プログラミングに焦点を当てる。最小限に必要なものだけで Java を構成する。

本の題名は必要だった。そのとき心に浮かんだのが、「コンピュータサイエンティストのように考えてみよう」(“How to think like a computer scientist”) である。

私の第一版は荒削りなものであったが有効だった。学生たちはそれを読み、高度な話題や興味あるテーマにもクラスの時間を割けるほどの理解を示した。さらに、最も重要なことであるが、学生たちがそれらのテーマについての実習もできたことだ。私はこの本を GNU Free Documentation License の下で出版した。このことはこの本のコピー、変更、さらに配布は利用者が自由にできることを意味していた。

次ぎに起きたことは実にかっこいいことだった。バージニア州の高校教師、Jeff Elkner が私の本を引き継ぎ、それを Python で書き直してくれた。そして、彼はその翻訳版を私に送ってくれた。私は私の書いた本を読んで Python を学ぶという希有な経験をした。2001 年には Green Tea Press から Python 版の初版を出版した。

2003 年に私は Olin College で教鞭を取り始め、そこで初めて Python を教えることになった。やってみると、Java のときと対照的に学生の躓きは少なく学習が進み、もっと興味有るテーマに取り組めるようになり、概して学生たちはより面白がった。

九年間以上かけて私は間違いを修正、サンプルプログラムを推敲、そして練習問題に多くの材料を追加し、この本を充実したものにしよう努力してきた。その結果誕生したのがこの本である。少し控え目に書名は Think Python とした。主な変更点を以下列記する。

- 各章の終わりにデバッギングに関する節を設けた。これらの節はバグを避けること、バグの発見の仕方の技術を扱っている。
- 理解を助ける小テストからかなり本格的な課題までに渡り、多くの練習問題を追加した。そしてそれらの殆ど全てに解答を書いた。
- 一連の事例研究を追加した。それらは課題、その解法、議論からなる少し長めの例題である。
- プログラム開発手法や基本的なデザインパターンの議論を拡張した。
- デバッギング、アルゴリズムの解析に関する付録を追加した。

第二版では以下のような改訂を行った：

- 本の内容および関連するコードを Python3 に即して更新した。
- 初心者が Python をインターネットブラウザで実行することができるように `web` に関する新たな一節を追加した。
- 第 4.1 節で導入した `Swampy` と呼んだタートルグラフィックパッケージを Python の標準のモジュールである `turtle` に変更した。
- 一つの新らしい章を追加した。この章は「便利グッズ」という題で必ずしも必要ではないがあれば便利だという Python の特徴を導入した。

私はこの本を使うことで学習が楽しみになり、あなたの役に立ち少しでもコンピュータサイエンティストのように考えることができるようになることを期待する。

Allen B. Downey
Olin College

第1章　プログラムが動くまで

この本の目標は、如何にしたらコンピュータサイエンティストのように考えることができるかをあなたに教えることである。そのような考え方は数学者、工学者、自然科学者のそれぞれの特徴を合わせ持っている。数学者のように、コンピュータサイエンティストは自分のアイデアをコンピュータ上で実限させるために形式言語を使う。工学者のように物ごとを設計し、部品を集めて一つのシステムを作り、さまざまの可能性の損得を評価する。また、自然科学者のように、複雑系の振る舞いを調べ、仮説を立て、予測を検証する。

このコンピュータサイエンティストに求められる最も重要な能力は**問題解決能力（problem solving）**である。問題解決能力は問題を定式化し、その解決について創造的に考え、その解決を明白にかつ正確に表現する能力のことである。徐々に明らかになるが、プログラミングを学習するプロセスはこの問題解決能力を耕す大変に貴重な機会である。この章を「プログラムが動くまで」とした理由もここにある。一面では、プログラムの学習それ自体は有意義な能力開発である。他面では、プログラミングは他の目的のための手段である。追々とその目的とは何かがはっきりしてくるはずだ。

1.1　プログラムとは何か

一つの**プログラム（program）**とは如何に計算を進めるのかということを特定した一連の命令の集まりである。ここで計算といっているものは例えば方程式の解を求める、多項式の根を求めるなどの数学的なものかもしれないが、しかし、それは文書の中やプログラムの中（不思議に思えるが）のテキストを検索したり、置き換えをしたりすることでもある。

異なったプログラム言語で細部は異なってみえるが、基本的な命令はどのような言語でも確認できる：

入力： キーボード、ファイルそしてその他の様々な装置からデータを得る。

出力： 画面にデータを表示、ファイルやその他の装置にデータを書き出す。

演算： 加算や乗算のような基本的な数学的操作。

条件実行： 特定の条件を調べ、条件が満たされていると目的のコードを実行する。

繰り返し： ある動作を繰り返し実行する。大抵の場合何かを少しずつ変えて実行する。

信じられないことだが、これらはプログラムが持つべき機能の殆ど全てである。あなたが既に使ったどのプログラムも、見かけは複雑にみえても、ここで述べた基本的な機能によく似た命令で出来上がっているのだ。だからプログラミングとは全てを盛り込んだ大きなタスクをより小さい複数のタスクに分解し、上記の基本的な機能の一つとして実行できるほど細かな多数のタスクに分解することである。

1.2　Python を走らせる

Python を勉強しようとして最初に挑戦しなければならないことの一つはあなたのコンピュータに Python や関連するソフトウエアをインストールすることかもしれない。そのコンピュータのオペレーティング・システムに馴染んでいて、特にコマンドベースのインタフェースに問題がなければ Python をインストールすることには何の問題もないであろう。しかし初心者にとってシステム管理っぽいこととプログラミングを同時に学ぶことは苦痛になるかもしれない。

この問題を避けるために、インターネットブラウザ上で Python を走らすことから始めることを推奨したい。後に Python に馴染んできた時点で Python をインストールするための幾つかの示唆を提供することにしたい。

インターネットブラウザ上で Python を動かす `web` ページはいくつもある。あなたが好みの一つがありそれを使っているならばそれを使うので構わない。さもなければ `PythonAnywhere` を推奨したい。これをから始めるための詳細を http://tinyurl.com/thinkpython2e で述べている。

Python には二つのヴァージョンがある。これらは Python2 と Python3 と呼ばれている。二と大変に似ており一つの学習すれば他のヴァージョンに移行することは極めて易しい。この本は Python3 について書かれた本であるが、必要に応じて Python2 にも触れた。

Python **インタプリタ（Interpreter）**は Python で書かれたプログラム（Python コード）を読み実行するためのプログラムである。環境によってアイコンをクリックしてこのインタプリタを起動することになるかコマンド入力で始めることになる。何れにしても以下のような出力が表示されるはずだ：

```
Python 3.4.0 (default, Jun 19 2015, 14:20:21)
[GCC 4.8.2] on linux
Type "help", "copyright", "credits" or "license" ......
>>>
```

最初の三行はこのインタプリタやこれを走らせているオペレーティング・ｌｓシステムに関する情報である。だからあなたの画面に表示されたものと若干異なるかもしれない。しかし Python のヴァージョンナンバー（3.4.0）はこのインタプリタが Python3 であることを示すものであり、確認してほしい。それが 2 で始まっているとするとインタプリタは Python2 である。

最後の行（山型の記号>>>） はインタプリタがユーザに入力可能状態を示す**入力請求記号（prompt）**である。あなたがキーボードから 1+1 と入力しエンタキーを敲くとインタプリタは 2 と表示する。

```
>>> 1 + 1
2
```

これで出発の準備はできた。これ以降は Python インタプリタを走らせコードを実行できる環境にあるとして話を進めることにしたい。

1.3 初めてのプログラム

伝統的に新しい言語であなたが書く最初のプログラムは “Hello World!” と呼ばれるものである。そのプログラムがすることの全ては画面に “Hello World” と表示することだからである。Python では以下のようにある。

```
print( 'Hello World!')
```

これが紙の上に何も印刷しないけれど、**print 文（print statement）**の一つの例である。print 文は一つの値を画面に表示する。いまの場合それは単語で、

```
Hello World!
```

である。プログラムでテキストの最初と最後を示す引用符号は結果には表示されない。括弧はこの print が関数であることを示す。第 3 章で関数は詳しく触れる。

Python2 では構文が少し異なり、それは関数ではないので括弧は使わない：

```
print 'Hello World!'
```

これらの区別は直ぐに意味のあるものであることがわかるが、今はこれで十分である。

1.4　算術演算子

“Hello, World” の後は次のステップは計算である。Python はそのための**演算子（operators）**を提供している。それらは加算や乗算の計算を表現するための特別な記号である。

演算子+、-そして*は以下のように加算、減算、乗算を実行する：

```
>>> 40 + 2
42
>>> 43 - 1
42
>>> 6 * 7
42
```

演算子/は除算を実行する：

```
>>> 84 / 2
42.0
```

何故結果が 42 でなく 42.0 なのか不思議に思うかもしれないが、次節で説明する。

最後は演算子**でべき乗算を実行する。ある数値をべき指数回乗算する：

```
>>> 6**2 + 6
42
```

ある種のコンピュータ言語では記号^がべき乗算に使われることがあるが、Python ではこの記号は XOR と言われるビット演算子である。このビット演算子に馴染みがないとする、以下の結果に驚くかもしれない：

```
>>> 6 ^ 2
4
```

この本ではビット演算については触れない。興味のある人は http://wiki.python.org/moin/BitwiseOperators. を読んでほしい。

1.5　値と型

値（value）は文字や数値のようなプログラムに関わる基本的なものである。これまでに出会った値は 2、42.0 それに’Hello World!’ である。

これらの値は異なった**型（types）**に属している。つまり 2 は**整数（integer）**であり、42.0 は**浮動小数点数（floating point）**であり、'Hello world!' は文字の列を含んでいるので、**文字列（string）**である。

ある値がどの型であるか不明なときはインタプリタが教えてくれる：

```
>>> type(2)
<class 'int'>
>>> type(42.0)
<class 'float'>
>>> type('Hello, World!')
<class 'str'>
```

これらの結果では "class" という単語は部類（カテゴリー）の意味で使われている。つまり一つの型は値が属する一つの類概念のことである。当然だが、整数は int 型に、文字列は str 型に属し、小数点を含む数値は float 型に属する。

ところで、'2' や '42.0' は何型かな？　数値のようにみえるが、文字列のように引用符で括られている。

```
>>> type('2')
<class 'str'>
>>> type('42.0')
<class 'str'>
```

これらは str 型に属する、つまり、文字列である。

大きな数を表示するとき、1,000,000 の様に三つの数字毎にカンマを入れることがある。これは Python の世界では不正な整数であるが、合法的だ。

```
>>> 1,000,000
(1, 0, 0)
```

まあ、期待したものではないが。Python インタプリタはカンマで区切られた三つの整数の配列だと解釈する。このような配列の関しては後にさらに学習するはずだ。

1.6　形式言語と自然言語

自然言語（natural language）は英語、スペイン語、フランス語のように人々が話している言語である。それらは（人々はこれらに対してある規範を課そうとするが）人工的に設計されたものではない。これらは自然と進化したのだ。

形式言語（formal language）はある特別の用途のために人工的に設計された言語である。例えば、数学で用いる記号は数や記号の間の関係を記述する上で適切な形式言語である。化学者は分子の化学構造を示すために形式言語を用いる。そして、さらに重要なことであるが、

プログラム言語は計算を表現するために設計されてきた形式言語である。

形式言語はその構文が厳格になる傾向がある。例えば、$3+3=6$ は数学的には正しい構文で作られた文である。しかし、$3+=3\ \$\ 6$ はそうでない。H_2O は化学的には正しい構文で作られた文である。 ${}_2Zz$ はそうでない。

構文のルールは二つ特徴物からなる、つまり**字句（tokens）**と**文法（sturucture）**である。字句は単語、数、化学元素のような言語の基本要素である。$3+=3\ \$\ 6$ が不正である一つの理由は$が数学的に正しい字句でないことである。また、${}_2Zz$ では省略形 で表現される元素がないことである。

構文エラーの二つ目は文の文法に関連するものだ。つまり、字句の並べ方のよるものだ。$3+=3$ という文は $+$ や $=$ は正しい字句であるが、隣接して使うことは不正である。同じように、化学式では元素記号の後に添え字が付き前ではない。

英文を例に取ると：

> This is @ well-structured Engli$h sentence with invalid t*kens in it.
> This sentence all valid tokens has, but invalid structure with.

ここで最初の文は不正な字句を含む文法的に正しい英文の例であり、二番目は有効な字句からなる文法的に不正な英文の例である。

あなたが英文や形式言語の文を読んでいるとき、あなたはその文の構文がどうなっているか確認しなければならない（自然言語のばあいはこれを無意識に行っている）。このような過程を**構文解析（parsing）**と言う。

形式言語と自然言語は多くの共通点、字句、文法、構文、意味があるが、いくつかの違いがある：

曖昧さ（ambiguity）： 自然言語は曖昧さに満ちあふれている。人々は文の中味に手懸かりを求め、他の情報を使ってこの曖昧さに対処している。形式言語は、文の中味に拘わらず、ほぼまたは完全に曖昧さを排除するように設計されている。

冗長性（redundancy）： 自然言語では曖昧さがあることや誤解を避けるために、冗長性を高めている。その結果、自然言語はだらだらしたものになる。形式言語は冗長性を減らしその結果簡明なものになっている。

逐語性（literalness）： 自然言語は慣用句や比喩に満ちあふれている。わたしは“The penny dropped”と言ったとしても、もしかするとそこには“penny”も“dropping”もないかもしれない（この慣用句的な表現は“ 目から鱗が落ちる ”という意味だ）。一方、形式言語ではそこで言われたことがそのままの意味になる。

自然言語を喋って育った人々は形式言語に慣れることに多々苦労する。ある意味では、この相違は詩と散文の違いに似ているが、それ以上だ：

詩（poetry）： 単語はその意味と同時に音韻のためにも使われる。詩全体によって感情の起伏を作り出す。曖昧さの高さは一般的ばかりでなく、意図的だ。

散文（prose）： 単語の逐語的な意味はもっと重要になり、文法はより意味を持つようになる。散文はより素直に解析できるが、多くのばあい曖昧さは多い。

プログラム（program）： 一つのコンピュータプログラムが持っている意味は唯一で曖昧さがなく、字句と文法の解析で完全に理解できる。

ここでプログラム（その他の形式言語でも）を読む上で留意すべき点を列記する。第一に形式言語は自然言語に比較して稠密なことだ。そのためそれを読むためには時間を要する。また、文法は極めて重要だ。従って、上から下、右から下へと読み下すことは薦められない。そのかわり、字句を特定し、文法的な解釈を頭でしながら、構文解析をすべきだ。最後に、細かなことであるが、自然言語では見過ごしてもよいような綴りや句読点の僅かな間違いが形式言語では重大な相異を生み出す。

1.7 デバッギング

プログラミングにはエラーは付き物だ。いたずら心から、プログラミングに起因するエラーは**バグ（bugs）**と呼ばれる。そしてこのバグ取りの過程を**デバッギング（debugging）**と呼ぶ。

プログラミングや特にデバッギングはときとして心理的な葛藤を呼び起こすことがある。もしあなたが困難なバグに格闘しているとすると、怒り、失望、恥ずかしさの感情を持つかもしれない。

人々がこのような状況におかれるとコンピュータがあたかもヒトであるように対応することはよく知られていることである。作業が上手くいっていると我々はコンピュータを僚友と感ずる、そして、コンピュータが意地悪で、言うことを聞いてくれないと、我々はそのようにヒトに対して振る舞うように、コンピュータ

に対しても振る舞う（Reeves and Nass, “The Media Equation: How People Treat Computers, Television, and New Media Like Real People and Plays”）。このようなヒトの反応に対して心構えをしておくことは賢明だ。一つの接近法はコンピュータを例えば、速度や精度に対して特殊な能力を持っているが、物事を斟酌し、大枠を捉えることに極端に弱点のある従業員と考えることである。

あなたの仕事は優秀なマネージャであることである。つまり、その長所を活かし、弱点を補強する方法を探すことだ。そして、コンピュータに対するあなたの感情が仕事をする上で邪魔にならないよう、あなたの感情を問題解決することに集中する方法を探すとよい。

デバッグの方法を学ぶことは期待したものではないかもしれない。しかし、これはプログラミングという枠を超えて多くに仕事にも役にたつものである。各章の終わりにここで述べたようなデバッグの節を設けた。あなたの助けになればなによりだ。

1.8 語句

問題解決能力（problem solving）： 問題を定式化し、解を見つけそしてそれを表現する過程。

高級言語（high-level Language）： Python のような、人にとって読み書きが容易になるように設計されたプログラム言語。

低級言語（low-level Language）： コンピュータにとって実行が容易であるよに設計されたプログラム言語。「機械語」とか「アセンブリ言語」と呼ばれる。

移植性（portability）： 一種以上のコンピュータ上で実行可能な性質。

入力請求記号（prompt）： インタプリタでユーザからの入力を受けることが可能になっていることを示すための文字列。

プログラム（program）： 一連の計算を実行するための一連の命令の集まり。.

print 文（print statement）： Python インタプリタが画面上に値を出力することを意図した命令。

演算子（operators）： 加法、乗法そして文字列の連結などの単純な計算を表現する特な記号。

値（value）： プログラムが操作する数値や文字列のような基本的データの実体。

型（types）： 値が持つ属性。これまで出会ったのは整数（int 型）、浮動小数点数（float 型）そして文字列 (str 型）である。

整数（integer）： 全ての数を表現する型。

浮動小数点数（floating point）： 小数点以下を持つ数値を表現する型。

文字列（string）： 文字の列を表現する型。

自然言語（natural language）： 人類の進化と共に進化したヒトが話す言語。

形式言語（formal language）： 数学的な考えやコンピュータのためのプログラムのように特別な目的のために設計された言語。そして全てのプログラミング言語は形式言語である。

字句（tokens）： 自然言語の単語に類似したプログラムの意味的構造上の基本要素の一つ。

構文（syntax）： プログラムの構造。

構文解析（parsing）： 形式文法に従ってプログラムを解析すること。

バグ（bugs）： プログラムに潜んでいるエラー。

デバッギング（debugging）： プログラムに潜む三種類のエラーを見つけ取り除く作業。

1.9 練習問題

練習問題 1.1 この本を読むときには、あなた自身で例題がどう動くか確認しつつ進めるようコンピュータを前にして読んでほしい。

新しいテーマに出会うたびに、間違いをすべきである。例えば、“Hello World!” プログラムである。print 文で使っている引用符（’）の一つがないと何が起きるか？二つないと何が起きるか？print の綴りを間違えると何がおきるか？

このような間違いを犯した経験は、このような間違いをするとそのとき表示されるエラーメッセージの意味を確認できるので、この本で読んだことが何であったかの記憶を助けてくれ、デバッギングの助けにもなる。早い内に、それも意図的に間違いをしてみる方は後に偶然にそれをするより賢明だ。

1. print 文でカッコの一つまたは双方を外すと何がおきるか？

2. 文字列の引用符の一つまたは双方を外すと何がおきるか？
3. -2 のような負の数を表現するのにマイナス符号-を付ける。もしある数の前に+の符号を付けると何がおきるか？また 2++2 では何がおきるか？
4. 数学では 02 のように数字の前にゼロを書くことは許されている。Python では何がおきるか？
5. 演算子なしで二つの数字を並べて書くと何がおきるか？

練習問題 1.2 Python インタプリタを起動し、計算をしてみよう。

1. 42 分 42 秒は何秒か？
2. 10 キロメートルは何マイルか？（ヒント：1 マイルは 1.61km である。）
3. 10 キロメートル競技を 42 分 42 秒で完走した。1 マイルあたり掛かった平均時間（分と秒を使って）はいくらか？また、マイル毎時で表現した速度はいくらか？

第2章　変数、表式、文

プログラム言語の強力な機能の一つは**変数（variable）**が扱えることだ。変数は値を参照するための付けられた名前である。

2.1　代入文

代入文（assignment statement）によって変数は生成され、それに値が与えられる：

```
>>> message='And now for something completely different'
>>> a=17
>>> pi=3.141592653597932
```

これらの例では三つの代入文を生成した。最初の例では文字列の代入を新しい変数 `message` に、二番目では整数 17 の代入を変数 `a` に、最後の例では、円周率 π の近似値を変数 `pi` に代入した。紙の上で変数を表現するために良く用いられる方法は新しい名前を書きその変数が持つ値に向けて矢印を書くというものだ。この種の図は各変数の現在の状態（心の変動状態のように考えて）**状態図（state diagram）**と呼ぶ。図 2.1 は前の例の状態図である。

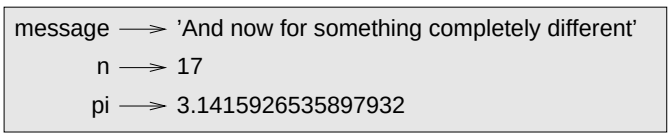


図 2.1: 状態図

2.2　変数名

プログラマーは一般的にそれが何のために用いられているかが分かるように意味がある変数名を選ぶ。

変数名はいくらでも長くできる。それらは文字と数字を含んでいても構わないが、先頭は文字でなければならない。大文字も合法的だが、先頭文字は小文字にするのが賢明だ（理由は後で分かる）。

アンダースコア文字、'_' も名前に付けてよい。これは例えば、
my_name や airspeed_of_unladen_swallow のように、単語を結びつけて使うときに頻繁に使う。

あなたが不正な名前を使うと構文エラーになる：

```
>>> 76trombones = 'big parade'
SyntaxError: invalid syntax
>>> more@ = 1000000
SyntaxError: invalid syntax
>>> class = 'Advanced Theoretical Zymurgy'
SyntaxError: invalid syntax
```

76trombones は先頭が文字で始まらないので不正な名前である。more@は不正な文字を含むので不正である。では、class は？実は、class は Python の**予約語（keywords）**である。インタプリタはプログラムの構造を認識するためにこれらの予約語を使い、変数名として使うことができない。

Python3 は以下のような予約語がある：

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

これらを記憶しておく必要はない。大抵の開発環境ではこれらの予約語は異なった色で表示されるので、それを間違って変数として使ったとしても直ぐ分る。

2.3　表式と文

一つの**表式（expression）**は値、変数、演算子の組み合わさったものである。一つの値のみも表式であり、変数一つもそうである。従って以下の表式は合法的である：

```
>>> 42
42
>>> n
17
>>> n + 25
42
```

入力請求記号（>>>）に続き表式が入力されると、インタプリタはそれを**評価（evaluates）**, つまり表式の値を求める。この例では n は値 17 を持ち、n+25 は値 42 を持つ。

一つの**文（statement）**は変数を生成するまたは値を表示するといった Python インタプリタが実行できるソースコードの単位である。

```
>>> n = 17
>>> print(n)
```

一行目は n の値を与える代入文であり、二行目は変数 n が持つ値を表示する print 文である。

文を入力するとインタプリタはそれを**評価（evaluates）**, つまり文が意味することは何かを探る。一般に文は値を持たない。

2.4 スクリプト・モード

これまではわれわれは Python をインタプリタは**インタラクティブ・モード（interactive mode）**で使ってきた。このモードではあなたがキーボードからプログラムを入力するとインタプリタは即結果を画面に表示した。このモードは習い始めにはよい方法だが数行に渡るコードを書こうとすると、このモードは足手まといになる。

別な方法はコードを**スクリプト（script）**と呼ばれるファイルに保存し、その後このスクリプトを実行するために Python を**スクリプト・モード（script mode）**で走らせる方法である。Python スクリプトは慣例に従って .py の拡張子を使う。

スクリプトの生成やコンピュータ上で如何にスクリプトを実行するかといったことが了解済みならば先に進めるはずだ。さもなければ PythonAnywhere を再度使うことをお薦めしたい。スクリプト・モードでの Python の実行方法は別途に http://tinyurl.com/thinkpython2e に纏めておいた。

.Python は二つのモードを提供しているので、ソースコードをスクリプトとしてまとめて書く前に、そのコードを少しずつテストできる。しかし、インタラクティ

ブ・モードとスクリプト・モードの間には若干の相異があり、混乱を引き起こすかもしれない。

例えば、以下は Python を電卓のように使う例である：

```
>>> miles = 26.2
>>> miles * 1.61
42.182
```

一行目は変数 `miles` への代入文であり、結果はなにも表示されない。二行目は表式で、インタプリタはこの表式を評価し、その結果を表示する。これでマラソン競技は約 42km の距離を走ることが分かる。

しかし、この二行をそっくりスクリプトとして書き、実行しても何も表示されない。スクリプト・モードでは表式それ自体には表示機能がない。スクリプト・モードでも表式は評価されるが、表示は別だということである。スクリプトでは以下のように書く：

```
miles = 26.2
print miles * 1.61
```

最初はこのことで混乱するかもしれない。

スクリプトは通常複数の文を含む。複数の文があるときは、結果はそれらの文が一度に一つずつ実行される。

スクリプトの例を示す：

```
print(1)
x = 2
print(x)
```

これを実行した結果の表示は

```
1
2
```

となる。代入文には表示するものがない。

理解を確認するために、以下のような表現を Python インタプリタで入力し、何が表示されるか調べよてみよう：

```
5
x = 5
x + 1
```

さて、同じ表現をスクリプトとして書き、そのスクリプトを実行してみよう。スクリプトを修正して、各表式を `print` 文のカッコの中に入れて実行してみよう。

2.5 演算子の順位

表式に一つ以上の演算子が含まれていると、その表式の評価は**優先順位（rule of precedence）**による。数学的な演算子については、Python は数学的慣用に従う。**PEMDAS** という頭字語として覚えるとよい。

- 括弧は最も高い優先度を持つ、そしてこれによってある表式の評価をあなたが必要とする順位で処理させることができる。括弧で括られた表式が最初に評価されるので `2 * (3-1)` は 4 であり、`(1+1)**(5-2)` は 8 である。また、あなたは括弧を表現の見やすさのために用いることもできる。`(minute * 100) / 60` では括弧はなくても結果は同じになる。

- べき乗算は次ぎに高い順位を持つ演算子である。だから、`2**1+1` は 3 であり、4 ではない。また、`3*1**3` は 3 であり、27 ではない。

- 乗算、除算は加算、減算（この二つも同じ優先度である）より優先度が高く、同じ優先度を持つ。だから、`2*3-1` は 5 であり、4 ではない。同じく、`6+4/2` は 8 であり、5 ではない。

- 同じ優先度を持つ演算子は左から右へ（例外はべき乗算）と評価される。だから、表式 `degree /2 * pi` は除算が最初で、その結果に `pi` がかけられる。2π で割ることを意図したのであれば、括弧を使うか、`degree/2/pi` とする。

わたしは他の演算子についての優先順位を憶えることにあまり熱心でない。表式がそれらしくみえないときは、括弧を使い明確にすることにしている。

2.6 文字列処理

一般に算術的な演算をそれが見かけの上で数のようであっても文字列に施すことはできない。従って、以下のような演算は不正である：

```
'2'-'1'     'eggs'/'easy'      'third'*'a charm'
```

演算子+は文字列に作用できるが、あなたが期待したものではないかもしれない。これは文字列の**連結（concatenation）**、つまり、文字列と文字列の端と端を結合することである。例を挙げると

```
>>> first = 'throat'
>>> second = 'warbler'
>>> first + second
throatwarbler
```

表示されるものはthroatwarblerである。

演算子*も文字列に使え、文字列の繰り返しを行う。例として、'Spam' * 3はSpamSpamSpamの表示を得る。被演算子の一つが文字列ならば、他は整数というかたちでなければならない。これらの演算子+、*の使用法は算術演算子として持つ機能を考えれば納得がゆくものであろう。4*3が4+4+4と同値であるように、'Spam' * 3は'Spam'+'Spam'+'Spam'であろうからである。

一方、この文字列の連結処理や繰り返し処理と整数の加算、乗算とは異なるところがある。文字の連結ではできないが整数の加算ではできる性質の一つを思い浮かべることができるか？

2.7 コメント

プログラムが大きくなり、より複雑になるとプログラムを読むことが難しくなる。形式言語は稠密なので、コードの一部分を眺めてそこで行われていることを理解することが多々困難なことがある。

そこでプログラムの中に自然言語でそこで行われていることについての注釈を追加することは賢明である。これらは**コメント（comments）**と言われ、記号#で始まる：

```
#compute the percentage of the hour that are elapsed
percentage = (minute * 100) / 60
```

この例ではコメントは一行に書いた。行の終わりに置くこともできる：

```
percentage = (minute * 100) / 60 # percentage of an hour
```

記号#から行の終わりまでがコメントで、プログラムには何の影響も持たない。

コメントはそのコードの部分に自明でない特徴がある場合に有効である。コードを直接眺めることでそこで何が行われているかが分かる方が説明を追加するより好ましいことには論を待たない。

以下のコメントは冗長で無意味である：

```
v = 5  #assign 5 to v
```

以下のコメントはコードのみでは得られない情報を含むので有用だ：

```
v = 5  #velocity in meters/second.
```

変数の付け方が上手いとコメントの量を少なくできるが、長い名前は表式を複雑にさせ読み辛いものになるので、損得を天秤かける必要がある。

2.8 デバッギング

プログラムでは三種類のエラーが起こりうる。つまり、構文エラー、実行時エラー、意味的エラーである。これらの三種類のエラーを区別し、バグ取りが効果的に行えるようにするとよい。

構文エラー： **構文（syntax）**とはプログラムの構造に言及したもので、その構造が持つべきルールである。例えば、カッコは右左を揃えて使わなければならない、だから (1+2) は合法であるが、8) は**構文エラー（syntax error）**になる。

あなたのプログラムが構文エラーを一つでも含んでいると、Python はそのエラーに対してエラーメッセージを出し、そこでプログラムの実行を停止する。プログラミングの経験の浅い初めの数週間はこの種の構文エラーのバグ取りにかなりの時間を費やすだろう。しかし、経験を積むに従ってエラーをする度合いは減るし、エラーを容易く発見できるようになる。

実行時エラー： 第二のエラーはプログラムが実行されるまでは現れないので実行時エラーと呼ばれているものである。これらのエラーはプログラムを実行しているときに何か例外的（そして悪い）な状況が起きたときに出るエラーなので**例外（exceptions）**とも呼ばれている。

簡単なプログラムでは実行時エラーは希であるので、それに遭遇するまでは暫くかかる。

意味的エラー： 第三のエラーは**意味的エラー（semantic error）**である。意味的エラーがあってもコンピュータはエラーメッセージを出さないでプログラムは成功裡に実行さたようにみえる。しかし、結果が間違っている。意図したことと別なことが実行された訳である。特に、間違っているにせよ、あなたがせよと命じたことが行われているからだ。

問題は書き上げたプログラムがあなたが書こうとしたプログラムでないことである。プログラムの内容（従ってその意味）が間違っている訳である。この意味的なエラーを探し出すことはきわどい仕事である。なぜならば、コンピュータの出力結果をみてソースプログラムの間違いを探り当てるという後ろ向きの作業をしなければならないからだ。

2.9 語句

変数（variable）： 一つの値を参照するための名前。

代入文（assignment statement）： 一つの値を一つの変数に割り当てる文。

状態図（state diagram）： 値の一組とそれらを参照する変数の一組の関係を示すグラフィックな表現。.

予約語（keywords）： コンパイラがプログラムを構文解析するために予め使うことを予定してある単語。あなたのプログラムでは `if`、`def` そして `while` のような単語を変数名として使うことができない。.

被演算子（operands）： 演算子が作用する値。

表式（expression）： 変数、演算子そして値を組み合わせて一つの計算を表現した式。結果は一つの値になる。 .

評価（evaluate）： 一つの単独な値を生成するために演算を実行して表式を簡潔にする。

文（statement）： 一つの命令や動作を表現するコードの一段落。 これまでに出会ったのは代入文そして `print` 文である。

実行（execute）： 一つ文を実行しすることそしてその文が意味することを行う。

インタラクティブ・モード（interactive mode）： 入力請求時にコマンドや表式を入力することによって `Python` インタプリタを使う方式。

スクリプト・モード（script mode）： 一つのスクリプトを読み込みそれを一気に実行するような `Python` インタプリタの使い方。

スクリプト（script）： 一つのファイルとして保存されたプログラム（通常はインタプリタで実行される）。

優先順位（rule of precedence）： 複数の演算子とを含む表式の評価（結果の値を出す）する際に適用される演算の順序に関する規則。

連結（concatenation）： 二つの被演算子を端と端で繋ぐこと。

コメント（comments）： 他のプログラマー（またはそのソースコードを読む）人たちのためにプログラムに書き込む情報。プログラムの実行には何の効果も無い。

構文エラー（syntax error）： 構文解析が不可能になるようなプログラム上のエラー（そしてそれ故インタプリタもエラーを出す）。

例外（exceptions）： プログラムの実行時に発生するエラー。

意味論（semantics）： プログラムの意味。

意味的エラー（semantic error）： プログラムが意図した以外の別な内容を実行してしまうこと。

2.10 練習問題

練習問題 2.1 前章の繰り返しになるが、新しいテーマに出会うたびにそれをインタラクティブ・モードを使って確認するように、そして意図的に間違いをして何がおこるかみてみよう。

- 代入文 `n = 42` は合法であることをみてきた。それでは `42 = n` では何が起きる？
- `x = y = 1` は？
- 文の終わりは必ずセミコロン（;）で終わりとする言語がある。文の終わりにセミコロンを書くと Python では何が起こる？
- 文の終わりにピリオド（.）を書くと Python では何が起きるか？
- 数学的な表記では x と y の乗算は xy と書けるが、これを Python でしたら何が起きる？

練習問題 2.2 Python インタプリタを電卓として使い以下の計算をしない。

1. 半径 `r` の球の体積は $\frac{4}{3}\pi r^3$、半径 5 の球の体積はいくつか？（ヒント：392.7 は間違い）
2. ある本の定価は 24.95 ドルだが、本屋は 40%の割引をしている。送料は最初の一冊では 3 ドルで、二冊目以降は 75 セントである。60 冊買うとして総額はいくらになるか？
3. わたしは 6:52am に家を出て、ゆっくりした歩行（8 分 15 秒毎マイル）で 1 マイル歩き、普通の歩行（7 分 12 秒毎マイル）で 3 マイル歩き、再びゆっくりした歩行で 1 マイル歩いて帰宅したとすると何時に朝食ができるか？

第3章　関数

プログラミングの範疇で言うと**関数（function）**とはある計算を実行する文の名前付きの一括りである。関数を定義するということは、名前を付けることと、一括りの文を定義することである。こうすると後にこの関数を名前で「呼び出す」ことができる。

3.1　関数呼び出し

既にわれわれはこのような**関数呼び出し（function call）**の一例をみてきた。

```
>>> type(42)
<type 'int'>
```

この関数の名前は type である。括弧の中の表式は関数の**引数（argument）**と言われるものである。この関数は結果としてこの引数の型を表示する。

一般の関数は引数を「受け取り」、結果を「返す」。この結果のことを**戻り値（return value）**と言う。

Python は値の型を他の型に変換する組み込み関数群を提供している。int 関数は任意の値を受け取ることができ、可能なときはその型を整数に変換する。さもないと、苦情を吐き出す：

```
>>> int('32')
32
>>> int('Hello')
ValueError: invalid literal for int() with base 10: 'Hello'
```

また、int 関数は浮動点小数も整数に変換するが、それは四捨五入ではなく小数点以下切り捨てになる。

```
>>> int(3.99999)
3
>>> int(-2.3)
-2
```

float 関数は整数や文字列を浮動小数点数に変換する：

```
>>> float(32)
32.0
>>> float('3.14159')
3.1415899999999999
```

最後の関数である str 関数は引数を文字列に変換する：

```
>>> str(32)
'32'
>>> str(3.14159)
'3.14159'
```

3.2 数学関数

Python は馴染み深い数学関数を一つの**モジュール（module）**として提供している。モジュールとは関連する関数類を一纏めしたファイルである。

モジュールは使う前にそれを**インポート文（import statement）**を使ってインポートしなければならない：

```
>>> import math
```

この文で math と言う名前の**モジュール・オブジェクト（module object）**が生成される。モジュール・オブジェクトを表示すると、このモジュールの情報が得られる：

```
>>> math
<module 'math' (built-in)>
```

このモジュール・オブジェクトはこのモジュールで定義された関数や変数を沢山に含んでいる。それらの関数の一つにアクセスするには、モジュール名と関数名をドット（終止符として知られている）で区切って特定しなければならない。この記法は**ドット表記（dot notation）**と言う。

```
>>> ratio = signal_power / noise_power
>>> decibels = 10 * math.log10(ratio)

>>> radians = 0.7
>>> height = math.sin(radians)
```

最初の例では関数 log10 と信号・雑音比をデシベルで計算するために使った（勿論、signal_power、noise_power は定義済みとしている）。数学モジュールは同様に log 関数、つまり底を e とする対数も提供している）。

二番目の例ではラジアンの sin である。変数の名前がヒントになっているように関数 sin や他の三角関数（cos，tan，etc）は引数としてラジアンで表現した角度を受け取る。角度をラジアンに変換するには 360 で割り、2π を掛けるとよいから

```
>>> degrees = 45
>>> radians = degrees / 360.0 * 2* math.pi
>>> math.sin(radians)
0.70710678118654746
```

ここで math.pi の表記は math モジュールで定義されている変数 pi を受け取る。この変数の値は π の 15 桁程度の精度の近似値である。三角関数の知識があるとこの結果を 2 の平方根を 2 で割ったもので確認できる：

```
>>> math.sqrt(2)/2.0
0.70710678118654757
```

3.3 混合計算

これまで変数、表式、文などのプログラムの構成要素を単独で、それらを混合する使い方の話なしで、眺めてきた。

プログラミングの強力な特徴の一つはこれらの構成要素を**混合（composition）**して使えることである。例えば、関数の引数には算術演算子を含む任意の表式が使える：

```
>>> x = math.sin(degrees / 360 * 2 * math.pi)
```

関数の呼び出しさえ構わない：

```
>>> x = math.exp(math.log(x + 1))
```

一つの例外を除けば、どの場所に値を置いてもよいし、任意の表式を置いてもよい。その例外とは代入文の左辺は一つの変数名でなければならないということだ。

左辺に変数名以外の表式を置くと構文エラーになる（この規則の例外は後でみる）。だから

```
>>> minutes = hours * 60    #正しい
>>> hours * 60 = minutes    #間違い
SyntaxError: can't assign to operator
```

3.4 新規関数の追加

これまではPythonが提供している関数のみを使ってきたが、新たな関数を作成することもできる。**関数定義（function definition）**はその新しい関数に名前を付けること、関数が呼ばれときに実行される一連の文を定義することである。

一例を示す：

```
def print_lyrics():
    print("I'm a lumberjack, and I'm okay.")
    print("I sleep all night and I work all day.")
    #（訳注：モンティパイソン「空飛ぶサーカス」より）
```

defは予約語で、これが関数定義であることを示す。関数名はprint_lyricsである。関数名に対する規則は変数名に対するものと同一である。変数名と関数名が同じというのは避けるべきだ。関数名の後の空の括弧はこの関数は何も引数を受け取らないことを示す。

関数の一行目は関数の**ヘッダー（header）**と言われ、残りは**ボディー（body）**と言われる。ヘッダーはコロンで終わり、ボディーはインデント（字下げ）される。習慣として、このインデントの量は4である。ボディーは任意の数の文を含んで構わない。

print文にある文字列は二重引用符で括られている。シングル引用符と二重引用符は同じ作用をもたらすが、一般にシングル引用符が使われる。例外は文字列の中に既にシングル引用符（またはアポストロフィ）が使われている場合である。

関数定義をインタラクティブ・モードで行っているときには、関数定義がまだ終わっていないことを知らせる省略記号 (...) が表示される：

```
>>> def print_lyrics():
... print("I'm a lumberjack, and I'm okay.")
... print("I sleep all night and I work all day.")
...
```

関数定義を終わりにしたいときには空行を入力する（これはスクリプト・モードでは不必要）。関数を定義することは同じ名前の**関数オブジェクト（function object）**生成することであり、このオブジェクトの型はfunction型である。

```
>>> print(print_lyrics)
<function print_lyrics at 0xb7e99e9c>
>>> type(print_lyrics)
<class 'function'>
```

関数の呼び出しは組み込み関数と同じである：

```
>>> print_lyrics()
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
```

一度関数が定義されると、その関数は他の関数の内部で呼ぶことができる。例として、前の歌詞を繰り返し表示する repeat_lyrics を作ってみる：

```
>>> def repeat_lyrics():
... print_lyrics()
... print_lyrics()..
```

そして関数 repeat_lyrics を呼び出してみる：

```
>>> repeat_lyrics()
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
```

この歌の実際の歌詞はこうではないが。

3.5 関数定義とその利用法

前節で紹介したプログラムを纏めると以下のようになる（スクリプト・モードで作成。）：

```
def print_lyrics():
    print "I'm a lumberjack, and I'm okay."
    print "I sleep all night and I work all day."

def repeat_lyrics():
    print_lyrics()
    print_lyrics()

repeat_lyrics()
```

このプログラムは二つの関数定義、print_lyrics と repeat_lyrics、を含んでいる。関数定義の実行は他の文と同じように行われるが、これは関数オブジェクトを生成する作業である。関数が呼ばれるまで関数内部の文の実行は行われない。そして、関数定義自体は何も出力しない。

予想したように関数の生成はそれが実行される前になされなければならない。別な言葉で言うと、関数はそれが呼ばれる前にその関数定義が実行されなければならないのだ。

練習問題として最後の行を最初に持ってきたスクリプトに変更して実行してみる。つまり関数呼び出しを関数定義の前で行うわけである。どのようなメッセージがでるか？

同様に関数定義の repeat_lyrics を print_lyrics の前で行うようにスクリプトを変更し、実行してみる。何が起こるか？

3.6 実行の流れ

関数はそれが最初に呼ばれる前に定義されなければならないということを確認するために、どの文から実行されるかという**実行の流れ（flow of execution）**を知らなければならない。

実行は常にプログラムの最初からだ。文は一度に一つずつ先頭から末尾に向かって実行される。

関数定義の実行もこの流れに沿って実行される。しかし、関数内部の文の実行はその関数が呼ばれるまで実行されない。

関数を呼ぶという文は実行の流れにできた回り道のようなものだ。次ぎの文が実行される替わりにそこではその関数のボディーにある文が実行され、そして別れたところに戻ってくる。

呼ばれた関数がまた別な関数を呼ぶことができると言う事態を想定しなければ、話は簡単だ。一つの関数の途中で他の関数の文を実行しなければならないかもしれない。また、この新たな関数が他の関数を呼ぶということもあり得る。

幸にして Python は実行状況を詳しく追跡することに優秀であるので、関数内の文が実行完了するとその関数を呼んだ関数内でこの関数に実行が移ったところに戻ってくる。プログラムの最後に来ると実行は終了になる。

プログラムを常に上から下へと読む必要はなく、ときには実行の流れに沿って読む方が賢明なこともあるのだということだ。

3.7 仮引数と実引数

これまでみてきた組み込み関数のいくつかは引数を要求したものである。例えば、math.sin は一つの数を引数として受け取る。いくつかの関数は一つ以上の引数を受け取る。math.pow は指数計算で底と指数の二つを受け取る。

関数の中では、その引数は**仮引数（parameters）**と呼ばれる変数に代入される。以下は引数を一つ受け取る新たに定義した関数の例である：

```
def print_twice(bruce):
    print(bruce)
    print(bruce)
```

この関数は引数を bruce と名前である仮引数に代入する。この関数が呼ばれると、仮引数の値をそれが何であれ）二回 print する。

この関数の引数は何であれ print できるものであれば有効である：

```
>>> print_twice('Spam')
Spam
Spam
>>> print_twice(42)
42
42
>>> print_twice(math.pi)
3.14159265359
3.14159265359
```

組み込み関数に適用できた混合計算の規則は新たに定義された関数にも適用される。従って、print_twice 関数の引数には任意の表記を書くことができる：

```
>>> print_twice('Spam' * 4)
SpamSpamSpamSpam
SpamSpamSpamSpam
>>> print_twice(math.cos(math.pi))
-1.0
-1.0
```

引数は関数が呼ばれる前に評価されるので、上記の二例の引数、'Spam' * 4 や math.cos(math.pi) は一度だけ評価されるだけである。

勿論、引数は一つの変数でもよい：

```
>>> michael = 'Eric, the half a bee.'
>>> print_twice(michael)
Eric, the half a bee.
Eric, the half a bee.
```

この関数で我々が引き渡す引数の名前 michael と関数の仮引数の名前 bruce とは何の関係もない。この関数を呼んだプログラムの中でその値が何と呼ばれていたかということには無関係で、print_twice では全てが bruce と呼ばれる。

3.8 変数や仮引数はローカルである

関数の内部で変数を生成したときには、この変数は**ローカル変数（local）**で関数の内部でしか存在しない。例を示す：

```
def cat_twice(part1, part2):
    cat = part1 + part2
    print_twice(cat)
```

この関数は二つの引数を受け取り、それらを連結し二回 print する。これを使ってみる：

```
>>> line1 = 'Bing tiddle '
>>> line2 = 'tiddle bang. '
>>> cat_twice(line1, line2)
Bing tiddle tiddle bang.
Bing tiddle tiddle bang.
#（訳注：モンティパイソン「空飛ぶサーカス」より）
```

関数 cat_twice が終了した後、変数 cat は消滅する。たからこの変数を print しようとすると例外になる：

```
>>> print(cat)
NameError: name 'cat' is not defined
```

仮引数もローカル変数である。例えば、関数 print_twice の外部では bruce というものは存在しない。

3.9 スタック図

どの変数がどこで使われているかを追跡するために、**スタック図（fig.stack）**を書いてみるのも有用だ。状態図と同じように、スタック図も各変数の値を表示するが、どの関数に属するかも示す。

各変数は**フレーム（frame）**によって区切られる。フレームはその脇に関数名を付した矩形枠で内部ではその関数の仮引数や変数の名前がある。先の例のスタック図は図 3.1 に示した。

フレームはどの関数がどの関数を呼んだかということが分かるように積み重ねた状態で表現されている。われわれの例では、`print_twice` は `cat_twice` によって呼ばれているし、`cat_twice` は特別な最上位のフレーム、`__main__`によって呼ばれている。関数の外側で変数を生成すると、この変数は`__main__`に属することになる。各仮引数は対応する引数が持つ値と同じものだ。だから、`part1` は `line1`

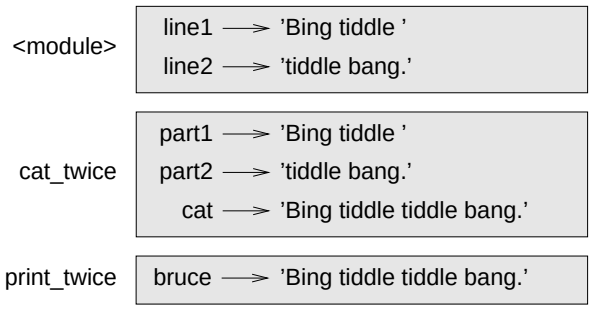


図 3.1: スタック図.

と同じ値を参照しているし、`line2` も然りである。`bruce` は `cat` と同じ値を持つことになる。

関数呼び出しのときにエラーを起こすと、Python はその関数の名前、関数を呼んだ関数の名前、その前というように`__main__`に至る全ての関数の名前を表示する。例えば、関数 `print_twice` 内で `cat` にアクセスしようとすると、`NameError` になる：

```
Traceback (innermost last):
  File "test.py", line 13, in __main__
    cat_twice(line1, line2)
  File "test.py", line 5, in cat_twice
    print_twice(cat)
  File "test.py", line 9, in print_twice
    print(cat)
```

```
NameError: name 'cat' is not defined
```

この関数のリストは**トレースバック（traceback）**と呼ばれるものである。これはプログラムのどこでこのエラーが発生したか、そして如何なる関数を実行しているときかといったことを示す。さらにこのエラーが起きたソースコードの行番号も示される。このトレースバックに現れる関数はスタック図と同じ順序で示される。現在実行中の関数は一番下になる。

（訳注: 上のトレースバックを発生させたスクリプトを以下に示す。ファイル名は test.py：

```
01 def print_twice(bruce):
02     print(bruce)
03     print(bruce)
04     print(cat) #誤り
05
06 def cat_twice(line1, line2):
07     cat = line1 + line2
08     print_twice(cat)
09
10 #main
11 cat_twice('bono', 'niko')
```

左端の行番号は便宜的に付けたものである。上のトレースバックはまずスクリプト test.py の 11 行目のメイン（module）で cat_twice('bono', 'niko') が実行され、次に 8 行の関数 cat_twice 内の print_twice(cat) が実行され、次に 4 行目の関数 print_twice 内の print cat でエラーが検出されたことを知らせている）

3.10 結果を生む関数とボイド関数

われわれが使っている関数、特に数学関数は結果を生む関数である。適当な名前がないのでこれを**結果を生む関数（fruitful function）**と名付けることにする。print_twice のような関数は実行されても結果を返さないので、これは**ボイド関数（void function）**と呼ばれている。

関数を呼ぶというときには、何時も何か返すものを伴う実行がなされると期待するものである。だからその結果をある変数に代入、表式の一部として使う。

```
x = math.cos(radians)
golden = (math.sqrt(5)+1)/2
```

関数をインタラクティブ・モードで使うと、Python は結果を表示する：

```
>>> math.sqrt(5)
2.2360679774997898
```

ところが、スクリプト・モードで結果を生む関数をそれ自体で呼ぶと、結果は永遠に失われてしまう：

```
math.sqrt(5)
```

スクリプトは 5 の平方根を計算するが、保存し、表示する機能を持っていないので不便なことになる。

ボイド関数も画面に表示をし、似た効果をもたらすが戻り値がない。その結果を変数で受け取ると、その値は None という特別な値になる：

```
>>> result = print_twice('Bing')
Bing
Bing
>>> print(result)
None
```

値 None は文字列'None' とは異なる。それはそれ自身の型をもつ特別な値である：

```
>>> type(None)
<class 'NoneType'>
```

これまで作成した関数は全てボイド関数であった。これからの数章では結果を生む関数を書くことになる。

3.11 なぜ関数？

プログラムを関数に分割することに心を砕く理由はあまりはっきりとしないかもしれない。その理由としていくつかある。

- 関数を生成することで一括りの文に名前を付けることができ、読みやすく、デバッグし易いプログラムを作ることができる。
- 関数化することで繰り返しをなくし、プログラムを短くすることができる。そして、何か変更が必要なことが起きたときには一個所を修正することで足りる。

- 長いプログラムを関数に分割することで関数ごとのデバッグが可能になり、その後にそれらを全体して纏めることができる。
- 良くできた関数は多くにプログラムで有用だ。一度書いてデバッグして完成させると、それを再利用することができる。

3.12　デバッギング

あなたが獲得しなければならない最も重要な能力の一つはデバッギングである。デバッギングはやる気を挫くものであるが、これはプログラミングの作業のなかで最も知的で、やりがいがあり、興味深い領域である。

ある意味では、デバッギングは探偵の仕事に似ている。幾つかの手懸かりを頼りにその結果を引き起こした過程やイベントを特定しなければならない。また、デバッギングは実験科学に似ている。何が間違っているかについて一つの考えが浮かんだら、プログラムを修正し、実行してみる。この仮説が正しかったとすると、その変更の結果を予測てき、前に一歩進める。仮説が間違っていたとすると、再度仮説を立て直さなければならない。シャーロック・ホームズも指摘したように、「あなたが不可能を取り除いてしまったとき、どんなものが残ろうとも、それがありそうもないことであっても、残ったものが真実にちがいない」（コナンドイル「四つの署名」）。

多くの人たちにとっては、プログラミングとデバッギングは同義語だ。つまり、プログラミングとは最終的に欲しい結果を生み出すプログラムを徐々に完成させるデバッギングの過程である。この考え方は、最初はほんの簡単なことができるプログラムを作成し、進捗するにつれて、何時もその限りでは動くプログラムを心がけつつ、これを徐々に変更、デバッグするということだ。

例えば、Linux はオペレーティング・システムの一つであり、ソースコードは数千行もあるプログラムであるが、Linus Torvalds が嘗て Intel 80386 チップの動きを調べるために作成した一つの簡単なプログラムであった。Larry Greenfield によれば、「Linux の初期のプロジェクトの一つは print AAAA と　print BBBB の切り替えを行うプログラムであった。後にこれが Linux へと進化した。」（The Linux Users ’ Guide Beta Version 1）

3.13　語句

関数（function）： 意味のある命令を実行する文の纏まりに名前を付けたもの。関数は引数を取るものもあり、結果を返すものもある。

関数定義（function definition）： 関数名、仮引数、実行する文の纏まりを指定して新たな関数を作成するための文。

関数オブジェクト（function object）： 関数定義で生成される一つの値。関数名はこの関数オブジェクトを参照する変数である。

ヘッダー（header）： 関数定義の一行目の部分。 . :

ボディー（body）： 関数定義の内部の本体（文の集合）。

仮引数（parameters）： 関数の中で引数として渡される値を参照する変数。

関数呼び出し（function call）： 関数を実行する文。関数名と必要な実引数のリストが続く。

引数（argument）： 関数が呼ばれたときその関数に付随して与えられる値。この値は関数の中では対応する仮引数に代入される。

ローカル変数（local）： 関数内で定義され使われる変数。ローカル変数はその関数内でのみ有効である。

戻り値（return value）： 関数の結果。もし関数が表式として使われているとすると、その戻り値がその表式の値となる。

結果を生む関数（fruitful function）： 結果を返す関数。

ボイド関数（void function）： 値を返さない関数。

モジュール（module）： 関連する関数群や付随する定義を一つに纏めたファイル。

インポート文（import statement）： 一つのモジュールを読み込みモジュール・オブジェクトを生成する文。

モジュール・オブジェクト（module object）： 一つのモジュール内で定義されている値へのアクセスを提供する `import` 文が生成する値。

ドット表記（dot notation）： 他のモジュールに属する関数や定義を使うとき、それらをモジュール名に続いてピリオドそして関数名（または定義変数）で指定する構文規則。

混合（composition）： より長い表式の一部として表式を、またはより複雑な文の一部として文を使うこと。

実行の流れ（flow of execution）： プログラムの実行時にどの文から実行されるかの順序。 .

スタック図（fig.stack）： 一つのモジュールで使われる関数群やそれらの内部で定義される変数、そしてそれらの変数が参照する値の対応を表示するグラフィカルな表現。

フレーム（frame）： スタック図内で関数呼び出しを表現する枠。

トレースバック（traceback）： 例外が発生したときに表示される実行されていた関数のリスト。

3.14 練習問題

練習問題 3.1 Pythonは文字列の長さを返す組み込み関数len、例えばlen('allen')は5を返す、を提供している。
文字列の表示で、文字列の最後の文字が70桁にくるようなright_justify関数を作成せよ。

```
>>> right_justify('monty')
                                                                    monty
```

練習問題 3.2 関数オブジェクト自体も関数の引数とすることができる。例えば、do_twice関数は関数オブジェクトを引数として受け取りその関数を二度実行する、つまり

```
def do_twice(f):
    f()
    f()
```

さて、do_twiceが以下の関数print_spamを引数とする使用例をみてみよう：

```
def print_spam():
    print('spam')


do_twice(print_spam)
```

この関数を以下の順序を踏んで変更せよ：

1. 上記のプログラムをスクリプトとして書き、実行せよ。

2. 引数を二つ持つように do_twice を変更せよ。一つの引数は関数オブジェクトであり、他はこの関数が受け取る引数の値である。

3. print_spam をより一般的な関数 print_twice に変更せよ。この新しい関数は引数を一つ持ち、それを二回表示する。

4. 新しい関数 do_four を作成せよ。この関数は関数オブジェクトを引数の一つとし、他の引数はこの関数が引数として受け取る値である。この新しい関数は引数として受け取った関数を四回実行する。この新しい関数の本体は二つの文で書ける。四つではないよ。

解答例："http://thinpython.com/code/do_four.py"

練習問題 3.3　この練習問題はこれまで学んだ文や他の機能を使って解答できるものである。

1．以下のような格子を描く関数を作成せよ。

```
+ - - - - + - - - - +
|         |         |
|         |         |
|         |         |
|         |         |
+ - - - - + - - - - +
|         |         |
|         |         |
|         |         |
|         |         |
+ - - - - + - - - - +
```

ヒント：print 文で一つ以上の値を一時に表示したいときは、値をカンマで区切ったもので print とするとよい：

```
print('+','-')
```

また通常は print 文は行の終わりに段落（\n）を吐いて次の行に移るが、この動きを無効にして例えば空白を吐いて終わりにすることもできる：

```
print('+', end=' ')
print('-')
```

この出力は'+ -' である。
　print 文それ自体で行が終わったら、改行で次ぎに行に移る。

　2. 同様に四行四列の格子を描く関数を作成せよ。
解答例："http://thinpython.com/code/grid.py"。
出典：この練習問題はOualline 著"Practical Programming. Third Edition (O'Relly Media, 1997)" の中の練習問題を基礎とした。

第4章　事例研究：インタフェース設計

この章はインタフェース設計のための諸機能を取り入れる過程を紹介する事例研究である。

まずturtleモジュールを導入しよう。このモジュールを使うとturtleグラフィックスで描画を作成できる。turtleモジュールは大抵のPythonのインストール・パッケージに含まれているはずだ。PythonAnywhereを使ってPythonを起動しているばあいにはturtleは動かないかもしれいない。

Pythonをインストールしてあるのであれば、ここのサンプルプログラムは動くはずだ。そうでなければインストールのよい機会である。仕方はに別途http://tinyurl.com/thinkpython2e. に纏めておいた。

この章のサンプルコードはhttps://thinkpython.com/code/polygon.pyにある。

4.1　turtleモジュール

turtleモジュールがインストールされているか確認するためにPythonインタプリタを起動して以下を入力してみよう。

```
>>> import turtle
>>> bob = turtle.Turtle()
```

これでカメを意味する小さい矢印が表示された新しい窓が現れるはずだ。確認できたら窓を閉じる。

さて、スクリプトモードで以下のコードを入力しmypolygon.pyと言う名前のファイルとして保存する：

```
import turtle
bob = turtle.Turtle()
print(bob)
turtle.mainloop()
```

turtle モジュール（t は小文字）は Turtle オブジェクトを生成する Turtle（T は大文字）関数 Turtle を含んでいる。ここではこのオブジェクトは変数 bob に代入されている。この bob を表示すると以下のようなものだ出るはずだ：

```
<turtle.Turtle object at 0xb7bfbf4c>
```

これは変数 bob は turtle モジュールで定義されている Turtle 型のオブジェクトを参照しているという意味だ。

mainloop は window に対してユーザが何か動作するのを待つようにと通知していることだ。もっとも今のばあいにユーザのできることは窓を閉じることぐらいしかないが。

一度 Turtle を生成していおくと、このカメ（今のばあいは bob）をこの窓の中で動かす**メソッド（method）**を呼び出すことができる。メソッドは関数に似ているが、少し異なった構文を使う。例えばカメを前進させるには：

```
bob.fd(100)
```

このメソッド fd は Turtle オブジェクトである bob に付随しているものである。メソッドを呼ぶということは要請するようなものだ。bob に前に進むことを要請したわけである。

fd の引数はピクセル単位の距離であり、実際の長さは使っているディスプレーによる。

Turtle に対して呼び出すことができるその他のメソッドには後退 bk、左回転 lt そして右回転 rt がある。メソッド lt と rt の引数は度を単位とした角度である。

また各 Turtle はペンを保持していて、それを上げたり下げたりする。それが下がっていると、Turtle は動くにつれて軌跡を画面に描く。メソッド pu と pd は「ペンを上げ」、「ペンを下げ」のつもりである。

直角を描くために以下の行をプログラムに追加してみよう（追加する場所は bob の生成の後で、mainloop の前である）。

```
bob.fd(100)
bob.lt(90)
bob.fd(100)
```

このプログラムを実行すると、bob は東に向いて前進し、その後北向きに進路を取り、直角を成す二つの線分が画面にできる。

このプログラムを変更して、正方形を描いてみよう。上手く行くまで続けてみよう。

4.2 簡単な繰り返し

多分あなたが行った変更は以下のようなものはずだ：

```
bob.fd(100)
bob.lt(90)

bob.fd(100)
bob.lt(90)

bob.fd(100)
bob.lt(90)

bob.fd(100)
```

これは for 文を使うともっとコンパクトに書ける。これらを mypolygon.py に追加して実行してみよう。

```
for i in range(4):
    print('hello!')
```

得られる結果は以下である：

```
hello!
hello!
hello!
hello!
```

これは簡単な for 文の利用法である。詳細は後にみることにする。正方形を描くプログラムに for 文を使うのには、上の例題で充分だ。 試してみよう。

以下は正方形描画の for 文版だ。

```
for i in range(4):
    bob.fd(100)
    bob.lt(90)
```

for 文の構文は関数定義と似ている。コロンで終わるヘッダーを持ち、インデントされたボディからなる。ボディは任意の数の文を含んで構わない。for 文は**ループ（loop）**と呼ばれる。実行の流れがボディを実行してあとボディのトップに戻るからである。今の場合はボディを 4 回繰り返す。

この for 文を使った正方形描画プログラムはオリジナルのものとホンの僅かだが異なっている。それは正方形を描いた後、bob が余計な左転回をすることだ。

この for 文版は指摘したように最後の左回転が余分である。これが実行時間に影響するかもしれないが、同じことをループを使って書けることでコードを簡素にすることができる。またこの版ではループを終えたときにはカメはループに入る前の方向を向いている効果も持っている。

4.3　練習問題

以下は TurtleWorld を使った練習問題のシリーズである。それらは面白いものであるが、学習のポイントも含まれている。作業をしながらその学習ポイントにも目を向けてほしい。

以下の節でその解答を提示するが、完成するまで（少なくも試みるまで）はそれをみてはいけない。

1. square という関数を作成せよ。仮引数として turtle である t を持つ。この turtle で正方形を描くようにせよ。そしてこの関数で bob を引数として受け取り関数呼び出しを書き実行せよ。

2. length という別な仮引数を追加せよ。関数のボディを変更して length を一辺の長さにするように変更し、関数呼び出しに第二の引数を持つように変更せよ。適当な長さの length を与え実行してみよう。

3. 関数 lt の回転角度の既定値は 90 である。しかし、lt(bob, 45) のように角度を与えることもできる。そこで square をコピーして名前を polygon とする。この関数は第二の仮引数 n を持ち、ボディを n 個の辺を持つ正多角形を描くように変更せよ。

   ヒント：n 個の辺を持つ正多角形の外角は 360/n である。

4. circle 関数を作成せよ。この関数は polygon 関数から派生し、仮引数として turtle を値に持つ t と、半径 r を持ち、近似的に円を描く関数である。

   ヒント：円周は近似的に正 n 辺多角形で近似すると、円周　= length * n で表現できる。ここで length は多角形の一辺の長さである。

   もう一つヒント：描画速度が余りのも遅いときは、bob,delay = 0.01 とするとよい。

5. 更に一般的な関数arc関数を作成せよ。この関数は追加の仮引数としてangleを持つ。描画はこのangleを角度とする孤を描くことにある。angle = 360で完全な円を描くことになる。

4.4 カプセル化

第一例は正方形を描画するコードを関数にする問題である。この関数の呼び出しではTurtleオブジェクトを引数として渡すようにする。解答例は

```
def square(t):
    for i in range(4):
        t.fd(100)
        t.lt(90)

square(bob)
```

最も内側のfdとltは二段にインデントされる。これらは関数定義のボディの始まりであるforループの内部にある。次の行にあるsquare(bob)は左余白をなくして入力される。このことはforループも関数定義も終了したことを意味する。

関数の内部ではtは同一のturtleであるbobを参照しているから、lt(t)はlt(bob)と同じ効果を持つ。それならば、何故直接bobを呼ばないのか？これは関数squareではtはどんなturtleにもなり得るからだ。二匹目のturtleを作成し、それをこの関数の引数にすることもできる、つまり：

```
alice = turtle.Turtle()
square(alice)
```

コードの一部を関数として纏める作業を**カプセル化（encapsulation）**と呼ぶ。カプセル化の利点は関数名としてコードに名前がつくのでドキュメントとなること。もう一つの利点はコードを再利用したいとき、関数呼び出しで事足りることである。

4.5 一般化

次ぎのステップはsquare関数に仮引数lengthを追加することである。解答例は

```
def square(t, length):
    for i in range(4):
        t.fd(length)
```

```
        t.lt(90)

square(bob, 100)
```

関数に仮引数を追加する作業を**一般化（generalization）**という。この作業で関数はさらに一般化される。

次ぎのステップも一般化で、正方形を描画する替わりに、正多角形を描画するものである。解答例は以下である：

```
def polygon(t, n, length):
    angle = 360 / n
    for i in range(n):
        t.fd(length)
        t.lt(angle)

polygon(bob, 7, 70)
```

7 辺を持ち、一辺の長さが 70 の正多角形を描画する。

Python2 を使っているとすると、角度の値が整数の割り算に伴なう打ち切り誤差を生じるかもしれない。簡単な解決は `angle = 360.0/ n` とすることだ。分子が浮動小数点数であるので結果も浮動小数点数になるからだ。

もし引数の数が増えて、その数値の意味が曖昧になり、順序が不明になったときは、仮引数の名前を付して引数を書くことは合法である：

```
polygon(bob, n=7, length=70)
```

これは**キー付き引数（keyword argument）**という。この構文規則はプログラムをより読みやすくする。

4.6　インタフェース設計

次ぎのステップは半径 r を仮引数として円を描画する関数の作成の問題である。以下は 50 辺を持つ多角形を描画する解答の一例である：

```
def circle(t, r):
    circumferance = 2 * math.pi * r
    n=50
    lenght = circumferance / n
    polygon(t, lenght, n)
```

第一行では円の円周を計算している。π の値は数学モジュールで与えられるので、math モジュールのインポートを忘れないように。慣例として import 文はプログラムの先頭に置く。

n は多角形の辺の数であり、この多角形で半径 r の円を近似的に描画することができることを使う。

この解答例の問題点は n が定数であることである。これでは大きな円では一辺の長さが大きくなって円の近似としては粗くなる。逆に、小さな円では一辺が小さくなりすぎて描画時間が無駄になる。n を仮引数にすることも一案だが、インタフェースとしては明確さを欠くことにある。

関数の**インタフェース（inteface）**とはその関数の使い方の指針である：「仮引数は何か？」、「この関数は何をする関数か？」、「戻り値は何か？」について纏めたものである。そのインタフェースは明確であるということは、「可能なかぎり簡単で、簡単すぎることもない」（アインシュタイン）ことである。

今の例では、r は描画しようとする円の半径の値を特定するものであるのでインタフェースに属すると考えられる。しかし、n は如何に円を描画するかに使うものなので、そうではない。これはインタフェースを乱雑にするだけなので、むしろ関数のボディのなかで円周から計算する方がよい。

```
def circle(t, r):
    circumference = 2 * math.pi * r
    n = int(circumference / 3) + 3
    length = circumference / n
    polygon(t, n, length)
```

これで一辺の数は circumferance/3 になり、一辺の長さは約 3 になる。これはできる多角形で充分に円を近似できる最小で、しかも大きな円でも充分な数である。これで如何なる大きさの円の描画に使える関数ができたことになる。

4.7 再因子分解

円の描画では関数 polygon を再利用できた。これは正多角形が円の近似になるからである。しかし、円弧を描画しようとすると polygon も circle も使えない。

一つの解決方法は関数 polygon のソースコードをコピーして、直接に関数 arc に加工するものである。結果は以下のようになる：

```
def arc(t, r, angle):
    arc_length = 2 * math.pi * r * angle / 360
```

```
    n = int(arc_length / 3) + 1
    step_length = arc_length / n
    step_angle = angle / n

    for i in range(n):
        t.fd(step_length)
        t.lt(step_angle)
```

後半は関数 polygon に似た処理をしているが、polygon を書き換えないと使えない。そこで、polygon を第三の引数を持たせるように書き換えることにする。名前も polyline とする：

```
def polyline(t, n, length, angle):
    for i in range(n):
        t.fd(length)
        t.lt(angle)
```

この新しく定義された関数は polygon にも使えるし、arc にも使える。polygon は以下のようになる：

```
def polygon(t, n, length):
    angle = 360.0 / n
    polyline(t, n, length, angle)
```

関数 arc は以下のようになる（訳注：ここでの angle は円弧を見込む角度）：

```
def arc(t, r, angle):
    arc_length = 2 * math.pi * r * angle / 360
    n = int(arc_length / 3) + 1
    step_length = arc_length / n
    step_angle = float(angle) / n
    polyline(t, n, step_length, step_angle)
```

円周は円弧の特別な形なので、関数 circle は関数 arc で書くことができる：

```
def circle(t, r):
    arc(t, r, 360)
```

以上のような過程、つまり関数インタフェースを改良し、コードの再利用を促進する方法を**再因子分解（refactoring）**という。いまの場合は関数 arc と関数

polygon とに共通する部分があり、この共通項として関数 polyline を「抽出」したことになる。

最初からこのことが分かっていたら、関数 polyline から始めたかもしれない。しかし、多くの場合プロジェクトの最初から全ての関数インタフェースを設計しておくことは難しい。プログラムを書き始めて問題が良く理解できるものである。ときとして再因子分解をしようと思うことは問題をよりよく理解できたサインである。

4.8 開発計画

開発計画（development plan）はプログラムを書く過程のことである。この過程としてわれわれがみてきたものは、「カプセル化と一般化」である。この過程は以下のように纏められる：

1. まず、関数を使わないで小さいプログラムを書く。
2. プログラムは実行できるようになったら、それを関数としてカプセル化し、名前を付ける。
3. 適宜仮引数を追加してそれを一般化する。
4. 1〜3 を繰り返し、プログラムが動くようにする。
5. 再因子分解でプログラムが改良できるかコードを眺めてみる。例えば、プログラムの中に同じようなコードを使っている個所あれば、適当な関数でそれらを置き換える。

この過程は欠点もあるが、もしプログラムをどのような関数に分解したらよいか事前に分からないときには助けになる（後に別なアプローチを示す）。この方法はプログラミングを進めながら設計も進める方法である。

4.9 ドキュメント文字列

ドキュメント文字列（docstring）とは関数のインタフェースを説明するために関数の初めの部分に置く文字列のことである（“doc” は “documentation” の省略）。例を示す。

```
def polyline(t, n, lenght, angle):
    """与えられた長さ（lenght）と与えられた線分間
    角度（angle）（度）を持つ n 個の線分を描画する。
    t は turtle である。"""

    for i in range(n):
        t.fd(length)
        t.lt(angle)
```

この文字列は三重の引用符で括られている。三重の引用符は複数行に渡る文字列を書くことができるからだ。

この例のようにドキュメント文字列は短いがこの関数を使おうとしたときに必要な情報を含んでいる。これは簡潔にこの関数の機能が関数の細部に渡らずに説明されているし、仮引数の役割（時には型についても）も示されている。

このようなドキュメントを書くことは関数のインタフェースの設計の重要な部分である。良く設計されたインタフェースは説明も明快にできるはずである。もし関数のインタフェースの説明に苦労するようであれば、それは多分にインタフェース自体の改良が必要なサインであり得る。

4.10　デバッギング

インタフェースは関数とその呼び手との接面のようなものである。呼び手は仮引数に対して値を与えることに同意し、関数はそれをもとに作業をすることに同意する。

例としてみると、polyline は四個の引数を要求している。t は turtle、n は線分の数、従って整数、length は線分の長さ、だから正の数値、angle は度を単位とする数値である。

これらの要求は**事前条件（precondition）**という。関数が実行される前に実現しておく必要のある条件だからである。この逆に、関数の終わりで示される条件は**事後条件（postcondition）**と言われるものである。これにはその関数が意図したもの（線分の描画のような）、その関数の実行による副産物（turtle の移動、世界に及ぼす変更のような）が含まれる。

事前条件は呼び手が責任を負うものである。呼び手がこの事前条件に反し、関数が正常に働かないとすると、バグは呼び手にあり、関数ではない。

4.11 語句

メソッド（method）： オブジェクトに付随している関数、そしてドット表記で呼ばれる。

ループ（loop）： 繰り返して実行されるプログラムの部分。

カプセル化（encapsulation）： 一つの目的のための一連の文の集合を一つの関数として変形する過程。.

一般化（generalization）： 必要なく特定されているもの（例えば数）を適当な一般的なもの（変数や仮引数）に置き換える過程。

キー付き引数（keyword argument）： 「キー」として仮引数の変数名を含めた実引数

インタフェース（inteface）： 関数名、引数や戻り値の説明を含めたその関数の使い方に関する叙述。

再因子分解（refactoring）： 関数のインタフェースやコードの質的な向上のため作動しているプログラムを改変する過程。

開発計画（development plan）： プログラムを作成する過程。

ドキュメント文字列（docstring）： 関数定義の中で 関数のインタフェースを叙述した文字列。

事前条件（precondition）： 関数を呼ぶ前に呼び手が満たすべき条件。

事後条件（postcondition）： 関数の実行が終わった時点でその関数が満たすべき条件。

4.12 練習問題

練習問題 4.1 https://thinkpython.com/code/polygon.py をダウンロードし以下の課題を行え。

1. 関数 `circle(bob, radius)` の実行に伴うスタック図を作成せよ。手書きでもよいし、コードに `print` 文で追加してもよい。

2. 4.7 節の関数 arc は、円の描画が常に想定した円の外側に沿った線分で近似されているので正確さに欠けるところがある。その結果 turtle が停止した位置は正しい位置から僅かにずれる。わたしの解答例ではこの誤差を減ずる工夫がされている。コードを読んで納得できるがみよ。図を描いてみるとそれが有効であることが分るかもしれいない。

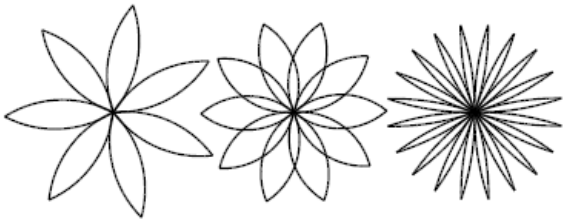


図 4.1: タートルの花々

練習問題 4.2　図 4.1 のような描画に対応する関数の典型的なセットを作成せよ。
解答例:https://thinkpython.com/code/flower.py　及びhttps://thinkpython.com/code/polygon.py. も必要である。
練習問題 4.3　図 4.2 のような描画に対応する関数の典型的なセットを作成せよ。
解答例：https://thinkpython.com/code/pie.py.

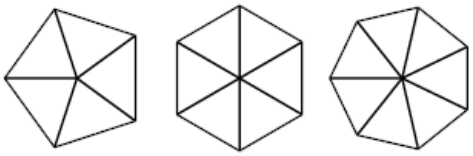


図 4.2: タートルのパイ

練習問題 4.4　アルファベットの文字は適当な数の基本要素、つまり横線、縦線、いくつかの曲線から生成できる。フォントをこのような要素から作れるようデザインせよ。そして、アルファベットを描画する関数を作成せよ。

先ず、各一文字を描画する関数を生成する。つまり、draw_a、draw_b といった具合である。これらを纏めて letters.py というファイルにする。あなたの制作した関数をテストするために “turtle typewriter” というモジュールを https://thinkpython.com/code/typewriter.py からダウンロードせよ。
解答例:https://thinkpython.com/code/letters.py 及びhttps://thinkpython.com/code/polygon.py. も必要である。
練習問題 4.5　渦巻について、http://en.wikipedia.org/wiki/Spiral（訳注：日本語ではhttp://ja.wikipedia.org/wiki/渦巻）の記事をよみ、アルキメデス

の渦巻（または、類似の渦巻）を描画するプログラムを作成せよ。
解答例：https://thinkpython.com/code/spiral.py.”

第5章　条件文と再帰

この章の主なテーマはプログラムの状態に依存して異なったコードを実行するif文である。しかしその前に新たな二つ演算子、打ち切り除算、モジュラ演算子を導入したい。

5.1　打ち切り除算とモジュラ演算子

打ち切り除算（Floor division）（演算子は//）は二つの数値の除算で結果の小数点以下を負の無限大の方向に丸める演算である。例として一つの映画の上演時間が105分であったとしよう。これが時を単位にすると幾らになるか知りたいと思ったとしよう。従来の除算では浮動小数点数が得られる：

```
>>> minutes = 105
>>> minutes / 60
1.75
```

時は通常は小数点を付けては使わない。打ち切り除算は小数点以下を切り捨て整数として時を表示する：

```
>>> minutes = 105
>>> hours = minutes // 60
>>> hours
1
```

残りは全体から一時間を分にしたものを引き算すると出せる：

```
>>> remainder = minutes - hours * 60
>>> remainder
45
```

他の方法は**モジュラ演算子（modulus operator）**を使う。これは整数に対する演算子で、最初の被演算子を二番目の被演算子で除算したときの余りを生成するものである。Pythonではこの演算子は%で表現される。構文は他の演算子と同じである：

```
>>> remainder = minutes % 60
>>> remainder
45
```

モジュラ演算子は極めて有用なものであることが分かってくる。例えば、ある整数が他の整数で割り切れるか調べたいときには、x % y がゼロならば、x が y で割り切れるということになる。

また、ある整数に一桁目の数字や下何桁かの数字を求めたときにも使える。例えば、x % 10 とすると、整数 x の十進一桁目の数字が、x % 100 で下二桁の数字が得られる。

Python2 を使っているとすると除算は異なった結果になる。除算の演算（演算子は/）はもしも被演算子が二つとも整数値であると打ち切り除算になり、被演算子のどちらかが浮動小数点数であると結果も float 型になる。

5.2 ブール代数表現

ブール代数表現（boolean expression）は**真（true）**と**偽（false）**の表現である。以下の例では、演算子==を使い二つの被演算子を比較し、等しい場合は真をそれ以外は偽を値として持つ。

```
>>> 5 == 5
True
>>> 5 == 6
False
```

True と False は bool 型に属する特別な値である。これらは文字列ではない：

```
>>> type(True)
<class 'bool'>
>>> type(False)
<class 'bool'>
```

演算子==は**比較演算子（relational operators）**の一つであり、他の演算子としては

```
x != y     #x は y に等しくない
x > y      #x は y より大きい
x < y      #x は y より小さい
```

```
x >= y     #x は y より大きいが等しい
x <= y     #x は y より小さいか等しい
```

これらの演算子は数学的な記号として馴染み深いものであるが、Python での記号の意味はそれらと若干異なる。もっともよくある間違いは二重等号（==）の替わりに、等号（=）を使ってしまうことだ。記号=は代入演算子であり、記号==が比較演算子である。また、=>や=<はない。

5.3 論理演算子

三つの**論理演算子（logical operators）**、and、or、 not がある。これらの意味はその英語の意味に似ている。例えば、x > 0 and x < 10 は x が 0 より大きく、かつ 10 より小さいとき真の値を持つ。x % 2 == 0 or x%3 == 0 はどちらかの条件が満たされると真の値を持つ。つまり、ある数が 2 か 3 かで割り切れればよい。最後は not でブール代数の否定演算子である。not（x > y）は、x > y が偽のとき真、つまり x が y より小さいか等しいときである。

厳密に言って、論理演算子の被演算子は論理表式であるべきだが、Python はそう厳密ではない。非ゼロの如何なる数値も論理的に真と解釈する。

```
>>> 42 and True
True
```

この柔軟性は有用であるが、混乱を引き起こす微妙な違いがある。このような使い方は避けた方がよい（何が起きているか把握できるまでは）。

5.4 条件処理

有用なプログラムを書くためにはわれわれは条件を調べ、それに従ってプログラムの振る舞いを替える機能が必要になる。**条件文（conditional statement）**はそのような機能である。この if 文の極簡単な例は

```
if x > 0:
    print('x is positive')
```

if の後のブール代数表式は**条件（condition）**と呼ばれる。それが真であると、インデントされた文が実行され、そうでないと実行されない。

条件文は関数の定義と同じ構造をしている。つまり、インデントされたボディを従えたヘッダー。このような文を**複合文（compound statements）**という。ボ

ディは無制限の数の文があってもよいが、最低でも一文は必要である。ときとして文なしのボディが必要なときがある（大抵の場合これはまだ書き終えていない部分を余白として残したいときだ）。このようなときは何もしないpass文を使うとよい：

```
if x < 0:
    pass      #負の場合の処理が必要
```

5.5 二者選択処理

if文の二番目のかたちは**二者選択処理（alternative execution）**である。これでは二つ可能性があり、条件はどっちを取るかを決める。構文は以下のようだ：

```
if x%2 == 0:
    print('x is even')
else:
    print('x is odd')
```

条件は真または偽のはずなので、二者の内の一つは必ず実行される。二者択一は、実行の流れの分岐であるから、**分岐処理（branches）**とも言われる。

5.6 条件文の連鎖

ときとして二つ以上の可能性がある場合があり、二つ以上の分岐が必要となる。このような処理を表現する一つの方法は**条件文の連鎖（chained conditional）**を使うものがある：

```
if x < y:
    print('x is less than y')
elif x > y:
    print('x is greater than y')
else:
    print('x and y are equal')
```

elifは “else if”の省略形である。ここでも一つの分枝が必ず実行される。elif文の個数には制限がない。else節を使うときはこれを最後に置く。これがある必要もない。

```
if choice == 'a':
    draw_a()
elif choice == 'b':
    draw_b()
elif choice == 'c':
    draw_c()
```

各条件は順次調べられる。もし初めが偽であるとすると、次ぎが調べられという具合である。それらの一つが真であるとその分枝が実行され、文の終わりになる。一つ以上の真があっても初めの真の分枝が実行されるだけである。

5.7 入れ子の条件処理

条件処理は入れ子にできる。三分枝の例題は以下のようになる：

```
if x== y:
    print('x and y are equal')
else:
    if x < y:
        print('x is less than y')
    else:
        print('x is greater than y')
```

外側の条件文には二つの分枝がある。一つの分枝は単純な一つの文からなるが、他の文枝はもう一つの if 文からなる。この if 文は二つの分枝がある。それぞれの分枝はまた条件文だということもあり得るが、今回は各分枝とも単純な一つの文である。インデントされた文の構造から**入れ子の条件処理（nested conditionals）**は読みやすいとは言えないので、なるべく避けるようにしたい。

論理演算子はこのような入れ子の条件処理を単純な処理に替えることができる。例として以下のプログラムは１つの条件文に書き替えることができることを示す：

```
if 0 < x:
    if x < 10:
        print('x は一桁の整数である')
```

print 文は二つの条件文を通過したときのみ実行される。これは論理積演算子 and を使うと簡潔に表現できる：

```
if 0 < x and x > 10:
    print('x は一桁の整数である')
```

5.8 再帰

一つの関数が他の関数を呼ぶことは合法であるが、自分自身を呼ぶことも合法だ。このようなことで何か得なことがあるとは俄には信じられないが、これはプログラムで実現できる魔法のような世界であることが徐々に明らかになる。

例として以下のようなプログラムを眺めてみよう：

```
def countdown(n):
    if n == 0:
        print('Blastoff!')
    else:
        print n
        countdown(n-1)
```

nが0か負になるとBlastoff!（発射！）'をprintする。それ以外はnを表示し、引数をn-1として関数countdown、つまり自分自身を呼ぶ。

main関数でこの関数を以下のように呼んだら何がおきるだろう：

```
>>> countdown(3)
```

関数countdownはn=3から始まる。nの値が0より大きいので、nの値3をprintして自分自身を呼ぶ。

> 関数countdownはn=2で実行される。nの値が0より大きいので、nの値2をprintして自分自身を呼ぶ。
>
> > 関数countdownはn=1で実行される。nの値が0より大きいので、nの値1をprintして自分自身を呼ぶ。
> >
> > > 関数countdownはn=0で実行される。nの値が0に等しいので、"Blastoff! "をprintして、関数をreturnする。
> > >
> > > n=1のcountdownが関数returnに達する。
> >
> > n=2のcountdownが関数returnに達する。
>
> n=3のcountdownが関数returnに達する。

そしてmain関数__main__に戻る。従って全表示は以下のようになる：

```
3
2
1
Blastoff!
```

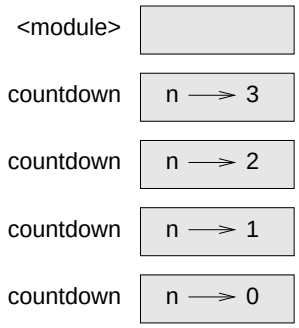


図 5.1: スタック図

自分自身を呼ぶ関数は再帰的で、このプロセスを**再帰処理（recursion）**という。

もう一つの例は文字列を表示するプログラムである：

```
def print_n(s, n):
    if n <= 0:
        return
    print(s)
    print_n(s, n-1)
```

もし `n <= 0` であるとすると、関数 `return` が実行される。実行は呼び出し関数に戻ることである。この関数の残りの部分は `countdown` に類似している。`n` が `0` より大きいと文字 `s` を `print` し、残りの `(n-1)` 回の `print` のために自分自身を呼んでいる。従って、文字 `s` の `print` は `1 + (n-1)`, つまり `n` 回の `print` がなされる。

このような簡単な例では `for` ループの方が易しいかもしれないが、後に `for` ループではプログラムを書くことが難しく、再帰では容易な例に出会うことになる。そのために少し早めに紹介した。

5.9 再帰関数のスタック図

3.9 節で関数の呼び出しに際してプログラムの状態を示すためにスタック図を用いた。同じ図が再帰関数の振る舞いを調べるのに便利だ。

関数は呼ばれる度に Python は関数フレームを生成する。このフレームには関数内のローカル変数や仮引数が含まれる。再帰関数の場合には、同時にスタック上にある関数は一つ以上であるかもしれない。

図 5.1 は `n=3` で呼ばれた `countdown` 関数のスタック図である。いつものようにスタックの最上段は`__main__`のフレームである。このフレームは`__main__`で生成された変数や引数が不明なので、空白である。四つ `countdown` フレームは仮引数の

n の値が異なっている。スタックの底は n=0 のフレームで**基底ケース（base case）**と呼ばれている。これは再帰的な呼び出しをしないのでこれ以上のフレームは存在しない。

練習問題として、print_n 関数が s = 'Hello'、n=2 で呼ばれたときのスタック図を描いてみよう。また仮引数として関数オブジェクトと一つの数 n を仮引数とする再帰関数 do_n を作成せよ。この関数は問題の関数を n 回実行する。

5.10　無制限な再帰

もしある再帰処理が基底ケースを持たないとすると、再帰的な呼び出しは永遠に続くことになる。これは**無制限な再帰（infinite recursion）**として知られていて、一般に健康なアイデアではない。以下はその最小なプログラムの例である：

```
def recures():
    recures()
```

大抵の開発環境では、このような無制限な再帰を含むプログラムは永遠に実行されることはない。Python は再帰の深さの限界値に達するとエラーメッセージを吐き出す：

```
  File "<stdin>", line 2, in recurse
  File "<stdin>", line 2, in recurse
  File "<stdin>", line 2, in recurse
                  .
                  .
                  .
  File "<stdin>", line 2, in recurse
RuntimeError: Maximum recursion depth exceeded
```

このトレースバックは前章でみたものより長めである。このエラー発生時には、スタックには 1000 個の recures フレームができる。

5.11　キーボード入力

これまで生成したプログラムはユーザからの入力を受け付ける機能がなく、いつも同じ動きをするもであった。

Python はキーボードから入力を受け付ける input という組み込み関数を提供している。Python2 では raw_input だ。この関数が呼ばれると、実行は停止し、ユー

ザからの入力待ちになる。ユーザはReturnキーやEnterキーを入力すると、プログラムは再実行され、input関数はユーザが入力したものを文字列として戻す。つまり、

```
>>> text = input()
What are you waiting for?
>>> text
'What are you waiting for?'
```

ユーザからの入力を得る前に、ユーザに何を入力すべきか知らせることが賢明である。関数inputは入力促進文字列を引数として受け取ることができる。つまり、

```
>>> name = input('What...is your name?\n')
What...is your name?
Arthur, King of the Britons!
>>> name
'Arthur, King of the Britons!'
```

入力促進文字列の最後にあるシークェンス\nは**改行（new line）**の記号でこれで改行が起きる。だから、入力文字はこの入力促進文字列の下に現れる訳である。

ユーザが整数値を入力すると期待できるときにはその戻り値を整数型に変換するとよい：

```
>>> prompt = 'What...is the airspeed velocity of an unladen swallow?\n'
>>> speed = input(prompt)
What...is the airspeed velocity of an unladen swallow?
42
>>> int(speed)
42
```

しかし、ユーザが数字以外のものを入力しようとするとエラーが出る：

```
>>> speed = input(prompt)
What...is the airspeed velocity of an unladen swallow?
What do you mean, an African or a European swallow?
>>> int(speed)
ValueError: invalid literal for int() with base 10
```

この種のエラーを処理する方法は後に学ぶ。

5.12　デバッギング

エラーが発生したとき表示される Python のトレースバックは多くの情報を含んでいる。しかし、スタック上のフレームが多数あるときには、その量の多さに圧倒される。その中で有用な情報は

- どのようなエラーか
- どこで起きたか

構文エラーの場所を特定するのは一般的に易しいが、意外なものもある。空白やタブは見えないし無視しがちなので、空白によるエラーは陥りやすい：

```
>>> x = 5
>>>  y = 6
  File "<stdin>", line 1
    y = 6
    ^
IndentationError: unexpected indent
```

この例では第二行が一つの空白でインデントされているのが問題である。しかし、エラーメッセージは y を指していて、誤解を生む可能性がある。

一般にエラーメッセージは問題が発生を確認した場所を示すが、実際のエラーはそれ以前に原因があることもあり得る。ある場合では前の行にそれがある。

実行時エラーでも同様だ。あなたは信号・雑音比をデシベルで計算しようしているとしよう。式は

$SNR_{db} = 10 \log_{10}(P_{signal}/P_{noise})$

と書ける。これを Python で書こうとすると以下のようになる：

```
import math
signal_power = 9
noise_power = 10
ratio = signal_power // noise_power
decibels = 10 * math.log10(ratio)
print(decibels)
```

実行してみるとエラーメッセージに出会う：

```
Traceback (most recent call last):
  File "snr.py", line 5, in ?
    decibels = 10 * math.log10(ratio)
ValueError: math domain error
```

このエラーメッセージは5行目を指しているが、ここにはおかしなところがない。変数 ratio を print してみることは助けになるかもしれない。結果は 0 である。つまり、問題は 4 行目にある。普通の除算の替わりに打ち切り除算が使われていることによる。問題の解決は浮動小数点で計算すればよい。つまり普通の除算にすればよい。

一般にエラーメッセージは問題をどこで発見したかをわれわれに告げる。しかし、多くの場合そこはエラーの原因になる場所ではない。(訳注：エラーメッセージにも注意する必要がある。この例では domain error となっている。つまり関数の適用範囲を超えていることからくるエラーである。)

5.13 語句

打ち切り除算（Floor division）： 二つの数値の除算で結果の小数点以下を負の無限大の方向に丸める演算 (例：5//3 -> 1、-5//3->-2)。

モジュラ演算子（modulus operator）： 一つの整数をもう一つの整数で割ったときの余りを生成する演算子でパーセント記号、(%) が用いられる。.

ブール代数表現（boolean expression）： その値が「真」(True) か「偽」(False) になるような表式。.

比較演算子（relational operators）： 二つの被演算子を比較し、「真」、「偽」の値を返す演算子。==、!=、>、<、>=, そして <=がある。.

論理演算子（logical operators）： ブール代数表現を結合させるための演算子。and,、or として not がある。.

条件文（conditional statement）： ある条件に従って実行の流れを制御するための文。

条件（condition）： どちらの分岐を実行するか決めるための条件文の中で使われるブール代数表現。

複合文（compound statements）： 一つのヘッダーと一つのボディから構成される文。ヘッダーはコロン（:）で終わり、ボディはヘッダーの対してインデントされる。

分岐処理（branches）： 条件文の中で二者択一的に実行される文の一方。

条件文の連鎖（chained conditional）： 二者択一の条件文を `elif` 文を使って連鎖させる。

入れ子の条件処理（nested conditionals）： 条件文の分岐処理の中で条件文を使う。

再帰処理（recursion）： 実行中の関数のなかでその関数を呼ぶ処理。

基底ケース（base case）： 再帰関数の中でその関数を再帰的に呼ばない分岐処理。

無制限な再帰（infinite recursion）： 基底ケースを持たないまたはこれに達することがない再帰処理。最終的には無制限な再帰は実行エラーとなる。

5.14　練習問題

練習問題 5.1 `time` モジュールは同じ名前の `time` 関数を提供している。この関数は今のグリニッジ平均時間を基点とした時間からの経過時間として戻す。`UNIX` ではこの基点は 1970 年 1 月 1 日である。

```
>>> import time
>>> time.time()
1437746094.5735958
```

この時間を読み込み、それを一日の時、分、秒に変換し、且つ基点から何日経過したかを表示するスクリプトを作成せよ。

練習問題 5.2 フェルマーの最後の定理は以下のような代数式を満たす整数 a,b,c は存在しないと言う：

$$a^n + b^n = c^n$$

ここで n は 2 より大きな任意の整数である。

1. 四つ仮引数 `a`,`b`,`c`,`n` を持つ関数 `check_fermat` を作成せよ。この関数は `n` が 2 より大きくしかも

   $$a^n + b^n = c^n$$

   が成立したら、「おやまあ、フェルマーは間違っている」と表示し、さもないと「だめだ。成り立たない」と表示する。

2. ユーザにたいして整数 a,b,c,n の入力請求をして、これらを整数に変換し、関数 check_fermat でこれらの整数値がフェルマーの定理を満たさないかどうかを示す関数を作成せよ。

練習問題 5.3 三つの楊枝を使って三角形を作ることを試みる。例えば、一つが 12 インチの楊枝で、他の二つは 1 インチの楊枝だとしよう。これでは明らかに端と端を接して三角形を作ることはできない。三つの任意の値に対して、三角形をつくる条件を調べることができる。

三角形の一つの辺の長さが他の二辺の長さの和より大きいとこれでは三角形は作れない。これ以外であれば作れる（二つの辺の長さは他の一辺に等しいときは、「縮退」三角形という）

1. 三つ整数を引数とする関数 is_triangle を作成せよ。この関数ではこれらの長さの楊枝から三角形が作れるときは"yes"を print し、それ以外は"no"を print する。

2. ユーザに三つの楊枝の長さを入力させ、関数 is_triangle で検定するプログラムを作成せよ。

練習問題 5.4 以下のプログラムはどのような出力を表示するか？結果を表示したときのスタック図を作成せよ。

```
def recurse(n, s):
    if n == 0:
        print(s)
    else:
        recurse(n-1, n+s)

recurse(3, 0)
```

1. この関数を recurse(-1, 0) で呼び出しと何が起こるか？

2. この関数を使うユーザのためにドキュメント文字列（過不足ないかたちで）を作成せよ。

以下の練習問題は第 4 章の turtle モジュールを使う。

練習問題 5.5 以下の関数を読みそれがどのようなことをしているか明らかにせよ。そしてこれを実行してみよう（第 4 章の例を参照のこと）。

```
def draw(t, length, n):
    if n == 0:
        return
    angle = 50
    t.fd(length*n)
    t.lt(angle)
    draw(t, length, n-1)
    t.rt(2*angle)
    draw(t, length, n-1)
    t.lt(angle)
    t.bk(length*n)
```

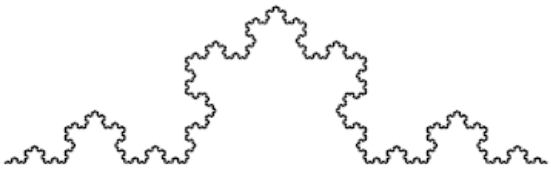


図 5.2: コッホ曲線.

練習問題 5.6 コッホ曲線は図 5.2 のようなフラクタル図形である。長さ x の線分からコッホ曲線を描くには以下のような手順が必要になる：

1. $x/3$ の長さでコッホ曲線を描く
2. 60 度左転回
3. $x/3$ の長さでコッホ曲線を描く
4. 120 度右転回
5. $x/3$ の長さでコッホ曲線を描く
6. 60 度左転回
7. $x/3$ の長さでコッホ曲線を描く

例外は x が 3 よい小さいときで、長さ x の直線を描くことでよい。

1. `turtle` と長さを仮引数とする関数 `koch` を作成せよ。この関数はこの与えられた長さでコッホ曲線を描く。

2. 関数 snowflake を三つのコッホ曲線から作成せよ。この関数は雪の微片の外周を表現する。
   解答例：https://thinkpython.com/code/koch.py.

3. コッホ曲線はいろんな具合に一般化できる。
   http://en.wikipedia.org/wiki/Koch_snowflake を参考にして自分の好みの図形を描くプログラムを作ってみよう。

第6章　結果を生む関数

これまで使ってきた数学関数のようなPythonの関数の多くは戻り値を返すものである。しかしこれまでに作成した関数は戻り値がないvoid型であった。それらはなにかをprintし、turtleが動きまわる動作をしたが、戻り値が何もなかった。

この章では結果を生む関数を書いてみることにする。

6.1　戻り値

このような関数を呼ぶと結果を返すので変数に代入するか、表式の一部として使う：

```
e = math.exp(1.0)
height = radius * math.sin(radians)
```

これまでに作成した関数は戻り値がないvoid型であった。それら戻り値が何もないが、厳密に言うとNoneという戻り値を持つ。

第一の例は関数areaである。この関数は半径の値を引数として円の面積の値を戻す関数である：

```
def area(radius):
    a = math.pi * radius**2
    return a
```

return文については既に触れたが、結果を生む関数では、return文には表式を付随する。この文は、「この関数から直ちに戻り、以下の表式を返り値として使え」という意味だ。この表式は任意で、以下のように書くこともできる：

```
def area(radius):
    return math.pi * radius**2
```

しかし、aのような**一時変数（temporary variables）**の導入はデバッグを容易にしてくれることがある。

ときとして、多重のreturn文の使用が有益なことがある。とくに条件文による分枝がある場合には、以下のようになる：

```
def absolute_value(x):
    if x < 0:
        return -x
    else:
        return x
```

これらの return 文は二者択一なので、return 文はどちらかが実行される。

関数で return 文が実行されると、関数は他の文を実行せず、直ちに関数の動作は終了する。このように return 文より後に置いた文や決して実行の流れが到達しないところにあるコードは**死コード（dead code）**と言われる。

結果を生む関数ではプログラムで想定される全ての可能な経路が必ず return 文に到達することを確認する必要がある：

```
def abusolute_value(x):
    if x < 0:
        return x
    if x > 0:
        return x
```

この関数は正しくない。x の値がたまたま 0 であると、この関数は return 文に到達できない。関数の実行は関数の終わりにある return 文で戻ることになる。この戻り値は**ノン（None）**であり、0 ではない。つまり

```
>>>print(absolute_value(0))
None
```

Python は組み込み関数として絶対値を計算する abs 関数を提供している。

練習問題として比較関数を作成せよ。つまり二つの引数 x と y を持ち、もしも x > y ならば戻り値は 1、x == y ならば、0、x < y ならば、-1 を返す関数である。

6.2　段階的な改良法

大きな関数を書き始めると、多分にデバッグに多くの時間を使うことになる。このような複雑なプログラムに取りかかると**段階的な改良法（incremental development）**と呼ばれている方法を取りたくなるだろう。この方法の目標は一時に追加・検証するコードを小刻みにし、膨大なデバッギングを避けることにある。

例をあげてみよう。与えられた二点 と からこの二点間の距離を求める問題だ。ピタゴラスの定理から

$$\text{distance} = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

第一ステップはこの関数 distance は Python ではどんな形になるかなと考察することである。換言すれば、入力（仮引数）は何で、出力（戻り値）は何か？

この例では入力は二つ点、つまり四つの数値であり、戻り値は計算された距離になるであろうから浮動小数点数の値である。

ここまで分かると関数の骨格が書ける：

```
def distance(x1, y1, x2, y2):
    return 0.0
```

この版では戻り値は常に 0 なので距離の計算はしていない。しかし、この関数は構文的には正しく、ちゃんと動く。もつと複雑に関数を仕上げる前に関数の検証としてこれは使える。

適当な引数を入れて検証する：

```
>>>distance(1, 2, 4, 6)
0.0
```

水平方向の距離が 3、垂直方向の距離が 4 の二点をわざわざ選んだ。これには理由があり二点間の距離は 5 となり、事前に答えが分かり関数の動作検証に都合がよいからだ。

この時点で関数 distance は構文的には問題ないことが確認できた。次ぎにくる作業は関数のボディを書くことである。これにはこの二点の x 方向の差、x2-x1、y 方向の差、y2-y1 を表示してみることだろう：

```
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    print('dx is',dx)
    print('dy is',dy)
    return 0.0
```

この関数を実行してみると、“dx is 3”、“dy is 4”と表示されるはずだ。

次ぎのステップは dx と dy の二乗の和を求めることになる：

```
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    dsquared = dx**2 + dy**2
    print('dsquared is: ',dquared)
    return 0.0
```

ここでまたこれを実行して出力を調べる。最後は平方根を計算する関数math.sqrtを使いその結果をreturn文で返す：

```
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    dsquared = dx**2 + dy**2
    result = math.sqrt(dsquared)
    return result
```

これがうまく動くことが確認できれば完了である。そうでなければ、変数resultをreturn文の前で表示してみるとよい。

この最終版の関数自体は何も表示しない。print文の挿入はデバッギングに有効だ。デバッグが終わったらこのprint文は削除する。このようなコードは**足場組み（scaffolding）**と呼ばれる。というのは、これらのprint文はプログラムを完成させることに有効だが、最終に生成されたものには含ませないからだ。

プログラミングを始めた最初のころは一度に追加するのは１～２行にしておくとよい。慣れてきたら、それより大きなまとまりを一度に書き、デバッグすることができるようになるはずだ。いずれにせよ、この段階的な改良法は時間の節約になる。

このプロセスの特徴を箇条書きにすると以下のようになる：

1. 実行可能な小さなプログラムから始めて、少しずつ段階的に変更を加える。いつの段階でもエラーが発生したら原因を突き止め修正しておく。
2. 一時変数を使い途中の値を保存し、それを必要に応じて表示、検証できるようにする。
3. 最終的なプログラムがうまく動くようになったら、足場の撤去やプログラムが読み難くならない範囲で、複数の文を一つの複合表式に置き換えなどの作業をする。

練習問題として直角三角形の二辺の長さを引数として受け取り、斜辺の長さを戻り値として返す関数hypotenuseを段階的な改良法を使って書け。各段階の状況を記録して置くこと。

6.3 合成関数

さて、これまでの議論で関数の中で別な関数を呼ぶことも可能だということを予想できたと思う。このような機能を**複合関数（composition）**という。

このような例として、円の中心の座標と円周上の座標の二点の座標を引数としてその円の面積を計算する関数を書いてみよう。

変数 xc、yc に円の中心の座標の値が保存され、xp、xp に円周上の座標の値が保存されているとしよう。第一にすべきはこの二点を使って円の半径（radius）を出すことである。これには関数 distance を使う：

```
radius = distance(xc, yc, xp, yp)
```

次はこの半径から面積を出す関数 area を使う。ここでは形式的にのみ書いて置く：

```
result = area(radius)
```

これらを一纏めにして一つの関数とする：

```
def circle_area(xc, yc, xp, yp):
    radius = distance(xc, yc, xp, yp)
    result = area(radius)
    return result
```

radius や result の一時変数はプログラムの開発途上やデバッグには有効であるが、最終的なプログラムでは関数を入れ子にして簡潔に書くこともできる：

```
def circle_area(xc, yc, xp, yp):
    return area(distance(xc, yc, xp, yp))
```

6.4 ブール代数関数

ブーリアン（真理値）を返す関数は関数内部で複雑な検証を隠す上で有効なものである。例を上げる：

```
def is_divisible(x, y)
    if x % y == 0:
        return True
    else:
        return False
```

ブール代数関数には二者択一のような関数名がよく付けられる。関数 is_divisible は x が y で割り切れるかどうかの結果を True や False として返す関数である。
実行例：

```
>>> is_divisible(6, 4)
False
>>> is_divisible(6, 3)
True
```

比較演算子の結果の値がブーリアンなので関数は以下のように書ける：

```
def is_divisible(x, y):
    return x % y == 0
```

ブール代数関数はよく条件文の中で使われる：

```
if is_divisible(x, y):
    print('x is divisible by y')
```

もしかしたら以下のように書くかもしれない：

```
if is_divisible(x, y)==0:
    print('x is divisible by y')
```

しかし、この比較は不必要である。

練習問題として以下のような関数 `is_between(x, y, z)` を書け。この関数は $x \leq y \leq z$ なら `True` を返し、これ以外ならば `False` を返す。

6.5　再帰関数の拡張

これまでの学習は Python の一部分であるが、これまでの学習で得たもので完全なプログラミング言語になっていることを確認することは興味深いことであろう。この完全という意味は計算したいと思われる全てのことがこの言語で書けるということである。あなたがこれまで作成した如何なるプログラムもこれまで学習した Python を使って書き換えることができる（多分、キーボード、マウス、ディスクといった装置を制御するコマンドは別途必要かもしれない)。

この主張の証明はアラン・チューリングによって初めて自明なことではない課題であることが証明された。かれは当時のコンピュータ科学者がそうであったように数学者であったが、コンピュータ科学者の嚆矢の一人である。従って、この主張はチューリング・テストと言われている。このチューリング定理のもっと完全で（従って厳密な）議論は “Introduction to the Theory of Computation” (Michael Sipser 著) を参照してほしい。

ここでは、これまで学習したツールを使って何ができるかの認識を得るために、再帰関数のいくつかを検討することにする。再帰的定義は循環的定義に似ている。それらの定義にはそこでは定義するべきものの参照を含んでいる。

循環的定義はあまり有用ではない。たとえばこうだ：

> vorpal: 何か vorpal であるものを記述する形容詞である。

こんな定義を辞書でみつけたらまごついてしまうかもしれない。

しかし、階乗関数が記号！で記述されているのを探し出したときには、その記述は以下のようなものだろう：

$$0! = 1$$
$$n! = n(n-1)!$$

この定義は 0 の階乗は 1、その他の任意の値、n の階乗は n-1 の階乗に n を乗じたものであることを意味する。だから 3! は 3 掛け 2! で、2! は 2 掛け 1! で、1! は 1 掛け 0! である。これを纏めると 3! は $3 \times 2 \times 1 \times 1$ で 6 となる。

何かを再帰的な定義で記述できたならば、それを Python でプログラムして評価することができる。

第一にすべきは、関数名とその仮引数を決めることである。今回の場合、その関数 factorial の仮引数は一つの整数である。つまり

```
def factorial(n):
```

となる。この引数がたまたま 0 であると、この関数の返すべき値は 1 であるので

```
def factorial(n):
    if n == 0:
        return 1
```

となる。さもないと、これが興味あるケースであるが、再帰呼び出しで n-1 の階乗を求め、それに n を乗ずる計算をすることになる。従って、プログラムは以下の様になる：

```
def factorial(n):
    if n == 0:
        return 1
else:
    recurse = factorial(n-1)
    result = n * recurse
    return result
```

このプログラムの実行の流れは 5.8 節で登場した countdown と似たものになる。n の値を 3 にしてこの関数を呼んだときの流れを以下に示す：

3 は 0 でないので、第二の分枝をとり、n-1 の階乗を計算する。
　2 は 0 でないので、第二の分枝をとり、n-1 の階乗を計算する。
　　1 は 0 でないので、第二の分枝をとり、n-1 の階乗を計算する。
　　0 は 0 であるので、第一の分枝をとり、単に 1 を戻り値として返す。
　　戻り値 1 を recurse に代入、これと n（即ち 1）とを乗じた値 1 を返す。
　戻り値 1 を recurse に代入、これと n（即ち 2）とを乗じた値 2 を返す。
戻り値 2 を recurse に代入、これと n（即ち 3）とを乗じた値 6 を返す。

これでこの関数を最初に呼んだ関数に戻る。

図 6.1 にこの関数呼び出しのスタック図を示した。戻り値はスタック図を繋ぐ経路に添付した。各フレームでは戻り値はそのフレームの result が持っている値である。最後のフレームでは流れは if 文の第一分枝になるので、ローカル変数 recures と result は存在しない。

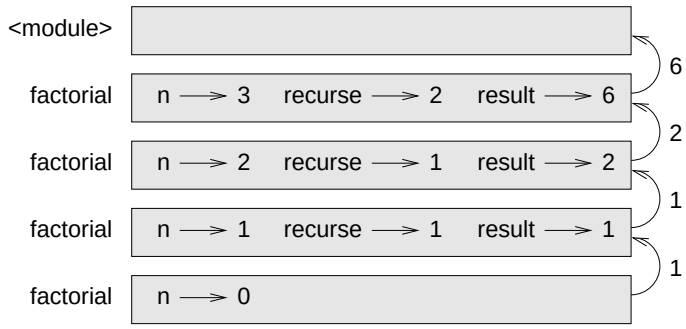


図 6.1: スタック図.

6.6　信用して跳び越える

実行の流れを追うことはプログラムを読むひとつの方法であるが、直ぐに複雑になる。別な方法は私が「信用して跳び越える」と呼ぶ方法である。関数に出会ったらその関数内の流れを追跡するかわりに、その関数は正常に動作し、正しい戻り値を返すとみなし、この関数内の実行の流れをスキップする。

このようなスキップは既に組み込み関数では経験している。math.exp や math.cos の関数ではこれらの関数内の流れには頓着しない。われわれはこれらの関数は優秀なプログラマーが作成したもので、正常に動作するとみなしている。

同じことが自作関数でもいえる。例えば、6.4 節で作成した divisible 関数だ。コードの吟味や検証で正常に動作することが確証できたら、以後はこのボディの実行の流れには頓着しない。

同じことが再帰関数でもいえる。再帰関数の呼び出しに出会ったら、その関数内の流れを追わず、その再帰的な呼び出しは正常に動作する（正しい戻り値を返す）とみなす。そして、「もし n-1 の階乗が計算できたら、それで n の階乗が計算できるか？」と自問してみる。この場合、n-1 の階乗に n を乗ずれば答えになることは明らかである。

勿論、まだ書き上げてもいない関数が正常に動作するとみなすことは少々奇妙であるが、これは「信用して跳び越える」という言葉の所以である。

6.7 もう１つの例題

関数 factorial に引き続き数学的な再帰関数として引き合いに出される fibonacci 関数である。この関数の定義は以下のようである：

$$\begin{aligned}\text{fibonacci}(0) &= 0\\ \text{fibonacci}(1) &= 1\\ \text{fibonacci}(n) &= \text{fibonacci}(n-1) + \text{fibonacci}(n-2)\end{aligned}$$

（http://en.wikipedia.org/wiki/Fibonacci_number も参照のこと）

これを Python に翻訳するとこんな風になる：

```
def fibonacci(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

この関数の実行の流れをまともに追跡すると、n が小さい場合でさえ頭が爆発してしまう。しかし、「信用して跳び越える」方法に従って、二つ再帰呼び出しは正常に実行されると仮定すると、第三分枝の計算で正しい答えが戻ることは自明となる。

6.8　型の検証

関数 factorial を引数の値に 1.5 を入れて呼ぶとどうなるだろうか？

```
>>>factorial(1.5)
RuntimeError: maximum recursion depth exceeded
```

この実行時のエラーは制限なしの再帰に起因するようにみえる。どうしてそんなことが起きるのか？この関数には基底ケースが考慮されているではないか。引数 n が整数でないと、この基底ケースをすり抜けてしまい、制限なしの再帰になるのだ。

上の引数では、最初の再帰で引数 n の値は 0.5 になり、二回目は-0.5 となる。だから基底ケースである n=0 は実現しない。

対策としては二つの方向がある。一つは factorial を拡張して引数は浮動小数点数にも対応できるようにする。二つ目は引数型を検証するように書き改めることである。第一はガンマ関数への拡張であり、この本の範囲を超えるので、第二番目の方法を紹介する。

引数の型を調べるために組み込み関数 isinstance を使う。これと同時に引数が正の数であることも調べることにする。プログラムは以下のようになる：

```
def factorial(n):
    if not isinstance(n, int):
        print('Factorial is only defined for integers.')
        return None
    elif n < 0:
        print('Factorial is not defined for negative integers.')
        return None
    elif n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

第一番目の基底ケースは非整数対策であり、第二番目の基底ケースは負整数対策である。どちらも何らかの問題があるというメッセージを吐き出し、戻り値は None である：

```
>>> print(factorial('fred'))
Factorial is only defined for integers.
None
```

```
>>> print(factorial(-2))
Factorial is not defined for negative integers.
None
```

この二つの検証を通過したら、引数 n は正か 0 の整数である。この状態になれば、この再帰呼び出しは有限回で停止することが証明できる。

このプログラムはときとして**保護回路（guardian）**と呼ばれる手法を例示したことになる。最初の二分枝は保護回路で実行時エラーを引き起こすだろうと思われる値からコードを保護する役目をしている。

11.4 節ではエラーの表示の替わりに例外処理を実行するもっと融通の効く手法に出会うはずだ。

6.9 デバッギング

大きなプログラムをそれより小さい関数に分割することはそれ自体でデバッグの極自然な確認の単位になる。もし関数が正常に動作しないとなると、考えなければならないことは以下の三点である：

- 関数の事前条件に違反するような引数の問題。
- 関数の事後条件に違反する問題。
- 関数の戻り値やそれの使われ方の問題。

最初の可能性を除外するには関数の先頭に print 文を挿入し、仮引数の値（そして型も）を表示してみるとよい。または、関数のコード自体にこの機能を持たせるとよい。仮引数に問題がないとすると、次は関数の return 文の前で戻り値を表示してみる。もし可能ならば、この戻り値を紙で計算してみる。簡単な引数の値であれば、その関数の戻り値を手計算することは容易い（6.2 節をみよ）。

関数の最初と最後に print 文を追加・挿入することは実行の流れを可視化する上で有用である。以下は factorial に print 文を追加したものである：

```
def factorial(n):
    space = ' ' * (4 * n)
    print(space, 'factorial', n)
    if n == 0:
        print(space, 'returning 1')
        return 1
    else:
```

```
    recurse = factorial(n-1)
    result = n * recurse
    print(space, 'returning', result)
    return result
```

変数 space は出力のインデントを制御する空白文字列である。factorial(5) の結果を示す：

```
                    factorial 5
                factorial 4
            factorial 3
        factorial 2
    factorial 1
factorial 0
returning 1
    returning 1
        returning 2
            returning 6
                returning 24
                    returning 120
```

実行の流れについて混乱してしまったときは、このような出力は有用になる。このような効果的な足場を据えることには時間が掛かるが、ちょっとした足場もデバッギングの時間を節約することができる。

6.10　語句

一時変数（temporary variables）： 複雑な計算の際に中間結果を保存する目的で使われる変数。

死コード（dead code）： 如何なるばあいでも到達しないプログラムの部分。return 文の後にあることが多い。

段階的な改良法（incremental development）： 一時に少量の追加とテストを行いつつプログラムを開発する手法。

足場組み（scaffolding）： 最終版では削除されるがプログラム開発の途上で使われるコード。

保護回路（guardian）： エラーを引き起こす状況を検出及び回避するために条件文を使うプログラムの部分。

6.11 練習問題

練習問題 6.1 以下のようなプログラムのスタック図を示せ。さらに実行したとき、如何なる表示が出るか答えよ。

```
def b(z):
    prod = a(z, z)
    print(z, prod)
    return prod

def a(x, y):
    x = x + 1
    return x * y

def c(x, y, z):
    total = x + y + z
    square = b(total)**2
    return square

x = 1
y = x + 1
print(c(x, y+3, x+y))
```

練習問題 6.2 Ackermann 関数は以下のように定義される：

$$A(m,n) = \left\{ \begin{array}{ll} n+1 & \text{if } m = 0 \\ A(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1, A(m,n-1)) & \text{if } m > 0 \text{ and } n > 0. \end{array} \right\}$$

（この関数については、 http://en.wikipedia.org/wiki/Ackermann_function. を参照のこと。）Ackermann 関数の値を計算する関数 ack を作成せよ。そして、ack(3, 4) の値 125 を調べよ。大きな m や n では何が起こるか調べよ。
解答例：https://thinkpython.com/code/ackermann.py

練習問題 6.3 回文とは、例えば “noon” や “redivider” のように後から読んでも、前から読んでも同じ綴りを持つ言葉である。再帰的には、もしある言葉の最初と最後の文字が同じで、残った中間の文字列が回文であるなら、この言葉は回文である。

次ぎのような文字列処理の関数はその文字列の最初の文字、最後の文字、中間の文字列を戻り値とする：

```
def first(word):
    return word[0]

def last(word):
    return word[-1]

def middle(word):
    return word[1:-1]
```

これらの関数の詳細は第 8 章で議論することになる。

- これらの関数を palindrome.py と名付けたファイルとして生成せよ。関数 middle に二文字の言葉を入れると戻り値はどうなるか？一文字ではどうか？一文字も含まない文字列、つまり” ではどうか？
- 引数として文字列を受け取る関数 is_ palindrome.py を作れ。この関数の戻り値はこの文字列が回文であると True、さもないと False である。文字列の長さは関数 len で調べられることに注意。

解答例：https://thinkpython.com/code/palindrome_soln.py.

練習問題 6.4 一つの数 a は、もしもこの数が b で割り切れ、しかも a/b が b のべき乗であるとき、b のべき乗である。

二つの仮引数 a、b を持つ関数 is_power を作成せよ。この関数は a が b のべき乗であるときに True を返す。
注）基底ケースについて考慮せよ。

練習問題 6.5 二つ整数 a、b の最大公約数（GCD）はこの二つの整数を除算して余りがない整数のなかで最も大きな整数である。

最大公約数を探す一つの方法はユーグリッド互除法である。この方法は a を b で割ったときの余りを r とすると、$gcd(a, b) = gcd(b, r)$ であることからきている。基底ケースは $gcd(a, 0) = a$ を使う。

二つの整数 a、b を仮引数とする関数 gcd を作成せよ。戻り値はこの二つの数の最大公約数である。
この練習問題の出典は、Abelson and Sussman,“Structure and Interpretation of Computer Programs”である。

第7章　繰り返し処理

この章では繰り返し処理について学ぶ。この種の繰り返し処理のお陰で文の集合を繰り返し実行できる。既に 5.8 節で再帰処理による繰り返しを見てきた。また 4.2 節では `for` ループによるによる処理を見てきた。この章では `while` 文によるもう一つの繰り返しをみよう。しかしその前に代入について考察したい。

7.1　多重代入

これまで経験したことから分かるが、同じ変数に複数回の代入をすることは合法である。この新しい代入で変数はこの新しい値を参照することになる（古い値の参照は停止する）。

```
>>> x = 5
>>> x
5
>>> x = 7
>>> x
7
```

最初の `x` を表示するとその値は `5` であったが、二回目ではその値は `7` となった。図 7.1 には**多重代入（multiple assignment）**のスタック図の様子を示した。

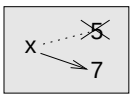


図 7.1: 状態図.

多重代入で重要なことは代入処理と等値の違いを明確にしておくことだ。Python では等号（=）は代入に使われるので、`a=b` のような文を等値と解釈しがちである。これは間違いだ。第一に、等値は左辺と右辺が対称だが、代入は対称でない。例えば、数学的では、$a = 7$ と $7 = a$ は同じ意味だが、Python では違う。更に、数

学では等値は常に真が偽の値を持つ。だから、もし $a = b$ なら a と b とは常に等値である。Python の代入文でも二つの変数の値を同じにすることができるが、常に同じ値を持つとは限らない。

```
>>> a = 5
>>> b = a #a と b とは同じ値を持つ
>>> a = 3 #a と b とは最早同じ値ではない
>>> b
5
```

第三行目で a の値が変わるが、b の値は変わらない。だから、この二つは最早等しくはない。

多重代入は頻繁に使うが、注意が必要だ。変数の値が頻繁に変わるのでプログラムが読み辛いものになりがちでデバッグが難しくなる。

7.2　変数更新

最も頻繁に遭遇する多重代入は**変数更新（update）**である。新しい値が古い値に依存するかたちをとる：

```
x = x + 1
```

この意味は、「x の現在の値に 1 を加えてものを x の新しい値として更新せよ」。

存在しない変数の値の更新は、Python は代入の前に右辺を評価しなければならないので、エラーになる。

```
>>> x = x + 1
NameError: name 'x' is not defined
```

変数の更新をする前に、変数は初期化（initialize）、一般に簡単な代入だが、しなければならない。

```
>>> x = 0
>>> x = x + 1
```

変数の更新で 1 を加えて更新することを**インクリメント（increment）**、1 を減じて更新することを**デクリメント（decrement）**と特別な名称が付けられている。

7.3 while文

コンピュータは繰り返しを自動的に実行するタスクに頻繁に使われる。同じか類似の処理を間違いなしで繰り返し実行するタスクはヒトでは苦手だが、コンピュータは得意としている。

これまで countdown と print_n というプログラムをみてきた。これらのプログラムは再帰を使っているが、**繰り返し処理（iteration）**の例だ。繰り返し処理は頻繁に使うので、Python は言語仕様として豊富な繰り返し機能を提供している。その一つが 4.2 節で紹介した for 文である。これは後に議論する。

もう一つが while 文である。countdown を while で書いたものが以下である：

```
def countdown(n):
    while n > 0:
        print n
        n = n - 1
    print('Blastoff!')
```

この while 文は英文を読むように読める。つまり、「n が 0 より大きい間、n の値を表示し、n の値を 1 減じなさい。n が 0 になったとき単語 Blastoff! を表示しなさい」。

もう少し厳密に while 文の実行の流れを述べてみる：

- 条件を評価し、真か偽を得る。
- 偽ならば、while 文のブロックを抜け、次ぎの文から実行を続ける。
- 真ならば、whike 文のボディを実行して、最初に戻る。

この種の実行の流れを**ループ（loop）**と称する。

ループの本体の中で一つか二つの変数を更新し、ループの終了が保証されているようにしなければならない。さもないとそのループは**無限ループ（infinte loop）**になる。コンピュータ科学者にとって、「よく泡立て、よく洗い、これを繰り返す」というシャンプーの使用説明はひやかしの尽きない源である。

関数 countdown のばあいは n の値が有限で、ループを繰り返す度に n が減少するので、いずれは 0 になることが証明される。

一般的にはこれは容易くない：

```
def sequence(n)
    while n != 1:
        print(n)
```

```
    if n%2 == 0:  #nは偶数
        n = n/2
    else:          #nは奇数
        n = n*3+1
```

このループの条件は n != 1、だからループは n が 1 に達し、条件が偽になるまで続く。

毎回 n の値が表示され、n が奇数か偶数かが検定される。もし偶数であるとすると、n は 2 で割られる。奇数であると、n の値は n*3+1 に置き換わる。例として 3 を引数としてこの関数が呼ばれると、表示は 3,10,5,16,8,4,2,1 となる。

n はときとして増加、またときとして減少するので、このプログラムで n が 1 に達し、プログラムが終了することを証明するのは自明ではない。例えば、n が 2 のべき乗で出発すると n は 1 に達すまで偶数のままである。前の例は 16 以降の系列はこのケースである。

如何なる正整数 n に対してもこのループが停止することを証明することは難しい問題だ。これまで、これをできた人がいないし、できないということを証明できた人もいない。

（参考：http://en.wikipedia.org/wiki/Collatz_conjecture）

練習問題として、関数 print_n を再帰でなく、繰り返し処理をするプログラムに書き換えよ。

7.4 ブレイク

ときとしてループの本体を実行している途中まで何時ループから抜け出るべきかを知ることができない場合がある。このときは break 文でループを脱出する。

例えば、done とタイプされるまでユーザの入力を受け付けることを考えてみよう。プログラムは以下のようになる：

```
while True:
    line = input('> ')
    if line == 'done':
        break
    print(line)

print('Done!')
```

ループの条件は常に真であるから、このループは break に遭遇するまで続く。その繰り返し毎にユーザ入力促進が一つのアングルブラケットで示される。ユーザ

が done と入力すると、break 文でループを抜ける。そうでなければ、入力したものが何であってもそれを表示してループの先頭に戻る。実行例：

```
> not done
not done
> done
Done!
```

これは while 文の使い方としては極普通である。ループの検定はどこでもできるし（ループの先頭でなくても）、終了条件を否定的（これが起きるまで続く）でなく、確定的（これを起きたとき終了）にも表現できる。

7.5 平方根

ループ処理は適当な答えから出発した数値計算でその答えを徐々に改良して行くときによく使われる。

例としてある数の平方根を求める手法としてニュートン法がある。ある数 a の平方根を知りたいとしよう。それに近い数 x から出発して、以下の式を使うとさらによい近似値、y が得られる：

$$y = \frac{x + a/x}{2}$$

例として、a が 4 で、x が 3 のばあいと計算してみる：

```
>>> a = 4
>>> x = 3
>>> y = (x + a/x) / 2
>>> y
2.16666666667
```

この値はより真の値（$\sqrt{4} = 2$ ）に近い。この得られた新しい値を使ってプロセスを繰り返すともっと近い値が得られる：

```
>>> x = y
>>> y = (x + a/x) / 2
>>> y
2.00641025641
```

これを数回繰り返すと推定値はほぼ正解になる：

```
>>> x = y
>>> y = (x + a/x) / 2
>>> y
2.00001024003
>>> x = y
>>> y = (x + a/x) / 2
>>> y
2.00000000003
```

一般に真の値を得るには何回の繰り返しが必要なのかは事前には分からない。しかし、推定値の表示が変動しなくなるところで、真の値の達したと考えてよい：

```
>>> x = y
>>> y = (x + a/x) / 2
>>> y
2.0
>>> x = y
>>> y = (x + a/x) / 2
>>> y
2.0
```

つまり、y == x で繰り返しを終了する。纏めると、初期推定値 x から出発してそれが変化しなくなるまで繰り返すことにする：

```
while True:
    print(x)
    y = (x + a/x) / 2
    if y == x:
        break
    x = y
```

この考えで大部分の a についてはよさそうだ、しかし浮動小数点数の等値に関連する問題を含んでいる。浮動小数点数は近似的に正しいのみである。例えば、1/3 のような有理数、$\sqrt{2}$ のような無理数は浮動小数点数では正確には表現できない。

そこで、x と y が正確に一致するという替わりに、この 2 つの値の差の絶対値を計算し、この差の絶対値に制限を置くことにする。ある値の絶対値は組み込み関数 abs を使う。

```
    if abs(y-x) < epsilon
        break
```

ここで epsilon は 0.0000001 のような数で、二つの値 x,y がどの程度近ければよいかを決める。

7.6 アルゴリズム

ニュートン法は**アルゴリズム（algorithms）**の一例である。それは機械的な方法である種の問題を解く方法のことである（いまの場合は、平方根を求める）。

アルゴリズムとは何かを定義することはそんなに易しくない。アルゴリズムでないものから議論したら理解が進むかもしれない。一桁のかけ算を学んだ過程で、九九の表を暗記することになるはずだ。この知識で特別な値を持った 100 個の場合のかけ算の解法が得られたことになる。しかし、このような知識はアルゴリズム的でない。

もしあなたが「怠け者」で、このような暗記をなるべく減らしたいとして頭を使ったとしよう。例えば、9 桁の九九は n を他の整数とすると、かけ算の答えは十の位は n-1 で、一の位は 10-n であるとすればよいと発見したとしよう。この発見は任意の整数 n に対して成り立つ操作だ。このようなものがアルゴリズムである。

同様に、加算での繰り上げ、減算の繰り下げの操作がアルゴリズムである。アルゴリズムの一つの特徴はその操作に知性を必要としないことだ。それらは最初から最後まで簡単な規則に沿って進められる機械的なプロセスである。

アルゴリズムの実行は退屈なものであるが、アルゴリズムを設計することは興味深く、知性的な挑戦であり、プログラミングの中枢である。

ヒトが何の苦もなく無意識にやっているいくつかのことをアルゴリズムとして定式化することは最も難しい。自然言語はその卑近な例である。ヒトはそれをやっているが、これまでどのようにしてヒトがそれをやっているのか説明できないでいるし、少なくともアルゴリズム的に定式化できないでいる。

7.7 デバッギング

大きなプログラムを書き始めると、デバッギングに掛ける時間が多くなる。より大きなコードはより多くのエラーの機会があり、より多くの場所にバグが潜んでいることを意味する。

時間を短縮する一つの方法は、「二分割デバッグ法」とでもいう方法だ。例えば、100 行あるコードを一時にチェックしようとすると 100 行をみる時間が必要になる。そこで、プログラムを半分に分割する。手頃な半分あたりでよい。そこに print 文（またはおなじ効果が得られるものでよい）を追加し、プログラムを実行して

みる。このチェックポイントまでで間違いが分かれば、前半に問題があることになるし、間違いがなければ、問題は後半にあることになる。

このように毎回二分割して行く。この操作を数回（これは 100 よりは少ない）すれば、理論的にはコードの１，２行に問題を絞り込むことができる。

実際には、「プログラムの半分」を見つけることが難しいことがあるが、行数を数えてまん中を見つけることは意味がない。エラーが潜んでいそうな場所や、値をチェックするのに都合がよい場所の目星をつける。そして、エラーがその前後で起きているらしいところをチェックポイントとして選べばよい。

7.8　語句

多重代入（multiple assignment）： プログラムの実行の過程で一つの変数に一回以上の代入をすること。

変数更新（update）： 変数の古い値に依存するかたちでその変数の値を更新する。

初期化（initialize）： 変数更新をするつもりの変数に特定の値（初期値）を与える。

インクリメント（increment）： 変数の値を 1 だけ増加させる変数更新。

デクリメント（decrement）： 変数の値を 1 だけ減ずる変数更新。

繰り返し処理（iteration）： 再帰呼び出しやループを使ってプログラムの一部を繰り返し実行する。

無限ループ（infinte loop）： 終了条件が満たされることのないループ。

アルゴリズム（algorithms）： 典型的な諸問題を解く一般的な手続き。

7.9　練習問題

練習問題 7.1 7.5 節の繰り返し処理をコピーしてそれを `mysqrt` と名前の関数としてカプセル化せよ。この関数は `a` を引数として持ち、`x` の適当な値を設定し、戻り値として
a の平方根の近似値を戻す。

この関数をテストするために `test_square_root` というプログラムを作り、組み込み関数 `math.sqrt` の結果と比較する以下のようなテーブルを作れ。

```
a   mysqrt(a)      math.sqrt(a)  diff
-   ---------      ------------  ----
1.0 1.0            1.0           0.0
2.0 1.41421356237  1.41421356237 2.22044604925e-16
3.0 1.73205080757  1.73205080757 0.0
4.0 2.0            2.0           0.0
5.0 2.2360679775   2.2360679775  0.0
6.0 2.44948974278  2.44948974278 0.0
7.0 2.64575131106  2.64575131106 0.0
8.0 2.82842712475  2.82842712475 4.4408920985e-16
9.0 3.0            3.0           0.0
```

第一列は数値 a、第二列は 7.5 節で検討した関数で計算した平方根の値、第三列は math.sqrt で計算した値、第四列は二つの計算結果の差の絶対値である。

練習問題 7.2 組み込み関数 eval は文字列を引数として受け取り、Python インタプリタでその文字列を評価する関数である。例えば

```
>>> eval('1 + 2 * 3')
7
>>> import math
>>> eval('math.sqrt(5)')
2.2360679774997898
>>> eval('type(math.pi)')
<class 'float'>
```

繰り返しユーザに入力請求をして文字列を入力、それを eval で評価して結果を表示するプルグラム eval_loop を作成せよ。この関数はユーザが'done' を入力したらループを抜け、最後に評価した値を戻り値とする。

練習問題 7.3 数学者 Srinivasa Ramanujan は以下の公式が $1/\pi$ の近似値を生成することを発見した：

$$\frac{1}{\pi} = \frac{2\sqrt{2}}{9801} \sum_{k=0}^{\infty} \frac{(4k)!(1103 + 26390k)}{(k!)^4 396^{4k}}$$

この公式を使い関数 estimate_pi を作成せよ。数列の項の和を求めるところに while ループを使い、項が充分に小さくなったところ（10^{-15}）でそのループを抜ける。結果を math.pi と比較せよ。
解答例：https://thinkpython.com/code/pi.py.

第8章　文字列

文字列は整数、浮動小数点数、ブーリアンと異なっている。文字列は**配列（sequence）**である。その意味は他の値を集めた順序だった集合である。この章では文字列を作っている個々の文字にアクセスする方法についてみる。そして文字列に付随するメソッドの幾つかについて学ぶ。

8.1　文字列は文字の配列

文字列は配列である。角括弧で文字列の一つの文字にアクセスすることができる。

```
>>> fruit = 'banana'
>>> letter = fruit[1]
```

第二の命令では文字列 fruit から一文字を選び、それを letter に代入している。
角括弧の中の表式は**インデックス（index）**と呼ばれている。このインデックスはどの文字を選びたいかを示すものである。
しかし、結果はあなたが期待したものでないかもしれない：

```
>>> letter
a
```

多くの人々にとっては、'banana' の最初の文字は b であって a ではない。しかし、コンピュータ科学者にとっては、インデックスは文字列の最初の文字からのオフセット（片寄り）であり、最初の文字のオフセットは 0 である。

```
>>> letter = fruit[0]
>>> letter
b
```

だから、b は 0 番目の文字であり、a は 1 番目の文字、n は 2 番目の文字である。
このインデックスの表式は、変数や演算子を含めて任意である：

```
>>> i = 1
>>> fruit[i]
'a'
>>> fruit[i+1]
'n'
```

しかしインデックスの値は整数でなければならない。さもないとエラーになる：

```
>>> letter = fruit[1.5]
TypeError: string indices must be integers, not float
```

8.2 len

組み込み関数 len は文字列の長さを返す関数である。

```
>>> fruit = 'banana'
>>> len(fruit)
6
```

文字列の最後の文字を得ようとしたとき、以下のようにするかもしれない：

```
>>> length = len(fruit)
>>> last = fruit[length]
IndexError: string index out of range
```

このエラーは文字列'banana' にはインデックスの値が 6 の文字がないことを意味している。文字は 0 番から数えたので最後の文字は 5 番となる。だから最後の文字は length から 1 減じたものになる：

```
>>> last = fruit[length-1]
>>> last
'a'
```

別な方法としては負のインデックスを用いるものだ。文字列の最後の文字から逆向きに数える。だから、fruit[-1] が最後の文字を示し、fruit[-2] が後から二番目の文字となる。

8.3 for ループによる横断処理

多くの文字列処理では一文字毎の扱いが沢山現れる。文字列の初めから一文字を抽出し、何かの処理を行い、これを文字列の最後まで続けるといったことがよくある。このような処理のパターンを**横断処理（traversal）**と呼ぶ。これを実現する一つの方法は while ループを使うものである：

```
index = 0
while index < len(fruit):
    letter = fruit[index]
    print(letter)
    index = index + 1
```

このループ横断処理では文字列の一文字毎に一行表示される。ループの継続条件は index < len(fruit) であるので、この文字列の最後に達したときにこの条件は偽になる。練習問題として、文字列を引数として受け取り逆順で文字を表示する関数を作れ。

別な方法は for ループで横断処理を行うものだ：

```
for char in fruit:
    print(char)
```

このループを回る毎に文字列の中の次ぎの文字が変数 char にコピーされる。そしてこのループは文字列の文字が尽きるまで続く。

以下の例は文字の連結と for ループでアルファベット順の単語を生成するものである。Robert McClosky の著書 “Make Way for Duckling” によれ、アヒルの鳴き声から連想される名前は Jack、Kack、Lack、Mack、Nack、Ouack、Pack、Quack だそうである。以下のループはこれらの名前の順次出力する：

```
prefixes = 'JKLMNOPQ'
sufix = 'ack'

for letter in prefixes:
    print(letter + suffix)
```

結果の出力は

```
Jack
Kack
Lack
Mack
```

```
Nack
Oack
Pack
Qack
```

となる。勿論これは正確に同じではない。Oack と Qack とはスペルが違っている。練習問題として、プログラムを修正してこの違いが出ないようにせよ。

8.4　文字列のスライス

文字の一部は**スライス（slice）**と呼ばれる。スライスを選択する方法は一文字選択と類似している：

```
>>> a= 'Monty Python'
>>> print a[0:5]
Monty
```

演算子 [n:m] で文字列の n 番目の文字から m 番目（n 番目は含めて、m 番目は除外して）までの文字列を返す。この振る舞いは直感的でないが、図 8.1 で示すようにインデックスは文字と文字の間にあるとすると理解できるかもしれない。　最初

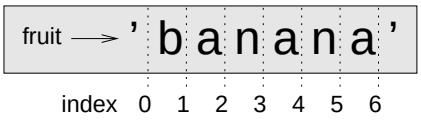


図 8.1: スライス・インデックス

のインデックス（コロンの前）を省略すると、スライスは先頭の文字からとなる。また二番目のインデックスを省略すると文字列の最後までとなる。

```
>>> fruit[:3]
'ban'
>>> fruit[3:]
'ana'
```

最初のインデックスが二番目より等しいか大きいと**空文字列（empty string）**を返す。これは二つの引用符で表現される：

```
>>> fruit = 'banana'
>>> fruit[3:3]
''
```

空文字列は文字を一つも含まないので文字列の長さは 0 であるが、他の性質は普通の文字列と同じである。

同様な例として fruit[:] は何を意味するだろうか？試して確認せよ。

8.5 文字列は変更不可

文字列の一部を変更しようとして以下のような代入を試みたとする：

```
>>> greeting = 'Hello World!'
>>> greeting[0] = 'J'
TypeError: 'str' object does not support item assignment
```

ここで「オブジェクト」は文字列、「アイテム」は文字列の中の代入しようとした文字である。この段階ではオブジェクト（object）とは値と同じものであるとしておく。後にこの定義は再吟味したい（10.10 節）。

ここでのエラーの理由は文字列は**変更不可（immutable）**であることから来ている。一旦作成した文字列は変更してはいけない。上の例で最良の解決は新しい文字列を生成することだ：

```
>>> greeting = 'Hello, world!'
>>> new_greeting = 'J' + greeting[1:]
>>> new_greeting
'Jello, world!'
```

この例では新しい文字と greeting のスライスしたものが連結されている。これでは元々の greeting には何も変更もない。

8.6 探索

次ぎのプログラムは何をしているのだろうか？

```
def find(word, letter):
    index = 0
    while index < len(word):
        if word[index] == letter:
            return index
        index = index + 1
    return -1
```

ある意味で、find は [] 演算子と逆の効果を持つ。インデックスに従って対応する文字を取り出す替わりに文字を受け取り、その文字を有するインデックスを返すわけである。文字が存在しないと-1 を返す。

また、この例がループの途中で return 文があるかたちの最初のものである。もし word[index] == letter が成立するとループが中断され return 文が実行される。問題の文字が文字列に存在しないとプログラムは通常のように戻り値-1 で関数を終了する。

配列を横断的に検査し、探しているものが見つかったら return が実行されるといった計算パターンは**探索（search）**と呼ばれている。

練習問題として、関数 find を第三の仮引数を持つように修正せよ。この第三仮引数は探索を始めるインデックスの値である。

8.7　ループ処理とカウンタ変数

以下の関数はある文字が文字列の中に何回現れるか調べるプログラムである：

```
word = 'banana'
count = 0
for letter in word:
    if letter == 'a':
        count = count + 1
print(count)
```

この例は**カウンタ（counter）**と呼ばれるプログラムパターンを例示したことになる。変数 count は 0 で初期化され、文字が発見される毎にインクリメントされる。ループが終了すると結果、つまり文字'a' が出現する回数が表示される。

練習問題として、上のプログラムをカプセル化して関数 count を作成せよ。この関数は仮引数として文字列と探索文字を持つものとする。それが済んだら関数 count を前節の第三の仮引数を持つ find 関数を使うように修正せよ。

8.8　文字列メソッド

文字列は各種の演算を実行するためのメソッドを提供している。メソッドは関数と似ている、つまり引数を受け取り、戻り値を返す。しかし、構文が異なっている。例えば、メソッド upper は文字列を受け取り、全てを大文字にした文字列を返す。

通常の関数の構文、upper(word) の替わりにメソッドの構文 word.upper() を使う。

```
>>> word = 'banana'
>>> new_word = word.upper()
>>> new_word
BANANA
```

ドット記法の形式はメッソド upper とそのメッソドを適用する文字列、word を特定している。空の丸括弧はこのメソッドが引数を何も取らないことを示している。

メソッドの呼び出しは**発動（invocation）**と呼ばれる。今の場合は word に対して upper を発動させたと言えばよい。

明らかになるように、文字列に対する find というメソッドは関数として書いた find と極めて類似した処理をするものである：

```
>>> word = 'banana'
>>> index = word.find('a')
>>> index
1
```

この例では word に対して find を探索する文字を引数として渡して発動している。

メソッド find は実際にはもっと複雑な処理ができる。引数は一文字でなく、文字列で構わない。

```
>>> word.find('na')
2
```

また、第二の引数を付けることも可能で、この引数は探索を開始するインデックスを意味している：

```
>>> word.find('na', 3)
4
```

これは**選択的な引数（optional argument）**の一例である。更に、第三の引数も取れる。これは探索を止めるインデックスを意味している：

```
>>> name='bob'
>>> name.find('b', 1, 2)
-1
```

この探索は失敗に終わっている。なぜなら、探索区間の 1 から 2 まで（2 は含まれない）の間に文字'b' はないからである。

8.9　in 演算子

英単語 in の表式で表現される演算子は二つの文字列を比較し、最初の文字列が第二の文字列に含まれているときには True を返すブール代数演算子である：

```
>>> 'a' in 'banana'
True
>>> 'seed' in 'banana'
False
```

この演算子の例として、二つの文字列に共通に含まれている文字を表示する関数を示す：

```
def in_both(word1, word2):
    for letter in word1:
        if letter in word2:
            print(letter)
```

変数の名前をうまく取っておくと、Python のプログラムは英文を読むように読める。この for ループは「第一の文字列になかにある各文字が第二の文字列の中にあるときにはその文字を print する」と読める。この関数を'apples' と'oranges' に適用すると以下のようになる：

```
>>>in_both('apples', 'oranges')
a
e
s
```

8.10　文字列の比較

比較演算子は文字列にも適用できる。二つの文字列が一致しているかどうかを調べるには

```
if word == 'banana':
    print('All right, bananas.')
```

他の比較演算子もアルファベット順に単語を並べるのに有効だ：

```
if word < 'banana':
    print('Your word, ' + word + ', comes before banana.')
elif word > 'banana':
    print('Your word, ' + word + ', comes after banana.')
else:
    print('All right, bananas.')
```

Python は大文字、小文字の扱いを普通人々がするようには処理しない。大文字で始まる全ての単語は小文字で始まる単語の前にくる。Pineapple と banana を比較すると以下のよな表示になる：

```
Your word, Pineapple, comes before banana.
```

これに対処する一般的な方法は文字列を標準化された形、例えば全てを小文字に変換した後に比較することである。手榴弾（Pineapple）で武装した人に対処しなければならないときのためにこのことを記憶しておくとよい。

8.11 デバッギング

配列の横断的な処理をするとき、その処理の初めと終わりを正しく選ぶことには注意が必要だ。以下は二つ単語を調べ、一つが他の逆順に並べたものであるときには `True` を戻す関数の一例である。しかし、二つのエラーを含んでいる：

```
def is_reverse(word1, word2):
    if len(word1) != len(word2):
        return False

    i = 0
    j = len(word2)

    while j > 0:
        if word1[i] != word2[j]:
            return False
        i = i + 1
        j = j - 1

    return True
```

最初の if 文は二つの単語の長さが等しいかどうか調べている。そうでなければ False を戻り値として関数は終了する。従ってこれ以降は二つの単語の長さは同じとして処理できる。これは 6.8 節で議論した保護回路の一例である。

変数 i、j はインデックスである。i は word1 を前方、j は word2 を後方に横断的にサーベイする。二つの文字が同じでないと、その場で False を戻り値として関数は終了する。ループの全過程を終了して抜けてきた場合は True を戻り値として関数は終了する。

単語 pots と stop を使ってこの関数をテストする。True が期待されるが、インデックスが領域外という IndexError が出る：

```
>>>is_reverse('pots', 'stop')
  File "is_reverse.py", line 9, in is_reverse
    if word1[i] != word2[j]:
IndexError: string index out of range
```

このようなときに私が最初にやることは、このエラーが発生する直前でこれらの二つのインデックスの値を表示させてみることである。

```
    while j > 0:
        print(i, j)  #ここで印刷
        if word1[i] != word2[j]:
            return False
        i = i + 1
        j = j - 1
```

実行してみると、新たな情報が得られるはずだ：

```
>>>is_reverse('pots', 'stop')
0 4...
IndexError: string index out of range
```

ループの最初で j の値が 4 になっているこのが、この値は文字列'pots' では範囲外になる。この文字列の最後の文字を示すインデックスの値は 3 で、j の最初の値は len(word2)-1 である。

そのように修正して、実行すると以下になる：

```
>>>is_reverse('pots', 'stop')
0 3
1 2
2 1
True
```

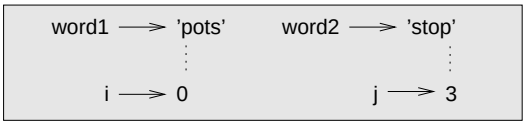


図 8.2: 状態図

今回の場合は正しい結果になったが、ループは三回しか回っていない。おかしい。このようなときは、この関数の状態図を書いてみるとよい。図 8.2 は `is_reverse` 関数の例で第一回目の繰り返し時の状態図である。私はフレーム内の変数が持つ値を表示する場所に少し工夫を施した。インデックス `i`、`j` が文字列 `word1`、`word2` のどの文字を指しているかが分かるようにその値と文字を点線で結んだ。

この図を紙に書いて繰り返し毎にインデックス `i`、`j` がどのような値をもつのか調べ、関数 `is_reverse` が持つ第二のエラーを修正せよ。

8.12 語句

オブジェクト（object）： 変数が参照するあるもの（訳注：定義に対してその実体）。

配列（sequence）： 整数インデックスによって同定される順序立った変数の組。.

アイテム（item）： 配列の一要素。

インデックス（index）： 文字列のような配列内の要素を選択ために使う整数値。

スライス（slice）： インデックスの範囲指定で取り出された文字列の一部。

空文字列（empty string）： 二つの引用符で表現された文字を含まない長さ 0 の文字列。

変更不可（immutable）： 文字列の要素を代入できない性質。

横断処理（traversal）： 似たような操作を配列の端から端まで繰り返し行う。

探索（search）： 捜していたものが見つかったときに停止する横断処理の一形態。

カウンタ（counter）： 何か数えるために使われる変数。通常は 0 で初期化され、その後インクリメントされる。

発動（invocation）： メソッドを呼ぶ文。

選択的な引数（optional argument）： 関数やメソッドで必ずしも必須としない引数。

8.13　練習問題

練習問題 8.1 文字列に対する各種のメソッドについて、
http://docs.python.org/3/library/stdtypes.html#string-methods をよく読み理解を深めよ。strip や replace の使い方を実際にやってみることは有益かもしれない。

ドキュメンテーションは若干混乱するかもしれない構文規則の表現になっているかもしれない。例えば find こうだ：

```
find(sub[, start[, end]])
```

角括弧は選択項目である。だから、sub は必修だが、start は選択項目であり、start を選択しても end は選択項目である。

練習問題 8.2 8.7 節に作成した count と類似した機能を持つメソッド count がある。このメソッドのドキュメントをよく読み、文字列'banana' に対して発動させ文字列に含まれる'a' の個数を表示せよ。

練習問題 8.3 文字列スライスは第三の引数を受け取ることができる。この第三の引数はスライスを何文字毎（ステップサイズ）にするかを指示するものである。ステップサイズが 2 であるということは、文字列内の文字を一つ置きに拾うことであり、3 であると、三つ毎になる。

```
>>> fruit = 'banana'
>>> fruit[0:5:2]
'bnn'
```

このステップサイズを-1 にすると後からのスライスになる。だからスライス [::-1] とするとその文字列の逆順の文字列が得られる。これを使って練習問題 6.6 で作成した文字列が回文である調べる関数 is_palidrome の一行版を作成せよ。

練習問題 8.4 以下の関数はいずれも文字列が小文字を含んでいるかを調べる意図に作成したものであるが、いずれも問題を含んでいる。各々の関数が実際にしていることを述べよ（関数に文字列を渡すとして）。

```
def any_lowercase1(s):
    for c in s:
        if c.islower():
            return True
        else:
            return False

def any_lowercase2(s):
    for c in s:
        if 'c'.islower():
            return 'True'
        else:
            return 'False'

def any_lowercase3(s):
    for c in s:
        flag = c.islower()
    return flag

def any_lowercase4(s):
    flag = False
    for c in s:
        flag = flag or c.islower()
    return flag

def any_lowercase5(s):
    for c in s:
        if not c.islower():
            return False
    return True
```

練習問題 8.5 シーザー暗号法は各文字をアルファベットで同一文字分文字を「回転」させる意味で微かに暗号のかたちを取っている。文字を回転させるとはアルファベットのなかでずらし、文字 Z を越えたときは先頭 A に戻るように折り返す操作である。だから、文字'A' を 3 だけ回転させると文字'D' になり、'Z' は 1 だけ回転させると'A' になる。

単語の回転ではその単語を構成する各文字を同一の量だけ回転させる。例えば、

“cheer” を 7 だけずらすと “jolly” になり、“melon” を 10 だけずらすと “cubed” になる。映画 “2001:Space Odyasey”（{2001 年宇宙の旅」）では、宇宙船内のコンピュータ（人工知能）は “HAL” と呼ばれているが、その名前を-1 だけ回転させると “IBM” となる。

関数 rotate_word を作成せよ。この関数は文字列と整数を仮引数として持つ。文字列の各文字を整数で指定された量だけ「回転」させ、その結果の文字列を戻り値とする関数である。

文字を文字コードに変換する組み込み関数 ord や文字コードを文字に変換する組み込み関数 chr を使ってもよい。アルファベットの各文字はアルファベットの順に符号化されている。例えば

```
>>> ord('c') - ord('a')
2
```

アルファベットの’c’ はアルファベットの二番目の文字である。しかし大文字の数値コードは小文字と異なることに注意。

インタネットに対する潜在的に悪意のあるジョークが回転文字数が 13 であるシーザー暗号法を使った ROT13 にコード化されていることがある。あまり気にならなければ、それを見つけ元に戻してみよう。

解答例：https://thinkpython.com/code/rotate.py.

第9章　事例研究：単語あそび

この章ではある種の性質を持つ英単語を探索するといった単語パズルを解くことを含むこの本の二番目の事例研究である。例えば、英単語の中で最長の回語は何か、また文字がまたアルファベットの順序で現れる単語にどんなものがあるかといった問題である。また新しい問題を既存の解決済みの問題に帰着させるというもう一つのプログラム開発手法を紹介する。

9.1　単語リストの読み込み

この章の演習では英単語のリストが必要だ。英単語のリストはWebで入手可能なものが沢山あるが、我々の目的に最適なものはMobyレキシコンプロジェクト（http://wikipedia.org/wiki/Moby_Project をみよ）の一部としてGrady Wardによって収集され、公開されている英単語リストがその一つである。

それは公式のクロスワードパズルや他の英単語ゲームに使える113,809個の英単語リストである。この集録は`113809f.fic`という名前のファイルになっているが、もっと簡単な名前、`.words.txt`で https://thinkpython.com/code/words.txt からダウンロードできる。

このファイルは単純なテキストファイルであるので、テキストエディタで閲覧できるし、Pythonで読むこともできる。組み込み関数`open`の引数ファイル名を与え実行すると**ファイルオブジェクト（file object）**を返してくる。これを使ってファイルを読むことができる：

```
>>> fin = open('words.txt')
```

変数`fin`は入力に使うファイルオブジェクトによく使われる名前である。モード`'r'`は読み込みモードでファイルがオープンされたことを示す（その逆は書き込みモードで`'w'`である）。

ファイルオブジェクトは読み込みのためのいくつかのメソッドを提供している。その一つは`readline`でこれは文字を改行記号に達すまで読み込み、その結果を文字列として返すメソッドである：

```
>>> fin.readline()
'aa\r\n'
```

このリストの第一番目の単語は “aa”である。これは溶岩の一種である。符号\r\nは二つ特殊記号、キャリージ・リターンと改行でこの単語を次ぎのものから分離するのに使われている。ファイルオブジェクトは今ファイルのどこにいるのかの軌跡を保存している。だから、次ぎに readline を実行すると次ぎの単語を読み出せる：

```
>>> fin.readline()
'aah\r\n'
```

次の単語は “aah”で、全く規則にあった単語である。そんなに怪訝な様子で私をみないでください。二つの特殊記号が邪魔ならば、文字列メッソド strip を使って取ってしまうこともできる：

```
>>>line = fin.readline()
>>>word = line.strip()
>>>word
aahed
```

for ループの一部にファイルオブジェクトを使うこともできる。以下のプログラムは words.txt を読み込み、一行に一単語毎に表示するものである：

```
fin = open('words.txt')
for line in fin:
    word = line.strip()
    print(word)
```

9.2　練習問題

これからの練習問題の解答は次の節にあるが、答えをみる前に一度は解答を試みてほしい。

練習問題 9.1 ファイル words.txt を読み、単語の長さが 20（特殊文字を含めないで）以上ある単語のみを表示するプログラムを作成せよ。

練習問題 9.2 Ernst Vincent Wright は 1939 年に Gadsby というタイトルの 50,000 単語の小説を出版した。この本は文字’e’ を全く含んでいない。文字’e’ は英語では最も頻度の高い文字であるので、この作業は簡単ではない。

> In fact, it is difficult to construct a solitary thought without using that most common symbol, it is slow going at first, but caution and hours of training can gradually gains facility.

この位で止めておこう。

与えられた単語が文字'e'を一つも含んでいないとTrueを返す関数has_no_eを作成せよ。

練習問題9.1のプログラムを単語が'e'を含んでいないときにのみ単語を表示するように変更し、文字'e'を含まない単語の全単語中に占めるパーセントを計算しなさい。

練習問題 9.3 文字列と文字を引数とする関数avoidsを作成せよ。この関数はその文字列に該当する文字が一つも含まれていないときはTrueを返す。

ユーザから入力された文字に対して、この文字を一つも含まない単語を表示するように練習問題9.1のプログラムを変更せよ。

また、このような文字5個を使ったときに排除される単語数が最少になるような5個の組み合わせを見つけることができるか？

練習問題 9.4 文字列と文字の組み合わせ（文字列）を引数とする関数uses_onlyを作成せよ。この関数はこの文字列がこの組み合わせ文字でのみ構成されているときはTrueを返す。また、組み合わせ文字、"acefhlo"のみを使って一つの文を作れるかな？"Hoe alfalfa"（アルファルファを刈り取る）以外には？

練習問題 9.5 文字列と要求される文字の組み合わせ（文字列）を引数とする関数uses_allを作成せよ。この関数はこの文字列が要求される文字を全て少なくとも一回は使っているときはTrueを返す関数である。全ての母音 "aeiou"を使うような単語は何個位あるのだろうか "aeiouy"ではどうかな？

練習問題 9.6 引数として受け取った単語の中の文字の並びがアルファベット順（同じ文字が並ぶのはよいとする）に並んでいるときTrueを返す関数is_abecedarianを作成せよ。この条件を満たす単語は何個位あるのだろうか？

9.3 探索

前節で取り上げた練習問題全ては共通した要素を持っている。それは8.6節でみた探索パターンで解答できることである。最も簡単な例は以下である：

```
def has_no_e(word):
    for letter in word:
        if letter == 'e':
            return False
    return True
```

この for ループは単語中の文字を横断的に探索する。もし文字 “e” が見つかれば直ちに False を返し関数は終了する。そうでなければ次ぎの文字に移る。ループを通常の姿で抜けたことは文字 “e” を一つも含まないことなので True を返す。

関数 avoids は関数 has_no_e を一般化したものであるが、プログラムの構造は同じだ：

```
def avoids(word, forbidden):
    for letter in word:
        if letter in forbidden:
            return False
    return True
```

文字の組み合わせ文字列 forbidden 内の文字を一つでも含んでいると直ちに False を返し関数は終了する。ループを通常の姿で抜けると True を返す。

関数 uses_only も条件は逆だが、同じ構造をしている。

```
def uses_only(word, available):
    for letter in word:
        if letter not in available:
            return False
    return True
```

禁止文字の組み合わせの替わりに、利用可能な文字の組み合わせを使う。利用可能文字以外の文字がみつかったら直ちに False を返して関数は終了する。

関数 uses_all では単語と組み合わせ文字の役割を逆にする。

```
def uses_all(word, required):
    for letter in required:
        if letter not in word:
            return False
    return True
```

単語（word）の中の文字を横断的に調べる替わりに、要求された文字の組み合わせ（required）の中の文字を横断的に調べる。要求された文字が一つでも欠けていたら直ちに False を返し関数を終了する。

本物のコンピュータ科学者のように考えるのであれば、uses_all は前の解いた問題の新たな例証にすぎないことを認識するはずだ。そして以下のように書くだろう：

```
def uses_all(word, required):
    return uses_only(required, word)
```

この例は**解決済み問題への帰着（reduction to a previously solved problem）**と言われるプログラム開発法の一例である。そこでは当該の問題がそれまで解決した問題の新たな例証にすぎないことを認識し、既に開発した解法を適用する。

9.4 インデックス付きループ

前節のプログラムでは for ループを伴う関数を作成した。文字列の中の任意の位置にある文字なので、そのループはインデックスが付くものではなかった。

関数 is_abecedarian では隣接する文字との比較が入るので、for ループは少し注意が必要だ。

```
def is_abecedarian(word):
    previous = word[0]
    for c in word:
        if c < previous:
            return False
        previous = c
    return True
```

再帰関数を使う版は以下のようだ：

```
def is_abecedarian(word):
    if len(word) <= 1:
        return True
    if word[0] > word[1]:
        return False
    return is_abecedarian(word[1:])
```

while ループを使う版は以下のようだ：

```
def is_abecedarian(word):
    i = 0
```

```
    while i < len(word)-1:
        if word[i+1] < word[i]:
            return False
        i = i + 1
    return True
```

このループは i=0 から始まり、i=len(word)-1 で終わる。ループを回る毎に第 i 番目の文字（これが現在の文字）と第 i+1 番目の文字（これが次ぎの文字）とが比較される。このループを失敗なしで通過すると問題の単語はテストを合格したことになる。この処理が正しいことを確認するために、“flossy” という単語を例に手作業をしてみる。この単語の長さは 6 である。従ってこのループを最後に回るときの i の値は 4 である。この値は後から二番目の文字を示す。この最後のループではこの後から二番目の文字と最後の文字を比較することになる。

以下は関数 is_palindrome（6.6 節参照のこと）を二つのインデックスを使う版である：

```
def is_palindrome(word):
    i = 0
    j = len(word)-1

    while i<j:
        if word[i] != word[j]:
            return False
        i = i+1
        j = j-1
    return True
```

また、この問題は前に解いた問題の一つの例証にすぎないことを見抜けば、以下のように書くこともできる：

```
def is_palindrome(word):
    return is_reverse(word, word)
```

と 8.11 節の is_reverse を使ってか書ける。

9.5　デバッギング

プログラムをテストすることは手強いものである。この章に現れた関数は手作業でテストできるので比較的易しいものである。それでも、潜在的なエラーの全

てを摘出するために必要な単語のセットを選ぶことは困難または不可能に近い問題である。

関数 has_no_e を例にしてみよう。'e' を含む単語のなかで、先頭にある単語、最後にある単語、中間にある単語でテストする必要がある。また、長い単語、短い単語、空文字列のような極端に短い単語も調べる必要がある。空文字列は**特別な例（special case）**の一例でときとしてエラーが潜むことがある。

テストとして用意した単語の他に words.txt のような既存の単語リストでプログラムをテストすることもできる。出力を調べることでエラーを見つけることができるかもしれないが、注意したいのは摘出したエラーはそのリストに含まれていない単語ではなく、含まれている単語に関してのみであることだ。

一般にテストすることはバグを探すのに有効だが、テストに使うよいセットを用意するのは容易ではない。そして、たとえ用意できたとしても、プログラムが正しいことの確証を持つことはできない。

伝説的なコンピュータ科学者によれば

> プログラムをテストすることはバグの存在を示すことには使えるが、決してバグがないことを示すことには使えない。
>
> —エドガー・ダイクストラ

9.6 語句

ファイルオブジェクト（file object）： 開かれたファイルを表現する値。.

解決済み問題への帰着（reduction to a previously solved problem）： 既に解かれた問題の例証として表現することで問題を解く。

特別な例（special case）： 典型的でなくまたは明白でない（そして正確に処理することができにくくみえる）検定用に使う例。.

9.7 練習問題

練習問題 9.7 この問題はラジオプログラム Car Talk (http://www.cartalk.com/content/puzzlers) で放送されたパズル名人を基礎とした問題である。

> 連続して二回同じ文字が続き、しかもこれが三回引き続いて起こるような単語を私にください。それに近いような単語の例を示す。例えば、committee つまり、c-o-m-m-i-t-t-e-e。途中にこっそりと i が入っ

ていることを除けば可成り近い例である。また、Mississippi つまり、M-i-s-s-i-s-s-i-p-p-i。もしも i を削除してもよいとすれば、まあ条件を満たす例になる。しかし、上の条件を完全に満たす単語が一つある。しかも、私の知る得る限りではこれが唯一の単語だ。条件をみたしている単語は多分 500 個近くあるかもしれないが、私はその一つしか知らない。さて、その単語は何か？

この単語を見つけるプログラムを作成せよ。
解答例： https://thinkpython.com/code/cartalk1.py

練習問題 9.8 これも Car Talk のパズル名人による。((http://www.cartalk.com/content/puzzlers)

先日ハイウエイをドライブしていた。そしてたまたま私の車の走行距離計が目に入った。多くの走行距離計と同じで 6 桁の数字で走行距離がマイルを単位として表示されている。従って、私の車が 300,000 マイル走ったのであると表示は 3-0-0-0-0-0 となる。さて、あの日に私がみた走行計の文字は興味あるものであった。最後の四桁の数字が回文になっていたのだ。つまり、それらは前から読んでも後から読んでも同じだったわけだ。例えば、5-4-4-5 は回文だ。距離計は例えば 3-1-5-4-4-5 となっていたわけである。ところが 1 マイル走って距離計をみると今度は最後の五桁が回文になっていた。例えば、3-6-5-3-5-6 ようだ。さらに 1 マイル走って距離計をみると六桁の内中間の四文字が回文になっていた。さらに驚くことに、1 マイル走って距離計をみるとなんと六桁全体が回文になっていた。問題は最初に眺めた距離計の表示は何か？

六桁全ての数字をテストし、上の条件を満たす組み合わせを表示する Python プログラムを書け。
解答例： https://thinkpython.com/code/cartalk2.py

練習問題 9.9 これもまた Car Talk からである。探索手法で解けるはずだ。(http://www.cartalk.com/content/puzzlers):

母親を最近に訪ねた、そして私の年齢の二桁の数字が母のそれと逆順になっていることに気が付いた。例えば、母が 73 で、私が 37 といった具合だ。このような状況がどの位の頻度で起ったのか話題になったが、話が横道に逸れてしまい、答えを得る機会がなかった。

自宅に帰った後に、この年齢の逆順はこれまで 6 回起きていたことが分かった。また、幸運ならば、この数年中にこの逆順が起こることも

> 分かった。さらに、充分に幸運ならばこの逆順はさらにもう一度訪れるだろうと期待できた。従って（母が 99 歳になるまで）8 回の頻度でこの現象が起きたことになる。問題は、私の現在の年齢はいくつか？

この問題を探索の問題として解く Python のプログラムを作成せよ。組み込み関数 `zfill` を使ってもよい (訳注：`zfill` は数字の先頭を 0 で埋める。`'5'.zfill(4)` のように使う)。
解答例：https://thinkpython.com/code/cartalk3.py

第10章　リスト

この章ではPythonで扱える最も便利な組み込み型の一つであるリストを取り上げる。さらにオブジェクトについて学び、同一のオブジェクトを一つ以上の変数で参照したとき何が起こるか調べる。

10.1　リストは配列である

文字列と同じように**リスト（list）**も値の配列である。文字列の値が文字であるのに対して、リストの値は任意の型で構わない。リストの値は**要素（elements）**または**アイテム（items）**と呼ばれる。

リストを生成する方法はいくつもあるが、最も簡単な方法は要素を角括弧（[と]）で括ることである：

```
[10, 20, 30, 40]
['crunchy frog', 'ram bladder', 'lark vomit']
```

一番目の例は四つの整数のリストである。二番目は三つの文字列のリストである。一つのリストの要素は同じ型にする必要はない。以下のリストは文字列、浮動小数点数、整数、それに、（あれ！）リストを含んでいる：

```
['spam', 2.0, 5, [10, 20]]
```

リストの中にリストがあると**入れ子のリスト（nested list）**になる。

要素を何も含まないリストは空リストと呼ばれ、空の角括弧 [] で生成できる。予想したであろうように、リストを変数に代入できる：

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> numbers = [17, 123]
>>> empty = []
>>> print(cheeses, numbers, empty)
['Cheddar', 'Edam', 'Gouda'] [17, 123] []
```

10.2　リストは変更可能

リストの要素にアクセスする構文は文字列の中の文字にアクセスするときと同じである。つまり、角括弧を使う。角括弧の中の表式はインデックスを表現する。インデックスは 0 から始まることに注意してほしい。

```
>>> cheeses[0]
Cheddar
```

文字列と異なり、リストは変更可能である。代入文で角括弧の演算が左に現れるとそれはリストのある要素への代入を表現する：

```
>>> numbers = [42, 123]
>>> numbers[1] = 5
>>> numbers
[42, 5]
```

リスト numbers の第一要素は 123 だったものが 5 になった。

リストはインデックスと要素との間にある関係と考えることもできる。この関係は**マッピング（mapping）**とも呼ばれている。つまり各インデックスは要素の一つを「写像」しているとみることができる。図 10.1 はリスト cheeses, numbers, empty の状態図を示している。これらのリストは箱によって表現されている。そ

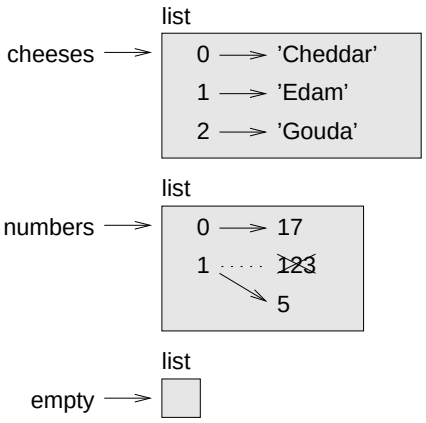


図 10.1: 状態図

の箱の外側にはリストの名前が、内側にはその要素が書かれている。cheeses は要素が 0, 1, 2 のインデックスを持つ一つリストを参照している。numbers は二つの要素を持つが、図はその第二要素は 123 から 5 に変更されたことを示している。empty は空リストを参照している。

リストのインデックスは文字列のそれと同じように働く。つまり

- 任意の整数表式がインデックスとして使える。
- もし存在しない要素を呼び出したり、書き出したりすると IndexError になる。
- もしもインデックスの値が負であるとすると、それはリストの後の要素からカウントしている意味だ。

in 演算子もリストにそのまま機能する。

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> 'Edam' in cheeses
True
>>> 'Brie' in cheeses
False
```

10.3 リストの横断的処理

リストの要素を横断的にアクセスする一般的な方法は for ループを使うものである。構文は文字列の場合と同じである：

```
for cheese in cheeses:
    print(cheese)
```

これはリストの要素を読み出すのみのときの場合である。要素の書き込みや更新をしたいときにはインデックスが必要になる。この場合の一般的な方法は range 関数を使うものである：

```
for i in range(len(numbers)):
    numbers[i] = numbers[i] * 2
```

このループでリストの各要素を横断的に更新できる。関数 len でリストの要素の数を出し、関数 range で 0 から n-1 までの数のリストを生成する。ここで n は問題のリストの長さである。ループが回る毎に i は次ぎの要素に対応するインデックスの値を持つ。代入文では i は古い要素に値を読むことに使われ、新しい値を代入することにも使われている。

空リストに対する for ループでは本体は一度も実行されない：

```
for x in []:
    print('This never happens.')
```

リストの中にリストを含ませることもできるが、そのリストは一つの要素としてカウントされる。従って以下のリストの要素の数は4である。

```
['spam', 1, ['Brie', 'Boquefort', 'Poi le Veq'], [1, 2, 3]]
```

10.4　リストに対する演算

演算子+はリストの連結を実現する。

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> c = a + b
>>> c
[1, 2, 3, 4, 5, 6]
```

同じように演算子*は与えられた回数だけ繰り返したリストを生成する。

```
>>> [0] * 4
[0, 0, 0, 0]
>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

最初の例では [0] を4回繰り返す。二番目では [1, 2, 3] を3回繰り返す。

10.5　リストのスライス

スライス演算もリストに適用できる：

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3]
['b', 'c']
>>> t[:4]
['a', 'b', 'c', 'd']
>>> t[3:]
['d', 'e', 'f']
```

一番目のインデックスを省略すると、スライスは先頭から始まる。二番目のインデックスを省略するとスライスは最後尾まで進行する。両インデックスを省略するとリスト全体がコピーされる。

```
>>> t[:]
['a', 'b', 'c', 'd', 'e', 'f']
```

リストは更新可能だから、折り畳み、回転、変更の操作をする前にそのコピーを作っておくことは多々有益である。

スライス演算子を代入文の左辺で使うと、複数要素を更新できる：

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3] = ['x', 'y']
>>> t
['a', 'x', 'y', 'd', 'e', 'f']
```

10.6 リストメソッド

Python はリストに適用できる各種のメソッドを提供している。例えば、append は要素の追加に使う：

```
>>> t = ['a', 'b', 'c']
>>> t.append('d')
>>> t
['a', 'b', 'c', 'd']
```

メソッド extend はリストを引数として受け取り、その要素の全てをリストに追加する：

```
>>> t1 = ['a', 'b', 'c']
>>> t2 = ['d', 'e']
>>> t1.extend(t2)
>>> t1
['a', 'b', 'c', 'd', 'e']
```

この場合リスト t2 の中味は変わらない。

メソッド sort はリストの要素を低位から高位に順に並べ替える：

```
>>> t = ['d', 'c', 'e', 'b', 'a']
>>> t.sort()
>>> t
['a', 'b', 'c', 'd', 'e']
```

リストのメソッドは全て戻り値がない、だからNoneを戻す。もし間違ってt=t.sort()とすると、期待はずれの結果になる。

10.7 写像・フィルタ・還元

リストの要素を全て足そうと思ったら、以下のようにループを使うだろう：

```
def add_all(t):
    total = 0
    for x in t:
        total += x
    return x
```

変数totalは0で初期化される。変数xはループを回る毎にリストtの要素を得る。演算子+=は変数の更新の操作の省略形である。**累積代入文（augmented assignment statement）**、total += xはtotal = total + xと等価である。ループが進むに連れて、totalは要素の和の累積になる。このような機能のために用いられる変数は**アキュームレイタ（accumulator）**と呼ばれる。

要素の全和を求めることは一般的なので、Pythonでは組み込み関数sumを提供している：

```
>>> t = [1, 2, 3]
>>> sum(t)
6
```

要素の列を一つの値にしてしまうような操作を**還元（reduce）**と呼ぶ。

ときとして他のリストを作成しつつ、リストを横断的に処理する必要がある。例えば

```
def capitalize_all(t):
    res = []
    for s in t:
        res.append(s.capitalize())
    return res
```

変数 res は空リストとして初期化される。ループが回る毎にオリジナルのリストの要素が処理され res に追加される。従って変数 res は別な種類のアキュームレイタとみなすことができる。関数 capitalize_all のような処理を**写像（map）**という。ここではあるリストの全要素に同一の処理（capitalize()）を施すからである。

もう一つの操作はリストの一部を選択して部分リストを作成するものである。例として文字列のリストから大文字のみを含む要素を選択し、部分リストを作る問題を考える：

```
def only_upper(t):
    res = []
    for s in t:
         if s.isupper():
            res.append(s)
    return res
```

メソッド isupper は文字列が大文字のみを含んでいるとき True を返す。

このような操作はリストの要素のなかで条件を満たすものだけを別なリストの要素にするから**フィルタ（filter）**と呼ばれる。

リストに対する操作で最も一般的なものはこれらの写像・フィルタ・還元を組み合わせたものである。

10.8 要素の削除

リストの要素を削除する方法はいくつかある。要素番号を知っているのであれば、pop を使うことができる：

```
>>> t = ['a', 'b', 'c']
>>> x = t.pop(1)
>>> t
['a', 'c']
>>> x
b
```

メソッド pop はリストを修正し、削除した要素を返す。もし pop をインデックスなして使うと最後尾の要素を削除し、その要素を返す。

要素を戻す必要がなければ、del 演算子が使える：

```
>>> t = ['a', 'b', 'c']
>>> del t[1]
>>> t
['a', 'c']
```

削除する要素を知っている（しかし、要素番号は不明）のであれば、メソッド remove が使える：

```
>>> t = ['a', 'b', 'c']
>>> t.remove('b')
>>> t
['a', 'c']
```

もし一つ以上の要素を一時に削除したいのであれば、スライス演算子で範囲を決め、del 演算子を使えばよい：

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> del t[1:5]
>>> t
['a', 'f']
```

スライスは第一番目のインデックスの要素から第二番目の要素まで選択する。注意、二番目のインデックスの要素は含まれない。

10.9 リストと文字列

文字列は文字の配列であり、リストは値の配列である。しかし、文字のリストは文字列とは異なる。

文字を要素とするリストに文字列を変換するには関数 list を使う：

```
>>> a = 'spam'
>>> t = list(a)
>>> t
['s', 'p', 'a', 'm']
```

list は組み込み関数であるので、変数名として避けるべきである。また l（エル）はあまりにも 1 と似ているので避け、t を用いた。

関数 list は文字列を文字に分解しリストにする。文字列を単語に分解するのは split を使う：

```
>>> s = 'pining for the fjords'
>>> t = s.split()
>>> t
['pining', 'for', 'the', 'fjords']
```

このメソッドに引数として**区切り文字（delimiter）**を与えることができる。これで如何なる文字を使って単語に分割するかを指定できる。

```
>>> s = 'spam-spam-spam'
>>> delimiter = '-'
>>> t = s.split(delimiter)
>>> t
['spam', 'spam', 'spam']
```

メソッド join は split の逆の操作だ。このメソッドは引数として文字列のリストを受け取り、要素の連結を作る。join は文字列のメソッドで区切り文字に対して発動される。リストはこのメソッドの引数として与える：

```
>>> t = ['pining', 'for', 'the', 'fjords']
>>> delimiter = ' '
>>> s = delimiter.join(t)
>>> s
'pining for the fjords'
```

このばあいは空白を区切り文字として単語を接合する。区切り文字なして接合するには空文字'' を区切り文字とすればよい。

10.10 オブジェクトと値

以下のような代入文を実行してみる：

```
>>> a = 'banana'
>>> b = 'banana'
```

変数 a、b とも同じ文字列を参照している。しかし、この二つが同一の文字列を参照しているかどうかは不明である。図 10.2 に示したように二つの可能性がある。一つの可能性は、この二つは同じ値を持つが異なった**オブジェクト（実体）（object）**を参照しているとするものだ。他方は同一のオブジェクトを参照しているとするものである。これをチェックするには is 演算子を使ってみればよい：

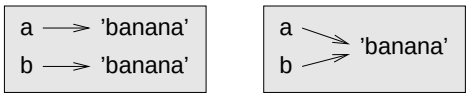


図 10.2: . 状態図

```
>>> a = 'banana'
>>> b = 'banana'
>>> a is b
True
```

この例では Python は一つ文字列オブジェクトを生成し、変数 a、b ともこれを参照している。

しかし、二つのリストでは、二つのオブジェクトが生成される：

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> a is b
False
```

従って状態図は図 10.3 のようになる。この場合二つのリストは同じ要素を持って

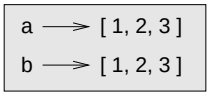


図 10.3: 状態図

いるので**等価（equivalent）**であるという。しかし異なったオブジェクトであるので**同一（identical）**ではない。同一である二つのオブジェクトは等価であるが、等価である二つのオブジェクトは必ずしも同一ではない。

これまで「オブジェクト（実体）」と「値」を交換可能な言葉として使ってきたが、もう少し厳密に言うとこうなる。オブジェクトは値を持つ。操作 [1, 2, 3] を行うとその要素が整数の値を持つリストオブジェクトが得られる。もしも他のリストが同じ値を持っていたとすると、2 つのリストは同じ値を持つというが、同一のオブジェクトではない。

10.11　別名参照

変数 a は一つのオブジェクトを参照し、b=a という代入文を実行すると、二つの変数は同一のオブジェクトを参照することになる：

```
>>> a = [1, 2, 3]
>>> b = a
>>> b is a
True
```

従って状態図は図 10.4 のようになる。一つの変数にオブジェクトを代入すること

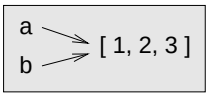


図 10.4: 状態図

は**参照（reference）**という。この例では一つのオブジェクトに対して二つ参照があることになる。一つのオブジェクトに対して二つ以上の参照があるとき、このオブジェクトは**別名参照された（aliased）**オブジェクトという。

もしも別名参照されたオブジェクトが変更可能であるとすると、一つの別名に行った変更は他にも反映される：

```
>>> b[0] = 42
>>> a
[42, 2, 3]
```

この振る舞いは利用できるときがあるが、エラーを起こしやすい。変更可能オブジェクトにたいする別名は避ける方が安全だ。変更不可のオブジェクトでは問題は少ない。

```
>>> a = 'banana'
>>> b = 'banana'
```

この例で変数 a、b が同じ文字列を参照しているかどうかは殆ど問題にならない。

10.12 リストを引数に使う

リストを関数の引数として渡すと、その関数はそのリストの参照ができる。関数はこの仮引数としてのリストを変更するとこの関数の呼び手のリストも変更を受ける。例えば、関数 delete_head はリストの先頭の要素を削除する関数としよう：

```
>>> def delete_head(t):
 del t[0]
```

この関数を使う：

```
>>> letters = ['a', 'b', 'c']
>>> delete_head(letters)
>>> letters
['b', 'c']
```

仮引数 t と変数 letters とは同一のリストの別名参照である。これらのスタック図は 10.5 図のようになる。このリストは二つのフレームで共用されているので中

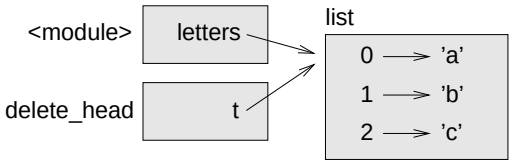


図 10.5: スタック図

間に置いた。

リストの変更を伴う操作と新たなリストを生成する操作を区別することは特別に重要だ。例えば、メソッド append はリストの変更を伴うが、演算子+はあらたなリストの生成になる。

```
>>> t1 = [1, 2]
>>> t2 = t1.append(3)
>>> t1
[1, 2, 3]
>>> t2
None

>>> t4 = t1 + [4]
>>> t4
[1, 2, 3, 4]
```

リストの変更を期待する関数を作ろうとするときもこの相異に注意が必要だ。

```
def bad_delete_head(t):
    t = t[1:]     #間違いだ
```

スライスは新しいリストを生成し代入文で変数 t はそれを参照する。しかし仮引数として渡された元のリストはそのままである。確認してみよう：

```
>>> t4 = [1, 2, 3]
>>> bad_delete_head(t4)
>>> t4
[1, 2, 3]
```

関数 bad_delete_head のはじめでは変数 t と t4 は同じリストを参照している。しかし終わりでは変数 t は新しいリストを参照し、t4 は元々の変更されていないリストを参照している。

そこで新規リストを生成し、そのリストを戻り値とする関数を作成した。関数 tail は以下のように先頭の要素以外をリストとして戻す関数である：

```
>>> def tail(t):
    return t[1:]
```

これも元のリストは変化なしだ。使ってみる：

```
>>> letters = ['a', 'b', 'c']
>>> rest = tail(letters)
>>> rest
['b', 'c']
```

10.13 デバッギング

リストや他の変更可能オブジェクトを不注意に使うとデバッグに時間をとられることになる。以下はよくある落とし穴とその回避法である。

1. 多くのリストに関するメソッドは引数のリストの変更を伴い、戻り値は None である。これに反して、文字列に関するメソッドは元の文字列は変更しないで、新たに文字列を生成する。だから、文字列の対するメソッドは

   ```
   word = word.strip()
   ```

   のつもりでリストにも以下のような記述をしたら、それは間違いである：

   ```
   t = t.sort()  #間違い
   ```

   メソッド sort は None を返すので、t に対する次の操作で問題が発覚する。リストに対するメソッドや演算を行う前にこれらに関するドキュメンテーションを注意深く読み、インタラクティブ・モードで試してみるべきである。

2. 慣用句を選択し、それに固執せよ。リストの対する問題は同じことをするのに多くの方法があることである。例えば、要素の削除はpop、remove、del、またはslice代入によっても実現できる。また、要素の追加はメソッドappendや演算子+で実現できる。tがリストでxが追加要素であると以下の書き方は正しい：

```
t.append(x)
t = t + [x]
```

しかし、以下は正しくない：

```
t.append([x])        #間違い!
t = t.append(x)      #間違い!
t = [x]               #間違い!
t = t + x             #間違い!
```

これらをインタラクティブ・モードで実行し、何が起こるかみてほしい。最後の例のみは実行時エラーを吐き出し、その他はエラーがでないが結果がおかしい。

3. コピーを作成し、別名参照を避けよ。引数に与えたリストを変更してしまうsortのようなメソッドを使い、且つ元のリストを原型でほしいときには、そのオリジナルのコピーを取ればよい：

```
>>> t = [3, 1, 2]
>>> t2 = t[:]
>>> t2.sort()
>>> t
[3, 1, 2]
>>> t2
[1, 2, 3]
```

組み込み関数sortedを使うのも一手である。この関数は戻り値として新たなリストを返す。

```
>>> t2 = sorted(t)
>>> t
[3, 1, 2]
>>> t2
[1, 2, 3]
```

10.14 語句

リスト（list）： 値の系列。

要素（elements）： リスト（または他の配列）の中の個々の値。アイテム（items）とも呼ぶ。

入れ子のリスト（nested list）： 他のリストの要素になっているリスト。

アキュームレイタ（accumulator）： ループの数え上げや単一の結果に累積する際に使う変数。.

累積代入文（augmented assignment statement）： +=. のような演算子を使って変数の値を更新する代入。

還元（reduce）： 配列を横断的してその要素を単独の結果に累積させる処理パターン。

写像（map）： 配列を横断的して各要素に同一の命令を施す処理パターン。

フィルタ（filter）： リストを横断してある条件を満たす要素だけを抽出する処理パターン。

オブジェクト（実体）（object）： 変数が参照する実体。オブジェクトは値とタイプを持つ。

等価（equivalent）： 同じ値を持つこと。

同一（identical）： 同じオブジェクトを参照していること。それ故それらは等価である。

参照（reference）： 変数と値との間の対応関係。

別名参照された（aliased）： 二つ以上の変数が同一の一つのオブジェクトを参照している状況。

区切り文字（delimiter）： 文字列を何処で区切るかを指示する文字または文字列。

10.15　練習問題

以下の練習問題の解答例は https://thinkpython.com/code/list_exercises.py からダウンロードできる。

練習問題 10.1 入れ子になっている整数の全要素の総和を求める関数 nested_sum を作成せよ。例を示す：

```
>>> t = [[1, 2], [3], [4, 5, 6]]
>>> nested_sum(t)
21
```

練習問題 10.2 整数値のリストを引数として受け取り、累積をリストとして返す関数 cumsum を作成せよ。ここで累積リストとは、新しいリストの i 番目の要素はオリジナルのリストの最初から i+1 個の要素の総和である。例を示す：

```
>>> t = [1, 2, 3]
>>> cumsum(t)
[1, 3, 6]
```

練習問題 10.3 リストを引数として受け取りそのリストの先頭と最後尾の要素を削除した新しいリストを返す関数 middle を作成せよ。例を示す：

```
>>> t = [1, 2, 3, 4]
>>> middle(t)
[2, 3]
```

練習問題 10.4 リストを引数として受け取り、先頭と最後尾の要素を削除する修正をそのリストに施す関数 chop を作成せよ。戻り値は None である。例を示す：

```
>>> t = [1, 2, 3, 4]
>>> chop(t)
>>> t
[2, 3]
```

練習問題 10.5 リストを引数として受け取り、そのリストが昇順にソートされていると True を、それ以外は False を返す関数 is_sorted 作成せよ。例を示す：

```
>>> is_sorted([1, 2, 2])
True
>>> is_sorted(['b', 'a'])
False
```

練習問題 10.6 二つの単語が一つの単語の文字の入れ替えで他方の単語になるとき、二つの単語はアナグラムであると言う。二つの単語を引数として受け取りそれらがアナグラムであると True を返す関数 is_anagram を作成せよ。

練習問題 10.7 リストを受け取りその要素が重複しているときは True を返す関数 has_duplicated を作成せよ。この関数ではリストを変更してはいけない。

練習問題 10.8 所謂誕生日パラドックスである（http://en.wikipedia.org/wiki/Birthday_paradox.）。クラスに 23 名の学生がいる。二人が同じ誕生日になる機会はどの程度あるか？学生数 23 人の誕生日（1 から 366 まで）を区間とする乱数を作り調べる。ヒント：モジュール random の中にある関数 randint を使う。
解答例：https://thinkpython.com/code/birthday.py

練習問題 10.9 単語ファイル words.txt を読み、一単語を一要素するリストを生成する関数を作成せよ。

二つの版、メソッド append を使うものと演算子+を使い t=t+[x] とするものを作れ。実行時間を計測しどちらが長い時間かかるか調べよ。まだ何故か？実行時間の計測は time モジュールの関数が使える。
解答例：https://thinkpython.com/code/wordlist.py

練習問題 10.10 単語が単語リストの中にあるかチェックしたい。in 演算子を使うことも考えられるが、この演算子は端から端へ一つずつ要素を調べて行くので時間がかかる。単語リストはアルファベット順になっているので、探索には二分法が使える。単語リストの中間から初めて、目的の単語がこの中間の単語より前にあるかどうかを調べる。そうであれば、前半を同様な方法で調べる。そうでないときは、後半を調べる。どちらであっても半分は探索をしないで済む。リストの単語数が 113,809 個であるとき、17 ステップで単語を見つけることができるか、存在しないかが分かる。ソートされたリストと目的の値を引数として、もし目的の値がリストにあるときは True を返し、存在しないときは False を返す関数 in_bisect を作れ。またはモジュール bisect のドキュメンテーションを読み、使ってみる。
解答例：https://thinkpython.com/code/inlist.py

練習問題 10.11 二つの単語がお互いに逆順であるとき二つは「逆順ペア」と呼ぶことにしよう。単語リストに存在する全ての「逆順ペア」を見つけるプログラムを書け。
解答例：https://thinkpython.com/code/reverse_pair.py

練習問題 10.12 二つの単語から構成する文字を互い違いに組み合わせると新しい単語は生成できるとき、この二つの単語は「咬み合っている」と呼ぶことにしよう。例えば、“shoe” と “cold” から “schooled” が作れる。

1. 「咬み合っている」単語ペア全てを見つけるプログラムを書け。ヒント：全てのペアを列挙するな。

2. 三つ方法で「咬み合っている」単語はあるだろうか？つまり、その単語の一番目の文字、二番目の文字、または三番目の文字から出発して、三番目毎に文字を抜き出し、三つの文字列を作るとこれらの全てが単語を作り出すような単語を見いだすことができるか？(訳注：逆にみると三つの単語があり、それらを構成する文字を一文字毎に交互に抜き出して並べると単語になるような三つの単語があるかという問題になる。)

出典：この練習問題は http://puzzlers.org よりヒントを得た。
解答例：https://thinkpython.com/code/interlock.py.

第11章　辞書

この章では辞書と呼ばれているもう一つの組み込み型を取り上げる。辞書はPython の最も優れた特徴の一つである。辞書型は多くの効果的で気が利いたアルゴリズムで構築されている。

11.1　辞書は写像

辞書（dictionary）はリストに似たものであるが、より一般的なものである。リストではインデックスは整数でなければならなかったが、辞書では（ほとんど）任意の型で構わない。

辞書をインデックス（これを**キー（keys）**と呼ぶ）の集合と値の集合との写像であると考えてもよい。一つのキーと一つの値の纏まりを**キーと値のペア（keys-value pair）**と呼ぶし、ときとして**アイテム（item）**と呼ぶ。

辞書の例として、英語からスペイン語への写像を示す辞書を作ってみよう。従って、キーも値も全て文字列である。関数 `dict` はアイテムなして空の辞書を生成する。言うまでもないが、`dict` は組み込み関数の名前であるので変数名として避けるべきである。

```
>>> eng2sp = dict()
>>> eng2sp
{}
```

波括弧は空の辞書を表している。この辞書にアイテムを追加するのは角括弧を使う：

```
>>> eng2sp['one'] = 'uno'
```

この行でキーを`'one'` として値を`'uno'` とするアイテムが生成されたことになる。`print` 文で表示してみると、キーと値をコロンで区切ったキーと値のペアが確認できる：

```
>>> eng2sp
{'one': 'uno'}
```

この出力形式は入力形式でもある。例として三つのアイテムを持つ新しい辞書を生成してみよう：

```
>>> eng2sp = {'one':'uno', 'two':'dos', 'three':'tres'}
```

これを print で表示すると：

```
>>> eng2sp
{'three': 'tres', 'two': 'dos', 'one': 'uno'}
```

となりちょっと戸惑う。キーと値のペアの順序が異なっている。あなたのコンピュータで実行するとまたこれと異なっているかもしれない。アイテムの並びは予想できないのだ。

しかし、辞書では整数でインデックスすることは決してないのでなにも問題はない。その替わり、キーが対応するアイテムの選択に使われる：

```
>>> eng2sp['two']
dos
```

キー'two' は常に値'dos' にマップされるので、アイテムの並ぶ順序はどうでもよい。

キーが辞書の中にないと、例外エラーになる：

```
>>> eng2sp['four']
KeyError: 'four'
```

関数 len も使えて、キーと値のペアの個数を返す。

```
>>> len(eng2sp)
3
```

演算子 in も使える。この演算ではキーだろうとしたものがその辞書にキーとしてあるかどうかのブール代数値を返す（値としてあることは充分でない）。

```
>>> 'one' in eng2sp
True
>>> 'uno' in eng2sp
False
```

辞書に値として存在するかどうかを確認したいならば、メソッド values を使うとよい：

```
>>> vais = eng2sp.values()
>>> 'uno' in vais
True
```

演算子 in はリストと辞書とでは異なったアルゴリズムを使っている。リストでは 8.6 節で議論した探索アルゴリズムを使っている。リストが長くなって行くに従ってそれに比例して探索時間が長くなる。辞書に対しては Python は**ハッシュ表（hashtable）**と呼ばれているアルゴリズムを使っている。このアルゴリズムはアイテムの数がどんなに増えてもほぼ同じ時間で演算子 in を実行できる。

付録 B.4 で如何にしてこれを可能にしているか説明するが、その説明はもう数章を読むまでは納得できないかもしれない。

11.2 カウンタの集合として辞書を使う

例えば文字列が与えられて、この文字列に使われている文字の頻度を調べたいとしよう。

この問題を処理する方法はいくつもある：

1. アルファベットに対応する 26 個の変数を用意する。そして、文字列を横断的に眺め、ある文字が現れたら対応する変数の値をインクリメントする。連鎖 if 文を使うことになるだろう。

2. 要素が 26 あるリストを作成する。文字列を横断的に眺め、出現した文字を整数値に変換する関数 ord を使って整数値に変換し、これをインデックスとしてリストの要素の値をインクリメントする。

3. 文字をキーとしてカウンタを値とする辞書を生成する。文字列を横断的に眺め、現れた文字が初めてのものであると、この文字をキーとして値を 1 とする要素を辞書に追加する。既出のものならば、該当する要素の値をインクリメントする。

どの方法を選択しても同じ結果が得られる。しかし、各々は異なった計算を実装したことになる。

一つの**実装（implementation）**は一つの計算方法による実行である。ある計算方法は他のものより優秀である。例えば、辞書を使った実装の優位な点は、事前にどんな文字がその文字列に現れるか知る必要がないことであり、現れた文字のみについてカウンタを作ればよいことだ。コードは以下のような感じになるはずだ：

```
def histogram(s):
    d = dict()
    for c in s:
        if c not in d:
            d[c] = 1
        else:
            d[c] += 1
    return d
```

関数名は**ヒストグラム（histogram）**である。これは統計学的な言葉で頻度分布を意味している。

関数の一行目は空の辞書の作成である。for ループは文字列を横断的に眺める。ループが回る毎にもし文字 c が辞書にないときには新たに文字 c をキーとし、値を 1 とするアイテムを生成する。もし c が既出ならばこの文字をキーとする要素 d[c] をインクリメントする。使ってみる：

```
>>>h = histogram('brontosaurus')
>>>h
{'a': 1, 'b': 1, 'o': 2, 'n': 1, 's': 2, 'r': 2, 'u': 2, 't': 1}
```

この頻度分布は文字'a' や'b' は一度だけ、文字'o' は二回などであることを示す。

辞書には get というメソッドがある。これは引数としてキーとその既定値を受けとる。そのキーが存在すればそのアイテムの値を戻すが、なければその既定値を戻す。

```
>>>h = histogram('a')
>>>h
{'a', 1}
>>>h.get('a', 0)
1
>>>h.get('b', 0)
0
```

この get を使って関数 histogram をよりコンパクトに書け。関数の本体にある if 文を無くすことができるはずだ。

11.3　ループ処理と辞書

辞書に対して for 文を使うと、それは辞書のキーを横断的に眺めることになる。例として各キーと対応する値を表示する関数 print_hist を以下示す：

```
def print_hist(h):
    for c in h:
        print(c, h[c])
```

実行してみる：

```
>>>h = histogram('parrot')
>>>print_hist(h)
a 1
p 1
r 2
t 1
o 1
```

ここでもキーの並びには特別な順序はない。ソートされた順序でキーを横断的に表示したければ組み込み関数 sorted を使えばよい：

```
>>> for key in sorted(h):
...     print(key, h[key])
a 1
o 1
p 1
r 2
t 1
```

11.4 逆ルックアップ

辞書 d とキー k が与えられたときそのキーに対応する値を見つけることは容易い、つまり、v = d(k) だ。この操作を**ルックアップ（look up）**という。

しかし、v が与えられ、これから k をみつけるとなると何が起こるだろうか？二つの問題がある。第一は v にマップされるキーは一つ以上あることである。アプリケーションによるが、そのキーの一つを選択すれば済みのこともあれば、全てのキーをリストにする必要があるかもしれない。第二はこの**逆ルックアップ（reverse lookup）**になっている探索を容易く行う方法がないことである。

以下は値を受け取りその値にマップされた最初のキーを返す関数の例である：

```
def reverse_lookup(d, v):
    for k in d:
```

```
        if d[k] == v:
            return k
    raise LookupError()
```

この関数は探索パターンのもう一つの例であるが、これまでにない raise を使っている。raise 文は例外を作りだす。今の場合は組み込み型例外である LookupError() を吐き出す。このエラーは引数の値が問題を含んでいることを示すものである。つまり、ループが終わってしまったということは、その辞書には v を持つ要素はないことになり、例外を送出する。

以下は成功裡に終わった逆ルックアップの例である：

```
>>> h = histogram('parrot')
>>> key = reverse_lookup(h, 2)
>>> key
'r'
```

そして不首尾の終わり例：

```
>>> key = reverse_lookup(h, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 5, in reverse_lookup
LookupError
```

例外の送出は Python のインタプリタが送出するものと同じである。つまり、トレースバックとエラーメッセージを表示する。

送出文はオプションとして詳細なエラーメッセージを受け取ることができる：

```
>>> raise LookupError('value does not appear in the dictionary')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
LookupError: value does not appear in the dictionary
```

逆ルックアップは順ルックアップに比較して鈍い。もしこれを頻繁に使ったり、辞書が大きかったりするときにはプログラムの実行はその影響を被る。

11.5 辞書とリスト

リストは辞書の値として現れることができる。例えば、文字から頻度をマップする辞書が与えられ、この逆引き辞書を作りたいとしよう。つまり、頻度から文字をマップする辞書を作りたいわけである。いくつかの文字が同じ頻度を持つことがあり得るので、この逆引き辞書の値は文字のリストにするのが妥当である。

```
def invert_dict(d):
    inverse = dict()
    for key in d:
        val = d[key]
        if val not in inverse:
            inverse[val] = [key]
        else:
            inverse[val].append(key)
    return inverse
```

ループを回る毎に変数 key は辞書からキーを貰い、変数 val は対応する値を貰う。もし val が辞書 inverse にないとその val がキーで、その key を唯一の要素とするリスト（これを**シングルトン（singleton）**と呼ぶ）を値とする新しい辞書要素を生成する。そうでないと、この val はキーとして既にあるので、対応する値 inverse[val] に key を追加する。実行してみる：

```
>>>hist = histogram('parrot')
>>>hist
{'a': 1, 'p': 1, 'r': 2, 't': 1, 'o': 1}
>>>inverse = invert_dict(hist)
>>>inverse
{1: ['a', 'p', 't', 'o'], 2: ['r']}
```

図 11.1 には辞書 hist と inverse の状態図を示した。辞書は上部に変数の型を示す dict のラベルをはり、箱の中味はキーと値のペアを要素として並べた。もし値が整数や文字列であるときにはそれらは箱の中に並べたが、それらがリストであるときには別な箱を用意してその中に纏めて並べた。

リストは辞書の値として取り得るが、キーとしては使えない。以下は間違ったその例である：

```
>>> t = [1, 2, 3]
>>> d = dict()
```

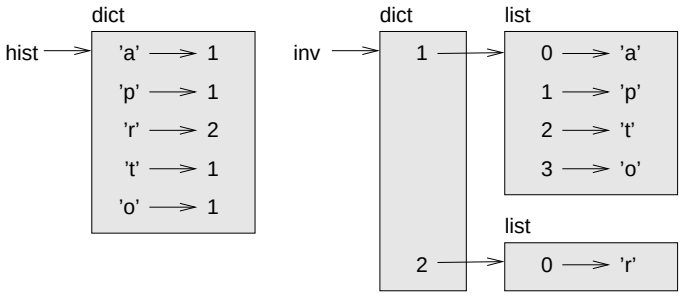


図 11.1: 状態図

```
>>> d[t] = 'oops'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: list objects are unhashable
```

既に述べたように辞書はハッシュテーブルを使って実装される。その意味するところは、キーは**ハッシュ可能（hashable）**でなければならないということだ。

ハッシュ（hash）は値（任意の型の）を受け取り、整数を返す関数である。辞書はこの整数（ハッシュ値という）をキーと値のペアの保存やルックアップに使う。

辞書のキーが変更不可である場合は問題ないが、キーがリストのような変更可能なものであるといろんなことが起こる。例えば、キーと値のペアを生成したとする。キーはハッシュされ、キーと値のペアは対応するところに保存される。そこでキーを変更したとしよう。変更されたキーはハッシュされる。新たなハッシュ値により、キーと値のペアは別なとことに保存される。こうなると、同一のキーに対して二つのアイテムができるとか、キーを発見できなくなるなどの混乱が起こる。

このため、キーはハッシュ化されなければならなく、リストのような変更可能なものであってはいけない訳である。このような制限を回避する方法は次章で触れるタプルを使うことである。

辞書自体は変更可能なのでキーには使えないが、値としては使える。

11.6　メモ

6.7 節の関数 `fibonacci` で引数の大きな数を与えれば与えるほど実行にかかる時間が長くなることに気が付く。さらに大きな数では計算時間が急激に増える。この理由を理解するために、引数 `n=4` で `fibonacci` を実行した場合を調べてみる。それには図 11.2 のような**呼び出しグラフ（call graph）**を作成してみるとよい。

呼び出しグラフは関数フレームの集合を示す。そこでは呼びだされた関数とその関数を呼んだ関数が線で繋がれている。グラフの頂点は引数 n=4 の fibonacci で、これが引数 n=3 と n=2 の fibinacci を呼んでいる。替わって、引数 n=3 の fibinacci は引数 n=2 と n=1 を呼んでいる。等々である。

このようなグラフから fibonacci(0) や fibonacci(1) が全体で呼ばれている回数を調べてみよう。これからこの fibonacci プログラムはこの問題の解答としては不充分であることが分かる。この欠点は引数が大きくなるほど強調される。

この解決法の一つは既に計算し終えた値を辞書に保存し値の軌跡を残して置くことだ。このように計算した結果を後のために保存することを**メモ（memo）**と呼ぶ。

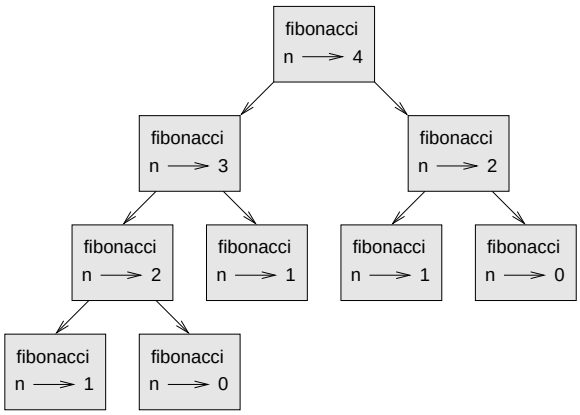


図 11.2: 呼び出しグラフ

以下はそれを使った関数 fibinacci の実装である：

```
known = {0:0, 1:1}

def fibonacci(n):
    if n in known:
        return known[n]
    res = fibonacci(n-1) + fibonacci(n-2)
    known[n]=res
    return res
```

大域変数 known は辞書で既に計算したフィボナッチ数を保存して置く。この辞書の初期値は 0 を 0 にマップ、1 を 1 にマップさせるものである。関数 fibinacci が呼ばれる度に辞書 known が検索される。結果があればそれを使う。なければ計算してその結果を辞書に残すと共に戻り値として返す。　この新たな fibonacci 関

数を実行してみると、さらに従来の fibinacci 関数と比較してかなり速いことを実感するはずだ。

11.7　大域変数

前節の例では変数 known は関数の外で生成された。これは__main__と呼ばれる特別なフレームにこの変数が属していることを意味している。__main__に属する変数は**大域変数（global variable）**と呼ばれる。この変数は任意の関数からアクセスすることができるからだ。関数に属するローカル変数が関数の終了と共に消失するのと異なり、大域変数は一つの関数呼ばれ、次が呼ばれようと残る。

大域変数としてよく使われるのがある条件が満たされたどうかを表示する**フラグ（flags）**機能を担った変数だ。以下 verbose というフラグを使い詳細表示を制御する例である：

```
verbose = True

def example1():
    if verbose:
         print('Running examle1')
```

もしこの大域変数の再代入をしようとすると予想外のことが起こる。

以下はある関数が呼ばれたことがあるかを追跡することを意図したプログラムである。

```
been_called = False

def example2():
    been_called = True    #間違い
```

このプログラムを実行すると、大域変数の値が変化していないことに気が付くはずだ。問題は関数 example2 では新たなローカル変数 been_called が作成されたことにある。関数が終了するとこのローカル変数は消失し、大域変数の値は何も変化はない。

関数の内部で大域変数へ再代入をしたいときは使う前に大域変数の宣言（declare）をする：

```
been_called = False
```

```
def example2():
    global been_called
    been_called = True
```

global 文はインタプリタに、「この関数では、been_called と呼ばれるものは大域変数で、ローカル変数として生成したものでないよ」と伝える役目をしている。
大域変数の更新についても同様だ：

```
count = 0


def example3():
    count = count + 1  #間違い
```

このまま実行すると以下のエラーが出る：

```
UnboundLocalError: local variable 'count' referenced before assignment
```

Python は変数 count をローカルとみなし、代入の前に参照していることでエラーを吐き出す。解決法は count を大域変数として宣言することだ：

```
def example3():
    global count
    count += 1
```

大域変数が変更可能なものであると、宣言なしで変更できる：

```
known = {0:0, 1:1}


def example4()
    known[2] = 1
```

従って大域リストに対しては、要素の追加、削除、変更は宣言なしでできる。しかし、再代入では宣言が必要となる：

```
def example5():
    global known
    known = dict()
```

大域変数は便利なものであるが、多用しその値を頻繁に変更するとデバッグが難しくなる。

11.8　デバッギング

扱うデータ（データセット）の量が多くなるに従って否応なしに表示によるデバッグや手作業によるデータのチェックが必要になる。ここではそのようなデータが多いプログラムに対するデバッグ作業のいくつかの示唆を示す：

入力のスケールダウン： もし可能なら、入力するデータ量を減らしてみる。例えば、テキストファイルを読むプログラムであるなら、最初の 10 行だけ読んでみる。または最少単位のデータを用意する。これはファイル自体を修正するか、もっといい方法はプログラムを修正して最初の n 行だけ読むようにする。

それでもエラーがあるときには、そのエラーを明らかにできる更に小さいデータ量で実行してみる。エラーが修正できたら、データ量を徐々に多くしてみる。

要約的把握や型の確認： データセットの全てを表示し、確認作業をする替わりに、例えば、辞書の要素の数や数のリストの総和といったデータの要約的な量を表示してみる。また、実行時のエラーは値が正しい型でないことからくることが多々ある。このような場合は単に値の型を表示してみることで済む場合がある。

自己点検の書き込み： 自己点検できるような機能をコードに書き込むことができる。例えば、数値のリストの要素の平均値を計算しているとしよう。この平均値はこのリストの最大要素の値より小さいはずであり、最小要素の値より大きいはずである。このような検証は結果が「不健全」であることを検出するから、「健全性の検証」と呼ばれている。

二つの異なった方法で得られた結果を比較するという検証はそれらが一貫しているかどうかの検証になる。これは「一貫性の検証」と呼ばれている。

出力を綺麗に表示： デバッグのための表示を綺麗に表示することはエラーの個所を特定することに役に立つ。その例を 6.9 節でみた。モジュール pprint の pprint 関数は組み込み型をより人間に読みやすい形式で表示する。（pprint は “pretty print” の略称である。）

足場建設のために費やした時間はデバッギングで消費する時間を縮めることができるのだ。

11.9 語句

写像（mapping）： 一つ集合の各要素がもう一つの集合の要素に対応している関係。

辞書（dictionary）： キーの集合から対応する値への写像。

キーと値のペア（keys-value pair）： キーから値への写像に具体的な表現。

アイテム（item）： 辞書におけるキーと値のペアの別名。

キー（keys）： 辞書のキーと値のペアにおいて対の最初に現れるオブジェクト。

値（values）： 辞書のキーと値のペアにおいて対の二番目に現れるオブジェクト。これはこれまで使っていた「値」よりもっと特定の状況での「値」の使い方である。

実装（implementation）： 計算の実際の実行方法。

ハッシュ表（hashtable）： Python の辞書を実装するためのアルゴリズム。

ハッシュ（hash）： キーの場所を計算するためにハッシュ表を用いること。

ハッシュ可能（hashable）： ハッシュ関数を持つデータ型。整数、浮動小数点型そして文字列のような変更不可の型はハッシュ可能であり、リストや辞書はそうでない。

ルックアップ（look up）： 一つキーを受け取りそれに対応する値を探すという辞書における操作。.

逆ルックアップ（reverse lookup）： 一つの値を受け取りその値にマップしている一つまたはそれ以上のキーを探すという辞書の操作。

シングルトン（singleton）： 一つの要素のみを含むリスト（または他の配列）。

呼び出しグラフ（call graph）： 一つのプログラムの実行過程で生成される各フレームで関数の呼び出し側と呼ばれた関数側とを矢印で結ぶグラフ表現。

メモ（memo）： 余計な計算を回避するために計算を終えた結果を保存して将来に備える。

大域変数（global variable）：: 関数の外で定義される変数。その変数は他の関数からもアクセスできる。

フラグ（flags）： 条件が満たされているかどうかを表示するために用いられるブーリアン変数。

宣言（declare）： `global` のように変数の属性をインタプリタに知らせる文。

11.10　練習問題

練習問題 11.1 ファイル `words.txt` を読み込み、英単語をキーとする辞書を作成せよ。値を何にするかは問わない。そして演算子 `in` を使ってある文字列がその辞書にあるかどうかを調べる関数を作成せよ。

もし練習問題 10.10 をやっていれば、二分探索とリストに対する演算子 `in` の実装の違いによる検索スピードを比較できる。

練習問題 11.2 辞書にたいするメソッド `setdefaut` のドキュメンテーションを読み、関数 `invert_dict` をコンパクトに書け。
解答例：https://thinkpython.com/code/invert_dict.py.

練習問題 11.3 練習問題 6.2 で扱った `Ackermann` 関数に対してメモ機能を付けたものせよ。そして、このメモ機能があると大きな値の引数でも関数の評価が可能かどうかを確認せよ。ヒント：なし。
解答例：https://thinkpython.com/code/ackermann_memo.py.

練習問題 11.4 練習問題 10.7 ではリストを引数として受け取りそのリストの要素が重複しているときは `True` を返す関数 `has_duplicates` を作った。辞書を利用してより高速でコンパクトな改訂版関数 `has_duplicates` を作成せよ。
解答例：https://thinkpython.com/code/has_duplicates.py

練習問題 11.5 二つの単語は一つを回転したら他方に一致するときこの二つの単語は「回転ペア」と呼ぶことにする。`words.txt` を読み込んで全ての「回転ペア」検出するプログラムを作成せよ（練習問題 8.5 の関数 `rotate_word` を参照せよ）。
解答例：https://thinkpython.com/code/rotate_pairs.py

練習問題 11.6 これも CarTalk のパズル名人から借用した。
(http://www.cartalk.com/content/puzzlers)
Dan O'leary という人物から送られてきたものだ。

> 五文字からなる一音節の単語を思いついたが以下のような面白い性質を持っているというのだ。まず、先頭の文字を削除すると四文字の単語になるがこれが元の単語と同音異義語である。つまりこの二つの単語の発音は厳密に一致するのだ。さらに、この文字を元のところにもどして、二番目の文字を削除すると四文字の単語になるが、これも元の単語の同音異義語になる。問題は、この元の単語はなにかだ。
>
> さて、成り立たないが例を示す。単語 `wrack` つまり `W-R-A-C-K` を例にとろう。この単語は “wrack with pain”（痛さに卒倒する）などに使う言葉だ。先頭の一文字を削除すると `R-A-C-K` になる。“Holy cow, did you see the rack on that buck! It must have been a nine pointer!”（おやまあ、あの雄ジカの角を見たかい。あれはナインポインタに違いないよ）などにみるね。そしてこの言葉は完全な同音異義語である。さて、’w’ を元に戻し、二番目の文字を削除すると、“wack”が得られる。これも実在する単語であるが、この言葉は前二者と同音異義語になっていない。
>
> しかし、Dan やわれわれが知っている単語で少なくとも一つは条件を満たす単語がある。その単語は何か？

ある文字列が単語リストにあるかどうかを調べる目的で練習問題 11.1 の辞書を使うことができる。二つの単語は同音異義語であるかどうかの検定には CMU 発音辞典を使うことができる：http://www.speech.cs.cmu.edu/cgi-bin/cmudict または https://thinkpython.com/code/c06d からダウンロードできる。また、https://thinkpython.com/code/pronounce.py をダウインロードできる。この中の `read_dictionary` は発音辞典から単語とその単語の主要な発音を Python の辞書として戻す関数である。

パズル名人のこの問題を解くプログラムを作成せよ。

解答例：https://thinkpython.com/code/homophone.py

第12章　タプル

この章ではさらにもう一つの組み込み型であるタプルを取り上げ、その後リスト、辞書そしてタプルが如何に相互に働くかを見ることにする。また可変長引数リスト、「纏める」そして「ばらす」といった操作の有益な特徴にも触れる。

閑話休題：“tuple”をどう発音するか一致して見解はない。“supple”と韻が同じように “tuhple” と発音する人たちがいる。一方プログラミングの中身を踏まえて多くの人たちは “quadruple”と韻が同じで “too-ple”と発音する。

12.1　タプルは変更不可

タプル（tuple）は値の配列である。その値は任意の型でよい。また、整数でインデックス化されている。これらの点で、タプルはリストに似ている。大きな相異は、タプルは変更不可なことである。構文的にはタプルは値をカンマで区切った配列である：

```
>>> t = 'a', 'b', 'c', 'd', 'e'
```

必ずしも必要でないが、タプルを括弧で括ることが普通である：

```
>>> t = ('a', 'b', 'c', 'd', 'e')
```

一つの要素だけでタプルを生成するときには最後にカンマが必要だ：

```
>>> t1 = 'a',
>>> type(t1)
<class 'tuple'>
```

値を括弧で括ったものはタプルではない：

```
>>> t2 = ('a')
>>> type(t2)
<class 'str'>
```

組み込み関数 tuple を使って空のタプルを作ることができる。引数なしでそれを使う：

```
>>> t = tuple()
>>> t
()
```

もしも引数が配列（文字列、リスト、タプル）であるとすると、結果はその配列要素を要素とするタプルが得られる：

```
>>> t = tuple('lupins')
>>> t
('l', 'u', 'p', 'i', 'n', 's')
```

組み込み関数名 tuple を変数として使うことは避けるべきである。

リストに対して使った演算はタプルで有効だ。角括弧は要素のインデックスとして使える：

```
>>> t = ('a', 'b', 'c', 'd', 'e')
>>> t[0]
a
```

スライス演算も区間要素を選択するのに使える：

```
>>> t[1:3]
('b', 'c')
```

しかし、タプルの要素を変更することはできない：

```
>>> t[0] = 'A'
TypeError: object doesn't support item assignment
```

しかし別なタプルに置き換えることはできる：

```
>>> t = ('A',) + t[1:]
>>> t
('A', 'b', 'c', 'd', 'e')
```

つまりこの文は新規にタプルを作り変数 t にそれを参照させている。

比較演算子はタプルを初めてとしてリストにも適用できる。Python は初めの要素を比較する。等しいならば次ぎの要素を比較する。異なる要素に出会うまでこれを繰り返す。ここでの比較が結果である。だから以降の要素は無視される。

```
>>> (0, 1, 2) < (0, 3, 4)
True
>>> (0, 1, 2000000) < (0, 3, 4)
True
```

12.2 タプルの代入

二つの変数の交換は有用なことが多々ある。伝統的な手法は一時的な変数を使うものである。変数 a と b の交換を例にみよう：

```
>>>temp = a
>>>a = b
>>>b = temp
```

タプルの代入（tuple assignment）はもっとエレガントが手法を提供している：

```
>>>a, b = b, a
```

左辺は変数のタプルであり、右辺は表式のタプルである。各要素が対応する変数に代入される。右辺の表式は代入する前に評価される。

左辺の変数の個数は右辺の値の個数と一致している必要がある。

```
>>> a, b = 1, 2, 3
ValueError: too many values to unpack
```

もっと一般的には、右辺は任意に配列（文字列、リスト、タプル）でもよい。電子メールアドレスをユーザ名とドメイン名に split を用いて分離する例を示す：

```
>>> addr = 'monty@python.org'
>>> uname, domain = addr.split('@')
```

右辺は二つの要素を持つリストであり、先頭の要素は uname に代入され、二番目は domain に代入される。

```
>>> uname
monty
>>> domain
python.org
```

12.3　タプルを戻り値

厳密にいうと関数の戻り値は一つの値のみであるが、それがタプルであると多数の値を戻り値にする効果がある。

例を示す。二つの整数のわり算で商と余りを出す問題を考えよう。組み込み関数 divmod を使うと二つ整数のわり算の商と余りをタプルとして戻してくれる。だから、この二つをタプルに代入することができる。

```
>>> t = divmod(7, 3)
>>> t
(2, 1)
```

または要素毎の代入でもよい：

```
>>> quot, rem = divmod(7, 3)
>>> quot
2
>>> rem
1
```

以下は簡単な例である：

```
def min_max(t):
    return min(t), max(t)
```

min、max は配列の最大と最小の要素を見つける組み込み関数である。関数 min_max はその両方を計算し、それを一つのタプルとして戻している。

12.4　可変長引数タプル

関数は可変長引数を受け取りことができる。記号*で始まる仮引数は複数の引数を一つのタプルに**纏める（gathers）**。例えば、関数 print_all は任意の数の引数を受け取り、それらを print する：

```
def print_all(*args):
    print(args)
```

纏め引数の名前は自由に取れるが伝統的に arg を使う。この関数が正確に動くことを確かめる：

```
>>> print_all(1, 2.0, '3')
(1, 2.0, '3')
```

「纏める」の補語は**ばらす（scatter）**である。値の配列があり、それを複数の引数を受け取る関数に通したいときは、記号*を使う。例として、divmod 関数は厳密に二つの引数を要求するので、一つのタプルではエラーになる：

```
>>> t = (7, 3)
>>> divmod(t)
TypeError: divmod expected 2 arguments, got 1
```

しかし、タプルをばらすと二つの引数になり正常に動く：

```
>>> divmod(*t)
(2, 1)
```

組み込み関数の多くは可変長引数タプルを使っている。例を挙げれば、min、max も任意の長さの引数を受け取ることができる。

```
>>> max(1, 2, 3)
3
```

しかし、関数 sum はそうでない。

```
>>> sum(1, 2, 3)
TypeError: sum expected at most 2 arguments, got 3
```

（訳注：関数 sum は配列を受け取りその要素の和を戻す。）
そこで任意の引数を受け取り、その総計を返す関数 sum_all を作成せよ。

12.5 リストとタプル

組み込み関数 zip は引数として二つ以上の配列を受け取り、それらの配列の各要素を一つずつ組みにして「纏めて」タプルとし、それらのリストを返す。この関数の名前は二つの歯を交互に組み合わせるジッパーに由来している。

```
>>> s = 'abc'
>>> t = [0, 1, 2]
>>> zip(s, t)
<zip object at 0x7f7d0a9e7c48>
```

結果は値のペアを繰り返し処理することに特化した**ジッパーオブジェクト（zip object）**である。最も一般的な zip の使い方は for ループの中の in 演算子によるものである：

```
>>> for pair in zip(s, t):
...     print(pair)
...
('a', 0)
('b', 1)
('c', 2)
```

ジッパーオブジェクトは配列を横断的に繰り返す**イテレータ（iterator）**の一種である。イテレータはある面でリストに似ているが、リストと異なり、イテレータの要素を選択するときインデッススは使えない。

もしもリストの演算やメソッドを使いたいのであれば、ジッパーオブジェクトをリストにするとよい：

```
>>> list(zip(s, t))
[('a', 0), ('b', 1), ('c', 2)]
```

ジッパーオブジェクトを生成するとき、もし二つの配列の長さが一致しないときは短い方に合わせる：

```
>>> list(zip('Anne', 'Elk'))
[('A', 'E'), ('n', 'l'), ('n', 'k')]
```

このようなタプルを要素とするリストに対してはリストの for を使った横断的な処理ではタプルへの代入を使うことができる：

```
>>> t = [('a', 0), ('b', 1), ('c', 2)]
>>> for letter, number in t:
 print(letter, number)
```

ループは回る度に Python はタプルを選択し、そのタプルの要素を letter と number に代入する。結果は以下のようになる：

```
a 0
b 1
c 2
```

これら関数 zip, for 文, そしてタプルの代入を使うと二つ以上の配列を一時に横断的調べることができる仕掛けが作れる。関数 has_match を例に示す。この関数は二つの配列の同じインデックスの要素が少なくとも一つは一致するかどうかを調べるものである。

```
def has_match(t1, t2):
    for x, y in zip(t1,t2):
        if x == y:
            return True
    return False
```

配列を横断的に眺めると同時にその要素のインデックスも必要なときには、関数 enumerate を使うとよい：

```
for index, element in enumerate('abc'):
    print index, element
```

このループの表示は以下のようになる：

```
0 a
1 b
2 c
```

12.6 辞書とタプル

辞書には items というメソッドがある。これは辞書のキー・値ペアをタプルとしたリストを返す関数である。

```
>>> d = {'a':0, 'b':1, 'c':2}
>>> t = d.items()
>>> t
dict_items([('c', 2), ('a', 0), ('b', 1)])
```

結果は dict_items オブジェクトで、キーと値のペアを繰り返し処理するイテレータである。このオブジェクトは以下のように for 文のなかで in 演算子を使う：

```
>>> for key, value in d.items():
...     print(key, value)
...
```

```
c 2
a 0
b 1
```

辞書で予想できたように並びは順不同である。
　逆方向の操作もある。タプルのリストを新規辞書に変換する：

```
>>> t = [('a', 0), ('c', 2), ('b', 1)]
>>> d = dict(t)
>>> d
{'a': 0, 'c': 2, 'b': 1}
```

この関数 dict と関数 zip を組み合わせると大変簡明な辞書生成の方法ができあがる：

```
>>> d = dict(zip('abc',range(3)))
>>> print d
{'a': 0, 'c': 2, 'b': 1}
```

辞書のメッソド update はタプルのリストを受け取り既存の辞書に追加する。
　タプルを辞書のキー（リストはこの目的には使えないので）に使うことがよくある方法である。例えば、ファーストネームとラストネームを対にして辞書のキーとして使い、電話番号を値とする辞書である。それらを first,last,number とすると以下のように書くことができる：

```
dictionary[last,first] = number
```

角括弧内の表式はタプルである。このような辞書の横断的な処理にもタプルの代入が使える。

```
for last, first in dictionary:
    print first, last, dictionary[last, first]
```

このループはタプルであるキーを横断的に眺める。各タプルの組みを last,first に代入し、それに対応する辞書の値を print する。

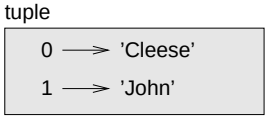


図 12.1: 状態図

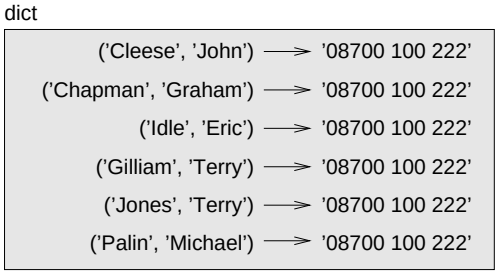


図 12.2: 状態図

この辞書のキーのタプルに対する状態図を描く方法には二つある。詳しい版ではリストと同じようにインデックスとそれが示す要素を書く。例として、(Cleese, John) のタプルを図 12.1 に示した。もっと大きな図ではそのような細部は無くてもよいと思うだろう。電話帳の例を図 12.2 に示した。そこでは Python で使うタプルの表式そのものを使った。対応する電話番号は BBC の苦情窓口のものだ。電話しないでほしい。

12.7 配列の配列

ここまではタプルのリストの話を進めてきたが、これらはリストのリスト、タプルのタプル、リストのタプルなどにも大凡適用できる。重複を避けるために配列の配列として話を纏めることにする。

多くの状況で、異なった形式の配列（文字列、リスト、タプル）は相互に交換可能である。それではどのようにして、何故に他の形式より、ある配列形式を好ましいとして選択すべきだろうか？

自明なことであるが、文字列は要素が文字のみであるということから他の形式の配列より制限がきつい。それらは変更不可でもある。もし文字の一部を変更したいと思ったら、文字列の替わりに文字のリストを使うに違いない。

リストは変更可能ということから、タプルより一般的と言える。しかし、タプルの方が好ましいと思われるいくつかの場合がある：

1. 例えば return 文のようないくつかの状況では、リストよりタプルで処理する方が構文的に簡単になることがある。その他の状況ではリストが好ましいかもしれない。

2. もし配列を辞書のキーとして使いたいなら、変更不可のタプルや文字列を使わざるをえない。

3. もし配列を関数の引数に渡したい場合があるなら、リストによる別名処理の弊害を避けるためにタプルを使った方が安全だ。

タプルは変更不可なので、既存のリストの変更を伴う sort や reverse などのメソッドは存在しない。しかし、sorted や reversed などの関数が用意されていて任意の配列を受け取り新しい配列を返すことができる。

12.8　デバッギング

リスト、辞書、タプルは一般にデータ構造（data structure）として知られているが、この章ではタプルのリスト、タプルをキーとしてリストを値とする辞書のような複合データ構造を初めてとりあげた。複合データ構造は有用だが、私が**型エラー（shape errors）**と名付けたエラーに陥りやすい。このエラーはデータ構造の間違った型や大きさ、複合の仕方が原因として起きる。

例えば、一つの整数を要素とするリストを期待していたのに、普通の整数が与えられたとすると、動かなくなるといったものだ。この種のエラーをデバッグするために、structshape というモジュールを作り、関数 structshape が使えるようにした。この関数は任意のデータ構造を引数として受け取りその型について纏めた文字列を返す関数である。ダウンロード先は https://thinkpython.com/code/structshape.py である。

以下いくつかの簡単な例を示す：

```
>>> from structshape import structshape
>>> t = [1,2,3]
>>> structshape(t)
list of 3 int
```

もっと凝ったプログラムでは “list of 3 ints” と書くかもしれない。しかし、複数形を斟酌しない方が簡単だ。
リストのリストだと以下のようになる：

```
>>> t2 = [[1,2], [3,4], [5,6]]
>>> structshape(t2)
list of 3 list of 2 int
```

もしも要素が異なった型であるとすると、structshape は型毎に纏めて表示する：

```
>>> t3 = [1,2,3,4.0,'5','6',[7],[8],9]
>>> structshape(t3)
list of (3 int, float, 2 str, 2 list of int, int)
```

タプルのリストでは：

```
>>> a = 'abc'
>>> lt = zip(t,a)
>>> structshape(lt)
list of 3 tuple of (int, str)
```

整数をキーとして文字列を値とする三つの要素を持つ辞書では

```
>>> d = dict(lt)
>>> structshape(d)
dict of 3 int->str
```

データ構造の追跡で何か問題が起きたら、この関数 `structshape` が有効だ。

12.9 語句

タプル（tuple）： 要素を変更できない配列。

タプルの代入（tuple assignment）： 右側に一つの配列、左側に複数の変数のタプルを置く代入文。右側が評価され対応する要素（複数の値からなる）が左側のタプルに代入される。

纏める（gathers）： 可変長引数タプルを纏める操作。

ばらす（scatter）： 関数の引数（複数）を一つの配列として扱う操作。

ジッパーオブジェクト（zip object）： 組み込み関数 `zip` によって生成されるオブジェクトでタプルを要素とする配列の形式をとり繰り返し処理に特化したオブジェクト。

イテレータ（iterator）： 配列の形式をとり繰り返し処理に特化したオブジェクト、しかしリストに対する演算やメソッドは持っていない。

データ構造（data structure）： リスト、辞書、そしてタプル等のかたちに纏められた値の集合。

型エラー（shape errors）： データ構造のような複雑に纏められたデータにアクセスする際に起こる型の不一致によるエラー。

12.10　練習問題

練習問題 12.1 文字列を受け取りその文字列に含まれている文字の頻度を降順に表示するプログラム most_frequent を作成せよ。色々な言語で書かれた文書を調べ文字の頻度分布が言語でどのように異なるかを調べ、以下の文献と比較せよ：
http://en.wikipedia.org/wiki/Letter_frequencies。
解答例：https://thinkpython.com/code/most_frequent.py

練習問題 12.2 アナグラム再論！

1. ファイルから単語集を読み込み（9.1 節をみよ）、アナグラムになっている単語の全てを表示するプログラムを作成せよ。
   どんな出力になるのかを例で示す：

   ```
   ['deltas', 'desalt', lasted', 'salted', 'slated', 'staled']
   ['retainers', 'ternaries']
   ['generating', 'greatening']
   ['resmelts', 'smelters', 'termless']
   ```

   ヒント：文字のセットとそれらの文字から作れる単語のリストとを写像する辞書を作ることになるかもしれない。問題はいかに文字のセットをキーとする辞書を作るかである。

2. 前のプログラムを修正して最大の数の単語を含むアナグラムを最初に表示し、次ぎは二番目というような順序に表示にせよ。

3. スクランブル（単語ゲーム）では、ボード上にある一文字とラックに用意された七個の文字タイルを全て使って八文字の長さの単語が作れると「ビンゴ」になる。どんな八文字のセットが最も「ビンゴ」になりやすいか？

ヒント：七つの単語が作れるものがある。
解答例：https://thinkpython.com/code/anagram_sets.py

練習問題 12.3 一つ単語の中にある二文字を入れ替えると他の単語になるときこの二つの単語は「字位転換ペア（metathesis）」という。辞書中の全ての字位転換ペアを見つけ表示するプログラムを作成せよ。
ヒント：単語集の全てのペアを検証するな、そして、一つの単語の可能な交換の全てを検証するな。（訳注：その心は含まれて文字が同じもののみ、つまりアナグラムになっていて二個所で文字が異なる単語だ）。
解答例：https://thinkpython.com/code/metathesis.py

出典：この練習問題は以下の例題に啓発された：http://puzzlers.org

練習問題 12.4 これも Car Talk のパズル名人の問題である。(http://www.cartalk.com/content/puzzlers)

> 英単語があり、それを構成している文字を一回に一文字を削除して文字を詰めると新たな単語になるような最も長い単語はなにか？さて、文字の削除は先頭でも中程でもよいが、文字の並び替えはできない。文字を削除したら単に文字を詰めてみると他の英単語になっている。これが成功したら、その新しい単語から一文字削除し、詰めると英単語が現れる。最後は一文字だけになるがこれも辞書にある英単語である。このような英単語の中で最も長い単語はなにか、また何文字からなるか？これが知りたいことである。

平凡で短い単語 sprite で例を示す。sprite で始める。まず、文字 r をとる、するとspite が作られる。さらに、文字 e をとると、spit ができる。またさらに、文字 s をとると、pit が生成され、文字 p をとると it が、さらに文字 t をとると I が出現する。最後の I も辞書にある単語だ。

上の例のように最終的に一文字の英単語に縮小できる（全縮小可能と呼ぶ）全ての英単語を見つけるプログラムを作成せよ、そして、その中で最も長い単語をみつけよ。この練習問題は今までのものより難しいので、いくつか示唆を与えておこう。

1. 単語を引数として受け取り、それから一文字削除してできる単語（子ども）の全てをリストにして返す関数を生成してみる。

2. ある単語はもしその子どもの一つでも全縮小可能ならば再帰的に全縮小可能である。そこでは空の文字列を再帰の基底ケースと考える。

3. 今まで使ってきた `words.txt` には一文字の単語が含まれていないので、“I”、“a” と空文字をそれに加える。

4. プログラムの効率を高めるために全縮小可能単語を記憶しておくことが考えられる。

解答例：https://thinkpython.com/code/reducible.py.

第13章　事例研究：データ構造・選択

この時点までで Python の中核になるデータ構造を学んだことになり、そしてそれらを使う上で有用な幾つかのアルゴリズムを学習した。もしもあなたが更にアルゴリズムについて深く学びたいのであれば、付録 B を読んでみるよい機会かもしれない。しかしこれからの議論の前にそれを読まなければならないというわけではない。それはあなたが興味を持ったときに読めばよい。

この章ではデータ構造の選択やそれらの使い方を考える練習問題を伴なった事例研究を取り上げる。

13.1　単語頻度分布解析

いつものように解答例をみる前に、少なくとも解答を試みるようにしてほしい。

練習問題 13.1 ファイルからテキストを読み、単語に分解し、区切り文字や句読点を取り除き且つ大文字を全て小文字に変換するプログラムを作成せよ。
ヒント：モジュール `string` は空白、タブ、改行文字等を含む文字の定義 `whitespace` や句読点を集めた文字の定義 `punctuation` を提供している。確かめてみよう：

```
>>> import string
>>> print string.punctuation
!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~
```

さらに、文字列に対するメソッド `strip`, `replace`, `translate` なども使える。

練習問題 13.2 グーテンベルグ・プロジェクト（http://gutenberg.org）から好みの本をテキストベースでダウンロードして Python で読めるようにせよ。そして、使われている単語を表示してみる。異なった時代、異なった著者の異なった著書で得られた結果を比較せよ。どの著者が最も多くの語彙を使っているか？

練習問題 13.3 前の練習問題を修正して取りあげた本で使われている最頻度度二十番目までの単語を表示せよ。

練習問題 13.4 第 9.1 節の単語集（words.txt）を読み込み、取りあげた本で使っている単語でこの単語集にないものを表示せよ。誤植はいくつあるか？単語集に掲載すべき通常の単語はいくつあるか？はっきりしないものはいくつあるか？

13.2　乱数

ある与えられた入力に対してコンピュータプログラムは常に同一の結果を出力する。この状況を**決定論的（deterministic）**と呼ぶ。われわれは常にこのような状況を期待しているので、決定論的な処理は良好だと思っている。しかし、ある種のアプリケーションではコンピュータに予測不可な振る舞いを期待する。ゲームはその明白な例であるがそれ以外にもある。

プログラムで真に非決定論的な振る舞いを実現することは易しくないが、非決定論的な振る舞いにみえるものを作るには方法がある。その一つが**疑似乱数（pseudorandom number）**を生成するアルゴリズムである。決定論的なプロセスで生成されるので、疑似乱数は真の意味では乱数ではないが、みた目には殆ど乱数と区別することは不可能だ。

モジュール `random` は疑似乱数を生成する関数を提供している（これ以降単に「乱数」と呼ぶことにする）。関数 `random` は区間 `0.0` から `1.0` までの間の乱数（`0.0` は含み `1.0` は含まない）を生成する。関数 random を呼び出す毎に長い数列の中の次ぎの数が得られる。例をみるために以下を実行してみよう：

```
import random
for I in range(10):
    x = random.random()
    print(x)
```

関数 `randint` は二つの引数 low と high を受け取り、この low と high（両方とも含む）の区間の整数乱数を生成する：

```
>>> random.randint(5, 10)
5
>>> random.randint(5, 10)
9
```

配列の要素を乱雑の選択するのであれば関数 `choice` が使える：

```
>>> t = [1, 2, 3]
>>> random.choice(t)
2
>>> random.choice(t)
3
```

モジュール random はガウス分布、指数分布、ガンマ分布などの連続分布関数に従って乱数を生成する関数を提供している。

練習問題 13.5 第 11.2 節で定義された頻度分布を引数として受け取り、その頻度分布に従って値を選択し、その値を戻り値とする関数 choose_from_hist を作成せよ。例を示す：

```
>>> t = ['a', 'a', 'b']
>>> hist = histogram(t)
>>> print(hist)
{'a':2, 'b':1}
```

従ってこの関数は'a' を 2/3 の確率、'b' を 1/3 の確率で選択しなければならない。

13.3 単語ヒストグラム

前節の演習問題は自分で解答を試みてほしいが、解答例は以下にある。
https://thinkpython.com/code/analyze_book1.py
この解答例で使った文献も必要だ (https://thinkpython.com/code/emma.txt.)。

以下はファイルから読み込んだ文献中の単語のヒストグラムを作成するプログラムである：

```
import string

def process_file(filename):
    hist = dict()
    fp = open(filename)
    for line in fp:
        process_line(line, hist)
    return hist
```

```
def process_line(line, hist):
    line = line.replace('-',' ')
    for word in line.split():
        word = word.strip(string.punctuation+string.whitespace)
        word = word.lower()
        hist[word] = hist.get(word, 0) + 1

hist = process_file('emma.txt')
```

このプログラムでは Jane Austen 著 “Emma” をプレーンテキストにしたファイル emma.txt を読み込む。関数 process_file 中のループはファイルから一行読みその都度それを関数 process_line に渡している。変数 hist は加算機の役割をしている。関数 process_line では split を用いて単語に分解する前に replace を用いて一行の文字列中に含まれるハイフンを空白に置き換えている（訳注：これは合成語の処理。ハイフネーションに対しては別の処理が必要）。その後に、単語のリストを strip で句読点を削除、lower で全ての文字を小文字に変換する処理を横断的に行っている（文字列が「変換された」というのは速断である。文字列は変更不可である、だから strip や lower では新たな文字列が生成されたのだ）。最後に process_line は新たなアイテムを追加または既存のアイテムの値を 1 増加するかして更新処理をする。

ファイル中の単語の総数は辞書の頻度の総和を求めることである：

```
def total_words(hist):
    return sum(hist.values())
```

異なった単語の総数は辞書の中のアイテムの数である：

```
def different_words(hist):
    return len(hist)
```

結果の表示の一例を以下に示す：

```
print('Total number of words:', total_words(hist))
print('Number of different words:', different_words(hist))
```

結果は：

```
Total number of words: 161080
Number of different words: 7214
```

13.4 頻度の高い単語

頻度の高い単語を検出するためには、単語とその頻度をタプルとするリストを作り、それをソートする。関数 most_common はヒストグラムを引数として受け取り、頻度で降順にソートし、頻度と単語をタプルとするリストを返す。

```
def most_common(hist):
    t = []
    for key, value in hist.items():
        t.append((value, key))
    t.sort(reverse=True)
    return t
```

各タプルで頻度が最初にきているので、ソートはこの頻度でソートされることになる。

降順にソートされているので先頭の十個のアイテムを表示してみる：

```
t = most_common(hist)
print('The most common words are:')
for freq, word in t[:10]:
    print(word, freq, sep='\t') #'\t' はタブ
```

Emma を解析した結果は以下のようになる：

```
The most common words are:
to      5242
the     5205
and     4897
of      4295
i       3191
a       3130
it      2529
her     2483
was     2400
she     2364
```

このコードは sort 関数のキー・パラメターを使うと簡単化できる。不思議に思うのであれば、Python で使う sort 関数の解説 https://wiki.python.org/moin/HowTo/Sorting. を参照せよ。

13.5　選択的な仮引数

これまで組み込み関数やメソッドの中に可変数の引数を受け取るものがあった。ユーザ定義関数でも選択的な仮引数を使うことでこれは可能だ。ヒストグラムから頻度の高い単語を表示する関数で例を示そう：

```
def print_most_common(hist, num=10):
    t = most_common(hist)
    print('The most common words are:')
    for freq, word in t[:num]:
        print(word, freq, sep='\t')
```

最初の仮引数は必須で、第二の仮引数は選択である。その仮引数 num の**既定値（default value）**は 10 である。もし引数一つのみでこの関数を呼び出すと、つまり、

```
print_most_common(hist)
```

変数 num の値はその既定値が使われる。もし二つの引数で呼べば、つまり

```
print_most_common(hist, 20)
```

変数 num の値は引数の値が使われる。換言すれば、選択的な引数を与えることは既定値を**無効にする（overrides）**わけである。

関数が必須引数と選択的な引数で構成されているときには、全ての必須引数をまず並べ、選択的な引数はそれに続けて並べる。

13.6　辞書の差し引き

ある本で見つかった単語で単語集 words.txt に存在しない単語を探したいという問題は一つの集合で他の集合にない全ての要素を見つける集合の差の問題である。関数 substract は二つの辞書 d1 と d2 を引数として受け取り、d2 にない d1 の要素を新たな辞書として返す関数である（新たな辞書はキーが重要で値は全て None とする）。

```
def substract(d1, d2):
    res = dict()
    for key in d1:
        if key not in d2:
            res[key] = None
    return res
```

辞書 d2 として words.txt を選び process_file で辞書化して使う：

```
words = process_file('words.txt')
diff = subtract(hist, words)

print("Words in the book that aren't in the word list:")
for word in diff:
    print(word, end=' ')
```

小説 Emma の結果は以下のようになる：

```
The word in the book that aren't in the word list are:
rencontre genlis jane's blanche woodhouses disingenuousness
friend's Venice apartment…….
```

いくつかは名前や所有格であり、“rencontre”のような最早普通には使われなくなったものもあるが、いくつかはリストに登録すべき普通の単語もある。

練習問題 13.6 Python は集合の演算をサポートする set と呼ばれるデータ構造を提供している。第 19.5 節でこれについて読むことができるし、http://docs.python.org/3/library/stdtypes.html#types-set. 文献を参照できる。本から抽出した単語で単語集にない単語を探すためにこの set を使ったプログラムを書け。

13.7 乱雑な単語選択

ヒストグラムに従って単語を乱雑に選択するためには、最も簡単な方法はその出現頻度に従って単語のコピーを作りリストの要素とし、そのリストに従い乱雑な要素を選択することであろう。つまり、

```
def random_word(h):
    t = []
    for word, freq in h.items():
        t.extend([word] * freq)
    return random.choice(t)
```

表式 [word]*freq は文字列 word を freq 個集めたリスト生成する。メソッド extend は append に似ているが引数が配列であることが違う。

上のアルゴリズムでも動く。しかし効率が悪いし、再構成されたリストは元の本程度の大きさになる。自明に近い改良は、リストはそのままで単語の選択を複数個同時選択させることだ。それにしてもリストは大きすぎだ。

別な解法を示す：

1. ヒストグラムのキーである単語をリストにする。
2. 単語の頻度の累積（練習問題 10.2 をみよ）を要素とするリストを作成する。したがってこの要素の最後の要素は全単語数、n になる。
3. 1 から n までの乱数を発生させる。二分探索法（練習問題 10.10）を使ってこの乱数を内挿値とする累積リストのインデックスを得る。
4. このインデックスに従って単語リストから単語を選択する。

練習問題 13.7 から乱雑に単語を選択するこのアルゴリズムを使ったプログラムを作成せよ。
解答例：https://thinkpython.com/code/analyze_book3.py.

13.8　マルコフ解析

本から乱雑に単語を選択して、並べると単語の並びができるが、それは文章にはなっていないだろう。例えばこうだ：

```
this the small regard harriet which knightley's it most things
```

連続する単語間に何も関係がないので、このような単語の並びは滅多に文章を作ることはない。例えば、現実の文章では、定冠詞 “the” の後には形容詞や名詞が続き、動詞や副詞がくることはない。

現実の単語間の関係を測定する方法の一つにマルコフ解析がある。ある単語の並びに対して、ある単語がそれに続く確率として表現される。例えば Eric, the Half a bee（訳注：歌詞の意味は http://en.wikipedia.org/wiki/Eric_the_Half-a-Be を参照のこと）の出だしはこうだ：

> Half a bee, philosophically, Must, ipso facto, half not be. But half the bee has got to be Vis a vis, its entity. D'you see?
>
> But can a bee be said to be Or not to be an entire bee When half the bee is not a bee Due to some ancient injury?

例えばこのテキストでは、句 “half the” の後には “bee” が常に続くし、句 “the bee” の後は、“is” か “has” かである。

マルコフ解析の結果はプレフィックス（“half the” や “the bee” のような）とサフィックス（“has” や “is” のような）の間の写像として表現される。

この写像が与えられると、まずあるプレフィックスから文章を始める。このプレフィックスに続くものは写像として許させるサフィックスの内から乱雑に一つ選ぶことで実現できる。次ぎには、このプレフィックスの最後の単語とサフィックスを繋いだものを新たなプレフィックスとして前と同じ手続きを繰り返す。

例えば、プレフィックスとして “Half a” で始めたとすると、サフィックスとしての候補は “bee” しかないので、この “bee” が続く。次のプレフィックスは “a bee” である。これに従うサフィックスの候補は “philosophically”、“be”、“due” である。この例ではプレフィックスの長さは二単語であるが、任意の長さのマルコフ解析ができる。この長さを解析のオーダーと呼ぶ。

練習問題 13.8 マルコフ解析

1. ファイルからテキストを読み、マルコフ解析を行え。結果はプレフィックスをキーとして可能なサフィックスをまとめたものを値とする辞書である。このまとめたものはタプル、リスト、辞書の形式があるが、これはあなたの選択に任せる。プログラムのテストではプレフィックスの長さは 2 でよいが、任意の長さに対応するようにせよ。

2. マルコフ解析に従ってテキストを乱雑に生成するプログラムを作り、前のプログラムに追加せよ。以下は Emma をオーダー 2 のマルコフ解析から得られた例である：

   > He was very clever, be it sweetness or be angry, ashamed or only amused, at such a stroke. She had never thought of Hannah till you were never meant for me?” ”I cannot make speeches, Emma:” he soon cut it all himself.

   この例では単語に付いていた句読点はそのままその単語に付けて用いた。結果は英語構文的にはほぼ、正確には充分でないが、正しい。意味論的には、ほぼ意味が通るが、正確には充分でない。

   マルコフ解析のオーダーを高くしたらどうなるがろう？もっと意味論的に意味が通るテキストができるのだろうか？

3. プログラムが動くようになったら、混合したらどうなるか確かめることにしたい。二つ以上の本をマルコフ解析して、それに従ってランダムテキストを

生成してみるとこれは解析に用いた本の単語や語句をブレンドしたものになるはずだ。

出典：Kernighan and Pike 著 “The Practice of Programming”(Addison-Wesley, 1999) の例に啓発されてこの事例研究を作成した。
例によって、練習問題は答えをみる前に解答を試みること。
解答例：https://thinkpython.com/code/markov.py. また解析に使ったテキストとして https://thinkpython.com/code/emma.txt. が必要。

13.9　データ構造

マルコフ解析によるランダムテキストの生成は面白いが、この事例研究はデータ構造の選択に関する問題も含んでいる。その選択では：

- プレフィックスをどのように表現するか？
- 可能なサフィックスの集合をどのように表現するか？
- 各プレフィックスからサフィックスの集合へのマッピングをどう表現するか？

最後の問題が最も簡単に決められそうである。つまり、辞書を使うことだ。プレフィックスに関しては、文字列、文字列のリスト、文字列のタプルなどが候補になるだろう。サフィックスに関しては、リストまたはヒスとグラム（辞書型）が考えられる。どのような基準で選択するか？

第一に考慮すべきは想定したデータ構造に対してしなければならない操作について検討することだ。新たなプレフィックスを生成するために、プレフィックスの先頭の単語を削除して、単語を一つ追加する操作が必要になる。例えば、今のプレフィックスは “Half a”であって、次ぎの単語が “bee”であったとすると、新しいプレフィックスは “a bee” となる（訳注：単語を一つずらして “Half” を消して “bee” を加える)。このように考えるとプレフィックスにはリストが便利なことが分かる。しかし、またプレフィックスは辞書のキーにもならなければならない。だとするとリストではダメである。タプルでは変更不可なので削除や追加はできないが、追加を関数で処理し、新しいタプルを生成することで代替できる：

```
def shift(prefix, word):
    return prefix[1:] + (word,)
```

関数 `shift` は単語のタプル prefix と文字列 word を引数として持ち、最初の単語を除外した単語のタプルと単語 word を追加した新しいタプルを返す。

サフィックスの集合に関しては、新たな要素を追加する（またはヒストグラムであると頻度を１上げる）、及びランダムにサフィックスを選択するという操作がある。リストからランダムに選択することは容易だが、ヒストグラムでは少し難しい操作（練習問題 13.7 をみよ）になる。

これまでは実装の容易さについてのみ考察したが、データ構造の選択は他の要素からも考察する必要がある。実行時間はその一つである。ある場合は理論的な推定からあるデータ構造が他のものより速いということが分かることがある。例えば、リストと比較して辞書にしておいた方が操作は速いことに既に言及した。しかし、多くの場合は前もってこのような実行時間の推定はできない。そのようなときに取る得る選択肢は双方のデータ構造を使って実行時間を実測してみることだ。これは**ベンチマーク・テスト（benchmarking）**と呼ばれている方法だ。

実際的には、実装することが最も容易なデータ構造を選択し、速度の点から実用に耐え得るかをみるのがよいだろう。もしよければこれ以上は必要ない。もしもそうでなければ、`profile` のようなツールを使ってプログラムで時間を食っている個所を特定してみるとよい。

考察の次ぎは記憶領域の消費の程度だ。例えば、サフィックスの集合をヒストグラムにすると保存すべきある単語がいかに多数回テキストに現れても一回だけの保存で済むのでリストより少ない記憶領域で処理できる。ある場合には少ない記憶領域で済むことは実行速度を速めることにもなる。メモリーを食い尽くしてしまったら動かなくなることもある。しかし、一般のアプリケーションでは記憶領域の大きさは速度に比較したら第二義的な問題だ。

最後にもう一つの考察をしよう。解析時とテキスト生成時のデータ構造は同じとしてきたが、これは二つが分離できることから、それぞれ異なったデータ構造を使う選択肢もある。一つのデータ構造を解析に用いて、それを必要に応じて変換したデータ構造でテキスト生成を行うわけである。テキスト生成時に於ける実行時間の節約が変換に必要な時間を超えていれば全体の時間は節約できる。

13.10 デバッギング

プログラムをデバッグしていて、特に見つけにくいバグに遭遇したときには以下の四つのことをしてみよう：

読め： 自分が書いたコードを吟味せよ。自分に言い聞かせるように読め。そして自分の言いたいことが書けているか調べよ。

実行せよ： プログラムを変更し、異なったバージョンを実行する実験をせよ。ときとして、しかるべきところで表示をしてみることで、問題が自明になると

いうこともある。しかし、このような足場を幾重にも組まなければならないときもある。

熟考せよ： 時間をかけて考えよ。それはどのようなエラーか、構文なのか、実行時なのか、意味的なエラーなのか？エラーメッセージまたはプログラムの表示から得られた情報は何なのか？どんなエラーが懸案の問題を引き起こすだろうか？問題が最初に現れたのは何をしたときか？

ラバー・ダックに話してみよ： 抱えている問題を他の人にはなしてみると、その問題を語り終える前に答えを発見してしまうことがある。あるばあいには他の人さえ必要としない。話相手はラバー・ダックでもよい。これが**ラバー・ダック・デバッギング（rubber duck debugging）**と呼ばれる戦略の由来である。作り話ではない。https://en.wikipedia.org/wiki/Rubber_duck_debugging. を見てほしい。

後退せよ： いくつかのタイミングで、最善の策は進行している変更を破棄し、プログラムがそれなりに動き、理解ができる時点のプログラムまで戻ることだ。そして、この状態からプログラムを再構築すればよい。

初心者はとかくこれらのやらなければならないことの一つのみに執着しがちだ。そして、他の可能性を忘れてしまう。これらの四つの行動は起因するエラーの種類に対応したものだ。

例えば、コードを詳しく読むことはコードが含むタイプミスを見つけるには役に立つが、問題が概念的な誤解からくるものであると役に立たない。自分がプログラムとして作っていることがらを理解していないならば、コードを百回読んでもエラーを見つけることはできない。エラーは自分の頭の中にあるのだから。

特に小規模で簡単なテストでプログラムを実行してエラーを探すことは有益である、しかし、考えもしないでコードを読まずに単に実行を繰り返すことは、私が「酔歩プログラミング」と呼ぶ欠陥に陥る。これはプログラムが正常に動くまで、適当に場当たり的に修正を繰り返すことになる。この酔歩プログラミングはデバッグに多大な時間がかかることは言うまでもない。考えることに時間を割くべきだ。デバッギングは実験科学のようなものだ。問題の所在について少なくとも一つの仮説を立てるべきだ。二つも三つもあるときは一つずつ潰していけばよい。

休憩は考察のために役に立つ、他の人に話をする機会もそうだ。問題を他の人、または自分自身でもよい、に説明しようとして、話が終わる前に答えが分かってしまうことがあるものだ。

エラーが大量で、小さいテストを実行するにはコードが膨大で複雑であるとこれら最善のデバッギングの技術を用いても失敗することがある。このような状況

では最善の選択肢は後退である場合がある。自分が理解でき、何かの手懸かりが掴まえられるまでプログラムを単純にしてみることだ。せっかく書きあげたコードであるので、それが間違いを含んでいてもそのコードの一行も削除することに我慢ができないので、初心者は後退に躊躇しがちである。コピーで気持が納まるのであれば、コピーを作り、コードを推敲すればよい。コピーされたものから少しずつ推敲中のコードに追加するようにする。

見つけにくいバグに遭遇したときは、読み、実行し、熟考し、ときとして後退することが必要だ。もし自分がこれらのやらなければならないことの一つのみに執着しているようだったら、他も試してみよう。

13.11 語句

決定論的（deterministic）： 同一の入力を与えられて実行されたプログラムが各時刻に同一の出力を出すといったプログラムの関係性。

疑似乱数（pseudorandom number）： 見かけの上では乱雑に見えるが決定論的な過程で生成された数の列が持つ性質。

既定値（default value）： 引数が与えられない場合に使う前もって準備された最適な値。

無効にする（overrides）： 既定値を引数で置き換える。

ベンチマーク・テスト（benchmarking）： データ構造の選択にあたって幾つかの候補になるデータ構造を実装し同じ実際のデータを与えて実行して優劣を決める過程。

ラバー・ダック・デバッギング（rubber duck debugging）： 抱えている問題を無機質の物体、例えばラバー・ダックに説明することによるデバッギング法。問題を声をだして話すことが、たとえそのラバー・ダックが Python を知らないとしても役に立つことがある。

13.12 練習問題

練習問題 13.9 単語出現頻度に従って単語を並べてときの単語の順位をその単語の「ランク」という。最頻度単語のランクは一位で、その次ぎ二位である。自然言語に対して、ジップの法則は単語の頻度とそのランクの間にある関係を述べてもの

である (http://en.wikipedia.org/wiki/Zipf's_law).。それによれば、単語の頻度 f はその単語のランク r から以下の関係で予測される：

$$f = cr^{-s}$$

ここで s と c は使用言語とテキストによって決まる定数である。両辺の対数をとると

$$\log f = \log c - s \log r$$

となる。従って、両対数グラフを作ると傾きが-s で接片が log c の直線が得られる。

テキストから英文を読み込み、単語出現頻度を調べよ。そして、頻度の降順に一行毎に各単語の log f、log r を表示せよ。手元にあるグラフ表示ソフトウエアを使って結果をグラフ化し、直線が現れるか調べよ。傾き s の値は推定できるか？
解答例：https://thinkpython.com/code/zipf.py.
この解答例を実行するためにはグラフ表示のモジュール matplotlib が必要である。もしも *Anaconda* がインストールしてあれば matplotlib は同梱されているはず、さもないと別途インストールする必要がある。

第14章　ファイル

この章では外部記憶装置にデータを保存することで実現できるプログラムの「永続性」の考えを導入し、ファイルやデータベースといった様々な記憶形式を如何に使うか考察する。

14.1　永続性

これまでみてきたプログラムは短時間実行し、結果を表示し、それが終われば消えてしまうという意味で過度的なものだ。もしプログラムを再度走らせようと思ったら、まっさらな状態からに再実行することしかない。

他のプログラムは**永続的（persistent）**だ。それらは長い時間（または常に）動いている：それらはデータの一部を外部記憶装置（例えばハードディスク）に保存しておく。そして、プログラムが終了しても、終了時のデータを使って再実行ができる。

このような永続的なプログラムの典型はオペレーティング・システムで、コンピュータが起動している間はほぼ動いている。WEB サーバはもう一つの例だ。これはネットワーク上の要求に対応するように常に動いている。

プログラムが扱うデータを保持する最も簡単な方法はテキストファイル（text file）の読み込み書き込みである。テキストファイルの読み込みについては既に触れたので、この章では書き込みを議論する。

また別な方法はプログラムの状態をデータベースに保存しておくものだ。この章では簡単なデータベースを示し、プログラムの状態をこのデータデースに保存するときに使うモジュール `pickle` を紹介する。

14.2　読み込み・書き込み

テキストファイルは外部記憶装置（ハードディスク、フラッシュ・メモリー、CD-ROM 等）に文字ベースで保存される。既に、9.1 節で読み込みのための `open` と `read` については学習した。

書き込みは'w' モードでファイルをオープンする：

```
>>> fout = open('output.txt', 'w')
```

既存のファイルであると古いデータを全て消去するので要注意。存在しないものであると新規にフィアルを作成する。

メソッド write はデータをフィアルに書き込む：

```
>>> line1 = "This here's the wattle,\n"
>>> fout.write(line1)
24
```

ファイルオブジェクトは現在ファイルの何処にいるのかの追尾機能を持っているので、write でもう一度書き込むとデータはそれまでの終わりに追加されることになる：

```
>>> line2 = "the emblem of our land.\n"
>>> fout.write(line2)
24
```

書き込みの操作が全て終わったらそのファイルを閉じる：

```
>>> fout.close()
```

もしも閉じることを忘れてしまっても、プログラムの終了時にそのファイルは閉められる。

14.3 記述演算子

メソッド write の引数は文字列でなければならない。従って他の値をファイルに書き込むにはそれらを文字列に変換する必要がある。その最も安易な方法は str を用いるものだ：

```
>>> x = 32
>>> f.write(str(x))
```

他の方法として、**記述演算子（format operator）**%を使うものがある。整数に適用した%はモジュラー演算子であるが、最初の被演算子が文字列であるときには記号%は記述演算子となる。

最初の被演算子は**記述文字列（format string）**で、第二の被演算子を如何に記述するかを指定する少なくとも一つの**記述子（format sequence）**を含んでいる必要がある。

整数を記述する'%d' 記述子（d は "digital" の略語である）を例にみよう。

```
>>> camels = 42
>>> '%d' % camels
'42'
```

結果は文字列の'42'である。整数の 42 と混乱しないように。

記述子は文字列の任意の場所に置くことができ、値を文章の任意の場所に挿入できる：

```
>>> camels = 42
>>> 'I have spotted %d camels,' % camels
'I have spotted 42 camels,'
```

二つ以上の記述子が文字列中にあるときは、第二被演算子はタプルでなければならない。以下の例では'%d' は整数のため、'%g' は浮動小数点数のため、'%s' は文字列のために使われている：

```
>>> 'In %d years I have spotted %g %s.' % (3, 0.1, 'camels')
'In 3 years I have spotted 0.1 camels.'
```

タプルの要素の個数は記述子の個数と一致している必要があるし、要素の型は記述文字列の中の型と合っている必要がある。

```
>>> '%d %d %d' % (1, 2)
TypeError: not enough arguments for format string
>>> '%d' % 'doliara'
TypeError: %d format: a number is required, not str
```

最初の例では個数が一致していないし、第二の例では型が一致していない。

記述演算子は強力であるが使い方が難しい。より詳細は https://docs.python.org/3/library/stdtypes.html#printf-style-string-formatting を参照のこと。さらに強力な別な方法は文字列メソッドや組み込み関数 `format()` である（https://docs.python.org/3/library/stdtypes.html#str.format.。

14.4 ファイル名とパス

ファイルは**ディレクトリー（directories）**を使って組織化されている。実行中のプログラムの全てが大部分の操作に対して既定値となる「カレント・ディレクトリー」と呼ばれるディレクトリーを持っている。例えば、プログラムでファイ

ル読み込みのためにopen文を実行すると、Pythonはこのカレント・ディレクトリーにそのファイルを探しに行く。

モジュールosはファイルやディレクトリーに関連する操作をサポートする関数を提供している（osは“operating system”の略である）。os.getcwdはカレント・ディレクトリーの名前を返す：

```
>>> import os
>>> cwd = os.getcwd()
>>> print cwd
/home/dinsdale
```

cwdは“current working directory”の略である。この例の結果は/home/dinsdaleでユーザdansdaleのホームディレクトリーである。

/home/dinsdaleのようなファイルの所在を示す文字列は**パス（path）**と呼ばれる。memo.txtのような単純なファイル名もパスであると考えられるがこれはカレント・ディレクトリーを起点とするパス表示で**相対パス（relative path）**と呼ばれている。もしもカレント・ディレクトリーが/home/dinsdaleであると、ファイル名memo.txtは/home/dinsdale/memo.txtを指していることになる。

絶対パス（absolute path）はファイルシステムのトップディレクトリーを起点としたパス表示である。

絶対パスを表示するにはos.path.abspathを使う：

```
>>>os.path.abspath('memo.txt')
'/home/dinsdale/memo.txt'
```

os.pathはファイル名やパスの関連する多くの関数を提供している。例えば、os.path.existsはファイルやディレクトリーが存在するかどうかを調べる（訳注：探索パス上に存在すかどうかである）。

```
>>>os.path.exists('memo.txt')
True
```

もし存在することが判明したら、os.path.isdirでそれがディレクトリーであるかどうかを調べることができる：

```
>>> os.path.isdir('memo.txt')
False
>>> os.path.isdir('/home/dinsdale')
True
```

同様に、os.path.isfile でそれがファイルであるかどうかを調べることができる。

os.path.listdir で与えられたディレクトリー内のファイル名や他のディレクトリー名のリストを得ることができる。

```
>>>os.path.listdir(cwd)
['musics', 'photos', 'memo.txt']
```

これらの関数の機能を以下のプログラムで示す。これは一つのディレクトリー内を「逍遙」して全てのファイル名を表示し、その中の全てのディレクトリーを再帰的に探索する。

```
def walk(dirname):
    for name in os.listdir(dirname):
        path = os.path.join(dirname, name)

        if os.path.isfile(path):
            print(path)
        else:
            walk(path)
```

os.path.join はディレクトリー名とファイル名を引数に取り、完全なパス名を返す。

モジュール os はここで提示した関数 walk と似たしかしもっと一般的な関数を提供している。この関数のドキュメンテーションを読み、あるディレクトリー内のファイル名、その中にあるディレクトリー内のファイル名というように全てのファイル名を表示するプログラムを作成せよ。
解答例：https://thinkpython.com/code/walk.py.

14.5 例外捕捉

ファイルの読み書きでは沢山の事柄が上手く行かないことがある。存在しないファイルを開こうとすると IOError が出る：

```
>>> fin = open('bad_file')
IOError: [Errno 2] No such file or directory: 'bad_file'
```

また、許可のないファイルにアクセスしようとすると同様にエラーになる：

```
>>> fout = open('/etc/passwd','w')
IOError: [Errno 13] Permission denied: '/etc/passwd'
```

更に、ディレクトリーを書き込みモードで開とエラーになる：

```
>>> fin = open('/home')
IOError: [Errno 21] Is a directory
```

これらのエラーを避けるためには、os.oath.exists や os.path.isfile のような関数を使うことも考えられるが、可能性のある全てのエラーを調べるには時間もかかるし、コードも余計に太る。

それよりもさし当たって問題が惹起されるか試みて、問題があれば対処するという方法がよりよい。これは try 文が意図するものである。構文は if 文に似た書き方をする：

```
try:
    fin = open('bad_file')
except:
    print('Something went wrong.')
```

Python は try 句を先ず実行する。何も問題がなければ、except 句は無視し次ぎに進む。もし何か例外が起こると try 句を中断して except 句を実行する。

try 文による例外の処理は**例外捕捉（catching）**と呼ばれている。上の例では except 句は単にエラーが起きたことを知らせる表示だけでありあまり役に立たないが、一般に except 句は問題の解決法、再実行、少なくとも優雅にプログラムを終了させる位のことはする。

14.6 データベース

データベース（database）はデータ保存のために組織化されたファイルである。大部分のデータベースはキーから値への写像という特徴をもつので辞書のように組織かされている。最大の違いは、データベースはハードディスクのような外部記憶装置上に作られ、従って、プログラムが終了しても永続的に残ることである。

モジュール anydbm(訳注：Python3 では dbm となった）はデータベース・ファイルの作成及び更新の操作を提供している。例としてここでは画像ファイルの脚注を保存するデータベースを作成する。データベースのオープンはファイルのそれと似ている：

```
>>> import anydbm
>>> db = anydbm.open('captions.db', 'c')
```

モード c はもしデータベースが存在しないときは作成するという意味である。結果は辞書が持つ操作の大部分を提供するデータベースオブジェクトを戻す。新しいアイテムを追加しようとすると、anydbm はデータベースを更新する。

```
>>> db['cleese.png'] = 'Photo of John Cleese.'
```

アイテムにアクセスしようとすると、anydbm はファイルからの読み込みをする：

```
>>> db['cleese.png']
b'Photo of John Cleese.'
```

結果は b をつけた文字列のバイトオブジェクト（bytes obeject）である。バイトは文字と多くの点でにかよったものである。Python の学習がもっと先に進むその違いが重要になるが、当面は無視していよう。

もし既存のキーに代入を試みると、古い値の上書きになる：

```
>>> db['cleese.png'] = 'Photo of John Cleese doing a silly walk.'
>>> db['cleese.png']
b'Photo of John Cleese doing a silly walk.'
```

辞書で使えたメソッド、例えば keys や items なども使える。また for 文での繰り返しもできる：

```
for key in db:
    print(key, db[key])
```

終了は、普通のファイルのようにクローズで終わる：

```
>>> db.close()
```

14.7 削ぎ落とし

anydbm の制限はキーも値も文字列のみが許されることだ。他の型ではエラーになる。モジュール pickle はこんなときに役に立つ。このモジュールはほぼ任意の型のデータを文字列に変換し、また逆に型データに逆変換する。

pickle.dumps は引数として任意の型を受け取り、対応する文字列による表現を返す（dumps は “dump string” の略）。

```
>>> import pickle
>>> t = [1, 2, 3]
>>> pickle.dumps(t)
b'\x80\x03]q\x00(K\x01K\x02K\x03e.'
```

この文字列の記述法はヒトには不明だ。これは pickle にとって容易に解釈できることを意味している。事実 `pickle.loads`（つまり "load string"）はオブジェクトを復元する。

```
>>> t1 = [1, 2, 3]
>>> s = pickle.dumps(t1)
>>> t2 = pickle.loads(s)
>>> t2
[1, 2, 3]
```

この新しいオブジェクトは古いものと同じ値を持っているが、一般に同一のオブジェクトではない：

```
>>> t1 == t2
True
>>> t1 is t2
False
```

つまり、「削ぎ落とし」と「その復元」はオブジェクトをコピーしたことに対応する。

この `pickle` モジュールを使って文字列でない型のデータをデータベースで扱うことができる。モジュール `shelve` はこれらの機能をカプセル化したものである。

14.8 パイプ

多くのオペレーティング・システムは**シェル（shell）**として知られているコマンドベースのインタフェースを持っている。シェルはファイルシステムを探索したり、アプリケーションを起動させたりするコマンドの体系を提供している。例えば、Unix ではディレクトリーの変更は `cd`、ディレクトリーの中味の表示は `ls`、また web ブラウザーの起動は（例えば）`firefox` のコマンドで行う。

シェルから起動できるプログラムは**パイプ（pipe）**を使って Python からも起動できる。パイプは起動されているプログラムを表現しているオブジェクトである。

例えば、Unix コマンド ls -ll はカレント・ディレクトリーの中味を詳細表示するものであるが、os.popen [1]でこのコマンドを実行できる：

```
>>> cmd = 'ls -l'
>>> fp = os.popen(cmd)
```

引数はコマンドを表現する文字列である。戻り値はファイルのオープン文で戻るようなオブジェクトである。メソッド readline で ls プロセスの表示を一行毎に表示できるし、read で一度に全てを表示できる。

```
>>>res = fp.read()
```

終わるときはファイル同じようにクローズする：

```
>>> stat = fp.close()
>>> print(stat)
None
```

戻り値は ls プロセスの終了状態である。None はそのプロセスが正常に終了したことを示す。

他の例である。多くの Unix システムはファイルの中味を読み、“checksum” という量を計算するコマンド md5sum を提供している。
この MD5 については http://en.wikipedia.org/wiki/Md5 を参照せよ。このコマンドは二つのファイルの中味が同一なものであるかどうかを調べる上で有効な方法を提供している。中味が異なるファイルが同じ checksum を生成する確率は極めて小さい（宇宙が崩壊する以前には起こりそうもない）。Python からパイプを使ってこの md5sum を起動することができ、結果も得られる。

```
>>> filename = 'book.tex'
>>> cmd = 'md5sum ' + filename
>>> fp = os.popen(cmd)
>>> res = fp.read()
>>> stat = fp.close()
>>> print(res)
1e0033f0ed0656636de0d75144ba32e0  book.tex
>>> print(stat)
None
```

[1]popen は旧式であると言われている。その意味はこれに替わる新しいモジュール subprocess を使うことが期待されている。しかし、簡単なケースでは subprocess は必要以上に複雑になってしまう。わたしはそれが解決されるまでは popen を使うつもりである。

14.9　モジュールを書く

任意のPythonコードを含むファイルはモジュールとしてインポートできる。例として、以下のようなコードを含むwc.pyを考える：

```
def linecount(filename):
    count = 0
    for line in open(filename):
        count += 1
    return count

print(linecount('wc.py'))
```

このプログラムを起動するとファイルの行数、つまり7を表示するはずだ。このファイルをインポートもできる：

```
>>> import wc
7
```

ここではwcはモジュールオブジェクトになっている：

```
>>> wc
<module 'wc' from 'wc.py'>
```

このモジュールはlinecountという関数を提供することになる：

```
>>> wc.linecount('wc.py')
7
```

これでPythonのモジュールを書いたことになる。

唯一の問題はこの例ではモジュールをインポートした時点で、コードの最後に書いたコードのテストが実行されてしまうことである。通常はモジュールのインポートは関数類の定義でありその実行までは必要ない。モジュールとして使う予定のプログラムはよく以下のような常套句を使う：

```
if __name__ == '__main__':
    print(linecount('wc.py'))
```

__name__は組み込み変数でプログラムが起動されたときに値が決まる。もしもプログラムがスクリプトとして起動された時はこの値は__main__である。その場合はテストコードが実行される。さもないと、つまりモジュールとしてインポートされるとテストコードはスキップされる。

練習問題として、wc.py をファイルとして作成せよ。そしてそれをスクリプトとして実行せよ。その後 Python インタプリタを起動して wc をインポートしてみる。このもモジュールがインポートされたときの__name__の値は如何なる値を持っているか？

注：もしもインポートしようとしたモジュールが既にインポートされているとすると、そのモジュールが変更を受けていようが Python は何もしない。再度モジュールをインポートしたいときには、組み込み関数 reload が使える。しかし、扱いにくいので、最も安全な方法はインタプリタを再起動し、再度モジュールをインポートすることだ。

14.10 デバッギング

ファイルの読み書きでデータ区切り文字の問題に遭遇するかもしれない。普通空白、タブ、改行は見えないので、この種のエラーはデバックが難しい：

```
>>> a = '1, 2\t 3\n 4'
>>> print(a)
1, 2  3
 4
```

組み込み関数 repr がこのときに役に立つ。この関数は引数として任意のオブジェクトをとり、そのオブジェクトを表現する文字列を返す。文字列であるとデータ区切り文字を含めて表示される：

```
>>> print(repr(a))
'1, 2\t 3\n 4'
```

他の問題としては行の終わりを示す文字が異種システム間で違っていることだろう。あるシステムでは行の終わりは\n になるが、他のシステムでは\r であり、またこの両方で行の終わりを示すシステムもある。異なったシステム間でファイルのやり取りをするときに問題になる可能性がある。多くのシステムでは変換のためのアプリケーションがある。それらを見つけてみよう。
更に http://en.wikipedia.org/wiki/Newline も参照のこと。勿論、あなた自身でそのプログラムを書くのもよし。

14.11 語句

永続的（persistent）： 休みなく実行され少なくともそのデータの一部が外部不揮発記憶装置に保存されようなプログラムの性格。

記述演算子（format operator）： 記述文字列と記述子（タプルになっている）を受け取り、記述子の各要素を記述文字列に従って文字列に変換することを含めた文字列を生成する演算子 `%`である。

記述文字列（format string）： 記述演算子と共に使われる記述子を含む文字列。

記述子（format sequence）： 記述文字列の中で値を如何に文字列に変換するかを指定する `%d` のような文字列。

テキストファイル（text file）： ハードディスクのような外部装置に保存される文字だけのデータ。

ディレクトリー（directories）： 固有の名前が付けられたファイルの集合。フォルダーとも言う。

パス（path）： 一つのファイルを同定するための文字列。

相対パス（relative path）： カレント・ディレクトリーから辿ったパス。

絶対パス（absolute path）： ファイルシステムの最上位のディレクトリーから辿ったパス。.

例外捕捉（catching）： `try` 文や `except` 文を用いてプログラムの異常終了を回避する手法。

データベース（database）： その内容がキーと対応する値を辞書で組織化した中身になっているファイル。

バイトオブジェクト（bytes object）： 文字列に似たオブジェクト。

シェル（shell）： ユーザのコマンド入力を受け取り他のプログラムによってそれを実行するといったことを可能とするプログラム。

パイプオブジェクト（pipe object）： プログラム実行を表現するオブジェクト。これにより Python プログラムがコマンドを実行し結果を読むことができる。

14.12　練習問題

練習問題 14.1 探索文字パターン、置換文字パターンの二つの文字列を引数に、更に二つのファイル名を引数とする関数 `sed` を作成せよ。一つのファイルは読み込み用のテキストファイルで、他は書き込み用のファイルである。関数は読み込み

用のファイルからテキストを読み込み、その中に探索文字パターンがあるときは、この文字列を置換パターンに置き換えてテキストを書き込みファイルに書き出す。ファイルのオープン、読み込み、書き込み、クローズに際してエラーがある場合には、例外捕捉でエラー表示をし、プログラムを終了するようにせよ。
解答例： https://thinkpython.com/code/sed.py.

練習問題 14.2 練習問題 12.2 の解答 https://thinkpython.com/code/anagram_sets.py をダウンロードしてコードを眺めてみると単語をキーにしてこの単語から派生するアナグラムにある単語のリストを値とする辞書を使っていることがわかる。例えば、'opst' のアナグラムにある単語のリストは（'opts', 'post', 'pots', 'spot', 'stop', 'tops'）となる。この `anagram_sets` をインポートし、二つの新しい関数を作成せよ。一つはアナグラム辞書を「棚」に保管する `store_anagrams`、もう一つはこのデータベースからキーに対応するアナグラムのリストを呼び出す関数 `read_anagrams` である。
解答例： https://thinkpython.com/code/anagram_db.py.

練習問題 14.3 MP3 ファイルの膨大なコレクションがある。多分同じ内容の曲が異なった名前や異なったディレクトリーにあることが少なからずあると思われる。この練習問題はこの重複を探索する方法である。

1. あるディレクトリー内及びそのサブディレクトリーと再帰的に調べ特定の拡張子（.mp3 のような）を持つ全てのファイル対する絶対パスを要素とするリストを生成するプログラムを作成せよ。

2. 重複を確認するために、各ファイルの "checksum" を計算するために `md5sun` を利用する。二つのファイルが同一の "checksum" だったら、この二つのファイルは中味が同じとみてよい。

3. 二重のチェックとして Unix コマンドの `diff` を使うこともできる。

解答例：https://thinkpython.com/code/find_duplicates.py.

第15章　クラスとオブジェクト

これまででコードを組織化する上での関数の役割やデータを組織化する上での組み込み型について学んできた。次のステップはコードとデータの双方を組織化できるユーザ定義型を使う「オブジェクト指向プログラミング」について学習する。オブジェクト指向プログラミングは大きなテーマであるので全体像を把握するために数章を当てる。

この章の例題のコードはhttps://thinkpython.com/code/Point1.pyで入手可能である。さらに例題の解答例はhttps://thinkpython.com/code/Point1_soln.pyにある。

15.1　ユーザ定義型

これまで組み込み型についてみてきたが、この章では新しい型を定義してみよう。例として二次元平面上の点を表現するPointと呼ばれる型を生成してみよう。

数学的な表現では二次元上の点は二つ座標の値をカンマで区切り、カッコで囲む。例えば、$(0, 0)$は原点を示し、(x, y)は右にx単位移動し、上にy単位だけ移動した点の表現になる。

Pythonでこの状況を表現する方法はいくつもある：

- 変数xと変数yを用意し、座標の値を保存する。
- リストやタプルを用意し、その要素として二つの値を保存する。
- 座標点の表現するような新しい型を創造してその型の変数に保存する。

新しい型を創造するのはちょっと複雑のように思われるが直ぐみるように利点も大きい。

ユーザ定義型は**クラス（class）**とも呼ばれている。クラスの定義は以下のようなものだ。

```
class Point(object):
    """Represents a point in 2-D space."""
```

ヘッダーはこの新しいクラスの名前が Point であることを示している。そして、このクラスが組み込み型 object の一種であることを宣言している。ボディはこのクラスの目的を記述した文書である。ここには変数の宣言やメソッドを記述することができるが、それは後の話題とする。

クラスを宣言することで新たな**クラスオブジェクト（class object）**を生成したことにもなる。

```
>>> Point
<class '__main__.Point'>
```

このクラスはプログラムのトップで宣言されたのでその完全名は__main__.Point になる。

クラスオブジェクトはオブジェクトを生成する工場のようなものである。一つの Point 型の変数を生成するには、関数を呼ぶようにクラスを使う：

```
>>> blank = Point()
>>> blank
<__main__.Point object at 0xb7e9d3ac>
```

戻り値は Point オブジェクトへの参照であり、これが変数 blank に代入される。新しいオブジェクトを生成することを**オブジェクト具現化（instantiation）**と呼び、生成されたオブジェクトをクラスの**インスタンス (instance)** という。インスタンスを print すると、それがどのクラスに属しているか、メモリー上のどこに保存されたのか（0x のプレフィックスが付いていることから十六進数）を表示する。

15.2　属性

以下のようにインスタンスに値を与えることができる：

```
>>> blank.x = 3.0
>>> blank.y = 4.0
```

ここでの構文はモジュールから変数を取り出す、math.pi や string.whitespace のようなとき使う構文に似ている。しかし、ここではオブジェクトの名前付きの要素へ値を代入することである。この要素のことを**属性（attributes）**という。この英単語は名詞としては第一音節のアクセントがあるので、“AT-trib-ute” と発音するが、動詞としては、第二音節にアクセントがあるので、“at-TRIB-ute” と発音させる。

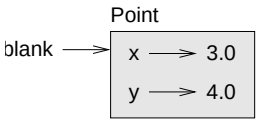


図 15.1: オブジェクト図

図 15.1 はこの代入の結果を表すものである。一つのオブジェクトとその属性の状態を示すものでオブジェクト図という。変数 blank は Point オブジェクトを参照し、二つの属性を持っていることを示す。各属性は浮動小数点数を参照している。

属性の値は普通のように扱える：

```
>>> blank.y
4.0
>>> x = blank.x
>>> x
3.0
```

表式 blank.x の意味は「blank が参照しているオブジェクトに行き、x の値を得よ」である。更にこの場合はその値を変数 x に代入している。変数 x と属性 x の間には何も対立はない。

ドット表記は任意の表式のなかで使える。例を示す：

```
>>> '(%g, %g)' % (blank.x, blank.y)
'(3.0, 4.0)'
>>> distance = math.sqrt(blank.x**2 + blank.y**2)
>>> distance
5.0
```

インスタンスを関数の引数とすることもできる：

```
def print_point(p):
    print('(%g, %g)' % (p.x, p.y))
```

関数 print_point は引数として point オブジェクトを受け取り、それを数学的な表記で表示する。これを発動させるときは引数にインスタンス blank を渡せばよい：

```
>>> print_point(blank)
(3, 4)
```

この関数の中では p は blank の別名であるので、関数の中で p を変更すると blank も変わる。

練習問題として、二つの point オブジェクトを引数に取り、二点間の距離を戻り値とする関数 distance_between_points を作成せよ。

15.3　長方形

オブジェクトの属性が明白であるものもあるし、そうでなく上手く決めなければならない場合もある。例として、長方形を表現するクラスを設計することを考えよう。その位置と大きさを指定するためにどのような属性が必要だろうか？簡単にするために長方形は垂直や水平に置かれたものとする。

指定の仕方に少なくとも二つの方法があるだろう：

- 長方形の一頂点（または中心）の座標、幅、高さを指定する。
- 対角線上にある二つの頂点の座標を指定する。

どちらがよいか決めかねるので、第一の方法を使う。

クラス定義は以下のようだ：

```
class Rectangle(object):
    """Represents a rectangle

    attributes: width, height, corner
    """
```

ドキュメント文字列にある属性 width, height は数、corner は Point オブジェクトである。

一つの長方形を表現することは Rectangle オブジェクトをオブジェクト具現化し、その属性に値を代入することである：

```
box = Rectangle()
box.width = 100.0
box.height = 200.0
box.corner = Point()
box.corner.x = 0.0
box.corner.y = 0.0
```

表式 box.corner.x の意味は「box が参照するオブジェクトへ行き、corner と名付けられた属性を選択し、そのオブジェクトへ行き、x と名付けられた属性を選択せよ」である。図 15.2 はこのオブジェクトの**オブジェクト状態図（object diagram）**を示している。他のオブジェクトの属性になっているオブジェクトは**埋め込まれたオブジェクト（embedded object）**である。

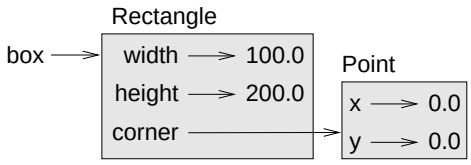


図 15.2: オブジェクト図

15.4 戻り値としてのインスタンス

関数の戻り値にインスタンスがくることがある。例を示す。関数 find_center は引数に Rectangle オブジェクトを取り、戻り値にこの Rectangle の中心の座標を含む Point オブジェクトを返す：

```
def find_center(rect):
    p = Point()
    p.x = rect.corner.x + rect.width/2.0
    p.y = rect.corner.y + rect.height/2.0
    return p
```

例としてインスタンス box を引数にしてみると、その中心の座標が戻ってくる：

```
>>>center = find_center(box)
>>>print_point(center)
(50, 100)
```

15.5 オブジェクトは変更可能

オブジェクトの属性の一つに値を代入することでオブジェクトの状態を変えることができる。例えば、一つの長方形の座標は変えずに、width と height を変えたいとしたら以下のようにする：

```
box.width = box.width + 50
box.height = box.height + 100
```

オブジェクトを変更する関数を書くこともできる。例えば、Rectangle オブジェクトと二つの数、dwidth、dheight を引数に持つ関数 grow_rectangle はこの二つの数値分だけ長方形を大きくする関数である：

```
def grow_rectangle(rect, dwidth, dheight):
    rect.width += dwidth
    rect.height += dheight
```

以下はこの関数の効果を示す例である：

```
>>> box.width, box.height
(150.0, 300.0)
>>> grow_rectangle(box, 50, 100)
>>> box.width, box.height
(200.0, 400.0)
```

関数の中では rect は box の別名である。従って関数内で rect を変更すれば、box も変更される。

練習問題として、引数としてオブジェクト Rectangle と二つ数 dx、dy を取り、長方形の場所の座標 x と y をそれぞれ dx と dy だけ移動する関数 move_rectangle を作成せよ。

15.6　コピー

別名呼称は一ヶ所を変えると他に思わぬ効果が出るためプログラムを読み難くしてしまう。その結果あるオブジェクトを参照する全ての変数の軌跡を追跡することを難しくしてしまう。

別名呼称に替わるものとしてコピーがある。モジュール copy は任意オブジェクトのコピーを生成する関数 copy を提供している：

```
>>> p1 = Point()
>>> p1.x = 3.0
>>> p1.y = 4.0

>>> import copy
>>> p2 = copy.copy(p1)
```

p1 と p2 は同じデータを含んでいるが、同一の Point オブジェクトではない。

```
>>> print_point(p1)
(3, 4)
>>> print_point(p2)
(3, 4)
```

```
>>> p1 is p2
False
>>> p1 == p2
False
```

is 演算子は二つが同一のオブジェクトでないことを意味し、これは予想したことだ。しかし、==演算子では結果は True と出ることを期待したかもしれないが、その予想は外れた。インスタンスについては既定値として、==演算子は is 演算子と同じ振る舞いをする。尤も、この振る舞いは変えられるが、これは後の話だ。

copy 関数を使うと Rectangle の複製を作ることができるが、埋め込まれたオブジェクト Point は違う。

```
>>> box2 = copy.copy(box)
>>> box2 is box
False
>>> box2.corner is box.corner
True
```

図 15.3 はオブジェクトの状態図がどうなるかを示した。この操作はオブジェクトとそれらが参照しているもののコピーは作成するが埋め込まれたオブジェクトはそのままであるので**浅いコピー（shallow copy）**と呼ばれる。大部分のアプリケー

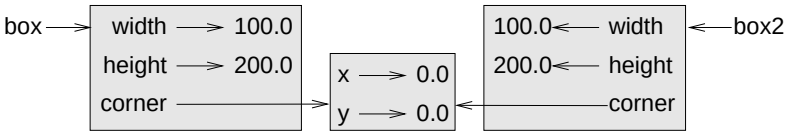


図 15.3: オブジェクト図

ションではこの事情はあって欲しくないものである。いまの例でいうとオブジェクト Rectangle に対して grow_rectangle を発動させるとコピーされた別のインスタンスは影響を受けない。しかし、関数 move_rectangle の発動では別のインスタンスも影響をうける。こんな振る舞いは混乱するし、エラーの元だ。

幸にして、オブジェクトばかりでなく、そのオブジェクトが参照するオブジェクト、また更にそれらが参照するオブジェクト等々のコピーをも生成する関数 deepcopy が提供されている。これを**深いコピー（deep copy）**と呼ぶ理由は自明だろう。

```
>>> box3 = copy.deepcopy(box)
>>> box3 is box
False
```

```
>>> box3.corner is box.corner
False
```

これで box と box3 は完全の別なオブジェクトである。

練習問題として、引数として Rectangle オブジェクトを受け取り、新しい Rctangle を返す関数 move_rectangle の改訂版を作成せよ。

15.7　デバッギング

オブジェクトの処理を含むプログラムを扱い始めると新た種類のエラーに遭遇する。もし存在しない属性にアクセスしようとすると、AttributeError が出る：

```
>>> p = Point()
>>> p.x = 3
>>> p.y = 4
>>> p.z
AttributeError: Point instance has no attribute 'z'
```

もし一つのオブジェクトがどうようなものか不明な時は、以下のようなコマンドを使うとよい：

```
>>> type(p)
<class '__main__.Point'>
```

もしある属性をそのオブジェクトが持っているか不明なときは組み込み関数 hasattr が使える：

```
>>> hasattr(p, 'x')
True
>>> hasattr(p, 'z')
False
```

第一番目の引数は任意のオブジェクトで、第二番目の引数はその属性とした名前の**文字列**である。

オブジェクトがその属性を持っているか不明なときは try 文を使うこともできる：

```
try:
    x = p.x
except AttributeError:
    x = 0
```

この手法は異なったタイプでも動作する関数を書こうとする上で便利なものである。このテーマの詳細は第 17.9 節で扱う。

15.8 語句

クラス（class）： ユーザ定義型。クラスの定義によって新しいクラスオブジェクトが生成される。

クラスオブジェクト（class object）： ユーザ定義型に関する情報を含むオブジェクト。クラスオブジェクトはその型のインスタンスを生成するときに使われる。

インスタンス (instance)： 一つのクラスを具現化したオブジェクト。

具現化（instantiate）： 新規のオブジェクトを作ること。

属性（attributes）： 一つのオブジェクトに付随した名前の付いた値。

埋め込まれたオブジェクト（embedded object）： 他のオブジェクトの属性として使われたオブジェクト。

浅いコピー（shallow copy）： 埋め込まれたオブジェクトへの参照を残したままオブジェクトの複製を作る。

深いコピー（deep copy）： そのなかに埋め込まれたオブジェクトがあるときその複製をも作るかたちでオブジェクトの複製を作る。

オブジェクト状態図（object diagram）： オブジェクト、その属性、その属性が参照している値を明示するグラフ表現。

15.9 練習問題

練習問題 15.1 Circle という名前のクラスを定義せよ。その属性は center と radius で center は Point オブジェクトであり、radius は数値である。

次にこの Circle オブジェクトを center の値を 150,100）そして radius の値を 75 の初期値で具現化せよ。

以下幾つかの関数を作成せよ：

- 一つの Circle オブジェクトと一つの Point オブジェクトを引数とする関数 point_in_circle を作成せよ。この関数のはこの Point がこの Circle の内側もしくは境界上にあるならば Ture を戻り値として返す。
- 一つの Circle オブジェクトと一つの Rectangle オブジェクトを引数とする関数 rect_in_circle を作成せよ。この関数はこの Rectangle 全体がこの Circle の内側もしくは境界上にあるならば Ture を戻り値として返す。

- 一つの Circle オブジェクトと一つの Rectangle オブジェクトを引数とする関数 rect_circle_overlap を作成せよ。この関数はこの Rectangle の頂点の一つでもこの Circle の内側にあるならば Ture を戻り値として返す。

  もう少し挑戦的な課題として、この Rectangle の辺のどこかの部分がこの Circle の内側にあるならば Ture を戻り値として返すとした関数を作成せよ。

解答例：https://thinkpython.com/code/Circle.py.

練習問題 15.2 Turtle オブジェクトと Rectangle オブジェクトとを引数に持ち、この Turtle でこの Rectangle を描画する関数 draw_rect を作成せよ。Turtle オブジェクトの使い方は第 4 章を参照のこと。
また同様に円を描画する関数 draw_circle を作成せよ。
解答例：https://thinkpython.com/code/draw.py

第16章　クラスと関数

新しい型の作り方がわかったので、次のステップはこのようなユーザ定義型のオブジェクトを仮引数に持ち、戻り値もこれらの型のオブジェクトを取るような関数を作ることである。またこの章では「関数プログラミング手法」と二つの新たなプログラム開発計画法についても触れる。

この章のサンプルコードは https://thinkpython.com/code/Time1.py にある。また練習問題の解答例は https://thinkpython.com/code/Time1_soln.py. にある。

16.1　時刻

ユーザ定義型のもう一つの例として一日の内の時刻を記録する Time を取り上げる。そのクラス定義は以下のようだ：

```
class Time:
    """Represents the time of day.

    attributes: hour, munute, second
    """
```

Time オブジェクトを生成し、属性に値を代入してみる：

```
time = Time()
time.hour = 11
time.minute = 59
time.second = 30
```

図 16.1 にこのオブジェクトの状態図を示した。

練習問題として、引数として Time オブジェクトを取り、hour:minute:second の形式で時刻表示をする関数 print_time を作成せよ。ヒント：記述子'%.2d' を使うと整数を先頭に 0 を詰めて二桁で表現する。

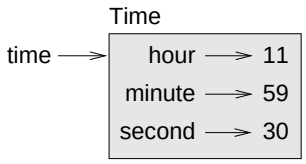


図 16.1: オブジェクト図

16.2　純関数

以下では時刻の和を求める関数 add_time を作成することにする。ここでは二つの関数が開発の途上で検討されることになる。それらは純関数と修正関数である。またこの二種類の関数は単純なものから初めて徐々に複雑なものにして行く開発計画で**原型とパッチ（prototype and patch）**と呼ばれるものに対応している。

関数 add_time の原型は以下のようなものだろう：

```
def add_time(t1, t2):
    sum = Time()
    sum.hour = t1.hour + t2.hour
    sum.minute = t1.minute + t2.minute
    sum.second = t1.second + t2.second
    return sum
```

この関数では新たに sum という Time オブジェクトを生成、属性を初期化、そのオブジェクトを戻り値としている。これは引数として受け取ったオブジェクトを何も変えないで、しかも値を表示するとか、ユーザ入力を求めるとかの効果もないという意味で**純関数（pure function）**と呼ぶ。

この純関数をテストするために映画 “Monty Python and the Holly Grail” の開始時間 start という Time オブジェクトとこの映画の上演時間（１時間３５分）を保存する duration という Time オブジェクトを生成する。

add_time で何時にこの映画が終わるか分かる：

```
>>> start = Time()
>>> start.hour = 9
>>> start.minute = 45
>>> start.second = 0

>>> duration = Time()
>>> duration.hour = 1
>>> duration.minute = 35
```

```
>>> duration.second = 0
>>> done = add_time(start, duration)
>>> print_time(done)
10:80:00
```

結果の 10:80:00 はわれわれが欲しいものではない。問題は秒や分で足し合わせた結果 60 より大きくなったときの処理をしていないからである。この状況が起きたときは、余分な秒を分に「桁上げ」、余分な分を時に「桁上げ」しなければならない。

これを考慮した改訂版である：

```
def add_time(t1, t2):
    sum = Time()
    sum.hour = t1.hour + t2.hour
    sum.minute = t1.minute + t2.minute
    sum.second = t1.second + t2.second

    if sum.second >= 60:
        sum.second -= 60
        sum.minute +=1

    if sum.minute >= 60:
        sum.minute -= 60
        sum.hour += 1

    return sum
```

改訂版は正しいが、コードは大きめになり初めている。次章ではこれに替わる修正の方法を示す。

16.3 修正関数

ある場合には引数として受け取ったオブジェクトを修正する関数の使用が有益なことがある。この場合にはこの関数を呼んだ側にも修正が反映される。この機能を取り入れた関数を**修正関数（modifiers）**と呼ぶ。

与えられた秒数だけ Time オブジェクトの時刻を増加させる関数 increment は素直に修正関数として書くことができる：

```
def increment(time, seconds):
    time.second += seconds

    if time.second >= 60:
        time.second -= 60
        time.minute +=1

    if time.minute >= 60:
        time.minute -= 60
        time.hour += 1
```

第一行目が基本演算で、残りは前にみたような特別な場合の処理である。

ところでこの関数は正しいだろうか？第二の引数 second が 60 秒よりずうっと大きな値であるとどうなるだろうか？その場合には桁上がりは一度では足りない。つまり、Time オブジェクトの second 属性の値が 60 より小さくなるまで桁上がりを続ける必要がある。一つの解決策は if 文を while 文に置き換えることだが、効率が悪い。練習問題として、関数 increment を正せ。繰り返しのループ無しでこれを達成せよ。

修正関数でできることは何でも純関数でできる。実際、純関数のみが許されているプログラミング言語もある。純関数を使ったプログラムが修正関数を使ったものと比べ開発や速くエラー誘発が少ないとする証拠がある。しかし修正関数が便利なこともあり、関数プログラムがそんなに効率が出ないときもある。

一般に特に問題がなければ純関数を使い、それを使う特別な理由があるときのみ修正関数を使うことを勧める。このアプローチはいわば**関数プログラミング作法（functional programming style）**と言ってもよい。

練習問題として、関数 increment の純関数版を作成せよ。引数の Time オブジェクトを修正する替わりに、結果を新たに生成した Time オブジェクトとして戻せ。

16.4　原型と開発計画

ここまで示してきたプログラム開発法は「原型とそのパッチ」と呼ばれる方法である。各関数について、まず基本的な計算をする原型を作成し、その後にこれをテストし、必要な修正を施す。

この手法は問題の深い理解がまだできていないとき特に有効な方法となる。しかし、この手法は徐々に複雑化するためどれほど複雑にしたらよいのかが不明なので必要以上にコードが複雑になる傾向がある。

別の手法は**計画に基づいた開発手法（planned development）**である。これは高いレベルの問題考察でプログラミングをより容易にしてくれる手法である。Time オブジェクトに関して言えば、その洞察は 60 進法（http://en.wikipedia.org/wiki/Sexagesimal を参照せよ。）を基礎にした三つの数字の問題で、属性 second はその 60 進数の一桁目の数字であり、属性 minute は二桁目、属性 hour は三桁目の数字であることを教えてくれる。関数 add_time や increment では桁上がりを行うことで実質的に 60 進法の和を実行していたことになる。

このような考察は全問題に対して異なる手法が採れることを教えてくれる。つまり、Time オブジェクトをコンピュータが桁上がりを自動的に処理できる 10 進数（つまり整数）に変換してしまうことだ。

Time オブジェクトを整数に変換は以下のようである：

```
def time_to_int(time):
    minutes = time.hour * 60 + time.minute
    seconds = minutes * 60 + time.second
    return seconds
```

この逆に、整数を Time オブジェクトに変換するコード（関数 divmod 二つの引数を取り、最初の引数を第二で割った商と余りをタプルとして戻す）は以下のようになる：

```
def int_to_time(seconds):
    time = Time()
    minutes, time.second = divmod(seconds, 60)
    time.hour, time.minute = divmod(minutes, 60)
    return time
```

これらの関数は期待通り動くことを確認するためにいろいろなテストをやってみるかもしれないが、一つの面白い実験は time_to_int(int_to_time(x)) == x が成り立つことを確かめることだ。これは一貫性のテストの例にもなっている。

関数が期待通りに動くことが確かめられたら、関数 add_time の作成に使ってきる：

```
def add_time(t1, t2):
    seconds = time_to_int(t1) + time_to_int(t2)
    return int_to_time(seconds)
```

この版はすっきりしているし、確認が楽である。練習問題として、time_to_int や int_to_time を使って関数 increment も同様に書き換えよ。

ある面では 60 進法数と 10 進法数への変換、またはその逆変換は時間についての観念のよる操作より難しいかもしれない。この変換はより抽象的で、日常の時間に対するわれわれの直感のほうが良いかもしれない。しかし、一度 60 進法数としての時間につての洞察ができ、その変換プログラムを作ってしまうと、より短く、読むことやデバッグがより易しく、より信頼性の高いプログラムを作ることができる。

また、この方法を取ると機能の追加をするのが楽である。例えば、二つの時間の差を求める問題を考えてみよう。従来の方法では、上位の桁からの借りが必要になる。変換関数を使う方法ではもっと簡単で、従って正常に動くプログラムが作れる。

16.5　デバッギング

Time オブジェクトは minute や second が 0 と 60 の間（0 は含み 60 は含まない）の値を持っていて、hour が正の値であると健全である。また hour や minute の値は整数値であるべきだが、second は小数点数であってもよい。このような要求はそれが何時も満たされていることが求められるものなので、**不変性（invariants）**と呼ばれている。換言すれば、それが満たされていないと何かが間違っているということになる。

この不変性をチェックするようにプログラムを書くと、エラーの検出やその原因解明にとっての助けになる。例えば、Time オブジェクトを引数としてそれば不変性の一つでも破っているときは False を返す関数 valid_time が作れる：

```
def valid_time(time):
    if time.hour < 0 or time.minute < 0 or time.second < 0:
        return False
    if time.minute >= 60 or time.second >= 60:
        return Flase
    return True
```

関数の先頭でこのチェックをして引数の妥当性を調べることができる：

```
def add_time(t1, t2):
    if not valid_time(t1) or not valid_time(t2):
        raise ValueError, 'invalid Time object in add_time'
    seconds = time_to_int(t1) + time_to_int(t2)
    return int_to_time(seconds)
```

または assert 文を使い、条件が満たされていないと例外を発生させることもできる：

```
def add_time(t1, t2):
    assert valid_time(t1) and valid_time(t2)
    seconds = time_to_int(t1) + time_to_int(t2)
    return int_to_time(seconds)
```

assert **文（assert statement）**は通常に実行されるコードとエラー検出用のコードを区別するので役に立つ。

16.6 語句

原型とパッチ（prototype and patch）： 大雑把なプログラムを書き、それをテストして見つかったエラーを修正するといった開発計画。

計画に基づいた開発手法（planned development）： 徐々に精査する開発法や原型とパッチ方式より問題自体を精査して高いレベルで問題を捉えなおしより計画性をもった開発計画。

純関数（pure function）： 引数として受け取ったオブジェクトに如何なる変更も施さない関数。多くは戻り値を持つ。

修正関数（modifiers）： 引数として受け取った一つまたはそれ以上のオブジェクトに修正を施すような関数。

関数プログラミング作法（functional programming style）： 多くの関数が純関数として計画されたプログラミング手法。

不変性（invariants）： プログラムの実行過程で常に「真」の状態にあるべきとされる条件。

assert **文（assert statement）：** 条件を調べ満たされていないときには例外（AssertionError）を発生させる。

16.7 練習問題

この章で使ったコードは以下から利用できる：https://thinkpython.com/code/Time1.py。また、練習問題の解答例は以下からダウンロードできる：

https://thinkpython.com/code/Time1_soln.py。

練習問題 16.1 Time オブジェクトと数を引数として、この Time オブジェクトにこの数を乗じた Time オブジェクトを返す関数 mul_time を作成せよ。更にこの関数 mul_time を使って、レースの所要時間を表す Time オブジェクトと距離（マイル）を表す数を引数として、レースの平均ペース（時間毎マイル）を示す Time オブジェクトを返す関数を作成せよ。

練習問題 16.2 モジュール datetime はこの章で議論したと同じような date や time オブジェクトを提供しているが、さらに多くのメソッドや操作を含んでいる。詳しくは http://docs.python.org/3/library/datetime.html. を参照せよ。

1. 日日を引数にとり、曜日を返す関数 datetime を作成せよ。
2. 誕生日の日付を受け取り、ユーザの年齢と次ぎの誕生日までの日数、時、分、秒を表示するプログラムを作成せよ。
3. 異なった日に生まれた二人にとって、一人が他の日数の二倍になる日が存在する。これは二人にとって「二倍日」である。二つの誕生日を引数として受け取り、この二人の「二倍日」を計算するプログラムを作成せよ。
4. もう少し挑戦的な問題として、一般に二人の誕生日を引数として、一人の年齢が他の年齢 n 倍のなる日を計算するプログラムを作成せよ。

解答例：https://thinkpython.com/code/double.py

第17章　クラスとメソッド

これまで Python の オブジェクト指向の特徴の幾つかを使ってきたが、前二章のプログラムはユーザ定義型とそれに作用する関数類の関係を表現したものでないという意味で真のオブジェクト指向プログラムではない。次のステップは前二章で展開した関数を定義型との関係を陽に表現するメソッドに変換することである。この章のサンプルコードは https://thinkpython.com/code/Time2.py にある。また練習問題の解答例は https://thinkpython.com/code/Point2_soln.py. にある。

17.1　オブジェクト指向の特徴

Python はオブジェクト指向プログラミングを行う機能を提供しているという意味で**オブジェクト指向言語（object-oriented programming language）**である。オブジェクト指向プログラミングは以下のような特徴を持っている：

- プログラムはオブジェクトの定義、メソッドの定義を含む。
- 多くの操作がオブジェクトに対する演算として表現される。
- 各オブジェクトは現実世界の実体や概念に対応していて、オブジェクトに対し適用される関数は現実世界の実体が相互作用している様に対応している。

例えば、第 16 章で定義した `Time` クラスは人々が時間を記録する様に対応し、その関数は人々がその時間について行う操作に対応している。同様に第 15 章で現れた `Point` クラスや `Rectangle` クラスは数学的な概念に対応して定義される。

いままでは Python が提供しているオブジェクト指向の特徴を利用してこなかった。それらの特徴は厳格には必要でない。それらの特徴の多くはこれまで作ってきたプログラムの異なった表現による解を提供するにすぎない。しかし、多くの場合、この別表現はより簡便で、より正確にプログラムの構造を伝達することができる。

例えば、Time クラスでは、クラスの定義とそれに続く関数定義との間には関連があるかどうかは自明ではない。詳しくみると、全ての関数の引数の内少なくとも一つは Time オブジェクトであることが分かる。

この観察はある特別なクラスに付随する関数、つまり**メソッド（method）**の導入の契機になる。これまで文字列、リスト、辞書そしてタプルのためのメソッドをみてきた。この章ではユーザ定義型に対するメソッドを定義する。

メソッドは意味論的には関数と同じであるが、構文的には二つの点で関数とは異なる：

- メソッドはクラスとの関連を明白にするため、クラスの内部で定義される。
- メソッド起動に対する構文は関数を呼ぶ場合の構文とは異なる。

次の数節では前二章に登場した関数を取り上げ、それらをメソッドに変換する。変換自体は極めて機械的である。ステップに従えばできる。この変換に問題を感じないのであれば、何をするときも、二つ表現の内から最良と思うものを選択して使ってもらえばよい。

17.2　オブジェクトの print

第 16 章では Time クラスを定義し、練習問題 16.1 では print_time という関数を作成した：

```
class Time(object):
    """Represents the time of day."""

def print_time(time):
    print('%02d:%02d:%02d' % (time.hour, time.minute, time.second))
```

この print_time 関数を呼ぶためには引数に Time オブジェクトを渡さなければならない：

```
>>> start = Time()
>>> start.hour = 9
>>> start.minute = 45
>>> start.second = 0
>>> print_time(start)
09:45:00
```

print_time をメソッドにするために、しなければならないことの全てはこの関数定義をクラス定義の中に移動することだ（インデントの調整も忘れずに）：

```
class Time(object):
    def print_time(time):
        print('%02d:%02d:%02d' % \
          (time.hour, time.minute, time.second))
```

さて、この print_time を呼ぶ方法には二通りある。最初は（あまり普通はやらないが）関数呼び出しの構文である：

```
>>> Time.print_time(start)
09:45:00
```

このドット表記では、Time はクラス名、print_time はメソッド名、start は引数である。

第二は（普通にやる）メソッド構文によりものだ：

```
>>> start.print_time()
09:45:00
```

このドット表記では、print_time はメソッド、start はこのメソッドを発動させるオブジェクトで、**主語（subject）**と言う。文章の主語がその文章が何についてであるか示すと正に同じように、メソッドを発動する主語はそのメソッドが何についてあるかを示すものである。

メソッドの中では主語は第一番目の仮引数として代入される。今の場合は start が仮引数 time に代入される。慣例で、メソッドの第一仮引数は self という名前が使われる。したがって、より通常の形に print_time を書けば以下のようになる：

```
    def print_time(self):
        print('%02d:%02d:%02d' % \
          (self.hour, self.minute, self.second))
```

この慣例で書く理由は直接的な比喩にある：

- 関数呼び出し print_time(start) の構文は関数が活動主体であると示唆している。差詰め、「さあ、print_time よ。ここにお前さんがプリントするオブジェクトが一つあるよ」と言っているようなものだ。
- オブジェクト指向プログラミングでは、オブジェクトが活動主体である。start.print_time() のようにメソッドを起動することは、「やあ、start。自分のことは自分で処理してください」と言っているようなものだ。

このような見方の変更はより洗練されたものかもしれないが、それが有用であることは自明ではない。これまでの例では有用とはなっていない。しかし、責任主体を関数からオブジェクトに移すことは時として用途が多い関数を書き、よりメンテナンスが容易で再利用可能な関数を書くことを可能にするのだ。
練習問題として、関数time_to_int（16.4節）をメソッドとして書き換えよ。関数int_to_timeはメソッドに書き換える理由がない。これを発動させるオブジェクトがあるかな？

17.3　別な例

16.3節で登場した関数incrementをメソッドとして書き換える：

```
# クラスTimeの定義に中で
    def increment(self, seconds):
        seconds += self.time_to_int()
        return int_to_time(seconds)
```

ここでtime_to_intは直前の練習問題のようにメソッドとして書かれているものとする。さらに、このメソッドは純関数をメソッドにしたもので、修正関数ではないことにも注意すること。
このincrementの発動は以下のようになる：

```
>>> start.print_time()
09:45:00
>>> end = start.increment(1337)
10:07:17
```

主語startが関数の第一仮引数selfに代入される。引数1337は第二仮引数secondsに代入される。
この機構は特にエラーに遭遇したときに混乱の元になる。例えば、incrementに二個の引数を与えて発動させると以下のようにエラーになる：

```
>>> end = start.increment(1337, 460)
TypeError: increment() takes 2 positional arguments but 3 were given
```

このエラーメッセージには最初混乱する。括弧の中は二つの引数になっているからだ。しかし、主語それ自体も引数として考えるので、合わせて三個ということになる。

序ながら**位置固定引数（positional argument）**と言っているのは仮引数名で指定するキーワード引数でない引数である。例えば以下のような関数呼び出しを例にとると：

```
sketch(parrot, cage, dead=True)
```

parrot と cage は位置固定引数で、dead はキーワード引数である。

17.4 もっと複雑な例

関数 is_after（16.1 節）は引数に二つの Time オブジェクトを取るから少し複雑になる。この場合最初の仮引数の名前は self、二番目を other とするのが慣例である：

```
# クラス Time の定義に中で
    def is_after(self, other):
         return self.time_to_int() > other.time_to_int()
```

このメソッドを使うには、一つのオブジェクトでこれを発動させ、他のオブジェクトは引数としてこのメソッドに渡す：

```
>>> end.is_after(start)
True
```

この構文の愉快なのは英文を読むような順序に並んでいることだ、つまり、“end is after start?”。

17.5 init メソッド

init メソッド（“initialization”の省略）はオブジェクト具現化の際に発動される特殊なメソッドである。正式名は__init__ （二つのアンダースコア記号に init が続き更に二つのアンダースコア記号）。

Time クラスの init メソッドは以下のようなものだろう：

```
# クラス Time の定義に中で
    def __init__(self, hour=0, minute=0, second=0):
        self.hour = hour
        self.minute = minute
        self.second = second
```

__init__の引数の名前を属性と合わせることは習慣だ。文 self.hour = hour の意味は引数 hour の値は self の属性 hour の値として保存されるということである。

```
>>> time = Time()
>>> time.print_time()
00:00:00
```

もし引数一つを与えると、その値は hour の値を再定義する：

```
>>> time = Time(9)
>>> time.print_time()
09:00:00
```

もし引数二つを与えると、それらは hour,minute の値を再定義する：

```
>>> time = Time(9, 45)
09:45:00
```

引数三つを与えると既定値の全てが無効になる。

練習問題として、Point クラスの init メソッドを作れ。引数は x と y で、対応する属性に代入されるようにせよ。

17.6　__str__メソッド

__str__メソッドはオブジェクトの文字列による表現を返す特殊なメソッドである。

Time オブジェクトのための str メソッドは以下のようなものだ：

```
# クラス Time の定義に中で
    def __str__(self):
        return '%02d:%02d:%02d' % \
          (self.hour, self.minute, self.second)
```

Time オブジェクトを print しようとすると、str メソッドが発動される：

```
>>> time = Time(9, 45)
>>> print(time)
09:45:00
```

わたしが新しいクラスを書くときは、殆ど常にメンテナンスの容易さから__init__メソッドを書き、デバッギングに有用なので__str__メソッドを書くことにしている。

練習問題として、Point クラスの str メソッドを作成し、Point オブジェクトを一つ生成し、それを print せよ。

17.7 演算子の多重定義

特殊なメソッドの定義によって、ユーザ定義型のオブジェクトに対する演算の振る舞いを変更することができる。例として、Time クラスの__add__メソッドで加算演算子+を定義してみよう：

```
# クラス Time の定義に中で
    def __add__(self, other):
        seconds = self.time_to_int() + other.time_to_int()
        return int_to_time(seconds)
```

これを使ってみる：

```
>>> start = Time(9, 45)
>>> duration = Time(1, 35)
>>> print start + duration
11:20:00
```

Time オブジェクトの+演算子を使うと Python はこの__add__を発動する。結果を print しようとすると、__str__が発動される。

ユーザ定義型のオブジェクトに対して演算子の振る舞いを変更することを**演算子の多重定義（operator overloading）**と言う。Python の全ての演算子には__add__のような対応する特殊メソッドがある。詳細はhttp://docs.python.org/3/reference/datamodel.html#specialnames. を参照せよ。

練習問題として、Point クラスのオブジェクトに対する add メソッドを作成せよ。

17.8 型別処理

前の節では二つの Time オブジェクトを被演算子とする add メソッドを作成したが、一つの Time オブジェクトに整数型の数を加えることも考えられる。以下は引数 other の型をチェックする__add__メソッド改訂版で、関数 add_time か、increment かの呼び出しに繋げる：

```
# クラス Time の定義に中で
    def __add__(self, other):
        if isinstance(other, Time):
            return self.add_time(other)
        else:
```

```
            return self.increment(other)

    def add_time(self, other):
        seconds = self.time_to_int() + other.time_to_int()
        return init_to_time(seconds)

    def increment(self, seconds):
        seconds += self.time_to_int()
        return int_to_time(seconds)
```

組み込み関数 isinstance は値とクラスオブジェクトを引数として受け取り、その値がそのクラスのインスタンスであると True を返す。もし other が Time オブジェクトであると、__add__メソッドは add_time を発動させる。そうでないときは increment を発動させる。引数の型によって異なった計算処理を行うのでこのような演算を**型別処理（type-based dispatch）**と呼ぶ。

異なった型の演算に使ってみる：

```
>>> start = Time(9, 45)
>>> duration = Time(1, 35)
>>> print(start + duration)
11:20:00
print(start + 1337)
10:07:17
```

不幸にしてこの加算演算は交換可能でない。整数を被演算子の最初に置くとエラーになる。

```
>>> print(1337 + start)
TypeError: unsupported operand type(s) for +: 'int' and 'instance'
```

問題は Time オブジェクトに整数を加えることを要請する替わりに、Python は整数に Time オブジェクトを加算することを要請することになり、どうしてよいか分からなくなっていることである。しかし、上手い解決策がある。__radd__ (“right side add”の略) は特殊なメソッドで Time オブジェクトが演算子+の右側にあるときにこのメソッドが発動される。

今の場合の定義は以下のようになる：

```
# クラス Time の定義に中で
    def __radd__(self, other):
```

```
        return self.__add__(other)
```

使ってみる：

```
>>> print(1337 + start)
10:07:17
```

練習問題として、クラス Point で Point オブジェクトでもタプルでも有効な add メソッドを作成せよ。

- 第二被演算子も Point オブジェクトであるときには、戻り値はその被演算子の x 座標の和、y 座標の和を要素とする新規の Point オブジェクトになる。
- 第二被演算子がタプルのときにはタプルの初めの要素を x 座標に加え、第二の要素を y 座標に加えた新たな Point オブジェクトを戻す。

17.9 多態性

型別処理は必要なときは便利にものだが、(幸いにして) 常に必要になる訳ではない。よくあるあこであるが、異なった型の引数に対して正確に動作する関数を書くことによってこれを避けることができる。例として 11.2 節で取りあげた histogram では一つの単語の中の文字頻度をカウントするのに使った。

```
def histogram(s):
    d = dict()
    for c in s:
        if c not in d:
            d[c] = 1
        else:
            d[c] += 1
    return d
```

この関数はその要素がハッシュ可能ならば、つまりキーとして使えるならば、引数はリスト、タプル、それに辞書でさえ有効だ。以下はリストの例である：

```
>>> t = ['spam', 'egg', 'spam', 'spam', 'bacon', 'spam']
>>> histogram(t)
{'bacon': 1, 'egg': 1, 'spam': 4}
```

いくつかの型に対しても有効な関数は**多態的 (polymorphic)** だと言う。多態性はコードの再利用に役立つ。例えば、組み込み関数 sum は配列の要素の総和を求める関数であるが、加算演算子が定義されている要素であれば、如何なる配列でも構わない。

```
>>> t1 = Time(7, 43)
>>> t2 = Time(7, 41)
>>> t3 = Time(7, 37)
>>> total = sum((t1, t2, t3))
>>> print(total)
23:01:00
```

一般に関数内の全ての操作がある型に対して有効であるならば、その関数はその型に対して有効である。

多態性の面白さは既に作成した関数が当初計画しないでいた型に対しても有効だというのを発見すると言った予期しないことがあることだ。

17.10　デバッギング

クラスの属性はプログラムの実行中に任意に追加することができるが、もし型に対して厳格な考え方を持つなら、同じ型のオブジェクトが異なった属性を持つことは疑惑に満ちた習慣だ。オブジェクトの初期化の際に必要な属性を初期化しておくことは健全であろう。あるオブジェクトがある特定の属性を持っているかどうかを調べるには組み込み関数 hasattr が役に立つ（15.7 節を参照のこと）。

あるオブジェクトの属性にアクセスするもう一つの方法は組み込み関数 vars を使うことだ。これはこのオブジェクトを引数に取り、そのオブジェクトの属性名（文字列である）とその値を辞書として戻す。

```
>>> p = Point(3, 4)
>>> vars(p)
{'y': 4, 'x': 3}
```

この機能をデバッギングに使うために、簡便な関数を作成する：

```
def print_attributes(obj):
    for attr in vars(obj):
        print(attr, getattr(obj, attr))
```

print_attributes はオブジェクトの辞書を横断的に眺め、その属性と対応する値を表示する。組み込み関数 getattr はオブジェクトと属性を引数に取り、属性の値を返す関数である。

17.11 インタフェースと実装

オブジェクト指向によるプログラム設計の一つの目標はソフトウエアをより管理・維持し易くすることである。この意味するところはシステムの他の部分の変更があってもプログラムが動くようにしておくことができ、且つ、その変更に適合するようにプログラムを変更できることだ。

この目標を達成するための設計原理はインタフェースと実装を分離して置くことである。オブジェクトに関して言えば、クラスが提供するメソッドはその属性が如何に表現されるかに依らないようにすべきである。

例えば、この章では時刻を表現するクラスを作成した。このクラスが提供するメソッドは time_to_int、 is_after、そして add_time である。これらのメソッドはいくつかの方法で実装できる。実装の細部は時刻の表現の仕方に依存している。この章では、Time オブジェクトの hour, minute, second という属性がそれである。他の表現方法として、深夜 12 時からの経過時間を秒にした一つ整数で表現することもできる。この実装に沿っていくつかのメソッドが作られることになる。is_after などは容易に書けるメソッドだ。書くのが難しいメソッドもあるだろう。

新たなクラスの展開に伴って、もっと改良された実装を見つけるかもしれない。プログラムの他の場所でこのクラスを使っていたとすると、この実装変更に伴ってインタフェースの変更が伴うとするとこれは時間の浪費とエラーの元になる。しかし、インタフェースを注意深く設計して置くとインタフェースの変更無しに実装の変更ができる。つまり、プログラムの他の部分への変更はナシで済む。

インタフェースを実装から分離しておくということは属性を隠せということだ。プログラムの他の部分（クラス定義の外）では、メソッドを使いオブジェクトの状態を読み、変更を行うようにするのだ。属性に直接にアクセスしないようにするわけだ。このような原理を**情報隠蔽（information hiding）**と言う。

17.12 語句

オブジェクト指向言語（object-oriented programming language）： ユーザ定義クラスやメソッド構文のようなオブジェクト指向プログラミングをサポートする特徴を持っている言語。

オブジェクト指向プログラミング（object-oriented programming）： データとそれを操作する命令をクラスとメソッドとして組織化するプログラミング・スタイル。

メソッド（method）： クラス定義の中で定義される関数そしてそのクラスのインスタンスによって発動される。

主語（subject）： メソッドを発動するオブジェクト（実体）。

位置固定引数（positional argument）： 仮引数名を伴なわない引数、つまりキーワード引数でない。

演算子の多重定義（operator overloading）： ユーザ定義型に直に有効なように+のような演算子の振る舞いを変える。

型別処理（type-based dispatch）： 被演算子の型を判定し異なった型には異なった関数を当てはめるプログラミング手法。

多態的（polymorphic）： 一つ以上のデータ型に適用できるようになっている関数の性格。

情報隠蔽（information hiding）： 一つのオブジェクトが提供するインタフェースがその実装特にその属性の表現に依存しないようにするという原理。

17.13　練習問題

練習問題 17.1 https://thinkpython.com/code/Time2.py. をダウンロードせよ。そして Time オブジェクトの属性を真夜中からの経過時間（秒）を表す一つの整数に変更せよ。そして、この変更に見合うようにメソッド（そして関数 int_to_time）を変更せよ。main は変更してはいけない。このプログラムを実行すると変更前の表示と同じものがでるはずである。
解答例：https://thinkpython.com/code/Time2_soln.py

練習問題 17.2 この練習問題は Python でよく遭遇するが、しかし発見が難しいエラーについての教訓物語である。以下のようなメソッドを持つ Kangaroo と言うクラスを書け。

1. 属性 pouch_contents を空のリストで初期化する__init__メソッド。
2. 任意の型を引数に取りそれを pouch_contents に追加する put_in_pouch メソッド。
3. Kangaroo オブジェクトの文字列による表現（つまり print(オブジェクト)）とポーチの中味を表示する__str__メソッド。

コードのテストとして二つの Kangaroo オブジェクトを生成し、kanga と roo という変数に代入し、一つのオブジェクト kanga のポーチに roo を追加してみよう。

以下をダウンロードせよ https://thinkpython.com/code/BadKangaroo.py。これは前問に含まれる大きなしかもたちの悪いバグを含んだ解答例である。そのバグを修正せよ。立ち往生してしまったら、解答例を参照しよう。

解答例：https://thinkpython.com/code/GoodKangaroo.py

第18章　継承

オブジェクト指向言語に付随する大きな言語的な特徴は**継承（inheritance）**である。継承は既存のクラスを改変することで新しいクラスを生成する能力のことだ。新しいクラスは既存のクラスのメソッドを引き継ぐので、これは「継承」と呼ばれている。この章ではトランプカードゲーム、カードの組み、ポーカーの手と言った表現のためのクラスを取りあげ、継承の意義を検証する。

ポーカーゲームを知らない読者は http://en.wikipedia.org/wiki/Poker を参照のこと。練習問題を行うのに必要な知識はその都度示すから知らなくても構わない。

この章のサンプルコードは https://thinkpython.com/code/Card.py. からダウンロードできる。

18.1　カードオブジェクト

一組のトランプカードは 4 つのスートと 13 のランクからなる 52 枚のカードで構成される。スートはブリッジの得点順序に従うとスペード、ハート、ダイアモンド、クラブがあり、ランクはエース、2，3，4，5，6，7，8，9，ジャック、クイーン、キングがある。ゲームに依るが、エースはキングより点が高いこともあれば、2 より低い場合もある。

この一枚のトランプカードを表現する新規のオブジェクトを定義したいとすると、それが持つべき属性は `rank` と `suit` であることは自明である。しかし、型についてはそうでもない。一つの可能性として、スートでは`'Spade'` というような、ランクでは`'Queen'` といいうような文字列だ。実装に伴う問題として、どちらのカードが上位かなどを計算することが文字列では難しい。

他の可能性はランクやスートを**符号化（encode）**した整数を使うことである。ここで「符号化」の意味するところはスートと数、ランクと数の写像を定義することである。この種の符号化は（暗号のような）秘密であることは意味しない。以下のテーブルはスートと対応する整数を示した：

Spades $\mapsto$ 3
Hearts $\mapsto$ 2
Diamonds $\mapsto$ 1
Clubs $\mapsto$ 0

高い得点のスートが大きな数に写像されているのでこの符号化はカードの比較が容易だ。ランクの写像はかなり自明で、ランクの数字を対応する数に写像すればよい、そして顔カードの写像は以下のようにする：

Jack $\mapsto$ 11
Queen $\mapsto$ 12
King $\mapsto$ 13

ここで写像には Python のコードの一部でないことを明瞭にするために $\mapsto$ 記号を使った。これはプログラム設計の一部であり、コードには直接には現われない。

カードオブジェクトは以下のようなものになる：

```
class Card:
    """Represents a standard playing card. """

    def __init__(self, suit=0, rank=2):
        self.suit = suit
        self.rank = rank
```

いつものように、init メソッドは各属性を選択機能つき引数として受け取る。既定値はクラブの 2 である。

一枚のカードを生成することはこの Card クラスを好きなスートとランクで呼ぶことである：

```
queen_of_diamonds = Card(1, 12)
```

18.2　クラスの属性

ヒトが理解できるかたちでカードを表示しようとすると、整数コードと対応するスートとランクの写像が必要になる。その自然なかたちは文字列のリストである。このリストを**クラス属性（class attributes）**として設計しよう：

```
# クラス Card の定義に中で
    suit_names = ['Clubs', 'Diamonds', 'Hearts', 'Spades']
    rank_names = [None, 'Ace', '2', '3', '4', '5', '6', '7',
                  '8', '9', '10', 'Jack', 'Queen', 'King']
```

```
    def __str__(self):
        return '%s of %s' % (Card.rank_names[self.rank], \
                             Card.suit_names[self.suit])
```

suit_names や rank_names のようにクラスの内部で定義され、クラス外の任意のメソッドから参照できる変数はそのクラスに付随しているのでクラス属性と呼ばれる。rank や suit の変数はインスタンスに付随しているので**インスタンス属性（instance attributes）**と呼ばれる。

両方の属性はドット表記でアクセスできる。例えば、__str__メソッド内では、self は一つのカードオブジェクトであり、rank はそのランクを表す。同様に Card はクラスオブジェクトであり、Card.rank_names はこのクラスに付随している文字列のリストである。各カードオブジェクトは固有の rank と suit を持つが、suit_names や rank_names は共通に一つだけである。

まとめると、Card.rank_names[self.rank] の表式の意味は「配列のインデックスとしてオブジェクト self の属性 rank をクラス Card の rank_names リストに代入し、ランクの名前として固有な文字列を選べ」となる。

ランクが 0 のカードはないので rank_name の最初の要素は None にした。この場所埋めの None を使うことでリストのインデックス 2 は文字列'2' に写像される嬉しい性質が得られる。こんな小細工をしないで辞書を使うこともできる。

これまでのメソッドを使うと、カードを生成し、そのカードをプリントすることができる：

```
>>> card1 = Card(2, 11)
>>> print(card1)
Jack of Hearts
```

図 18.1 は Card クラスオブジェクトと 1 枚のカードの実体を示す Card インスタンスの図である。Card はクラスオブジェクトであるので枠のラベルは type であり、card1 の枠のラベルは Card である。スペースの関係で suit_names と rank_names の中身は表示していない。

18.3 カードの比較

組み込み型に関しては比較演算子（<,>,==,etc）がより大きい、より小さい、他と等しいなどの値の比較に使える。ユーザ定義型については、“less than”を意味する__lt__メソッドが提供する組み込み型の操作を再定義することになる。

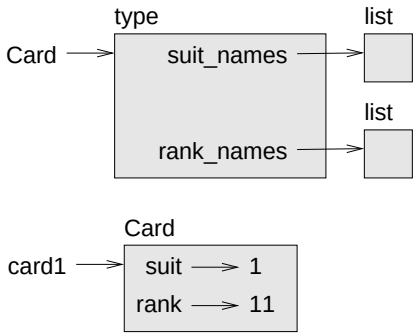


図 18.1: オブジェクト図

メソッド__lt__は二つの引数 self と other を受け取り、self が other 未満であると Ture を返す。

カードの順位は自明ではない。例えば、クラブの 3 とダイアモンドの 2 ではどちらが上か？一つは高いランクを持つが、他はスートでは上である。比較を可能にするためには、ランクとスートのどちらが重要かを決めないといけない。多分この答えは実際のゲームのルールによる。しかし、ここでは問題を簡単にするために、スートはより重要だと恣意的だが設定しよう。従って、スペードの全てのカードはどんなダイアモンドより高位になる、そして以下同様である。

このように決めると比較演算子のメソッド__lt__を書くことができる。

```
# クラス Card の定義に中で:

    def __lt__(self, other):
        #  まずスートを調べる
        if self.suit < other.suit: return True
        if self.suit > other.suit: return False

        # スートは同じなので、ランクを調べる
        return self.rank < other.rank
```

タプルの比較を用いるともっと簡便に書ける：

```
# クラス Card の定義に中で:

    def __lt__(self, other):
        t1 = self.suit, self.rank
        t2 = other.suit, other.rank
        return t1 < t2
```

(訳注：ここの演算子<はタプルに定義されている比較演算子である。)

練習問題として、Time オブジェクトのための__lt__メソッドを書け。タプルの比較を用いることでもよいが、整数の引き算を使うこともできる。

18.4 積み札

カードの定義が終わったので、次は積み札の定義である。積み札は複数のカードからなるので、カードのリストを属性に持つことは自然なことである。以下は Deck クラスの定義である。init メソッドでは属性 cards を生成、ここでは 52 枚の標準カードを持たせた。

```
class Deck:

    def __init__(self):
        self.cards = []
        for suit in range(4):
            for rank in range(1, 14):
                card = Card(suit, rank)
                self.cards.append(card)
```

カードの山を作成する最も安易な方法は入れ子のループを使うことだ。外側のループは 0 から 3 までスートを列挙し、内側のループは 1 から 13 までランクを列挙している。繰り返し毎に新しいカードが生成され、self.cards に追加される。

18.5 積み札のプリント

Deck のための__str__メソッドは以下のようになる：

```
# クラス Deck の定義に中で
    def __str__(self):
        res = []
        for card in self.cards:
            res.append(str(card))
        return '\n'.join(res)
```

このメソッドは大きな文字列を纏める有効な方法を示している。つまり、文字列のリストを作り、join を使い纏める。カードに対する組み込み関数 str はカード

毎にその__str__メソッドを発動させ、文字列の表現を返す（訳注：Card オブジェクトで定義された__str__メソッドは関数 str(card) でも発動される）。join メソッドは改行文字で発動されているので、各カードは一行毎にプリントされる。以下は結果である：

```
>>> deck = Dec()
>>> print deck
Ace of Clubs
2 of Clubs
3 of Clubs
.........
10 of Spades
Jack of Spades
Queen of Spades
King of Spades
```

結果は 52 行の表示であるが、これは改行文字を含む一つの長い文字列である。

18.6　追加・移送・シャッフル・ソート

カードの扱いとして欲しいメソッドとしては、カードの山から一枚のカードを移送する、及び追加するがある。

第一のメソッドに対して、pop が便利な方法を提供できる：

```
# クラス Deck の定義に中で
    def pop_card(self):
        return self.cards.pop()
```

pop はリストの最後の要素を削除するので、積み札の底からカードを抜き取ることになる。実際のゲームでは「底から抜き取り」は眉をひそめる行為であるが、今の場合はよしとする。

カードの追加は append を使う：

```
# クラス Deck の定義に中で
    def add_card(self, card):
        self.cards.append(card)
```

このような大したこともしないで他の関数を使うメソッドは**ベニヤ（veneer）**と呼ばれる。これは良質の木材の薄い層を張り合わせて廉価な木材を作り出す木工

技術の比喩からきている。ここでは積み札に対して妥当なリストの処理を「薄い」メソッドとして定義している。

シャッフルも同様にして、random モジュールの shuffle 関数を使って shuffle メソッドを Deck の内部に作る：

```
# クラス Deck の定義に中で
    def shuffle(self):
        random.shuffle(self.cards)
```

random モジュールのインポートを忘れないように。

練習問題として、Deck のもう一つのメソッド sort をリストのメソッド sort を使って作成せよ。メソッド sort はソートの順位を決めるために Card オブジェクトの__lt__メソッドを使う。

18.7 継承

継承は既存のクラスを改変することで新しいクラスを生成する能力のことだ。新しいクラスは既存のクラスのメソッドを引き継ぐので、これは「継承」と呼ばれている。この比喩を拡張して、既存のクラスは**親クラス（parent class）**、新しいクラスは**子クラス（child class）**と呼ばれる。

例として、一人のカードプレーヤが保持しているカード、つまり手札を考えてみよう。手札は積み札に似ている。どちらもカードの集合からなっていて、追加や移送の操作を必要とする。

手札は積み札と異なる面もある。積み札では意味を成さないような操作が手札については必要になることがある。例えば、ポーカーではどちらが勝者かを決めるために二人の手札を比較する必要がある。また、ブリッジでは賭けをするために、手札のスコアを計算する必要がある。

このようなクラス間の異同はそれ自体で継承という概念に繋がっている。子のクラスの定義は他のクラスの定義と似ているが、親クラスの名前が括弧の中に入る：

```
class Hand(Deck):
    """Represents a hand of playing cards."""
```

この定義では Hand クラスは Deck を継承している。つまり、Deck クラスのメソッド pop_card や add_card が Hand でも使えるわけだ。Hand は Deck クラスの__init__メソッドも継承できるが、それは欲しいものではない。52 枚のカードを手札とする替わりに init メソッドは空のリストで始めることが似つかわしい。以下は Hand クラスの init メソッドであるが、これは Deck クラスのそれを再定義することになる：

```
# クラス Hand の定義に中で
    def __init__(self, label=''):
        self.cards = []
        self.label = label
```

従って、一つの手札を生成するときには Python はこの init メソッドを発動する：

```
hand = Hand('new hand')
print(hand.cards)
[]
print(hand.label)  #訳注：属性 label（単なる文字列）の印刷
new hand
```

しかし、その他のメソッドは継承するので、pop_card や add_card はここでも使える：

```
>>> deck = Deck()
>>> card = deck.pop_card()
>>> hand.add_card(card)
>>> print(hand)  #訳注 Hand クラスのインスタンス hand そのものの印刷
King of Spades
```

この手続きを Deck のメソッド move_card としてカプセル化することは自然なことである：

```
# クラス Deck の定義に中で
    def move_cards(self, hand, num):
        for i in range(num):
            hand.add_card(self.pop_card())
```

move_cards は引数を二つ持ち、一つは Hand オブジェクトそして操作するカードの枚数 num である。このメソッドは self も hand を修正し、戻り値は None であることに注意。

カードゲームのいくつかではカードの移動は手札から手札、手札から積み札へ戻すなどがあるが、この move_cards を使えばよい。self は Deck でも Hand でもよい。更に仮引数は hand となっているけれど、これは deck でもよい。

継承は有用な機能だ。継承なしては繰り返しになってしまうプログラムも継承によって綺麗にかけることがある。継承は親クラスを変更することをしないでも親クラスの振る舞いをカスタマイズできるので、コードの再利用の機能を果たす。あるケースでは、継承は問題自体が持つ性質を反映することがあり、プログラム

をより読みやすくする。一方、継承はプログラムを読みにくくすることもある。あるメソッドが発動されたとき、その定義がどこにあるのかが明白でないこともある。関連するコードがいくつかのモジュールに渡って散乱していることもあるうる。継承を使ってなし得る多くのことがそれ無しでも充分にやれることもある。

18.8 クラス図

これまでプログラムの状態を示すスタック図、オブジェクトの属性とそれらの値の関係を示したオブジェクト図をみてきた。これらの図はプログラム進行中のスナップショットであり、プログラムの進行に連れて変わる。それらは極めて詳細を究め、ある目的には詳細過ぎる。**クラス図（class diagram）**はプログラムの構造をより抽象的に表現したものである。個々のオブジェクトを示すかわりに、クラスとクラス間の関係を示す。

複数のクラスの間にある関係としては以下のものがあり得る：

- 一つのクラス・オブジェクトが他のクラスのオブジェクトの参照を含む。例えば、個々の `Rectangle` オブジェクトは `Point` オブジェクトの参照を含んでいるし、個々の Deck オブジェクトは多くの Card オブジェクトの参照を含んでいる。この関係は「一つの `Rectangle` は一つの `Point` を持っている」にあるように **HAS-A 関係（HAS-A relationship）**と呼ぶ。
- 一つのクラスが他のクラスを継承する。この関係は「Hand は Deck の一種である」にあるように **IS-A 関係（IS-A relationship）**と呼ぶ。
- 一つのクラスの変更が他のもう一つのクラスの変更を要求するといった依存関係にある二つのクラス間の関係。

クラス図はこれらの関係を図式したものだ。例えば、図 18.2 は `Card`、`Deck`、`Hand` 間の関係を示したものだ。白抜きの矢印は `IS-A` の関係を表示している。この図で

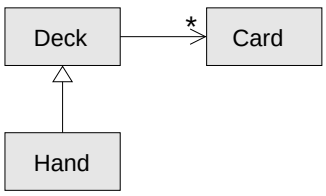


図 18.2: クラス図

は Hand は Deck を継承していることを示している。普通の矢印は `HAS-A` の関係を

表示している。この図では Deck は Card の参照を持っていることを示している。矢印の上部のスター（*）は**重複度（multiplicity）**である。これは Deck クラスには何個の Card オブジェクトの参照があるかを表示するものである。これは単に 52 といった単なる数字、5..7 と言った区間、任意の数を示すスターなどでよい。この図では Deck は任意の数のカードを持ち得るのでスターになっている。

この図では依存関係は何もない（訳注：「依存関係」の説明は語句の**依存関係（dependency）**をみよ）。依存関係があるとこれは破線の矢印で示すか、沢山の依存関係があるときにはそれらは省略される。

もっと詳細に渡る図にしようとすれば、Deck オブジェクトはカードのリストを含んでいることを示すことになるが、クラス図にはリスト、辞書などの組み込み型は含めない。

18.9　デバッギング

継承はあるオブジェクトのあるメソッドの発動がどちらのメソッドの発動か不明であるということがあるので、デバッグは新たな問題をはらんでくる。

いま Hand オブジェクトに付随した関数を書こうとしたとする。その関数はポーカーの手札、ブリッジの手札だといったどのような手札でも有効したいと思ったとしよう。こうなると、例えば shuffle というメソッドを発動させたときに、Deck で定義されたものか、そのサブクラスで再定義されたものの 1 つかということになる。このように、どのメソッドが発動されたか不確かになってしまう。この簡単な解決法は当該のメソッドの先頭に print 文を追加しておくことだ。もし Deck.shuffle が発動されると、'Running Deck.shuffle' とメッセージが出るわけである。これで実行の流れが追跡できる。

別な方法はオブジェクトとメソッド名（文字列）を引数に取り、そのメソッドを定義しているクラスを返す以下の関数を使うことだ：

```
def find_defining_class(obj, meth_name):
    for ty in type(obj).mro():
        if meth_name in ty.__dict__:
            return ty
```

例だ：

```
>>> hand = Hand()
>>> find_defining_class(hand, 'shuffle')
<class 'Card.Deck'>
```

（訳注：モジュール名が Card の場合である）

従って Hand の shuffle メソッドはクラス Deck で定義されたものを使っていることが分かる。関数 find_defining_class ではメソッドが探索されるクラスのリストを得るメソッド mro が使われている（“MRO” は “method resolution order” の略である）。

以下はプログラム設計に際しての示唆である。何らかの理由でメソッドを書き換える必要があるときにはいつでも、そのインタフェースは古いものと同じにすべきである。同じ型の仮引数で、戻り値も同じ型にする。つまり、同じ事前条件と同じ事後条件に従うようにする。この条件に従ってプログラムを設計すると、上位クラス（例えば Deck）のインスタンスで有効な関数を作っておくと、この関数はそのサブクラス（例えば Hand や PokerHand）のインスタンスでも有効に使えることになる。このルール（リスコフ (*Liskov*) の置換原理）に違反すると、そのコードはカードで作った家のように簡単に崩壊する。

18.10 データカプセル化

前章では「オブジェクト指向設計」とも呼ぶべきプログラム開発設計の過程を示した。まず必要な実体、Time、Point、Rectangle を確定した。そして、これらを表現するためのクラスを定義した。各々において、オブジェクトと現実世界の実体との明白な対応関係（少なくとも数学的世界の）がある。

しかし、どうようなオブジェクトが必要で、それらがいかに相互作用するのかが明白でない場合も往々にしてある。このような場合には別の方法を取る必要がある。関数のインタフェースはカプセル化や一般化の原則の実現体であるように、クラスのインタフェースも**データカプセル化（data encapsulation）**の原則を実現したものにすべきだ。

13.8 節で議論したマルコフ解析はよい実例を提供してくれる。
https://thinkpython.com/code/markov.py をダウンロートしてそのコードを眺めると、いくつかの関数から読み書きされる二つの大域変数 suffix_map と prefix とが定義されている：

```
suffix_map = {}
prefix = ()
```

これらの変数は大域変数であるので、一度に一つのテキストの解析ができるだけである。もし二つのテキストを同時に解析するとなると、同じデータ構造に追加することになる（これ自体は興味のある結果を生むが）。複数の解析を別々なデータ構造に収めようとすると、各解析の状態を一つのオブジェクトにカプセル化する必要がある。それは以下のようになる：

```
class Markov:
    def __init__(self):
         self.sufix_map = {}
         self.prefix = ()
```

次ぎに関数をメソッドに変換する。例として、process_word 関数を変換してみる：

```
    def process_word(self, word, order=2):
        if len(self.prefix) < order:
            self.prefix += (word,)
            return

        try:
            self.suffix_map[self.prefix].append(word)
        except KeyError:
            # if there is no entry for this prefix, make one
            self.suffix_map[self.prefix] = [word]

        self.prefix = shift(self.prefix, word)
```

関数の中味を変えずにデザインを変える手法はプログラムの変更方法に対する再因子分解のもう一つの事例である（4.7 節）。

この例はオブジェクトやメソッドをデザインするための開発計画を示唆している：

1. まず、大域変数（もし必要なら）を使って読み・書きする関数を書く。
2. そのプログラムが動くようになったら、それらの大域変数とそれらを使う関数の関係を探す。
3. 関連する変数を一つのオブジェクトの属性としてカプセル化する。
4. 付随する関数は新たなクラスのメソッドになるように変換する。

練習問題として、https://thinkpython.com/code/markov.py をダウンロードし、上記のステップに従い大域変数を新たなクラス Markov の属性としてカプセル化せよ。

解答例：https://thinkpython.com/code/Markov.py（大文字の M に注目）。

18.11 語句

符号化（encode）： 二つの集合の対応関係を構築することによって一つ値の集合を他の値の集合で表現する。

クラス属性（class attributes）： クラスオブジェクトに属する属性。クラス属性はクラスの定義の中で定義されるが如何なるメソッドの外側に置く。

インスタンス属性（instance attributes）： クラスのインスタンスに属する属性。

ベニヤ（veneer）： 大した計算過程も無く他の関数に異なったインターフェースを提供するメソッドや関数。

継承（inheritance）： 既存のクラスの改訂版になるような新しいクラスを定義する能力。

親クラス（parent class）： それから子クラスが派生する元のクラス。

子クラス（child class）： 既存のクラスから派生して作られた新しいクラス。「下位クラス」とも呼ばれる。

IS-A 関係（IS_A relationship）： 子クラスとその親クラスといった関係。

HAS-A 関係 （HAS_A relationship）： 二つのクラスで一つクラスのインスタンスが他のクラスのインスタンスへの参照を含むような関係。

依存関係（dependency）： 一つのクラスのインスタンスが他のインスタンスへの参照を含むような二つのクラスの関係。

クラス図（class diagram）： 一つのプログラムの中に現れるクラスとそれらの間の関係を示すグラフィカルな表現。

重複度（multiplicity）： HAS-A 関係の関係にある二つのクラスで他のクラスののインスタンスへの参照が何回あるかを示す表示法。

データカプセル化（data encapsulation）： プログラムの原型で使っていた大域変数を最終版ではインスタンス属性にして隠すプログラム開発法。

18.12　練習問題

練習問題 18.1 以下のようなプログラムに対してクラスやこのクラスの間の関係を示す UML クラス図を作成せよ。（訳注：統一モデリング言語（UML）によるクラス図は本文の図 18.2 で表示されているクラス図と思えばよい。この練習問題では「依存関係」もあることに注意。）

```
class PingPongParent:
    pass

class Ping(PingPongParent):
    def __init__(self, pong):
        self.pong = pong


class Pong(PingPongParent):
    def __init__(self, pings=None):
        if pings is None:
            self.pings = []
        else:
            self.pings = pings

    def add_ping(self, ping):
        self.pings.append(ping)

pong = Pong()
ping = Ping(pong)
pong.add_ping(ping)
```

練習問題 18.2 以下はポーカーの可能な手を値打ちの少ない順序（従って確率の大きな順序）に並べたものである：

pair: 同じランクを持った二枚のカード

two pair: 同じランクを持った二枚のカードの二組

three of a kind: 同じランクを持った三枚のカード

straight: 引き続くランクからなる五枚のカード（エースは高くも低くにもなれる。従って、Ace-2-3-4-5 は straight、10-Jack-Queen-King-Ace も straight、しかし、Queen-King-Ace-2-3 は straight ではない）

flush: 同じスートの五枚のカード

full house: 一つのランクの三枚のカードと他のランクの二枚のカード

four of a kind: 同じランクの四枚のカード

straight flush: 同じスートで引き続くランク（上で定義したように）からなる五枚のカード

さてこの練習問題の目標は上のような手を引き当てる確率を推定することにある。

1. 以下のファイルを https://thinkpython.com/code からダウンロードせよ。
   `Card.py` ：これは `Card`、`Deck`、`Hand` のクラス記述の完全版である。
   `PokerHand.py` ：ポーカーの手を表現するためのクラスの不完全な記述とそのテストコードである。

2. `PokerHand.py` を起動すると “7-card” ポーカーの七枚の手札が配られ、この手札に flush が含まれているかをチェックするようになっている。

3. `PokerHand` に `has_pair`、`has_twopair` というメソッドを追加せよ。これらのメソッドは手札に当該の手が含まれていると `True` を、そうでなければ `False` を返す。コードは任意の枚数の手札でも手を検出できること（5 とか 7 とかがよくある数であるが）。

4. 一つの手札に含まれる最高位の手を調べる `classify` メソッドを作成せよ。そして属性の `label` をそれに従ってセットせよ。例えば、“7-card” の手として flush を含んでいたら、`'flush'` という文字列を `label` に代入する。

5. この `classify` メソッドが動くことが確認できたら、次ぎのステップは種々の手の確率を求める番だ。`PokerHand.py` 内に積み札をシャッフルしてそれを手札に分割し、各手札を `classify` で分類し、特定の分類値が起こる回数を調べる関数を書け。

6. 分類値とその確率を表にしてプリントせよ。充分長く実行して沢山の手を発生させその確率が適当な桁で落ち着くまでプログラムを実行せよ。結果を http://en.wikipedia.org/wiki/Hand_rankings と比較せよ。

解答例： https://thinkpython.com/code/PokerHandSoln.py.

第19章　便利グッズあれこれ

この本の目的は最小限の範囲でPythonを学習できるようにすることである。それ故何か行う上で二つの方法があるときには、１つだけを取り上げ他は伏せてきた。ときとして二番目の方法は練習問題に含めてきた。

この章ではこれまで取り上げなかった便利グッズの幾つかについて触れることにしたい。Pythonは必ずしも必要としない「呼び物」を幾つか提供している。それら無しで優れたコードを書くことができるが、ときとしてそれらを使うと、すっきりして、読みやすく、効率のよい、時としてこの三つを兼ね揃えたコードを書くことができる。

19.1　条件付き表式

条件文については5.4節でみた。条件文は二つの値の内一つを選択するケースでよく用いられる。例を上げると：

```
if x > 0:
    y = math.log(x)
else:
    y = float('nan')
```

この文ではxが正の値か調べる。そうであるとmath.logが実行される。そうでないとmath.logはValueErrorの例外を発生してしまう。これを避けるために、ここでは“NaN”を生成する。これは“Not a Number”の意味の特別な浮動小数点値である。

さてこれを**条件付き表式（conditional expression）**で書くと以下のようによりすっきりと書ける：

```
y = math.log(x) if x > 0 else float('nan')
```

これは英語を読むように読める：“y gets log-x if x is greater than 0; otherwise it gets NaN”.

再帰関数もこの条件付き表式を使って書き直すことができる。例としてfactorial関数の再帰版を取り上げる。元のコードは：

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

これを書き直すと：

```
def factorial(n):
    return 1 if n == 0 else n * factorial(n-1)
```

となる。

もう一つの利用法は選択的引数の処理である。例として練習問題17.2のGoodKangarooのinitメソッドを取り上げる。元は：

```
    def __init__(self, name, contents=None):
        self.name = name
        if contents == None:
            contents = []
        self.pouch_contents = contents
```

書き直すと

```
    def __init__(self, name, contents=None):
        self.name = name
        self.pouch_contents = [] if contents == None else contents
```

となる。

一般に条件分岐の両方がreturn文とか同じ変数への代入とかの簡単な表式であるときには条件文を条件付き表式に置き換えることができる。

19.2　リスト内包

10.7節で写像とフィルターのコードパターンで考察した。例えば以下の関数は文字列のリストを引数として文字列の組み込みメソッドcapitalizeを各要素に適用し新たなリストを戻す：

```
def capitalize_all(t):
    res = []
    for s in t:
        res.append(s.capitalize())
    return res
```

これを**リスト内包（list comprehension）**を使って書き直すと：

```
def capitalize_all(t):
    return [s.capitalize() for s in t]
```

となる。角括弧の演算子があることから新たなリスト作成であることが分る。この角括弧の中の表式はリストの要素を表現し、for 節がどのような配列を横断的に処理するのか示している。

リスト内包の構文は変数 s が定義する前に表式に現われるので少し不恰好である。

リスト内包はデータフィルタリングでも使うことができる。以下はリスト t から大文字だけでできている文字列の要素を選択し新たなリストを作る関数である：

```
def only_upper(t):
    res = []
    for s in t:
        if s.isupper():
            res.append(s)
    return res
```

これを書き直すと：

```
def only_upper(t):
    return [s for s in t if s.isupper()]
```

となる。

リスト内包表現はコンパクトで for ループによる同等のものより速く、あるばあいにはずっと速い。それならばもっと早いところでこれに触れてほしかったとあなたは苦情を言うかもしれないが、それは予想していたことである。

言い訳になるが、リスト内包表現はループの中に print 文を入れることができないのでデバックが困難になる難点がある。リスト内包は計算が最初から明白に正しいことがわかっているような十分に単純なばあいに使うとよい。こんなことは初心者ではないはずだ。

19.3　ジェネレータ表式

ジェネレータ表式（generator expressions）はリスト内包に似ているが、角括弧の替わりに括弧を使う：

```
>>> g = (x**2 for x in range(5))
>>> g
<generator object <genexpr> at 0x7f4c45a786c0>
```

結果は値の配列を通じて如何に繰り返しの演算をするか表現したジェネレータ・オブジェクトである。しかしリスト内包と異なり、一時に値の計算をすることをしない。問い合わせがあるまで待っている。組み込み関数 next がこのジェネレータから次の値を得るのに使われる：

```
>>> next(g)
0
>>> next(g)
1
```

配列の最後の到達すると、next 関数は StopIteration 例外を吐き出す。値を for ループで繰り返し取り出す処理でも使える：

```
>>> for val in g:
...     print(val)
4
9
16
```

ジェネレータ・オブジェクトは常に配列の何処にいるかを追尾している for ループは next が抜け出たところを取り出す。ジェネレータを使い切ると、それ以降は StopIteration 例外を吐き出す：

```
>>> next(g)
StopIteration
```

ジェネレータ表式は sum、max そして min といった関数と一緒に使われることが多い：

```
>>> sum(x**2 for x in range(5))
30
```

19.4 any と all

Python はブーリアン値を要素とする配列を受け取りそれらの値に一つでも Ture があると Ture を返す組み込み関数 any を提供している。これはリストに対して動作する：

```
>>> any([False, False, True])
True
```

この関数はジェネレータ表式と一緒に使われることが多い：

```
>>> any(letter == 't' for letter in 'monty')
True
```

この例題では in 演算子と同じことをするだけなので有難味がない。しかし 9.3 節で展開したような探索関数の幾つかはこの any を使って書き直すことができる。例えば avoids は以下のようになる。：

```
def avoids(word, forbidden):
    return not any(letter in forbidden for letter in word)
```

この関数はほぼ英文を読むように書けている：“word avoids forbidden if there are not any forbidden letters in word.”

関数 any をジェネレータ表式と合わせて使うことは any が Ture を検出したらジェネレータは停止するので効率がよい。全ての配列を評価しないで済む。

Python はもう一つの組み込み関数 all を提供している。この関数は配列の要素全てが Ture のとき Ture を返す。練習問題として 9.3 節の uses_all をこの関数を使って書き直せ。

19.5 セット

13.6 節では文書に出現したが単語のリストにはない単語を見つける問題に出会った。そこで作成した関数は文書から出現した単語をキーとする辞書 d1 と単語のリストを含む辞書 d2 を引数とした。その関数の戻り値も辞書で d1 にあって d2 のない単語をキーとしたものであった。つまり：

```
def subtract(d1, d2):
    res = dict()
    for key in d1:
```

```
        if key not in d2:
            res[key] = None
    return res
```

これら辞書では値は全て None であった。これは記憶領域を無駄にしている。

Python は set と呼ばれる組み込み型を提供している。これは辞書に似ているがキーの集合で値を持たない。セットへの要素の追加は高速で、要素の検索を速い。そしてよく知られた集合演算を実行するメソッドや関数が提供されている。例えば集合の差を求めるためには difference というメソッドか減算演算子-を使えばよい。これを使えば subtract は以下のように書ける：

```
def subtract(d1, d2):
    return set(d1) - set(d2)
```

戻り値は辞書でなく、セットである。しかし繰り返しのような操作は振る舞いは同じである。

この本の練習問題の幾つかはこのセットを使うとコンパクトで効率のよいものに書き直すことができる。例えば、辞書を使った練習問題 10.7 の関数 has_duplication の解答例は：

```
def has_duplicates(t):
    d = {}
    for x in t:
        if x in d:
            return True
        d[x] = True
    return False
```

であるが、これは要素がはじめて現れたものであるとそれは辞書に追加され、同じ要素が現れると関数は Ture を戻り値として終了する。

これをセットを使って書き直すと：

```
def has_duplicates(t):
    return len(set(t)) < len(t)
```

一つの要素はセットでは一回のみである。だから配列で同じ要素が二回以上現れるとそのセットの大きさは配列 t より小さくなる。配列 t の重複がなければ、そのセットの大きさは配列の大きさと同じいなる。

第 9 章の練習問題の幾つかもセットで書き直すことができる。例えば関数 use_only を取り上げてみよう：

```
def uses_only(word, available):
    for letter in word:
        if letter not in available:
            return False
    return True
```

この関数は文字列 word の中の全ての文字が文字列 available の中にあるものか調べている。書き直すと：

```
def uses_only(word, available):
    return set(word) <= set(available)
```

演算子<=は一つのセットが他のサブセット（それらが等しいことも含めて）であるかどうかを調べる。だから word のセットが available のサブセット（または等しい）と Ture を返す。

練習問題として、関数 avoids をセットを使って書き直せ。

19.6 カウンター

カウンターは要素の重複を許すことを除けばセットに似ている。カウンターはその要素が何回現れたを常に追跡している。数学の**多重集合（multiset）**の概念に馴染みがあればカウンターは多重集合を表現する自然な方法である。

Counter は標準モジュール collections の中に納められているのでそれをインポートする必要がある。カウンターの初期化は文字列、リスト等繰り返しの操作を伴なう型データで行う：

```
>>> from collections import Counter
>>> count = Counter('parrot')
>>> count
Counter({'r': 2, 't': 1, 'o': 1, 'p': 1, 'a': 1})
```

カウンターは辞書のように振舞う。各々のキーはそれが何回出現したという回数に写像される。辞書の同じく、キーはハッシュ可能でなければならない。

辞書と異なり、存在しないキーにアクセスしようとしても例外を発生しない。替わりに 0 を返す。

```
>>> count['d']
0
```

練習問題 10.6 の is_anagram はこのカウンターを使うとコンパクトに書ける：

```
def is_anagram(word1, word2):
    return Counter(word1) == Counter(word2)
```

もしも二つの単語がアナグラムであると、それらは同じ文字を同じ回数で含んでいるはずである。それらのカウンターは同等であるはずだ。

カウンターは集合の加法、減法、合同、交差などの集合の演算を実行するメソッドや関数を提供している。そして中でも有用な most_common メソッドがある。これはカウンターの要素を最大頻度順でソートした値ー頻度の対を要素としたリストを返すメソッドである。

```
>>> count = Counter('parrot')
>>> for val, freq in count.most_common(3):
...     print(val, freq)
r 2
p 1
a 1
```

19.7　デフォルト辞書

モジュール collections は辞書に似ているが存在しないキーにアクセスしようとすると例外ではなくその場で新たに新規の値を作ることができる defaultdict を提供している。

defaultdict を作るとき新しい値を如何に作るかを示す関数を合わせて指定する。この関数をときとしてファクトリー factory と呼ぶ。リスト、セットそして他の型のデータを作りだす組み込み関数がファクトリーとして使える。

```
>>> from collections import defaultdict
>>> d = defaultdict(list)
```

ここで引数が空のリストを作る関数 list() ではなくクラスオブジェクトの list であることに注意。提供した関数は存在しないキーへのアクセスがあるまで呼ばれることはない：

```
>>> t = d['new key']
>>> t
[]
```

t と呼ばれている新しいリストがこのキーと対になる値として辞書に追加される。それ故に t を修正すると、この修正はこの辞書に反映される：

```
>>> t.append('new value')
>>> d
defaultdict(<class 'list'>, {'new key': ['new value']})
```

リストを値として持つような辞書の作成に絡む問題ではこのデフォルト辞書を使うとより簡単にコードを書くことができる。練習問題 12.2（解答例：https://thinkpython.com/code/anagram_sets.py）では文字をソートした一つの文字列をそれらの文字から作られる単語のリストの写像する辞書を作る問題だった。例えば、 opst' はリスト ['opts', 'post', 'pots', 'spot', 'stop', 'tops'] に写像される。元のコードは

```
def all_anagrams(filename):
    d = {}
    for line in open(filename):
        word = line.strip().lower()
        t = signature(word)
        if t not in d:
            d[t] = [word]
        else:
            d[t].append(word)
    return d
```

（訳注：ここで関数 signature は単語 word を受け取りその中の文字をアルファベット順に並び替えた文字列を戻す。）

この例題は setdefualt を使うと簡単化できる（これは練習問題 11.2）：

```
def all_anagrams(filename):
    d = {}
    for line in open(filename):
        word = line.strip().lower()
        t = signature(word)
        d.setdefault(t, []).append(word)
    return d
```

この解等はそれが必要であるかどうかに拘わらず毎回新たなリストを作るという欠点がある。リストではこれは大きな問題ではないが、ファクトリー関数が複雑になると問題になるかもしれない。

defaultdict を使うとこの問題を避けることができる：

```
def all_anagrams(filename):
    d = defaultdict(list)
    for line in open(filename):
        word = line.strip().lower()
        t = signature(word)
        d[t].append(word)
    return d
```

練習問題 18.3 の解答例（https://thinkpython.com/code/PokerHandSoln.py）では has_straightflush 関数で setdefault を使っていた。この解は必要の如何に拘わらずループの繰り返しごとに Hand オブジェクトを生成するといった欠点を持っていた。練習問題として、その関数を defaultdict を使って書き直せ。

19.8　名前付きタプル

多くの単純なオブジェクトは基本的に関連性のあるデータの集合である。例えば第 15 章の Point オブジェクトは二つの数値 x と y を属性として持っている。このようなクラスを定義するときには、一般的には init メソッドと str メソッドで初めてきた：

```
class Point:

    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def __str__(self):
        return '(%g, %g)' % (self.x, self.y)
```

これは少し大袈裟すぎる。Python はもっとコンパクトにする方法を提供している：

```
from collections import namedtuple
Point = namedtuple('Point', ['x', 'y'])
```

この関数の第 1 引数は生成したいクラス名、第 2 引数はそのオブジェクトに持たせる属性（文字列で）のリストである。namedtuple からの戻り値はクラスオブジェクトである。

```
>>> Point
<class '__main__.Point'>
```

このクラスオブジェクトは自動的に__init__メソッドや__str__メソッドを提供しているので、改めてそれらを書く必要はない。
インスタンスを作るにはこのクラスオブジェクトを関数のように使えばよい：

```
>>> p = Point(1, 2)
>>> p
Point(x=1, y=2)
```

init メソッドは引数を指定した変数（オブジェクトの属性）に代入する。str メソッドはクラスオブジェクトとその属性を表示する。
このインスタンス p が持つ属性へのアクセスは通常のように名前でアクセスできる：

```
>>> p.x, p.y
(1, 2)
```

しかしまたこのインスタンス p はタプルのように振舞う。

```
>>> p[0], p[1]
(1, 2)


>>> x, y = p
>>> x, y
(1, 2)
```

名前付きタプル（named tuple）を使うと簡単なクラスの定義を素早くできる。しかし欠点は簡単はいつも簡単ではないことである。それは名前付きタペルにメソッドを追加するときである。その名前付きタペルを継承した新たなクラスを定義すること以外に方法がない：

```
class Pointier(Point):
    # add more methods here
```

このばあいは従来のクラス定義に戻ることも考えた方がよい。

19.9　キーワード付き引数を纏める

12.4 節では複数の引数を一つのタプルに纏めるような仮引数を持った関数を書いた：

```
def printall(*args):
    print(args)
```

位置固定引数であれば如何なる個数であっても引数としてこの関数を呼ぶことができる：

```
>>> printall(1, 2.0, '3')
(1, 2.0, '3')
```

しかしこの*演算子はキーワード付き引数を纏めることはできない：

```
>>> printall(1, 2.0, third='3')
TypeError: printall() got an unexpected keyword argument 'third'
```

キーワード付き引数を纏めるには**演算子を使う：

```
def printall(*args, **kwargs):
    print(args, kwargs)
```

このばあいキーワード付き引数は如何なる個数であってもよい。結果はキーワードを値へ写像する辞書である：

```
>>> printall(1, 2.0, third='3')
(1, 2.0) {'third': '3'}
```

もしもキーワードと値の辞書があると、これをばらす演算子**を付けて関数を呼ぶことができる：

```
>>> d = dict(x=1, y=2)
>>> Point(**d)
Point(x=1, y=2)
```

ばらす演算子をつけないで関数を呼ぶとその辞書は一つの位置固定引数と見なされるのでそれは x への代入であり、y への代入がないので関数は苦情を言う：

```
>>> d = dict(x=1, y=2)
>>> Point(d)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: __new__() missing 1 required positional argument: 'y'
```

もし沢山の仮引数を持つ関数で作業をするときには、頻繁に使う選択値を特定する辞書を作りこれを関数に渡してやるとよい。

19.10 語句

条件付き表式（conditional expression）： 条件に依存して二つ値の片方を選択するような表式

リスト内包（list comprehension）： 新規にリストを生成するための角括弧内で `for` ループを伴なう表式。

ジェネレータ表式（generator expressions）： ジェネレータオブジェクトを生成する括弧内で `for` ループを伴なう表式。

多重集合（multiset）： 集合の要素とそれが何回出現するかを示す整数との間の写像を表現する数学的な実体。

ファクトリー（factory）： 通常オブジェクトを生成するための引数として渡される関数。

19.11 練習問題

練習問題 19.1 以下の関数は二項係数を再帰的に計算する関数である：

```
def binomial_coeff(n, k):
    """Compute the binomial coefficient "n choose k".

    n: number of trials
    k: number of successes

    returns: int
    """
    if k == 0:
        return 1
    if n == 0:
        return 0

    res = binomial_coeff(n-1, k) + binomial_coeff(n-1, k-1)
```

```
    return res
```

この関数の本体を条件付き表式を使って書き直せ。

注意：この関数は同じ値を繰り返し計算で終わるので効率が悪い。11.6 節のメモ機能を使うと効率を上げることができる。しかし条件付き表式でこの機能を使うのは大変に難しい。

付 録A　デバッギング

プログラムにはさまざまのエラーが起こり得る。エラーをより早く追跡するためにそれらを区別しておくことが大変有益だ。

- 構文エラーはPythonがソースコードをバイトコードに変換している過程でPython自身によって吐き出されるものである。それらは通常はプログラム上に構文違反があることを示す。例えば、def文の最後にコロンを忘れると、少しばかり冗長なメッセージ、SyntaxError: invalid syntaxが出る。
- 実行時エラーはプログラムが実行されている過程で何かおかしなことが起こるとインタプリタによって吐き出される。大抵の実行時エラーメッセージにはどこでそのエラーが発生したか、どの関数を実行中だったかについての情報が含まれる。例えば、無限の再帰処理は最終的には実行時エラー、maximum recursion depth exceededになる。
- 意味的エラーはエラーメッセージが発せられないけれど、正しい結果が得られないという問題である。例えば、ある表現が、あなたが期待したような順序で評価されず、そのため間違った結果になってしまったというような状況である。

デバッギングの第一歩はあなたが格闘しているエラーはどのような種類のエラーか見極めることである。以下の節はエラーの種類に従って叙述されるが、いくつかの技法は一つ状況だけでなく他の状況でも適用できるものである。

A.1　構文エラー

構文エラーはそれが何を意味するか分かれば対処しやすいものである。不幸にして、エラーメッセージはときとして役に立たない。最も頻繁なメッセージはSyntaxError: invalid sysntaxやSyntaxerror: invalid tokenだが、これらは必要な情報をあまり含むものではない。一方、メッセージは問題がどこで発生したかを教えてくれる。実際、Pythonはどこで問題に気づいたかを示す。しか

し、これはエラーのある場所であるとは限らない。ときとして、エラーはエラーメッセージの場所より前のことがあり、よくあることはその前の行だったりする。

プログラムを少しずつ大きくしているのであれば、そのエラーは新たに追加した個所であると疑ってみることは有益だ。

また、プログラムが文献からのコピーである場合には、一字一句の比較が必要である。ときとして、その本が間違いを含んでいるかもしれないので、もし構文エラーらしいものを見つけたとすると、実はそれはその本の間違いかもしれない。

以下はよくある構文エラーを避けるいくつかの方法である：

1. 変数名として Python の予約語を使っていないことを確かめる。
2. 複合文（`for`、`while`、`if`、`def`）の先頭行の末尾にコロンがあるか確かめる。
3. 文字列を表すクオート記号は前後で合っているか確かめる。
4. 多重行文字列を三重クオート（シングルクオートかダブルクオート）で括るとき、末尾が正常に終わっているかを確かめる。閉じていない文字列はプログラムの最後で `invalid token` のエラーになる。または、次の文字列が現れるまでプログラムは文字列とみなされてしまう。第二のケースではエラーメッセージは全く現れない。
5. 括弧で展開する表式―(、{、[―を閉じないと、Python は次ぎの行も文の一部をみなす。一般に次ぎの行でエラーメッセージが出る。
6. 条件文の中で==の替わりに=にしてしまう古典的な間違い。
7. インデントが意図通りに使われているか調べる。Python はタブでも空白でも処理できるが、それらを混在して使うと問題が起こる可能性あり。問題を避ける最善の方法は自動インデント可能なエディタを使うことだ。

これで解決しないときは、次ぎの節に進んでほしい。

ずうっと修正をしているのに変化なし

インタプリタがエラーを指摘しているのに、エラーが見つからないのはあなたとインタプリタとが同一のコードを眺めていない可能性がある。プログラム開発環境をチェックして編集をしているプログラムが Python が実行しようとしているものであることを確かめること。もし不安ならば、プログラムの先頭に意図的に分かり易い構文エラーを起こす文を挿入してみることだ。再実行してインタプリタがこのエラーを指摘しないとすれば、あなたは更新されたコードを走らせているのでないことが分かる。このようなことが起こるいくつかの犯人を示す：

- ファイルを編集したが、実行する前に保存するのを忘れた。開発環境によってはこの保存を替わりにやってくれるものもあるが、そうでないものもある。
- ファイル名を変更したが、実行しているものは古い名前のものだった。
- 開発環境が正常に構築されていない。
- モジュールを作成していてインポートを使っているとしたら、Python の標準モジュール名と同じ名前は使わないようにする。
- インポートでモジュールを読み込むことをしているときには、もしもそれが変更されたモジュールであるならば、インタプリタの再起動か `reload` コマンドで再読み込みを行うこと。さもないと変更が反映されない。

これでも行き詰まってしまったら、`'Hello World!'` のような簡単なプログラムから再出発し、確認が取れているプログラムが正常に動くことを確かめるのも一つの方法だ。そして、徐々に元のプログラムの一部を新規のプログラムに追加して行くようにする。

A.2 実行時エラー

プログラムが構文的の正しいとすると、Pyhton はそれをコンパイルし、少なくとも実行を開始する。次ぎに起こるとしたらどんなエラーだろうか？

全く反応なし

この問題の状況はファイルが関数やクラス定義からなるときで、実行を開始するために必要な何ものも発動していない場合である。そのモジュールをクラスや関数を提供する目的のためにだけ必要な場合は意図的に行うことがある。そうでないのであれば実行を開始するために関数を発動させるか、インタラクティブモードで関数の一つを実行しなければならない。以下の「実行の流れ」の項も参照のこと。

プログラムが終わらない

プログラムが終了しても何もしなかったようにみえるときは、多分に「ハング」した状態にあるためだ。多くの場合それは無限ループや無限の再帰処理に陥ったことを意味している。

- もしある特定のループが怪しいと思われるときは、そのループの直前に「ループ開始」の print 文を入れ、ループの直後に「ループ終了」の print 文を入れて再実行してみる。もし最初のメッセージが出て、二番目が出ないとすれば、このループが無限ループである。

- 大抵の場合、無限再帰処理では実行は暫く続き、その後に'RuntimeError: Maximum recursion depth exceeded' のエラーが出る。これが状況なら「無限再帰」の項を参考にしてほしい。このエラーが出ないにしても、再帰メソッドや関数に問題ありと思うときも「無限再帰」の節の手法が役に立つ。

- これらのステップが有効でないなら、別の個所のループや再帰処理をテストしてみよう。

- これでもうまく行かないときは、あなたがプログラムの実行の流れを理解しきれていない可能性がある。「実行の流れ」の項をみてほしい。

無限ループ :無限ループがあると思えてその原因になっているループが特定できるときは、ループの終わりに print 文を挿入し、ループの条件に関わる変数の値とループの条件を表示してみる。
例えばこうだ：

```
while x > 0 and y < 0 :
    # do something to x
    # do something to y

    print('x: ', x)
    print('y: ', y)
    print("condition: ", (x > 0 and y < 0))
```

さて、プログラムを再実行すると、そのループを通過する度に上の三行の表示が出力される。ループが終了するときには、最後の条件は false になるからだ。ループが止まらないときには、変数 x、y の値が表示されるので、なぜこれらの変数が正常に更新されないのか検討ができるはずだ。
無限再帰 :大抵の場合、無限再帰処理では実行は暫く続き、その後に'RuntimeError: Maximum recursion depth exceeded' のエラーが出る。もし疑いがある再帰処理の個所が特定できたならば、先ず初めに、その処理に基底ケースがあるかどうか確かめる。換言すれば、再帰的な実行を止めてその関数またはメソッドが return に達する条件があるはずだということである。そうでなければ、アルゴリズムの再検討と基底ケースの特定が必要となる。基底ケースがあるにも拘わらずそこに

達していないように思えるときには、その関数またはメソッドの先頭で仮引数を表示するためにprint文を挿入してみる。再実行してみると、その関数またはメソッドが発動される度にこのprint文による表示が現れる。仮引数の値が基底ケースの条件の方向に向かっていないとすれば、問題の所在についてのヒントが得られるはずである。
実行の流れ:プログラムの実行の流れが把握できないときには、「関数fooに入る」（このfooに関数名が入る）というようなメッセージを表示するprint文を関数の先頭に挿入する。プログラムを実行してみるとそのprint文はその関数が発動される軌跡となる。

プログラムを実行すると例外が発生する

実行時に何か不正があると、Pythonは例外の名称、発生したプログラムの行番号、トレースバックを含んだメッセージを吐き出す。トレースバックは現在実行されている関数名、その関数を発動した関数、さらにその関数を発動した関数等の特定情報である。換言すれは、あなたが現在どこにいるかを示す関数の系列を遡って表示する。さらにそれらの関数呼び出しが起きた行番号の情報も含まれる。

最初にすべきことはプログラムのどこでそのエラーが発生したかを特定し、何が起きたかを認識できるかどうか確かめることである。以下はよく遭遇する実行時エラーとその意味だ：

NameError： 現在存在しない変数名を参照しようとした。局所変数はローカルでしか通用しない。それらが定義された関数の外でそれらを参照することはできない。

TypeError： いくつかの原因がある：

- 値の代入が不正である。例えば、文字列、リスト、タプルのインデックスとして整数以外のものを代入した。
- 変換記述文字列と変換に渡されるアイテムの間に不一致がある。アイテムの個数の不一致でも、不正な変換でもこのエラーは出る。
- 関数やメソッドに渡す引数の個数が間違っている。メソッドでは、よく定義を眺めること、第一番目の引数はselfである。次ぎにそのメソッドを発動させる側を調べる。そのメソッドを発動させるオブジェクトが正しい型か、引数は正確かどうかを調べる。

KeyError： 辞書に存在しないキーを使って辞書の要素にアクセスしようとした。

`AttributeError`： 存在しない属性やメソッドにアクセスしようとした。まずスペルをチェックしよう。存在する属性を表示する `dir` コマンドを使うこともできる。もしその `AttrubuteError` がオブジェクトは `NoneType` であると表示したら、そのオブジェクトは `None` である。よく遭遇する原因は関数の戻り値を書き忘れたときである。関数の戻り値を与えないで関数定義を終えると、それは `None` を返す。他の例としては、リストに関連するメソッド、例えば `sort` のような、を使っているときだ。この戻り値は `None` だ。

`IndexError`： リスト、文字列、タプルにアクセスするために使っているインデックスが（それらの長さ-1）を越えている。直ちにエラー発生の直前にインデックスの値と配列の大きさを表示する `print` 文を挿入する。配列は予期した長さになっているか？インデックスは正常な値を示しているか？

Pyhton デバッガー（`pdb`）は種々の例外の原因を突き止めるときに役に立つ。それはそのエラーは発生する直前までのプログラムの状態を吟味できるからだ。`pdb` については `pdb` at https://docs.python.org/3/library/pdb.html を参照のこと。

多くの print 文の追加で出力に埋没

`print` 文を多用したデバッギングの問題は出力に埋没することだ。二つの回避方法がある。出力を簡単化するか、プログラムを簡素化するかだ。

出力の簡単化は余分な `print` 文を削除またはコメントにし、理解し易いように表示形式を工夫することだ。プログラムの簡素化はいくつも方法がある。

第一に、プログラムで取り組んでいる問題の規模を縮小してみることだ。例えば、リストの検索の問題であれば、問題のリストを小さいリストで行ってみることだ。プログラムがユーザから入力を受け取る部分もあるのであれば、問題の引き起こすに足りる最も簡単な入力にしてみることだ。

第二に、プログラムを整理してみる。死文化されたコードは削除、理解し易いようにプログラムを再構成してみることだ。例えば、問題は多重な入れ子に関連するところかという疑いがあるなら、その部分をもっと簡単な構造に書き直してみることだ。また、問題が大きな関数に由来していると思われるときは、この関数を複数の小さい関数に分割し、それらを別々に検証してみる。

ときとして最小単位のテストをしようとする過程がバグを発見することに繋がることがある。プログラムがある状況では問題なく、他のケースでは動かないというのであると、このことがヒントになる。同様に、コードの部分的な書き直しは隠されているバグの発見にも役立つ。プログラムには影響しないと思って変更したことがそうでなかったら、これこそバグの在処を教えているようなものだ。

A.3 意味的エラー

ある意味この意味的エラーが最も厄介なエラーである。何故なら、インタプリタは何が悪いのかの情報を一切持っていなからである。あなたたけがそのプログラムが為すべきことを知っている。

第一に、プログラム全体と観察から得られたプログラムの振る舞いとの間の関連を付けることだ。プログラムが実際に行っていることについて仮説を持つ必要がある。困ることの一つはコンピュータによる実行があまりにも速いことである。プログラムの実行速度はヒトの速度なみに減速され、デバッカが使えたらと思うことがあるかもしれないが、適当の位置に `print` 文を挿入することの方がデバッガーを起動させ、ブレイクポイントを挿入・除去し、エラーを見つけるためにプログラムを「ステップ実行」させるより往々にして短時間で済む。

プログラムがまともではない

以下を自問してみよう：

- プログラムの中で、実行されるべきなのに実行されていないように思える個所はないか？その機能を実行している個所を見つけ、それが実行されるべきときに実行されていることを確かめる。
- 起こるはずがないことが起きていることはないか？その機能を実行している個所をみつけ、起こるはずがないときに実行されているか確認する。
- 期待していない効果を生成しているようなコードの個所はないか?問題のプログラムコードを自分は理解しているか確認しよう。特にそのコードがPythonの他のモジュールの発動に関与しているときはそれが必要だ。発動させている関数のドキュメンテーションをよく読みなさい。小さなプログラムでそれらを使って結果を確かめよう。

プログラムを作成するためには、そのプログラムが如何に動くかについてのメンタルモデルが必要だ。もしプログラムを闇雲に書いているとすれば、問題はプログラムにあるのではなく、あなたのメンタルモデルに問題があるのだ。

メンタルモデルを修正する最善の方法はプログラムをいくつかの部分（通常は関数やメソッド）に分割し、その分割した部分を個別にテストすることだ。一度メンタルモデルと現実との不一致が確認できれば抱えている問題は解決できるはずだ。勿論、それらの分割したものを再構築し、プログラムを開発するに過程に応じてテストしなければならない。問題に遭遇したら、プログラムに追加する未知の部分は極少量にすべきだ。

大きく不格好な表現があり、それが予想したようには動いていない

複雑な表現でもそれが読み易く書いてあれば問題ないが、デバッグは難しくなる。複雑な表現を一時的な変数への代入の組み合わせで分割するのは有益なことが多い。

例を上げよう：

```
self.hands[1].addCard(self.hands[self.findNeighbor(1)].popcard())
```

これは以下のように書ける：

```
neighbor = self.findNeighbor(1)
pickedCard = self.hands[neighbor].popCard()
self.hands[1].addCard(pickedCard)
```

一時的な変数名が追加の情報になるのでこのような明確に展開された表現は読みやすい。中間の変数の値や型も表示し、チェックできるのでデバッグも容易だ。

長い表現のもう一つの問題は評価の順序が期待したものにならないというものだ。例えば、$\frac{x}{2\pi}$ の表現を Python に置き換えるとするとき以下のように書いたとする：

```
x = x / 2 * math.pi
```

これは正しい表式ではない、何故なら乗算と除算は同じ優先度を持つので、左から右への順序で式は評価されるので、$x\pi/2$ が計算されることになる。括弧を追加して評価の順序を明確に示しておくとデバッグし易いものになる：

```
x = x / (2 * math.pi)
```

評価の順序が不安なときはこのように括弧を使うとよい。プログラムが意図したように動くという意味で正しく動くばかりでなく、他の人がみてもより読み易くなる。

期待した戻り値を返さない関数やメソッドができてしまった

複雑な表現を持つ表式を持たせて return 文を書くと、その直前でその戻り値を表示することができない。ここでも一時的な変数を使うとよい。以下はその例である：

```
return self.hands[i].removeMatches()
```

の替わりに

```
count =  self.hands[i].removeMatches()
retrun count
```

とする。これで retrun 文の直前で戻り値 count の値を表示できる。

行き詰まってしまった、助けが必要だ

まず第一に数分間でよいのでコンピュータの前から離れてみる。コンピュータは以下のような症状を引き起こす波動を放射している：

- フラストレーションや怒り。
- 「コンピュータは私を憎んでいる」という妄想や「私が帽子を後ろ向きに被ったとみにのみプログラムは動く」という迷信。
- 酔歩プログラミング（思いつくままにあれこれとプログラムを変更してみる）。

これらの症状の一つにでも当てはまるようだったら、椅子を立ち散歩に行くとよい。落ち着いてきたらプログラムのことを考えてみよう。何が起きているのか？何があのような振る舞いの原因なのか？ちゃんと動いたプログラムだったのかいつか？それに何を追加したのか？

ときとして時間が解決してくれるときもある。私はよくバグが見つけられるのはコンピュータから離れ、あれこれを考えているときだ。それは列車の中だったり、シャワーを浴びているときだったり、眠りに落ちる寸前のベッドの中だったりする。

もうダメだ、助けが必要だ

これは起こる。最高のプログラマでもときとして行き詰まる。一つのプログラムに長い間掛かり切りになっているが故にエラーが見えないことがある。新たな目が必要なことがある。

他の人を呼び込む前に充分な用意ができていることを確認しよう。プログラムはできうる限り簡単にすべきだ。また問題を引き起こすに必要な最少の入力データを用意すべきだ。さらに、適当な場所に print 文（その出力も充分に意味が分かるもの）を追加すべきである。最後に、起きている問題を充分に理解できていて、それを簡潔に叙述できる必要がある。他の人に助けを求める前に、その人が必要な情報が揃っているか確かめよう。

- もしエラーメッセージがあるなら、それは何で、プログラムの何処を示しているのか？
- このエラーメッセージが発生するようになる直前にあなたは何をしたのか？あなたが最後に書き加えた部分は何処か？失敗した最近の事列は何か？
- これまで試したことは何か？それで解ったことは何か？

バグは見つかったときは、もっと早く発見するためには何をすればよかったのかを僅かな時間でよいから考えてみよう。次回には同じような状況になったときは今より早くバグを見つけるようになるだろう。そのプログラムが動くようになることだけが目標ではないことを思いだそう。目標は如何にしたら動くプログラムが作れるかを学ぶことだ。

付 録B　アルゴリズムの解析

この章のアプローチはAllan B. Downey著 “Think Complexity”(O'Reilly Media: 2011)から編集した抄録である。今読んでいるこの本を終えてしまったら、そちらの本に進んでみたいと思うかもね。

アルゴリズムの解析（analysis of algorithms）はアルゴリズムの効率、特に実行時間とメモリー消費を調べるコンピュータ科学の一分野である http://en.wikipedia.org/wiki/Analysis_of_algorithms を参照せよ)。

このアルゴリズムの解析の実践的な目標はプログラムを設計するときの目安として異なったアルゴリズムの実行効率を予測することである。

2008年の米国大統領選挙キャンペーンの最中、オバマ候補はグーグル社を訪問したとき前触れなしの質問を受けた。主経営責任者のエリックシュミットは冗談交じりに、「100万個の32ビット整数をソートするのに最も効率のよいのは何でしょうね」と聞いた。「バブルソートではダメだよね」と即座に答えたことからどうやらオバマ候補は耳打ちされていたようだ。(http://www.youtube.com/watch?v=k4RRi_ntQc8 を参照せよ)

確かにバブルソートは原理的には単純だが、大きなデータセットでは効率が悪い。シュミットが求めていた答えは多分「基数ソート」だろう（http://en.wikipedia.org/wiki/Radix_sort を参照のこと)。[1]

アルゴリズムの解析の目標はアルゴリズム間の意味ある比較である。しかし、いくつかの問題がある：

- アルゴリズムの相対的な実行効率はハードウエアの特性によるかもしれない。従ってあるアルゴリズムはマシンAではより速いが、マシンBでは他が速いということがある。このような状況の解決策は**マシンモデル（machne model）**を特定させることだ。そして、この仮想的なマシンモデルの下で

[1] しかし、インタビューで同じような質問を受けたとき、もっとましな答えは以下のようだと私は思う。「100万個の整数をソートする最速な方法は私が使っている言語が提供しているソート関数をまず使うことです。それは大抵のアプリケーションに対して充分な性能を持っています。もしそのアプリケーションに対してそれが遅いことが判明したら、プロファイラを使いどの部分で時間を食っているのか調べます。そして、より速いソートアルゴリズムがこれに対して充分な効果があると判明したら、基数ソートのうまい実装を探すことにします。」

アルゴリズムを実行したときに必要なステップ数、または操作を解析することだ。

- 相対的な実行効率はテストに使うデータセットの細部に依存するかもしれない。例えば、あるソートのアルゴリズムは部分的にソートされているデータセットに対してより高速であるが、今の場合は更に低速だというようなことだ。このような事態を回避する通常の方法は**最悪の状況（worst case）**で解析することだ。平均的な実行効率を解析することも有用だが、平均を取るべきデータセットの集まりが何であるかも自明ではない。
- 相対的な実行効率は問題の大きさにも依存する。小さいリストに対して高速のソートのアルゴリズムも大きなリストに対しては低速だということもあり得る。この状況に対する解決法は問題の大きさの関数として実行時間（または操作の回数）を表現し、問題のサイズが大きくなって行くにつれての漸近的な振る舞いを比較することである。

これらの比較の有用な点はそれ自体がアルゴリズムの分類に繋がっていることである。例えば、入力のサイズ n に対してアルゴリズム A は n に比例し、アルゴリズム B は n^2 に比例するとしよう。大きな n に対してアルゴリズム A はより高速だと期待できる。

この種の解析の結論は注意も必要だ。これについては後に触れる。

B.1　増加の次数

二つのアルゴリズムを解析し、実行時間を入力データの個数 n の関数と表現したとしよう。アルゴリズム A はある問題を個数 n で解くのに $100n + 1$ ステップ掛かったとしよう。一方アルゴリズム B は n^2 ステップであったとしよう。

これから異なった問題サイズ n に対して二つ実行時間の表にしてみる：

入力データの個数	アルゴリズム A の実行時間	アルゴリズム B の実行時間
10	1 001	111
100	10 001	10 101
1 000	100 001	1 001 001
10 000	1 000 001	$> 10^{10}$

この表から $n = 10$ ではアルゴリズム A は酷く効率が悪いようにみえる。アルゴリズム B と比較すると 10 倍も時間が掛かる。しかし、$n = 100$ では殆ど同じ、更に大きな n に対しては A はかなりよい。

大きなnに対しての振る舞いの基本的な理由は のn^2項を含む任意の関数はn項を支配項とする任意の関数と比較してより速く増大するからである。**支配項（leading term）**とは最高の次数を持つ項である。

アルゴリズム A の支配項の係数は 100 と大きく、これが n が小さいときに B が有利な原因である。しかしこの係数にも拘わらず $an^2 > bn$ を満たす n は必ずある。

この議論は支配的でない項に対しても当てはめることができる。アルゴリズム A の実行時間が $n + 1000000$ と書けたとしても大きな n に対してはアルゴリズム B より優秀だと言える。

一般に大きな問題に対しては次数の小さい支配項を持つアルゴリズムがより優秀だといえるが、小さな問題になってくると他のアルゴリズムが有利になる**切り換え点（crossover point）**がある。この切り換え点はアルゴリズムの細部、入力データ、ハードウエアに依存する。従って、アルゴリズム解析では通常は無視する。しかし、この事実は忘れてよいわけではない。

もし二つのアルゴリズムの支配項の次数が同じであるときは、どちらが優秀かをいうのは難しい。これも細部による。従ってアルゴリズム解析ではその支配項が持つ係数が異なっていてもこの二つは同等だと考える。

「**増加の次数（order of growth）**:」はそれらの漸近的な振る舞いが同等とみなされる関数の集合に与えられる。例えば、$2n$、$100n$、$n+1$ は同じ増加の次数に属する。つまり、**ビック-O 記法（Big-Oh notation）**(訳注：O はギリシャ文字）では $O(n)$ と書く。そしてこの関数の集合は n に比例して増大するので、この次数を**線形（linear）**と呼ぶ。

支配項が n^2 である全ての関数は O (n^2) に属し、それらは**方形（quadratic）**と呼ばれる。支配項が を持つ関数に付けられたものとしてはおかしな名前だ。

以下の表は通常アルゴリズム解析で登場する増加の次数を劣性が増加する順に並べてものである。

増加の次数	名称
$O(1)$	定数
$O(\log_b n)$	対数的（任意の底 b で）
$O(n)$	線形
$O(n \log_b n)$	線形対数
$O(n^2)$	方形（二次）
$O(n^3)$	立方形（三次）
$O(c^n)$	指数的（任意の底 c で）

対数項の場合その底の値は問題ではない。底が異なることは定数をその項に乗

ずることに同値なので増加の次数を変えるものではないからだ。同様に底が何であろうと指数的な次数の関数は同じ増加の次数に属する。指数関数は急激に増加するので、指数的なアルゴリズムは小さい問題でしか使えない。

練習問題 B.1 ビック-O 記法 http://en.wikipedia.org/wiki/Big_O_notation を読み、以下の問いに答えよ。

1. $n^3 + n^2$ の増加の次数は何か？それではこれは $1000000n^3 + n^2$？ではこれは $n^3 + 1000000n^2$？
2. $(n^2 + n) \cdot (n + 1)$ の増加の次数は何か？乗算を始める前に、支配項のみが必要なことを思い出してみよう。
3. もしも f が特別でない関数 g に対して $O(g)$, のオーダーならば、t $af + b$ について何が言えるか？
4. もしも f_1 と f_2 とが $O(g)$ のオーダーであるならば、$f_1 + f_2$ について何が言えるか？
5. もしも f_1 が $O(g)$ のオーダーで f_2 が $O(h)$ のオーダーであるならば、$f_1 + f_2$ について何が言えるか？
6. もしも f_1 が $O(g)$ のオーダーで f_2 が $O(h)$ のオーダーであるならば、$f_1 \cdot f_2$ について何が言えるか？

実行速度を気にするプログラマはときとしてこの種の解析結果に納得するのが難しい。係数の大きさや支配項でない項が実際の差に影響することがあるからだ。ハードウエアの詳細、プログラム言語、入力データ特性が大きな差を作り出すこともある。そして、小さなサイズの問題では、漸近的な振る舞いは問題とならない。

しかしこれらの注意を心に留めて置くとしても、アルゴリズム解析は有益なツールになる。少なくとも、大きなサイズの問題に対しては、「より優秀」と判定されたアルゴリズムは実際にもより優秀であり、ときとしてかなり優秀である。同じ増加の次数を持つ二つのアルゴリズムの差はサイズが大きくなっても定数に留まるが、異なった次数を持つアルゴリズムの差はサイズと共に限界なしで大きくなる。

B.2　Python の基本操作の解析

大抵の代数演算は一定時間で実行される。乗算は加算、減算より時間を要する。除算はさらに時間が掛かる。しかし、これらの演算は被演算子の大きさには依らない。極端に大きな整数は例外で、その実行時間は桁数と共に増加する。

配列や辞書の要素の読み書きに出てくるインデックスを使って要素を指定する操作はデータ構造の大きさに依らず一定時間で実行される。

配列や辞書を横断的に眺める for ループは本体での操作が一定時間である限り、通常は線形だ。例えば、リストの要素の総和を求める操作は線形である：

```
total = 0
for a in t:
    total += a
```

組み込み関数 sum も同じことをやっているので線形であるが、もっと効果的な方法を実装している。アルゴリズム解析の言葉では、支配項の係数がより小さいということになる。

同じループを文字列のリストの「数え上げ」に適用するとその次数は方形（二次）となる。文字が基本操作の単位であり文字列の連結は線形となるが、その連結の操作のループだからである。（訳注：文字列の連結は既存の文字列の最後に新たな文字列を追加することになり、この既存の文字列の最後を探す操作が線形である。）文字列メソッド join は文字列が基本単位であり文字列の全長に対して線形なので、その次数は線形だ。

経験からループの本体の次数が $O(n^a)$ であると、全ループは $O(n^{a+1})$ になる。例外はこのループがある決まった有限回数で終了することが示されたときである。ループは n にも拘わらず k 回で終了するのであれば、その次数は如何に k が大きくても $O(n^a)$ である。定数 k の乗算は次数を変えないし、除算も然りである。従って本体は $O(n^a)$ であり、ループの回数が n/k 回であっても全体の次数は $O(n^{a+1})$ となる。

多くの文字列やタプルに対する操作は線形である。例外はインデッスを使ったアクセスと関数 len で、これらは定数である。組み込み関数 min、max は線形である。スライスを使った操作は出力の大きさに比例するが、入力のサイズとは独立である。

全ての文字列に対する操作は線形であるが、文字の長さがある定数によって有限であるとすると実行時間の次数は定数となる。その例としては単独文字の文字列に対する操作がある。

リストに対する操作は大抵線形である。これには例外がある：

- リストの末尾に要素を追加するのは平均にすると一定時間だ。もし領域が不足してより大きな領域の全体をコピーするという事態であるとそれは線形となる。しかし、n 回の追加で必要となる実行時間は $O(n)$ であるので、一操作当たりの「償却時間は $O(1)$ になる。

- リストの末尾のある要素の削除は一定時間である。

- リストのソートは $O(nlogn)$ である。

辞書の操作やメソッドの多くは一定時間である。これも例外がある：

- copy の操作は辞書の要素の数に比例する。しかし、要素の大きさにはよらない（これは参照のコピーであって、要素のコピーでないならばのことである）。
- update の操作は引数として受け取る辞書の大きさに比例するが、更新される辞書にはよらない。
- メソッド keys、values、items は辞書全体を新規リストとして返すので線形である。メソッド iterkeys、itervalues、iteritems はイテレータを返すので一定時間である。しかし、for ループでそのイテレータを使って横断的な処理をすると、そのループ処理は線形である。関数 iter を使うことは初期処理を節約できるが、処理する要素の数が有限でなければ増加の次数は変わらない。

辞書の実行速度はコンピュータ科学に於けるちょっとした奇跡である。この話題は B.4 節で取りあげる。

練習問題 B.2 ソートのアルゴリズムについて http://en.wikipedia.org/wiki/Sorting_algorithm を読み、以下の問いに答えよ。

1. 「比較ソート」とは何か？比較ソートの最悪状況下での最高実行時間の増加の次数は何か？　また、任意のソートアルゴリズムの増加の次数は何か？
2. バブルソートの増加の次数は何か？また、何故にオバマ候補はこれでは「ダメだよね」と思ったか？
3. 基数ソートの増加の次数は何か？またこのソートが使える前提条件は何か？
4. 安定ソートとは何か？またその機能が実際問題として問題になるのは何故か？
5. 最も効率の悪いソートは何か（名前が付いている）？
6. C 言語のライブラリで使われているソートアルゴリズムは何か？Python ではどうか？それらは安定ソートか？web で調べることも手かもしれない。
7. 比較ソート以外のソートの多くは線形である。Python では何故に $O(nlogn)$ の比較ソートを使っているのか？

B.3 探索アルゴリズムの解析

探索（search）は配列と目的のアイテムとを受け取りそのアイテムがその配列の中に存在するかどうか、よくあるのは目的アイテムと一致する配列のインデックスを返すアルゴリズムである。

最も単純な探索アルゴリズムは「線形探索」である。これは配列のアイテムを一つ一つ横断的に調べ、目的アイテムと一致するものを見つけたら終了する。最悪のばあいは配列の全ての要素を調べることになるので線形である。

配列処理に使う `in` 演算子は線形である。`find`、`count` のような文字列で使うメソッドも同様である。

もし配列が順序付けられているならば、二分探索が使える。この次数は$O(logn)$である。辞書（本当の）で単語を探すときに使っているアルゴリズムである。最初から一つ一つ調べる替わりに真ん中の単語あたりから当たり、探している単語はそれより前か後かを調べる。それがそれ以前になるならば、前後の前半を調べる。そうでなければ後半を調べる。どちらにしても残りの半分は無視する。配列は1000000要素を持っていたとしても約20回で目的の単語に行き着くか、該当する単語はないという結論を得る。これは線形探索に比較して50,000倍程度高速である。

二分探索は線形探索に比較するとずうっと速いが、配列が順序付けられている必要があり、そのための余分な手間が必要である。**ハッショ表（hashtable）**と呼ばれているもう一つのデータ構造がある。これはさらに高速である。これは探索を一定時間で行い、しかもアイテムがソートされている必要もない。Pythonの辞書はそのハッシュ表を使って実装されている。従って、`in` 演算子を含め辞書に対する大半の操作は一定時間である。

B.4 ハッシュ表

如何にハッシュ表が有用で、効率がよいかを説明するために、マップの簡単な実装からハッシュ表に到達するまで徐々にそれを改良して行くことにする。

この実装を論証するためにPythonを用いる。しかし、実際にはそのようなコードをPythonで書くことはないだろう。何故ならPythonでは辞書型を使えばよいからだ。この章では辞書型が存在しない世界を想像し、キーから値を写像するデータ構造を実装したいのだと思ってほしい。実装したい操作は以下のようだ：

`add(k, v)`： キー`k`から値`v`の写像の追加。Pyhtonの辞書型`d`では`d[k] = v`。

`get(k)`： `k`がキーであるような値を探索し、その値を返す。Pyhtonの辞書型`d`では`d[k]`または`d.get(k)`。

さて以下では各キーは唯一であると仮定しよう。このようなインタフェースの最も簡単な実装はキーと値のペアをタプルとするリストにすることだ：

```
class LinearMap:

    def __init__(self):
        self.items = []

    def add(self, k, v):
        self.items.append((k, v))

    def get(self, k):
        for key, val in self.items:
            if key == k:
                return val
        raise KeyError
```

メソッド add はキー・値のペアの一つをアイテムのリストに追加する。これは（マップの大きさに関係なく）一定時間でできる。メソッド get は一つの for ループ使ってリストを探索しその値を返す、なければ KeyError の例外を発生させる。この処理は（マップの大きさに比例して時間がかかるので）線形である。

別な方法は予めリストをキーでソートしておくことが考えられる。これであるとメソッド get には二分探索が使えるので、この処理時間は $O(logn)$ のオーダーだ。しかし、新規のアイテムをリストに挿入するためにはリストの横断的処理が必要なのでこの処理は線形になる。従ってこれはあまり期待できない。

LinearMap を改良する一つの方法はキー・値ペアのリストを小さいリストに分割することだ。以下はその実装で BetterMap と名付けた。このリストは 100 個の LinearMap からなる。直ぐにわかるがメソッド get は最終的には LinearMap の一つを横断的探索しなければならないので線形のオーダーである。しかし、BetterMap はハッシュ表へ向かう第一歩である：

```
class BetterMap:

    def __init__(self, n=100):
        self.maps = []
        for i in range(n):
            self.maps.append(LinearMap())
```

```
    def find_map(self, k):
        index = hash(k) % len(self.maps)
        return self.maps[index]

    def add(self, k, v):
        m = self.find_map(k)
        m.add(k, v)

    def get(self, k):
        m = self.find_map(k)
        return m.get(k)
```

メソッド__init__はLinearMapを複数個入れるリストを作成する。
メソッドfind_mapは組み込み関数hashを使う。この関数はPythonの殆ど全ての型を引数として整数を戻す。実装で唯一の制限はハッシュ可能なデータ型、つまり変更不可のデータ型であることだ。リストや辞書はハッシュには使えない。ハッショ化するオブジェクトが同じであるとみなされると同じ整数が返ってくる、しかし逆は必ずしも成り立たない。異なったオブジェクトが同じハッシュ値を持つこともある（だから同一のLinearMapに保存される）。find_mapはまたモジュラ演算子を使ってハッシュ値を0からlen(self.maps)までの整数値に包み込んでいる。従ってリストの大きさの制限を満たす整数値になっている（勿論このことによって異なったハッシュ値が同一のインデックス値をもつことにもなる）。しかし元のハッシュ値に大きな片寄りがなければ（これはハッシュ関数の設計によるが）、個々のLinearMapのアイテムの個数は$n/100$に近いと期待できる。

LinearMapの実行時間はアイテムの数に比例するので、BetterMapの実行速度はLinearMapの100倍になる。しかし、LinearMapのオーダーは線形のままなのでBetterMapも線形である。支配項の係数は小さくなったのはよいことだが、ハッシュ表のようには行かない。

ここでハッシュ表を高速にする決定的な考えを導入する。LinearMapの長さを有限にできれば、メソッドgetは一定時間でできる。一つのLinearMap当たりのアイテムの数の増加を監視し、制限値を超えたら複数のLinearMapを追加してハッシュ表を再構成する。以下のその実装である：

```
class HashMap:

    def __init__(self):
        self.maps = BetterMap(2)
        self.num = 0
```

```
    def get(self, k):
        return self.maps.get(k)

    def add(self, k, v):
        if self.num == len(self.maps.maps):
            self.resize()

        self.maps.add(k, v)
        self.num += 1

    def resize(self):
        new_maps = BetterMap(self.num * 2)

        for m in self.maps.maps:
            for k, v in m.items:
                new_maps.add(k, v)

        self.maps = new_maps
```

各HashMapは一つのBetterMapを含んでいる。メソッド__init__ではそれをたった二つのLinearMapで始める。変数numはリスト全体に保存するアイテムの個数を監視するのに使う。メソッドgetはBetterMapのそれをそのまま実行する。核心はメソッドaddで、まずアイテムの個数とBetterMapのサイズ（つまり、LinearMapの個数）を比較する。それらが等しいとき、つまりLinearMap一つ当たりのアイテムの数が平均で1であるときには、resizeが呼び出される。メソッドresizeはそれまでの二倍の個数のLinearMapを持つ新規のBetterMapを生成し、それまでのBetterMapに保存されているアイテムを「再ハッシュ」し、この新規マップに保存する（約半数の個数のLinearMapで充足する）。この「再ハッシュ」はfind_mapsで使っているモジュラ演算子の分母であるLinearMapの個数が変わるので必要である。これで同じLinearMapに割り当たされてしまったいくつかオブジェクトは分散することになる（このこと自体は期待したことだよね）。

「再ハッシュ」はアイテムの個数に比例するので線形であり、resizeも線形である。これだけではaddメソッドがアイテムの個数に関わらない定数時間でできるという約束からすると期待外れのようにみえるかもしれない。しかし、resizeは毎回する必要はない、従って操作addは、いつもは定数時間で、時々線形になる。n回のaddを実行するのに必要な操作はnに比例することになるから操作add

一つあたりの時間はアイテムの数に関わらない定数時間となる。

このことを明らかにするために空のハッシュ表から初めて、次々とアイテムを追加して行くと考えよう。2 つの LinearMap があるので最初の 2 つの add は速い。例えば各々に必要とする時間を 1 単位としよう。次ぎの add は resize が必要だ。そこで最初の 2 つは「再ハッシュ」される（それには全てで 2 単位の時間が必要だ)。そして三番目のアイテムを add するが、これは 1 単位時間が必要だ。次ぎは 1 単位、全てで 6 単位が 4 つのアイテムに対して必要になる。

さて次は resize(四のアイテムの) と一個の新規アイテムの add で 5 単位必要になる。しかし引き続く 3 アイテムは各 1 単位で済むから、これで 8 アイテムに対して必要な時間は 8+5+3=14 単位になる。

さらに次ぎの add は resize（8 アイテムの）と一個の新規アイテムの add で 9 単位、しかしそれ以降の 7 アイテムは各々1 単位だから 16 個のアイテムに対して 30 単位の時間を使う。32 個のアイテムの add の処理で 62 単位が必要となるといった具合で、もうあなたもからくりが分かったと思う。n が 2 のべき数であるとき、n 個の add に要する時間は $2n-2$ 単位になる。従って一つ add 操作当たり 2 単位余りの時間が必要となる。n が 2 のべき数のときが最高の効率で、それ以外は若干落ちるがそれは重要ではない。重要なことが add の処理が $O(1)$ であることだ。

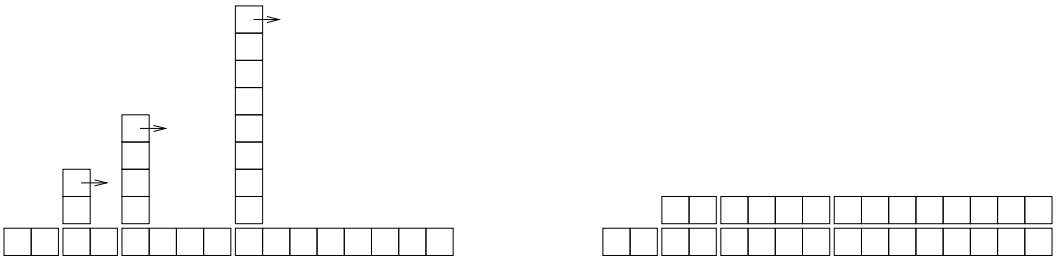


図 B.1: ハッシュ表における add 操作のコスト

図 B.1 は視覚的な説明の図である。各ブロックは作業の単位時間を表す。列は左から右へと各 add で必要とする作業時間を表現している。「再ハッシュ」で要する作業時間は縦のタワーの系列として表現されている。このタワーを倒して基本的な add に要する時間ブロックに重ねてみると n 個のアイテムを追加するときに必要な作業時間は $2n-2$ 単位であることが視覚的にも分かる。

このアルゴリズムの重要な特徴はハッシュ表のサイズを大きくして行くときに、幾何級数的に増やしていることだ。もしサイズを算術的に増加させる、つまり毎回一定の固定した数だけ増やすと操作 add 一つ当たりの平均時間は線形となる。

ハッシュマップの実装は https://thinkpython.com/code/Map.py からダウンロードできる。しかし、Python ではそれを使う必要はない。単に辞書を使えば済むからだ。

B.5　語句

アルゴリズムの解析（analysis of algorithms）： 複数のアルゴリズムを実行時間と必要な記憶域で比較する方法。

マシンモデル（machine model）： アルゴリズムを記述するために用いられるコンピュータの簡単化された表現。

最悪の状況（worst case）： そのアルゴリズムにとって実行が最遅（または記憶域を最大）になるような入力データ。.

支配項（leading term）： 多項式で最大の指数を持つ項。

切り換え点（crossover point）： 二つのアルゴリズムで実行速度や必要とする記憶域が拮抗する問題のサイズ。

増加の次数（order of growth）： アルゴリズムの解析のために増加が同等とみなされる関数の集合。例えば増加が線形である全ての関数は細部を無視して同じ増加の次数を持つという。

ビック-O 記法（Big-Oh notation）： 増加の次数を表現する記法。例えば、$O(n)$ は線形に増加する関数の照合を表現する。

線形（linear）： 問題のサイズが大きなところで見ると、実行時間が問題のサイズに比例するようなアルゴリズム。

方形（quadratic）： 問題のサイズの指標 n に対して、実行時間が n^2 に比例するようなアルゴリズム。

探索（search）： 集合（例えばリストや辞書）の中の特定の要素の場所を特定（または存在しないこと示す）する問題。

ハッショ表（hashtable）： キーと値のペアの集合であって要素の探索が集合の大きさのよらず一定 j 時間でなされるような特性を持つデータ構造。

付 録C　日本語の処理（Python2版）

ここではPythonで日本語の文字を扱うときのポイントを述べる。

日本語の文字をコンピュータで扱うときの問題点は第一にどのような文字が扱えるとよいかということがある。ひらがな、カタカナは文字数が少ないからよいが漢字は文字数が多いから漢字を完璧に扱えるようにするには大変である。「超漢字」システム（http://ja.wikipedia.org/wiki/超漢字）は十七万漢字を取り扱うことができるが、これは例外的である。よく使われる漢字に対して**JIS 第一水準漢字**、**JIS 第二水準漢字**として規格化されている約一万個の漢字がある（https://ja.wikipedia.org/wiki/JIS漢字コード）。後発ではあるがこれとは別の体系として**ユニコード（Unicode）**がある。東アジア圏の言語を纏めたCKJ統合漢字の中で扱える漢字が約二万個ある（http://ja.wikipedia.org/wiki/Unicodeを参照せよ）。ユニコードで扱える漢字はJIS第一水準漢字、JIS第二水準漢字を完全に含んでいる。

一方漢字をどのように符号化するかも複雑である。歴史的な経緯からJISの水準漢字に対しては、シフトJIS、JIS、EUC-JPの三つ**エンコード方式（符号化方式）**が使われている。ユニコードに対してはUTF-8のエンコード方式がよく使われる。

Pythonではこの四つのエンコード方式をサポートしている。これらのエンコード方式は全て8ビットの数値を一文字として扱う。漢字一文字も複数バイトに分解して扱うので8ビット文字列と呼ぶ。

C.1　ユニコード文字列の生成

これに対してユニコード文字列は英数字、ひらがな、カタカナ、漢字の一文字を平等に一文字として扱うことができる。Pythonはこのユニコード文字列で文字列処理を行う機能を持っている。

ユニコード文字列を生成する方法の一つが8ビット文字列の前に「u」をつける方方式である：

```
a = 'あいう'
au = u'あいう'
print len(a), len(au)
```

この print 文で文字数を表示すると 9 3 とでる。つまり、ユニコード文字列ではひらがな文字（カタカナ、漢字も）も文字数が正確に計測される。このようにユニコード文字列で日本語を扱うと英数字の文字列に使えた関数やメソッドがそのまま日本語にも適用できる。

さて、インタラクティブ・モードであれスクリプト・モードであれそこに現れた日本語は何らかエンコード方式を使ってコード化されたものである。このエンコード方式を確かめる簡単な方法がある：

```
a = 'あいう'
au = unicode(a, 'utf-8')
```

これは a に代入された文字列が utf-8 でエンコードされたものみなし、ユニコードに変換を試みる関数 unicode を使った例である。第二の引数に間違ったエンコード方式を指定するとエラーとなる。例を示す：

```
>>> a = 'あ'
>>> unicode(a,'utf-8')

Traceback (most recent call last):
  File '<pyshell#1>', line 1, in <module>
    unicode(a,'utf-8')
UnicodeDecodeError: 'utf8' codec can't decode byte 0x82 \
       in position 0: unexpected code byte
```

このエラーが吐き出されないエンコード方式がこの日本語をエンコードしているものだとわかる。

C.2　エンコード方式の指定

スクリプト・モードではファイルの先頭もしくは二行目にそのファイルの日本語のエンコード方式を明示的に指定できる：

```
#coding: sjis
```

このようにエンコード名（今の場合はシフト JIS）を指定するとこのファイル内の日本語はこの指定したエンコード方式でエンコードされることになる。このエンコード方式を明示することで簡単にユニコード文字列が生成できる。つまり、前に紹介したように文字列の前に「u」を付ける：

```
u’あいう’
```

である（インタラクティブ・モードではこの指定はできないので「u」を付ける方法では期待したものにならない場合があるので注意）。

Python で使えるエンコード名は以下の表のようになる：

エンコード名の名称	Python で使うエンコード名
シフト JIS	sjis, shift-jis, shift_jis
JIS	iso-2022-jp
EUC-JP	euc-jp
UTF-8	utf-8

ユニコード文字列の生成の第二の方法は unicode 関数を使うものだ：

```
a = ’あいう’
au = unicode(a, ’sjis’)
```

関数の第二の引数はエンコード名で明示したエンコード名を入れる。

第三の方法は文字列メソッドを使う方法である。

```
a = ’あいう’
au = a.decode(’sjis’)
```

ここでもメソッドの引数には指定したエンコード名が入る。

さて、日本語のエンコード方式にはいくつかあることが分かったが、どのエンコード指定を使えばよいのだろうか？スクリプト・モードでは日本語をユニコード文字列として扱うことが理想であるので、ユニコードを一貫的に扱えるエンコード方式である UTF-8 が妥当ということになる。

C.3 ユニコード文字列のエンコード変換

ユニコード文字列を明示的な仕方で 8 ビット文字列に変換したいときには、ユニコード文字列のメソッド encode を使う：

```
au = u'あいう'
a = au.encode('utf-8')
```

これでユニコード文字列 au は utf-8 でエンコード変換され変数 a に代入される。

C.4　辞書やタプルで日本語

日本語を辞書やタプルで使ってみる。辞書、リスト、タプルの要素としてユニコードを使う。以下はその例である：

```
#coding: utf-8
d = { u'甲':(u'コウ', u'きのえ'),
      u'乙':(u'オツ', u'きのと'),
      u'丙':(u'ヘイ', u'ひのえ'),
      u'丁':(u'テイ', u'ひのと'),
      u'戊':(u'ボ', u'つちのえ')}
for key, value in d.items():
    print key, value
```

結果の表示は以下のようになる。

```
乙 (u'\u30aa\u30c4', u'\u304d\u306e\u3068')
丙 (u'\u30d8\u30a4', u'\u3072\u306e\u3048')
丁 (u'\u30c6\u30a4', u'\u3072\u306e\u3068')
甲 (u'\u30b3\u30a6', u'\u304d\u306e\u3048')
戊 (u'\u30dc', u'\u3064\u3061\u306e\u3048')
```

ユニコード文字列が辞書のキーに使えることが分かる。この例では辞書の値はユニコード文字列のタプルになっている。この値のタプル表示はユニコードのコードポインタに対する数字である（この状況はユニコード文字列を辞書などのデータ構造の要素とする場合に起こる）。これを対応する文字列として表示するためにはタプルの要素を直接表示するとよい。つまり

```
for key, (v1,v2) in d.items():
    print key, v1, v2
```

結果は以下のようになる：

乙 オツ きのと
丙 ヘイ ひのえ
丁 テイ ひのと
甲 コウ きのえ
戊 ボ つちのえ

C.5 日本語を含むファイル

簡単な日本語を含むテキストファイルを作成する。ファイル名は abc.txt で文字コードは sjis とする。このファイルを Python で作成する。以下はそのプログラムである：

```
#coding: utf-8
fout = open('abc.txt','w')
uword = u'あいうえお'
fout.write(uword.encode('sjis'))
fout.close()
```

ユニコード文字列を sjis でエンコード変換してファイルに書き込んだ。

今度はこのファイルを読み込んでみよう。プログラムは以下である：

```
#coding: utf-8
fin = open('abc.txt','r')
for line in fin:
    word = line.strip()
    print word, len(word)
    uword = unicode(word, 'sjis')
    print uword, len(uword)
fin.close()
```

word は 8 ビット文字列（sjis）で、uword はユニコード文字列である。結果の表示は以下のようになる：

```
あいうえお 10
あいうえお 5
```

文字列の長さを len で調べてとユニコード文字列は日本語の長さを適切に処理していることが分かる。

データベースに使ってみる第 14 章で扱ったデータベースにユニコード文字列を使ってみる。

```
#coding: utf-8
import anydbm
import pickle
d = {u'子':u'ね', u'丑':u'うし',u'寅':u'とら',
     u'卯':u'う', u'辰':u'たつ'}

db = anydbm.open('abc.db','c')
for key, value in d.items():
    print key, value
    skey = pickle.dumps(key)
    svalue = pickle.dumps(value)
    db[skey] = svalue
db.close()
```

データベースのキーや値としてユニコード文字列を直接使うことができないので、dumps 関数でユニコード文字列を string に変換して使う。

このデータベースの読み込みには、loads 関数を使い元のユニコードに戻す。一例を示す：

```
#coding: utf-8
import anydbm
import pickle
db = anydbm.open('abc.db')
for skey, svalue in db.items():
    key = pickle.loads(skey)
    value = pickle.loads(svalue)
    print key, len(key), value, len(value)
db.close()
```

表示の結果は以下のようになる：

```
丑 1 うし 2
寅 1 とら 2
卯 1 う 1
子 1 ね 1
辰 1 たつ 2
```

日本語文字の長さが正確に計測されているのが分かる。

[**参考文献**]

柴田 淳「みんなの Python」（ソフトバンククリエイティブ：2006 年）

付 録D　日本語の処理（Python3版）

コンピュータで扱える日本語文字の範囲については付録 C の冒頭を参照してほしい。

Python3 ではユニコードの位置付けが著しく改良された。これは日本語にとっても朗報である。Python3 では引用符（’)、二重引用符（")、三連二重引用符（”””）で囲まれた文字は全てユニコード文字として扱う。つまり str 型はユニコード文字からなる：

```
>>> a=’これも’
>>> print(a,type(a),len(a))
これも <class ’str’> 3
>>> b="これも"
>>> print(b,type(b),len(b))
これも <class ’str’> 3
>>> c="""これもユニコード"""
>>> print(c,type(c),len(c))
これもユニコード <class ’str’> 8
```

文字列の長さが適切の処理されていることに注目。

8 ビット文字列はこのユニコード文字列に b をつけると得られる。

```
>>> d=b’a’
>>> print(d, type(d), len(d))
b’a’ <class ’bytes’> 1
```

しかしこの方法はマルチバイト文字では構文エラーとなる。

```
>>> e=b’あ’
SyntaxError: bytes can only contain ASCII literal characters.
```

このばあいは encode メソッドを使う：

```
>>> e='あ'
>>> f=e.encode('utf-8')
>>> print(f, type(f), len(f))
b'\xe3\x81\x82' <class 'bytes'> 3
```

Python3 で文字列を扱う上でよく引用される助言は：

> ソフトウェアは内部では ユニコード文字列のみを利用し、入力データはできるだけ早期にデコードし、出力の直前でエンコードすべきです。

D.1 辞書やタプルで日本語

日本語を辞書やタプルで使ってみる。辞書、リスト、タプルの要素としてユニコードを使う。以下はその例である：

```
#coding: utf-8
d = { '甲':('コウ', 'きのえ'),
      '乙':('オツ', 'きのと'),
      '丙':('ヘイ', 'ひのえ'),
      '丁':('テイ', 'ひのと'),
      '戊':('ボ', 'つちのえ')}
for key, value in d.items():
    print(key, value)
```

結果は

```
甲 ('コウ', 'きのえ')
乙 ('オツ', 'きのと')
丙 ('ヘイ', 'ひのえ')
丁 ('テイ', 'ひのと')
戊 ('ボ', 'つちのえ')
```

となる。辞書、タプルでも日本語が問題なく使えることがわかる（Python2 ではユニコード文字列がコードポインタに対する数字になる難点があった）。

D.2 日本語を含むファイル

簡単な日本語を含むテキストファイルを作成する：

```
#coding: utf-8
fout = open('abc.txt','w',encoding='utf-8')
word = '五月雨坊主'
fout.write(word)
fout.close()
```

ファイル名は abc.txt でユニコード文字列はパラメータ encoding で指定されたエンコード方式で 8 ビット文字列として保存される。今のばあいは utf-8 。

このファイルを読み込むには：

```
#coding: utf-8
fin = open('abc.txt','r',encoding='utf-8')
for line in fin:
    word = line.strip()
    print(word, len(word))
    print(type(word))
fin.close()
```

読み込むばあいもエンコード方式を指定する（どのような方式でエンコードされたデータであるかを指定)。これが誤っていると読み込まれた文字列は文字化けしたものになってしまう。

今のばあいの結果の表示は以下のようになる：

```
五月雨坊主 5
<class 'str'>
```

適切なユニコード文字列になっていることがわかる。

データベースに使ってみる第 14 章で扱ったデータベースにユニコード文字列を使ってみる。例として：

```
#coding: utf-8
import dbm
d = {'子':'ね', '丑':'うし','寅':'とら',
     '卯':'う', '辰':'たつ'}

db = dbm.open('abc2.db','c')
for key, value in d.items():
    print(key, value)
    db[key] = value
db.close()
```

ここではユニコード文字列でできた辞書をデータベース abc2.bd に書き込んだ。Python3 ではデータベースにユニコードを書く込む際には文字列は自動的にエンコードされて保存される。エンコードは utf-8 が既定値である。

読み込みはそのままでは 8 ビット文字列であるので decode メソッドを使ってデコードしてユニコード文字列とする：

```
#coding: utf-8
import dbm
db = dbm.open('abc2.db')
for bkey, bvalue in db.items():
    key = bkey.decode('utf-8')
    value = bvalue.decode('utf-8')
    print(key, len(key), value, len(value))
db.close()
```

print 文の出力を見ると：

```
子 1 ね 1
丑 1 うし 2
寅 1 とら 2
卯 1 う 1
辰 1 たつ 2
```

とユニコードとして適切に処理されていることがわかる。

D.3 クラスで使う

クラス定義で再定義された__str__メソッドでユニコード文字列を扱ってみる。例として：

```
coding: utf-8
class Garden(object):
    def __init__(self, plants):
        self.plants = plants

    def __str__(self):
        t = [ object.__str__(self) + '咲いている草花は' ]
        for obj in self.plants:
```

```
            s = '    ' + object.__str__(obj)
            t.append(s)
        return '\n'.join(t)

mygarden=Garden(['なずな','はこべら','ほどけのざ'])
print(mygarden)
```

このクラス定義の__str__はこの Garden クラスのインスタンスオブジェクトの print に使う。この例では mygarden がインスタンスで print(mygarden) がそれである。結果の表示：

```
<__main__.Garden object at 0x0000000002F32400>咲いている草花は
    'なずな'
    'はこべら'
    'ほどけのざ'
```

この例でもユニコード文字列が適切に処理されている（Python2 ではユニコード文字列に対する print 文がコードポイントの数になる難点があった）。

[参考文献]

https://docs.python.jp/3/howto/unicode.html

訳者あとがき

Allen Downey 著 “Think Python: How to Think Like a Computer Scientist(2nd Edition)”の日本語訳である。訳者も大学の情報科学系の学生に初級プログラミングの担当をしたことがある。C 言語を取りあげたがいくつかの問題点を感じていた。それらは、

1. C 言語の教育にも拘わらずポインタは難易度が高いとして省略せざるをえない。D.Knuth によればポインタは C 言語の宝庫である。
2. オブジェクト指向プログラミングについては別の言語で学習する必要がある。
3. 日本語は継子扱いである。

Python はこれらの問題点のほとんどを解決してくれると訳者には思えた。

Python の大きな難点は日本語で読めるよい教科書がないことである。そのようなときに出会ったのがこの Think Python である。原著者の「はじめに」にもあるように、この本は大学のプログラミングの教科書として書かれたものでる。教科書として特徴的なことはプログラムを作成する上で不可欠のデバッグについて多くのページを割いていることである。この点が原書を翻訳してみようとした大きな動機でもある。原著の第二版では第一版と比べより基礎的なところから丁寧に叙述がなされおり、より教科書として整備されてきていると感じた。

日本語の取り扱いについては原著では一切触れていない。Python3 になってユニコードの位置づけが大幅に変更になったので訳書では付録 C と付録 D とし Python2 と Python3 による日本語の処理を纏めておいた。

この訳書が如何にプログラミングをするかといったことに感心がある読者にとって助けになれば幸である。

相川利樹
仙台