Python による

プログラミング入門

第 6 版

愛知大学経営学部

有澤健治

はじめに

Python はセンスの良い現代的なプログラミング言語である。現在では Google の主力言語の 1 つにまでになっており、日本でも Python のテキストがかなり揃いだした。 Python は以下に挙げる特徴を持っている。こうした特徴は Python が今後さらに大きな支持を得て広がっていくであろう事を示唆している。

1. 無料で手に入る。
2. 殆どの OS で使用できる。
3. 電卓的な気楽な使い方から C++ のような高度な使い方まで広範囲にカバーする。
4. 簡潔、かつ、可読性の高いプログラミングが可能である。
5. アルゴリズムが率直に表現できる。
6. 高い拡張性を備えている。
7. グラフィクスが使える。

ここではあえて Python の最大の特徴である「オブジェクト指向」を挙げていない。Python では言語設計がオブジェクト指向で一貫している。一貫したオブジェクト指向による設計は、オブジェクト指向特有の小難しい言葉を忘れさせてくれるのである。

Python はインターネットで手に入る。パソコンを持っている読者は `www.python.org` から取り寄せてインストールしてほしい。インストールはとても簡単である[1]。Python の最新版 (2012 年 2 月現在) は第 3 版 (3.2.2) である。しかしながら、このテキストは第 2 版 (2.6.2) を基にして解説している。Python3 と互換性がない部分に関しては簡単なコメントを添えた。Python は Python3(第 3 版) と Python2(第 2 版) とでは基本的な違いが存在する。この違いのために、Python3 への移行がすんなりとは行かないのが現状であり、流通している Python の本も Python2 に基づくものが殆どで、また Google や Mac においても、標準的には Python2 が使われている。

日本語に翻訳されている Python の本は現在では多数存在する。残念ながらどれも他のプログラミング言語に充分精通したプログラマのための Python の入門書である。それらのうち定評のあるものとしては例えば Mark Lutz 著、飯坂剛一監訳、村山敏夫、戸田英子共訳の

- 「Python 入門」(O'Reilly Japan, 1998)
- 「Python プログラミング」(O'Reilly Japan, 1998)

がある。実はこの翻訳書は

- Mark Lutz:"Programming Python",(O'Reilly & Associates,Inc. 1996)

の分厚い英語版を 2 つに分割したに過ぎない。Python を使って本格的にプログラムを作成したい場合には

[1] インストールに関してはこの本では採り上げない。`http://ar.aichi-u.ac.jp/python/` に解説を載せておく。

- デビット・M・ビーズリー著、習志野弥治朗訳「Python テクニカルリファレンス」(ピアソン・エデュケーション)

が推奨される 。

Python のグラフィックスプログラミングに関しては紙面の都合上、本当に必要最小限の解説だけがこの本の中で加えられている。英文の本では

- John E.Grayson: “Python and Tkinter Programming” (Manning, 2000)

があるが日本語訳はまだ出ていない。ただし網羅的ではないがこの本より丁寧な解説がある。

- 有澤健治「Python による GUI の構築法 I - ウィジェットの配置 -」
  (愛知大学情報処理センター「Com」Vol.11, No.1,2000)
- 有澤健治「Python による GUI の構築法 II - ウィジェット各論 -」
  (愛知大学情報処理センター「Com」Vol.11, No.2,2000)
- 有澤健治「Python による GUI の構築法 III - Text ウィジェット -」
  (愛知大学情報処理センター「Com」Vol.12, No.1,2001)

これらは愛知大学の情報処理センターの紀要に載せた記事であるが、私のホームページからも手に入るので興味があれば読んでみるがよい。

`http://ar.aichi-u.ac.jp/python/`

Amazon で 「Python」と検索すると、最近は多数の入門書が出版され出しているのが分かる。筆者はそうした入門書に目を通した事が無いのでどれが推奨できるのかは知らない。しかし入門書はこのテキストで事足りるのではないかと思っている。

Python は非常に素直なプログラミング言語なのでプログラミング入門用に適していると思える。このテキストはそのような思いで書きあげており、他のプログラミング言語の知識を想定していない。

2012 年 2 月 20 日　　有 澤 健 治

目 次

第 I 部　Python 入門　1

第 1 章　Python の会話モード　3
1.1 四則演算 4
1.2 数学関数 6
1.3 変数 7
1.4 関数の定義と繰り返し 8
1.5 練習問題 8

第 2 章　プログラムの実行法　11
2.1 準備 11
2.2 例 1 11
2.3 例 2 13
2.3.1 打ち込みの注意点 13
2.3.2 ファイル名 15
2.3.3 実行結果 15
2.3.4 もしも間違えがあれば 15
2.4 雑談 16

第 3 章　サンプルプログラムの解説　17
3.1 表示する 17
3.2 連続した数字を書き出す 18
3.3 10 回繰り返す 19
3.4 曜日を書き出す 20
3.5 条件に応じて処理を変える 21
3.6 整形する 22
3.7 バグ (Bug) 24
3.8 関数を作る (1) 26
3.9 関数を作る (2) 27
3.10 大域 (global) 変数 28
3.11 コメント 29
3.12 好ましいプログラムスタイル 30
3.13 閏年の判定 31
3.14 Zeller の公式 34

3.15 1752 年 9 月 35

第 4 章　プログラムの作り方　37

4.1 フィボナッチ数列 37
4.2 数の総和 40
4.2.1 リストの要素の和 41
4.3 ちょっと変わった掛け算 44
4.4 グラフの交点 46
4.5 誕生日の怪 47
4.5.1 問題の提起 47
4.5.2 コンピュータシミュレーション 47
4.6 フェルマーの問題 49
4.7 再帰的アルゴリズム 50
4.8 置き石ゲーム 54
4.8.1 ゲームのルール 54
4.8.2 プログラムの考え方 54
4.9 Boyce の壷の問題 57
4.9.1 問題の提起 57
4.9.2 プログラムの考え方 57
4.9.3 譜 4.15 の改良 58

第 II 部　Python 文法　61

第 1 章　定数　63

1.1 定数の分類 63
1.1.1 基本概念 63
1.1.2 データの体系 64
1.2 数 66
1.3 文字列 (string) 66
1.3.1 文字列定数 67
1.3.2 文字列に対する基本演算 68
1.3.3 文字列の順序関係 69
1.3.4 文字列に対する基本操作 70
1.3.5 日本語の扱い 71
1.4 タプル (tuple) 73
1.4.1 タプルの基本演算 73
1.4.2 任意個の引数を持つ関数 74
1.5 リスト (list) 75
1.5.1 リストの演算 75
1.5.2 リストへの作用関数 77

1.5.3 2 次元の配列 . . . 79
1.5.4 プログラム例 (素因数分解) . . . 80
1.6 辞書 (dictionary) . . . 81
1.6.1 辞書の基本操作 . . . 81
1.6.2 辞書のメソッド . . . 82
1.6.3 プログラム例 (伝票の処理) . . . 82
1.6.4 キーとして許されるデータの種類 . . . 84
1.6.5 関数のキーワード引数 . . . 84
1.7 集合 . . . 85

第 2 章 変数と式 **87**
2.1 式 . . . 88
2.2 算術式 . . . 90
2.3 関係式 . . . 91
2.4 論理式 . . . 92
2.5 ビット演算 . . . 93
2.6 集合演算 . . . 94
2.7 lambda 式 . . . 95

第 3 章 プログラムの構成 **97**
3.1 基本概念 . . . 97
3.1.1 文 . . . 97
3.1.2 コメント . . . 97
3.1.3 行 . . . 97
3.1.4 字下げ . . . 98
3.1.5 ブロック . . . 98
3.1.6 特殊記号と空白 . . . 100
3.2 print 文 . . . 101
3.2.1 書式 . . . 101
3.3 代入文 . . . 103
3.4 式 . . . 106
3.5 条件文 . . . 107
3.6 繰り返し文 . . . 108
3.6.1 while と for . . . 108
3.6.2 range . . . 109
3.6.3 continue と break . . . 109
3.7 関数定義文 . . . 110
3.7.1 def . . . 110
3.7.2 return . . . 111
3.7.3 引数並び . . . 112

3.7.4 局所変数と大域変数 . . . 112
3.7.5 いろいろな関数引数 . . . 113
3.8 クラス定義文 . . . 114
3.8.1 class . . . 114
3.8.2 プログラム例 . . . 115
3.8.3 self . . . 116
3.8.4 クラス定義で使用可能な関数 . . . 117
3.9 エラーの捕捉 . . . 118
3.9.1 try . . . 118
3.9.2 raise . . . 119
3.9.3 except . . . 119
3.9.4 組み込みの例外名 . . . 120
3.10 import 文 (モジュール) . . . 121

第 4 章 ファイル **123**
4.1 ファイルの中の数の総和 . . . 123
4.2 ファイルの基本操作 . . . 126
4.3 sys.stdin . . . 129

第 5 章 基本関数 **131**
5.1 input . . . 131
5.2 raw_input . . . 133
5.3 rstr, repr, ‘...‘ . . . 133
5.4 filter . . . 134
5.5 map . . . 135
5.6 reduce . . . 135
5.7 eval . . . 136
5.8 exec . . . 136
5.9 execfile . . . 137
5.10 compile . . . 137
5.11 id . . . 138
5.12 type . . . 138
5.13 hash . . . 138
5.14 callable . . . 139
5.15 dir . . . 139
5.16 vars . . . 139
5.17 locals . . . 140
5.18 globals . . . 140

第 III 部　Python によるグラフィックス　141

第 1 章　グラフィックスの基礎　143

1.1 サンプルプログラム . . . 143
1.2 キャンバス . . . 146
1.3 座標系 . . . 147
1.4 基本図形 . . . 148
1.4.1 create_line . . . 148
1.4.2 create_rectangle . . . 150
1.4.3 create_polygon . . . 151
1.4.4 create_oval . . . 152
1.4.5 create_arc . . . 153
1.4.6 create_text . . . 154
1.5 オプション (補足) . . . 156
1.5.1 font . . . 156
1.5.2 color . . . 158
1.5.3 stipple . . . 159
1.5.4 capstyle . . . 160
1.5.5 joinstyle . . . 160
1.5.6 tags . . . 160
1.6 印刷 . . . 164

付 録 A　基本演算と基本関数　167

A.1 演算子と算術関数 . . . 167
A.2 基本関数 . . . 168

付 録 B　基本モジュール　171

B.1 math モジュール . . . 171
B.2 文字列のメソッド . . . 173

付 録 C　colors1.py　175

付 録 D　colors2.py　177

第I部

Python 入門

第1章　Python の会話モード

Python はプログラミング言語の 1 つではあるが、プログラムの概念を知らなくても Python を使うことができる。数学関数を備え、式を扱うことができ、メモリーを自由に使える高級な電卓の様に振る舞うのである。そのためには Python を会話モードで実行する。

Windows の場合、Python を会話的に実行するには、Windows の「スタート」から「全てのプログラム」を選び、Python の IDLE を実行する[1]。

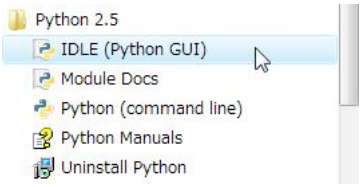


図 1.1: Python IDLE

すると図 1.2 の画面が生成される。

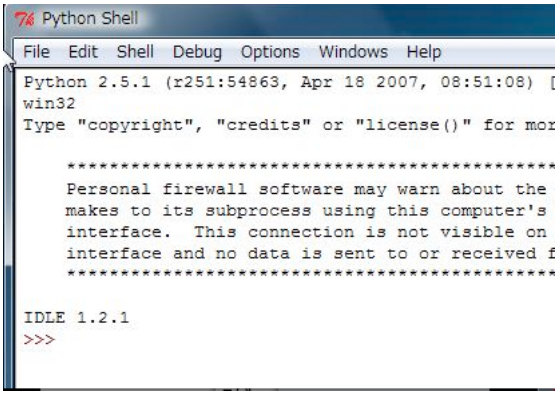


図 1.2: Python Shell

[1]Windows の場合、Python は標準的にはインストールされていないので www.python.org から取り寄せる必要がある。MacOSX や UNIX では標準でインストール済みである。IDLE は初心者向けの環境であり、本格的な利用には向かない。MacOSX で IDLE を使いたい場合には MacPython が使える。詳しくは http://ar.aichi-u.ac.jp/python/ を見よ。

1.1　四則演算

Python の会話モードでは

```
>>> 3*(5+8)
39
```

この様に、単に式を入れるだけで計算してくれる。ここに現われる * は掛け算の記号である。そして 39 は Python が計算してくれた結果である。

演算の優先順位はほぼ数学の習慣に従っている。乗除算の方が加減算よりも優先度が高い。数式の中で計算の優先順位を指定する括弧は丸い括弧 () だけが使える。他の括弧 [] や {} は他の意味に使用されているので優先順位の指定には使用しない。だから括弧が入り組んでいる場合にも

```
3*(7*(5*(1+2)+3)+11)
```

のように丸い括弧だけで計算の優先順位を指定する。

割り算に関してはプログラミング言語 C と同様な注意が必要である[2]。即ち Python では整数を整数で割ると整数の範囲で商を出す。

```
>>> 1/3
0
```

実数の範囲で商を求めるには 1/3 の代わりに 1.0/3 あるいは 1/3.0 と書く。そうすると

```
0.333333333333
```

を出力するであろう[3]。小数点以下のもっと多くの表示を求めたい場合とか、逆にもっと簡潔に表示したい場合には明示的に書式を与える必要がある。しかしながら会話的な実行の下ではその必要は殆ど無いであろう。従って書式の解説は後回しにする。ここでは計算機の常識として、実数の計算はあくまで近似計算であり厳密な精度を持っているのではない事だけを指摘しておく。(Python では 10 進数表示で 16 桁の精度である。)

整数を整数で割った時の剰余を求める場合には演算子 % を使用する。例えば、

```
>>> 18 % 7
4
```

となる[4]。乗除に関係した 3 つの演算子 (*, /, %) は優先度が等しい。その優先度が等しい演算は左から右へ計算を行う。従って、例えば

```
31/3*6%7
```

[2] プログラミング言語 C はベル研究所の Kernighan と Ritchie によって、1970 年代に UNIX のために開発されたシステム記述用の言語である。そして、現代のコンピュータ文明を支える基本的な言語となっている。

[3] この割り算記号 "/" の説明は Python2 に適用される。Python3 では割り算の記号の使い方が変更されている。Python3 では整数の範囲での割り算は "/" ではなく、"//" を使う。そして "/" を使った場合には 、実数の範囲で答えを出す。このテキストでは Python2 の割り算規則を使っている。

[4] 負の整数に対する剰余の計算規則は第 II 部第 2 章 2 節を見よ

は (演算の結果生成される結果を便宜的に → で示すと)

```
31/3 → 10
10*6 → 60
60%7 → 4
```

の順で計算が行われている事を意味している。

べき乗の計算には ** を使用する。

```
>>> 2**10
1024
```

殆どのプログラミング言語では 2 の 100 乗を計算できない。しながら Python ではメモリが許す限り大きな整数を計算してくれる[5]。

```
>>> 2**100
1267650600228229401496703205376L
```

最後の L は、大きな整数の場合に表示される[6]

絶対値が非常に大きい実数あるいは非常に 0 に近い実数では指数表現を使用できる。例えば 1.23×10^5 は

```
1.23E5
```

と書く。ここに現われる E5 は $\times 10^5$ を意味している。従って 1.23E5 は 123000 と同じである。同様に 1.23E-5 の E-5 は $\times 10^{-5}$ を意味し、0.0000123 と同じである。E は小文字で書いてもよい。

Python は 8 進数や 16 進数も扱うことができる。8 進数は数字の先頭に 0 (ゼロとオー) を付ける[7]。8 進数の 177 を 10 進数で知りたい場合には

```
>>> 0177
127
```

で知ることができる。

16 進数は 0x で始める。16 進数の 7F を 10 進数で知りたい場合には

```
>>> 0x7f
127
```

で知ることができる。

10 進数ではなく、8 進数や 16 進数で出力したい場合もある。その場合には関数 oct あるいは hex を通せばよい。例えば

[5] 2**100 の計算結果には数字の末尾に L が付いて表示されるかも知れない。これは以前の版の Python の名残である。

[6] これは古い Python の名残で、現在では必要の無い表示である。Python3 では表示されない。

[7] Python2 では、ゼロとオーを添えて 0o177 のように書いてもよい。Python3 では、0o177 の書き方に統一されている。

```
>>> oct(127)
'0177'
>>> hex(127)
'0x7f'
>>> hex(0xc+0xd)
'0x19'
```

とすればよい。

Python は複素数の計算も行う。虚数単位は j で表す。

```
>>> (2+3j)*(3-5j)
(21-1j)
>>> (2+3j)**5
(122-597j)
>>> abs(2+3j)
3.605551275464
```

虚数単位 j を使用する時には必ず j の前に数字を書く。例えば

```
1j
```

の様に j に掛ける数字が 1 であっても 1 を書く。でないと j は変数であると見なされる。

1.2　数学関数

数学関数を使用する場合には

```
>>> from math import *
```

を実行しておく。この命令の意味は、「math モジュールから全ての関数を取り込め」である。「全て」は「*」で表現されている。sin と cos だけを取り込みたい場合には、* の代わりに、sin,cos と書く。モジュールとは関連した関数を集めたファイルの事である[8]。以下に簡単な例を挙げる。

```
>>> pi
3.14159265359
>>> e
2.718281828459
>>> tan(1)
1.557407724655
>>> s=sin(1)
>>> c=cos(1)
```

[8]詳しくはの付録の math モジュールを見よ。

```
>>> s,c
(0.841470984808, 0.540302305868)
>>> s/c
1.557407724655
```

ここで pi は円周率 π、e は自然対数 e であり、Python ではこの 2 つの数学上の基本定数が math モジュールの中で定義されている。三角関数 sin(x) 、 cos(x) 、 tan(x) がこの例に現われている。

関数は数学の習慣通り、名前の後の () の中に関数に渡すデータを書く。このデータを引数と言う。三角関数の引数の単位はラジアンである。[9] 従って例えば 60 度の sin の値を知りたい場合には

```
>>> sin(pi/180*60)
0.866025403784
```

のように角度 60 に pi/180 を掛けておく必要がある。

1.3 変数

第 1.2 節の例題では変数が現われている。この例題に現われる

```
s=sin(1)
```

は sin(1) が生成するデータに対して s と言う名前を付けたのである。このようにデータに名前を付けておくと、このデータを後に利用する時に便利である。データに付けられた名前を変数と言う。記号 ‘ =’ の左に名前を書き、右に名付けるべきデータがくる。データに付けられた名前は新たに名前が付け直されるまで有効である。次に例を示す。

```
>>> a=3
>>> a
3
>>> b=2*a
>>> b
6
>>> a=5
>>> b
6
>>>
```

a が 5 になっても、b は (10 ではなく) 6 のままである事に注意する。「=」は「代入する」であって「等しい」ではないからである。

また実際のプログラムの中で頻出する次のような例も、また変数と「=」の意味を理解する上で有用である。

[9] 360 度で 2π ラジアンである。

```
>>> n=3
>>> n=n+1
>>> n
4
>>>
```

n + 1 の計算結果 (4 となる) が、n に代入され、n が 4 になったのである。

変数名として任意の文字列が許される訳ではない。1 文字の英字 (アルファベット) は変数名となる資格を持っている。英字で始まり記号を含まない文字列もまた変数名となる資格を持っている[10]。

変数を使用すると込み入った計算を行う場合に便利である。例えば

```
>>> 3*6.276429*(6.276429+7)
```

のような計算を行いたいとせよ。この場合にはむしろ

```
>>> x=6.276429
>>> 3*x*(x+7)
```

とする方が利口である。

数学に慣れていると 3*x を 3x にするなど、掛け算の記号 * を書くのを忘れがちである。プログラムの中の式の表現では掛け算の記号を省略できない。

1.4　関数の定義と繰り返し

Python では会話モードで自分で関数を定義する事もできる。また繰り返し等の高級な処理を行う事もできる。しかし、こうした事を会話モードで行う事は推奨できない。会話モードでは誤りが発生した場合のロスが大きい。ファイルを作成し、そこにプログラムを記述して、それを実行した方が利口なのである。従って関数の定義や繰り返しに関しては第 2 章以降で解説する。

1.5　練習問題

問 1 次の *5* つの数の和を求めなさい。

```
923746
736427
783644
103847
348582
```

[10] Python の変数に関する詳しい解説は第 II 部の第 2 章と第 3 章を見よ。

また電卓を使ってこの計算を行う事の長所と短所について考えて見なさい。

問 2 Python で書かれた次の計算式を最初に手で計算し、その答えをプログラムを実行する事によって確認しなさい。

```
(3*(2+5)-1)*2
17/3*2
3*(2+3)**3
1+2**3
-2**4+1
```

問 3 次の 2 つの数 7636272 と 93838 について和、差、積、割り算の商、割り算の余り、実数の範囲での割り算を求めなさい。割り算については 7636272 を 93838 で割るものとする。またこの計算を変数を活用して行いなさい。

ヒント: 実数の範囲での割り算の計算を行うには、一方の数に `.0` を付加し実数として扱えばよい。変数を使っている場合には `1.0` を掛ければよい。

問 4 四捨五入によって得られた 2 つの数字 3.278 と 73.2 がある。この 2 つの数字の掛け算を計算機で計算すると 239.9496 になるのだが、この数字のどこまでが信頼できるかを元の数字の曖昧さを考慮した上で吟味しなさい。

ヒント: 3.278 は四捨五入で得られた数字てあるから、元の数字は、3.2775 以上、3.2785 未満である。

問 5 $7^1, 7^2, 7^3, 7^4, 7^5, 7^6, 7^7, 7^8, 7^9, 7^{10}$ を求めなさい。またこれらを 11 で割った余りを求めなさい。特に最後の 7^{10} を 11 で割った余りが 1 になっている事に注目しなさい。

注釈: この問題は少し意地悪である。面倒だけど、会話モードで何とかやっていける。面倒だと言う事を味わって欲しかったのである。むしろプログラムで処理すべき問題である。

問 6 上の問題は、次の有名な定理 (Fermat の定理):

> m が素数の時に x^{m-1} を m で割った余りは 1 である[11]。ここに x は $1, 2, \cdots, m-1$ のどれでもよい。

の特別な場合 ($x = 7$、$m = 11$) です。$x = 2$ として、いろいろな m について試し、m が素数でなくとも余りが 1 になる m の値を見つけなさい。

注釈: この問題はさらに意地悪である。会話モードでは、よほど根気が無いと見つけられないであろう。この問題は第 3 章を読み終えた頃には、手頃なプログラミングの練習問題になっているはずである。

[11] m が素数でなくても 1 になることがあるので注意せよ

第2章　プログラムの実行法

プログラムが複雑になれば、間違いも多くなる。その場合には会話的な実行法は現実的ではない。ファイルにプログラムを書いてそれを実行した方が、間違えた時の訂正が容易なので開発効率が良い。以下では Windows 7 の環境を前提にプログラムによる Python の使い方を述べる。

ここではそのために Python IDLE 付属のエディタを用いて、これをファイルに打ち込み、実行することがこの章のテーマである。

2.1　準備

ドキュメントフォルダの中に Python と言う名のフォルダを作成し、そこに自分が作成した Python 関係のファイルを保存する事にする。そのためにはドキュメントフォルダを開き「新しいフォルダ」をクリックする (図 2.1)。すると「新しいフォルダ」と言う名のフォルダが作成されるので、それを「Python」に変更する。

図 2.1: フォルダの作成

2.2　例 1

Python Shell の File メニューから New Window を開けば Python Editor が起動される (図 2.2)。

まず最初に、たった一行

```
print "OK"
```

から構成される簡単なプログラムを実験的に実行してみよう[1]。

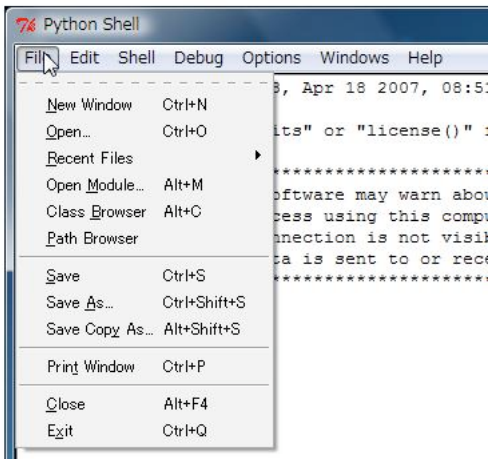


図 2.2: Python Editor の起動

そのためには図 2.3 のように Python Editor にプログラムを打ち込む。編集画面に ”Untitled” の表示が出ている状態では実行はできない。実行する前に、この内容をファイルに保存する必要がある。

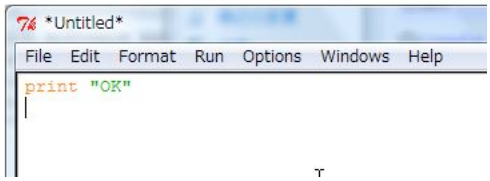


図 2.3: OK を表示するプログラム

「File」メニューから「Save」を選び、ホームフォルダの Python フォルダに、適当な名前 (例えば a.py) を付けて保存する。

Python のファイル名には習慣的に拡張子 ”.py” を添える。これを添える事によって、Python のファイルが蛇のアイコンで表示され、またマウスクリックでプログラムが実行される。

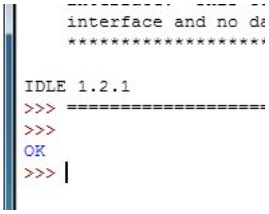


図 2.4: プログラムの実行結果

[1] ここでの “print” の使い方は Python2 に添っている。Python3 では
print("OK")
のように、丸い括弧でデータを括らなくてはならない。この違いは形式的な小さな違いであるかのような印象を受けるかも知れないが、実は基本的な考え方に大きな違いがある。この違いはあまりにも大きいので、Python3 への移行の障壁になっている。このテキストでは Python2 の簡素な print に基づいて解説する。

実行には「Run」メニューの「Run Module」をクリックする。結果は Python Shell の画面に表示される (図 2.4)。

2.3 例 2

もう少し大きなプログラムの例としてカレンダーを表示するプログラム (譜 2.1) を実行してみる。

譜 2.1 プログラム 1. cal.py

```
# coding: shift-jis
wday=("sun","mon","tue","wed","thu","fri","sat")
def cal0(n,m):
    # cal0(n,m) は 1 つの月のカレンダーを作成する。
    # n は月の開始日の曜日を表す数字 (0 から 6) であり、
    # 0 は日曜日を意味する。
    # m はその月の日数である。例えば 1 月は m=31 である。
    # 従って 2000 年 1 月のカレンダーは cal0(6,31) で表示される。
    for x in wday: print " ",x,
    print
    for x in range(0, n): print "  ---",
    for x in range(1, m+1):
        print "%5d"%x,
        if (x+n)%7==0 : print
    print
cal0(6,31)
```

2.3.1 打ち込みの注意点

以下に Python のプログラムを打ち込む時の注意点を挙げる。

1. この中で # 記号で始まる行はコメントである。コメントはプログラムを読む人のために書かれる。即ち、コメント部分はプログラムの実行結果に影響を与えない[2]。(従って今の場合、コメントを書き写さなくても構わない。)

2. 日本語の文字 (今の場合にはコメントの中に現れている) がプログラムの中で使用される場合には、最初の行に文字コード情報を追加する事。例えば日本語 Windows 環境であれば SHIFT-JIS が文字コードとして採用されているので、プログラムは

   ```
   # coding: shift-jis
   ```

   で始める[3]。

[2] 最初の行の `# coding: shift-jis` は文字コードの解釈を与えており、プログラムの実行に影響を与える。その意味で純粋なコメントではない。

[3] Mac や UNIX など、Windows 以外では国際文字コードである UTF-8 が使われる事が多い。その場合には `utf-8` を指定する。

3. 各行の左側の空白 (これを字下げ空白と言う) の個数は重要な意味を持っている。図 2.5 にこのプログラムの字下げ空白を影で示す。コメントは省略されている。

```
wday=("sun","mon","tue","wed","thu","fri","sat")
def cal0(n,m):
    for x in wday: print " ",x,
    print
    for x in range(0, n): print "  ---",
    for x in range(1, m+1):
        print "%5d"%x,
        if (x+n)%7==0 : print
    print
cal0(6,31)
```

図 2.5: Python の字下げ空白
字下げ空白の部分は影で示されている。

この例では字下げの深さ (各行の字下げ空白の個数) は 4 の倍数になっている。この例の様に 4 ずつ深くする必要はないが、各行の深さの順序は守らなければならない。字下げは TAB キーによって行ってもよいが、TAB キーによる字下げと空白キーによる字下げは混在してはならない。

4. プログラムは普通の英文と同様に単語と区切り記号と空白から構成されている。但しプログラムは計算機に命令を下す為の人工言語なので単語や区切り記号の意味は必ずしも英語の文章と同じではない。英文と同様に、単語と単語の間の空白は単語と単語を区切る役割を果たしている。このような空白は省略できない。この例では単語と単語を全て 1 個の空白で区切っているが、数個の空白で区切っても構わない。例えば

   ```
   def cal0(n,m):
   ```

   は

   ```
   def    cal0(n,m):
   ```

   と書いてもよい。また、他の区切り記号によって区切られていれば、空白は無くても構わない。例えば

   ```
   range(0, m)
   ```

   は

   ```
   range(0,m)
   ```

   と書いてもよい。空白を多用するか、最小限に留めるかは単に好みの問題である。

プログラムは小さな命令の寄せ集めであり、それらの命令の意味は第 3 章で解説する予定である。

2.3.2 ファイル名

プログラムを書き込むファイルの名称は、この場合にはカレンダーを連想させるものがよいであろう。例えば calendar の先頭 3 文字を借用して `cal.py` としよう[4]。

2.3.3 実行結果

打ち間違えが無ければ次の様に表示されるはずである。

```
sun   mon   tue   wed   thu   fri   sat
---   ---   ---   ---   ---   ---     1
  2     3     4     5     6     7     8
  9    10    11    12    13    14    15
 16    17    18    19    20    21    22
 23    24    25    26    27    28    29
 30    31
```

これは 2000 年の 1 月のカレンダーである。

任意の年の任意の月のカレンダーを作成するには、その月が何曜日から始まるのか、その月は何日で構成されるのかが分ればよい。そのような情報を与えた時にカレンダーを表示してくれるのが、ここで実行してみたプログラムである。(でも本当に難しいのは、昔の暦の制度がどうなっていたかなどの歴史学的な正確な知識を手に入れる事です...)

2.3.4 もしも間違えがあれば ...

もしも間違えがあれば、例えば

```
  File "cal.py", line 11
      for x in range(0, n+m);
                            ^
     SyntaxError: invalid syntax
```

のように表示される。この場合には「11 行目の最後がセミコロン (;) になっていますが変じゃないですか?」って言っているのである。セミコロンの下の山印 (^) が問題の箇所を指している。

問題を引き起こした行の内容が表示されるので、小さなプログラムでは指摘された行を見つけるのは容易であるが、大きなプログラ

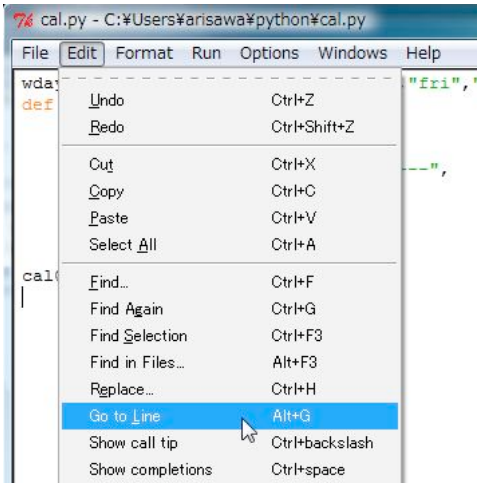


図 2.6: 行番号を指定して行を探す

[4] ファイル名には空白文字や日本の文字は使わない方が良い。トラブルの原因になる。

ムでは (行番号がエディタの中で表示されていないので) 見つけにくい。そのような場合には、Python Editor では「Edit」メニューの「Go to Line」によって、行番号を指定して行を見つける事ができる (図 2.6)。

`SyntaxError` と言うのは、文法上のエラーを意味する。この他に実行に至らないエラーとして `NameError`(存在しない変数を参照した) など様々な種類のエラーが存在する。

このケースでは山印 (^) は正しく誤りを指摘したが、いつもうまく行くとは限らない。Python のプログラムを処理するプログラム (=Python) はプログラムを英文を読むのと同じ順序で読み、この山印 (^) の箇所まで読んだ時に誤りを発見したのである。従って大抵の場合にはその場所に誤りはあるが、その前の部分が本当の誤りの可能性もある。

練習問題

問 1 2000 年 4 月のカレンダーを表示しなさい。(プログラムの中のコメントにヒントが書かれています。)

問 2 SyntaxError と NameError 以外の、実行に至らないエラーを見つけなさい。

問 3 最後まで実行はするのだが変な結果を表示するプログラムの例をプログラム 1 を 1 文字だけ変更して見つけなさい。

2.4　雑談

IDLE 環境は初心者向けの環境であり、Python の本来の姿が見えてこない。普通のテキストエディタと python コマンドの組み合わせが通常のプログラム作成法である。Mac OSX や UNIX 環境であればそのようなスタイルで解説するところであるが、Windows 環境ではコマンドは極めて使いにくい。

商品として出回っているソフトウェア (プログラム) はグラフィックスを使って美しく飾り、マウスでメニューを選択しながら操作を進めていく。これはこれで良い面もあるが、欠点もある。プログラムが重くなるばかりか、必ず人間が介在しなくてはならないために大量のデータの処理が困難になり、さらに、自動処理が不可能になるのである。

コンピュータを使う最大のメリットは処理の自動化にある。人手を介して行えば膨大な労力と時間を要する処理が、プログララムさえ作れば人手を介する事無く一瞬の内に完了するのである。Python は分かりやすく、かつ、強力なプログラミング言語であり、コマンド環境で実行される Python のプログラムは現在ではインターネットのサーバーなどで多く利用されるようになっている[5]。

[5]Python は Google 社の基本ソフトウェア開発言語の一つである。例えば Google App Engine では Java と Python がサポートされている

第3章　サンプルプログラムの解説

ここでは第 2 章で扱ったサンプルプログラムを解説する。このプログラムの中にはいくつかのキーワード print、def、if、for、in、range が現れる。これらの意味を理解するのがこの章の目標である。

3.1　表示する

print *X*

データ *X* を出力する

例 1 次の 2 つの数 7636272 と 93838 について和、差、積、割り算の商、割り算の余り、実数の範囲での割り算を求めなさい。但し割り算については 7636272 を 93838 で割るものとする。

譜 3.1 四則演算のプログラム

```
x=7636272
y=93838
print x+y
print x-y
print x*y
print x/y
print x%y
print 1.0*x/y
```

解説

この問題は第 1 章の問題 3 である。非会話モード (以下では単にプログラムと言う) では計算結果は自動的には表示されない。従ってプログラムの中でのデータの表示には print を使用する。6 つ目の print で 1.0 を掛けているのは、この計算を実数の範囲で行うべきである事を Python に知らせている。命令の実行は上の行から順に行われるのでその計算結果もその順で表示される。

プログラムは色々な方法で書く事が可能なのである。例えば

```
print 1.0*x/y
```

は

```
print float(x)/y
```

と書いてもよい。むしろ訓練されたプログラマはこのように書くであろうが、ここでは初等的な方法を採用する。なお `float(x)` は `x` を実数に変換する関数である。

また 6 個の計算結果を 1 つの行に表示するには

```
print x+y, x-y, x*y, x/y, x%y, 1.0*x/y
```

と書けばよい。面倒ではあるが、

```
print x+y,
print x-y,
print x*y,
print x/y,
print x%y,
print 1.0*x/y
```

によっても 1 つの行に出力される。`print` 命令の最後のコンマ (`,`) が存在する事によって改行動作の発生を押さえているのである。

3.2 連続した数字を書き出す

for *x* in range(*n*,*m*):

x を $n \leqq x < m$ の範囲で順に実行する

例 2 1 から 31 までの数字を表示しなさい。

譜 3.2 連続した数字を書き出すプログラム

```
for x in range(1,32):
    print x
```

解説

1 月のカレンダーを書くには 1 から 31 までの数字を書き出す必要がある。カレンダーの場合にはそれらの数字を置く位置を決めるという難しい問題が含まれるが、それについて考える前に、ともかくも 1 から 31 までの数字を書き出せなくてはならない。

書き出す数字の範囲は range(1,32) に現れる 2 つの数字、1 と 32 で決定されている。31 まで書き出すのだから 31 と書きたくなるのだが、そうではない事に注意する。range(1,32) の意味している範囲は 1 以上 32 未満なのである。

「x の初期値を 1 に設定し、x を 1 づつ増やして、range(1,32) の範囲で print x を実行しなさい。」と言うのが、例題のプログラムの意味である。

問 1 このプログラムに現れる x は他の名前でもよい。x の代わりに、n と置き換えても、同じ動作をすることを確認しなさい。

問 2 3 から 30 までの数字を書き出しなさい。

問 3 range(1,32) を range(1,32,3) と置き換えると、1,4,7,.... のように、増分が 3 になる。実際にプログラムを実行して、最後の数字を求めなさい。

問 4 1 から 31 の数を横方向に書きなさい。(ヒント: print 命令の改行抑制の “,”。)

3.3 10 回繰り返す

for *x* in range(0,10):

10 回繰り返す。*x* は 0,1,2,...,9 と変化するが、この *x* を利用しても、利用しなくてもよい

例 3 次のように “---” を 10 個書き出しなさい。

```
--- --- --- --- --- --- --- --- --- ---
```

譜 3.3 10 回繰り返すプログラム

```
for x in range(0,10):
    print "---",
```

解説

譜 3.3 では x は利用されていない。単に

print "---", が 10 回実行されるだけである。

同じ事を行うプログラムは (x が利用されていないために)

```
for x in range(1,11):
    print "---",
```

とも書けるが、譜 3.3 の方が「10 回」である事が分かりやすくて良い。

3.4　曜日を書き出す

for *x* in *L*:

L の要素 *x* を順に実行する

例 4 カレンダーの表示には、曜日 “sun”,· · · ,“sat” が含まれる。この曜日を for 文を使って書きなさい。曜日の表示は 1 つの行に書き出して、全ての曜日を書き出したら改行しなさい。

譜 3.4 曜日を書き出すプログラム

```
for x in ("sun","mon","tue","wed","thu","fri","sat"):
    print x,
print
```

解説

先の譜 3.2 に含まれている for の本質的な使い方は

for *x* in *L*:

である。ここに *x* は任意の変数、*L* はリストやタプルのような、要素を持っているデータである。リストは角括弧 [ ] で囲われ、タプルは丸括弧 ( ) で囲われている。リストの要素は変更可能であるが、タプルの要素は変更できない。譜 3.4 ではタプルが使われている。for 文では、リストやタプルの要素を順に実行して行く。*x* を for 文の制御変数と言う。

譜 3.4 の最後には、書き出すデータが書かれていない、単なる print が現れるが、その場合には何も書かないで単に改行するだけである。これによって曜日を書いた後、次に書き出すデータは行の先頭になる。(この事は、次のように、最後に何かデータ書き出してみれば確認できる。

```
for x in ("sun","mon","tue","wed","thu","fri","sat"):
    print x,
print
print "abc"
```

さらに次のプログラムを実行してみよう。

```
for x in ("sun","mon","tue","wed","thu","fri","sat"):
    print x,
    print
print "abc"
```

今度は 1 つの曜日を書き出す毎に改行されている。各々の繰り返しについて for 文はインデントされている部分を実行する。例えば x が "sun" の時には次の 2 つの文が実行される。

```
print x,
print
```

従って sun を書き出して改行する。他の曜日についても同様である。

プログラムの中で最もよく使用されるリストは最も単純なもの、即ち、[3,4,5,6,7] のように連続した整数の並びである。そのような単純な構造のリストを容易に扱える様にするために関数 range(n,m) が存在する。range(n,m) は n から始まり、増分が 1 で、m 未満の整数を要素とするリストを生成する関数である。これを使用すると例えば [3,4,5,6,7] は range(3,8) と表現できる。リストの要素が少ない場合にはリストを明示的に書くことも可能であるが

```
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,
25,26,27,28,29,30,31]
```

とは誰しも書きたくはないであろう。これを簡単に range(1,32) で生成してくれるのであるから range は有り難い関数である。

問 1 譜 3.4 にコードを追加して、曜日を書き出した後、行を改め、1 から 31 までの数字を改行せずに書き出し、最後に改行するプログラムにしなさい。

問 2 譜 3.2 に現れた range の正体をみるために

```
print range(1,6)
```

を実行してみよう。これによって range はデータを生成するもの、すなわち関数であることが分かる。生成されたデータはタプルかそれともリストか?

3.5 条件に応じて処理を変える

if *C*:

C には条件を書く

例 5 1 から 31 までを表示しなさい。ただし 7 の倍数 (7,14,21,28) の直後で改行しなさい。すなわち次の表示結果を出すプログラムを作りなさい。

```
1 2 3 4 5 6 7
8 9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
29 30 31
```

譜 3.5 7 の倍数で改行するプログラム

```
for x in range(1,32):
    print x,
    if x%7 == 0:
        print
print
```

解説

譜 3.5 では、x は 1 から 31 まで変化する。そして各々の x の値について、

```
print x,
if x%7 == 0:
    print
```

が実行される。まず、x の値を書き出して、その次に、x が 7 の倍数であれば print が実行され、改行される。ここでは x が 7 の倍数である事を、7 で割った余りが 0、すなわち、x%7 == 0 で判定している。ここの “==” は等しい事を表している。(Python では “=” は代入記号である事に注意)

条件が成立しない場合 (この場合 x が 7 の倍数では無い場合)、print は実行されない。

問 1 月の最初が水曜日から始まるプログラムでは 4,11,18,25 で改行しなくてはならないであろう。そこで、次の結果が得られるプログラムを書きなさい。(ヒント: これらは 7 で割って 4 余る)

```
1 2 3 4
5 6 7 8 9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28 29 30 31
```

3.6　整形する

```
"%5d"%x
```

整数 *x* を 10 進数の文字列に変換し、
5 カラムになるよう空白を付加する

例 6 日曜日から始まり、ひと月が 31 日のカレンダーには次のように数字が現われる。

```
  1     2     3     4     5     6     7
  8     9    10    11    12    13    14
 15    16    17    18    19    20    21
 22    23    24    25    26    27    28
 29    30    31
```

数字の表示位置が揃っていることに注意する。(2 と 3 の間には 5 個の空白があり、23 と 24 の間には 4 個の空白がある) これを書いてくれるプログラムを作りなさい。

譜 3.6 整形したプログラム

```
for x in range(1, 32):
    print "%5d"%x,
    if x%7==0 :
        print
```

解説

整形して出力する問題はプログラミングの初心者にとって難しい課題である。

```
print "%5d"%x,
```

の中の文字列

```
"%5d"
```

は書き方 (書式) を指定している。この場合は整数を 5 カラムで表示しなさいと言っているのである。この文字列の中の % 記号に続く数字が表示カラム幅の指定である。その数字の後に、d を書けば 10 進数で表示するのである。書式を指定する文字列の後に % 記号が来て、それに続いて表示すべき整数が来る。今の場合には表示すべき整数は x で指定されている。文字列をカラム幅を指定して表示するには書式文字列の中で % 記号に続けてカラム幅数を書き、その数字の後に s を書く。例えば次のプログラム

```
for x in ("sun","mon","tue","wed","thu","fri","sat"):
    print "%5s"%x,
print
```

の中の print 文の書式指定文字列 "%5s" は文字列を 5 カラムで出力する事を指示している。出力の対象となる文字列は x で与えられており、x はタプル

```
("sun","mon","tue","wed","thu","fri","sat")
```

の要素である。このプログラムを実行すると

```
  sun   mon   tue   wed   thu   fri   sat
```

が出力される。

注意深い読者は、5 カラムの出力を指定したのに 6 カラムで出力しているではないかと疑うかもしれない。Python の print 文はデータとデータの間に 1 個の空白を入れる。そして "%5d"%1 は "␣␣␣␣1"を生成し (空白文字を ␣ で示した)、"%5d"%23 は "␣␣␣23"を生成する。Python はこのようにして生成された (空白を含む) 文字列データを、データの間に 1 個の空白を挿入して表示するのである。(従って各行の先頭のデータは 5 カラムで表示されている。)

問 1 次の出力が得られるプログラムを作りなさい。

```
sun   mon   tue   wed   thu   fri   sat
  1     2     3     4     5     6     7
  8     9    10    11    12    13    14
 15    16    17    18    19    20    21
 22    23    24    25    26    27    28
 29    30    31
```

問 2 次の出力が得られるプログラムを作りなさい。

```
sun   mon   tue   wed   thu   fri   sat
---   ---   ---     1     2     3     4
  5     6     7     8     9    10    11
 12    13    14    15    16    17    18
 19    20    21    22    23    24    25
 26    27    28    29    30    31
```

3.7　バグ (Bug)

ここまで来れば、かなりカレンダーのプログラムらしくなる。しかし、いくつかの不便を解消してプログラムを改良して行きたい。

例 7 次の出力

```
---   ---   ---     1     2     3     4
  5     6     7     8     9    10    11
 12    13    14    15    16    17    18
 19    20    21    22    23    24    25
 26    27    28    29    30    31
```

はプログラム

```
for x in range(0,3):
    print "%5s"%"---",
for x in range(1,32):
    print "%5d"%x,
    if x%7 == 4:
        print
print
```

によって得られる。このプログラムでは木曜日から始まるカレンダーに変更する時に、2 カ所の変更が必要になる。一カ所の変更で済むようにプログラムを改良しなさい。

解説

2 カ所の変更とは `range(0,3)` の 3 と、`x%7 == 4` の 4 である。3 + 4 = 7 の関係にあるから、以下のようなプログラムで良さそうである。n に代入する数字 (ここでは 3) を調整しようと言う訳である。この場合 n の値と曜日をどのように関係づけるのが良いのだろうか? 自然な考え方は、
`n=0` を日曜日、`n=1` を月曜日、…、`n=6` を土曜日
として対応付ける事だろう。

読者は実際に、n にいろいろな数字を与えてみてプログラムの動作を確認して欲しい。

```
n=3
for x in range(0,n):
    print "%5s"%"---",
for x in range(1,32):
    print "%5d"%x,
    if x%7 == (7-n):
        print
print
```

実際に実験してみれば分かるように、日曜日以外では正しくカレンダーが表示されているのに、日曜日だけが正しく表示されない。`n=0` としても、`n=7` としてもである!

プログラムの中の欠陥をバグ (Bug = 虫) と言う。全然動かないプログラムは捨てられるので実害は無い。やっかいなのは、この例のように、殆どのケースで正しく動くが、ある特定の条件の下で発生するバグである。この例でもそうであるが、処理すべき数値が、ある範囲にある時、バグは端で発生しやすい。肝に命じておくべし。

何故、このプログラムが正しく働かないのか? `n=0` の時には改行の条件は

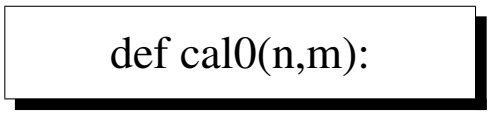
```
if x%7 == 7:
    print
```

となる。しかし、`x%7` は (x を 7 で割った余りなので)、7 になることはあり得ない!

問 1 `n=0` の場合にも正しく動くようにプログラムを修正しなさい。

ヒント: 初等的な解決法を 2 つ挙げる。(1) 2 つ目の for 文の前に、`n==0` 場合に `n=7` となるような命令を追加する。(2) `(7-n)` を `(7-n)%7` に置き換える。 もっと高度な解決法は、サンプルプログラム (譜 2.1) で採用されている方法である。すなわち `(x+n)%7==0` の条件で改行する。この考え方は高度すぎて、普通の人は気づかなくて当たり前である。嘆く必要はない。

3.8　関数を作る (1)



新しい関数 cal0(n,m) を作る

プログラミング言語には、1 つの機能を果たすコードの纏まりの再利用を容易にする仕組みがある。Python では関数やモジュールがそのために存在する。通常、関数はプログラムの中で直接定義するが、関連した関数が多い場合には、それらの関数群はモジュールに纏められる[1]。

他の言語では「手続き」と呼ばれるものも存在する[2]。「手続き」はコードを纏める点で関数と似ているのだが、関数と異なりデータ (値) を生成せず、プログラムの中では命令のように使われる。Python の場合には「手続き」は None と言う特殊な値を生成する関数として扱われる。None は演算の対象にならないので式の中では使わない。従って None を返す関数は、「手続き」と同様に、命令のような使い方になる。関数が生成する値は return で指定される。return で明示的に値を指定されない場合には None が関数の値になる (すなわち「手続き」のようになる)。

例 8　1 つの月のカレンダーを表示する関数 cal0(n,m) を作成しなさい。ここに n は月の最初の日の曜日を表す数字で、日曜日を 0、月曜日を 1、…、土曜日を 6 とする。m はその月の日数である。また、cal0(n,m) を使って、木曜日から始まり、31 日で終わるカレンダーを表示しなさい。

譜 3.7 cal0(n,m) の定義

```
def cal0(n,m):
    for x in ("sun","mon","tue","wed","thu","fri","sat"):
        print "%5s"%x,
    print
    for x in range(0,n):
        print "%5s"%"---",
    for x in range(1,m+1):
        print "%5d"%n,
        if (n+x)%7 == 0:
            print
    print
cal0(4,31)
```

解説

関数の定義は def で行う。def に続けて、定義する関数の名前と、その関数にデータを渡す窓口 (関数引数と言う) を指定する。関数の定義部はインデントする。従って、次のインデ

[1] 例えば数学モジュールには sin(x) や cos(x) など多数の数学に固有の関数が納められている。
[2] 他に「サブルーチン」と呼ばれることもある。

ントされていない行までが関数の定義部である。

関数はそれが使用される前に定義しておく必要がある。この例では cal0(4,31) で定義された関数が使用される。(これを関数の呼び出しと言う。)

関数の呼び出し時に指定された 4 と 31 は、定義部の n と m に渡される。つまり、関数 cal0 が呼び出された時に、定義部の n と m の値は 4 と 31 になっている。

以上の事を簡単に次の図で示す。

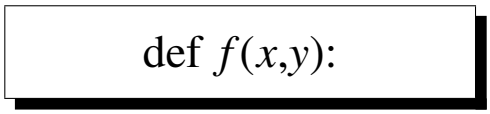
```
def cal0(n,m):
      (a) ここに関数 cal0(n,m) の定義を書く。
      (b) 関数が使用される前に、関数を定義しておく。
      (c) 定義の範囲はインデントで示す。
      (d) 引数 n,m の初期値は n=4, m=31 となり、その下で定義部のコードが実行される。
cal0(4,31) # ここで関数が使用されている。(関数を呼び出している)
```

問 1 次のプログラムの出力を予測しなさい。

```
def f(x,y):
     print (x+y)*y
f(2,3)
print f(2,5)
```

ヒント: 最後に出力された None は、この関数の値である。

3.9 関数を作る (2)



新しい関数 $f(x,y)$ を作る

カレンダーのプログラムの理解とは直接関係がないが、ここで、通常の関数の解説をしておく。通常の関数は値を生成する。生成する値は return で指定される。

例 9

$$f(x,y) = (x+y) \times y$$

となる関数 $f(x,y)$ を定義し、$f(2,3)$ と $f(2,5)$ の値を表示しなさい。

譜 3.8 f(x,y) の定義

```
def f(x,y):
    return (x+y)*y
print f(2,3)
print f(2,5)
```

解説

`print f(2,3)` が実行されると、関数引数 (2,3) が 関数定義文の (x,y) に渡され、x=2, y=3 となる。その下で (x+y)*y が計算され、`return 15` が実行され、この 15 が f(2,3) の値になり、print 15 が実行され、15 が表示される。`print f(2,5)` についても同様である。

問 1

$$f(x) = \frac{1}{(x^2 + 1)}$$

となる関数 $f(x)$ を定義し、$f(2)$ と $f(3)$ の値を表示しなさい。

ヒント: 実数の割り算にしておかないと、頭の中が ???? となる[3]。

問 2 半径 r の球の体積 V は

$$V = \frac{4}{3}\pi r^3$$

で求められる。これを求める関数を `V(r)` とし、`V(2)` と `V(3)` の値を表示しなさい。

ヒント: 円周率 π は数学モジュール `math` に `pi` として含まれている。`math` の中の `pi` を使うには

```
from math import pi
```

を最初に実行しておけばよい。

3.10　大域 (global) 変数

譜 2.1 では、

```
wday=("sun","mon","tue","wed","thu","fri","sat")
def cal0(n,m):
    for x in wday: print " ",x,
    print
    ........
cal0(6,31)
```

のように[4]、曜日のデータを一旦、変数 `wday` に入れられ、それが `cal0` の定義の中で使われた。この `wday` への代入は、行の先頭から始まっている。つまり、関数定義の外で代入され

[3] Python3 では割り算記号 "/" は常に実数を生成する。そして整数の範囲での割り算は "//" に変更された。Python3 の割り算記号の使い方は、数学計算に伴う誤りを減らす上で良いアイデアである。

[4] プログラムの中に現れる "........" の所には何かプログラムコードが続く。

ている。このような変数を大域変数と言う。この例では wday を利用する関数は 1 つだけであるが、一般に大域変数はいろいろな関数から利用できる。

データを変数に入れるメリットの 1 つは、変数に適切な名前をつける事によって、そこに代入されているデータの性質が分かることにある。しかし、単にそのメリットを得たいだけであれば

```
def cal0(n,m):
    wday=("sun","mon","tue","wed","thu","fri","sat")
    for x in wday: print " ",x,
    print
    ........
cal0(6,31)
```

と書いてもよい。このように関数の中で定義された変数はローカル変数と呼ばれ、大域変数ではない。ローカル変数は、定義されている関数の中でのみ参照できる。

大域変数の使用にはデメリットもある。このような簡単な例では分からないが、大域変数はいろいろな関数から利用されている可能性があるので、大域変数を使ったプログラムは、後で変更する時に細心の注意を要する。つまり、保守性が悪くなるのである。使う必然性がない限り、大域変数は使わない方が良い。

ローカル変数にせよ、変数を使えば、それが複数の箇所で利用されている可能性がある。その事に気づかずに、後で修正を加えた時に、問題が発生する事がある。そのような事を考えた時には、変数にする必然性がない限り、変数を使わない方が良いと言える。つまり、次のプログラムの方が好ましい。

```
def cal0(n,m):
    for x in ("sun","mon","tue","wed","thu","fri","sat"): print " ",x,
    print
    ........
cal0(6,31)
```

3.11 コメント

コメントを関数定義の最初に入れておくと、後で再利用しやすい。譜 2.1 では関数 cal0(n,m) に対するコメントを次のように入れていた。

```
def cal0(n,m):
    # cal0(n,m) は 1 つの月のカレンダーを作成する。
    # n は月の開始日の曜日を表す数字 (0 から 6) であり、
    # 0 は日曜日を意味する。
    # m はその月の日数である。例えば 1 月は m=31 である。
    # 従って 2000 年 1 月のカレンダーは cal0(6,31) で表示される。
    ........
cal0(6,31)
```

公式配布されている Python のプログラムでは次のようにコメントが入っている。

```
def cal0(n,m):
    """ cal0(n,m) は 1 つの月のカレンダーを作成する。
    n は月の開始日の曜日を表す数字 (0 から 6) であり、
    0 は日曜日を意味する。
    m はその月の日数である。例えば 1 月は m=31 である。
    従って 2000 年 1 月のカレンダーは cal0(6,31) で表示される。"""
    ........
cal0(6,31)
```

ここに現れる、二重引用符を 3 つ並べた """ は、引用符 ' や " と同様に、文字列を表す引用符であるが、行をまたがって使える。一重の引用符 (') を使ってもよいが、文字列の始まりと終わりには同じ引用符を使う必要がある[5]。

既に述べたように、Python ではデータをプログラムの他の部分から利用しても利用しなくてもよい。ここでは文字列を書いているのだが、その文字列は利用されていない。単に書かれているだけである。そのような文字列はコメントとして代用できる。

3.12　好ましいプログラムスタイル

譜 2.1 では、

```
if (x+n)%7==0 : print
```

と書かれているが、この部分は

```
if (x+n)%7==0 :
    print
```

と書いてもよい。むしろ、この方が好ましい。

好ましい書き方は、首尾一貫した書き方である。Python ではインデントでプログラムの論理構造を表現している。既に気づいた読者もいると思うが、範囲を示さなくてはならない命令の後には、コロン (:) が続く。例えば

```
def cal0(n,m):
for x in range(1,32) :
if x%7 == 0:
```

などである。そして、その後にはコードブロックが続く。そのコードブロックは、簡単な場合には、コロンの後に改行せずに続けてもよいが[6]、その中にコロンが含まれるような複雑な

[5] 引用符を 3 つ並べた長い行にまたがるコメントは、コメントの始まりと終りが分かりづらい。この問題は工夫によって解決すべきである。

[6] 同じ行に複数個の命令を書く場合には命令をセミコロン (;) で区切る。しかし、複数の命令を 1 つの行に書くのは避けるべきである。

場合には必ず改行して、次の行からインデントのレベルを深くして書かなくてはならない。プログラムを見やすく、首尾一貫した書き方にするには、コロンで改行し、インデントで範囲を示す書き方に統一すべきである。

検索エンジンで有名な Google 社は、プログラムのスタイルについての社内規則を持っており、その日本語訳が

http://works.surgo.jp/translation/pyguide.html

に載っている。

3.13 閏年の判定

カレンダーの例題から逸れるが、閏年の判定関数について考えてみよう。

例 10 西暦年号を与えてその年が閏年であれば 1 を、閏年でなければ 0 を返す関数を定義しなさい。(この関数の名前を leap としましょう。英語では閏年の事を leap year と言います) そして、その関数が正しく働いていることを 1996 年、1999 年、1900 年、2000 年 の 4 つの年について確認しなさい。

譜 3.9 条件に応じて処理を変える関数

```
def leap(x):
    if x%400 == 0: return 1
    if x%100 == 0: return 0
    if x%4 == 0: return 1
    return 0
print 1996, leap(1996)
print 1999, leap(1999)
print 1900, leap(1900)
print 2000, leap(2000)
```

解説

閏年は次の規則で判定される:

a. 4 で割り切れる年は閏年である。

b. 100 で割り切れる年は閏年でない。

c. 400 で割り切れる年は閏年である。

判定の優先順位は c, b, a の順である。例えば 1900 は a, b に適合するが b の判定が優先度が高いので閏年ではない。2000 は a, b, c に適合するが c の優先度が最も高いので閏年である。従って実際の判定は c, b, a の順に行うのがよい。

さて、判定のための関数名を `leap` とした。関数名は 1 文字だけではなく長い名前も許される。但し名前は数字と英字 (アルファベット) だけで構成され、英字で始まる必要がある。変数名についても同様である。

閏年を判定する関数の定義は 1 行で書くことができるほど簡単ではない。関数の定義が複数の行にわたる場合には Python では字下げによって定義文の範囲を表現する。従って

```
def leap(x):
    if x%400 == 0: return 1
    if x%100 == 0: return 0
    if x%4 == 0: return 1
    return 0
```

が関数 leat(x) の定義である。この定義文の最初に現われる

```
    if x%400 == 0: return 1
```

は x を 400 で割った余りが 0 に等しい場合には return 1 を実行しなさいと言う意味である。そしてその場合には return 1 によって関数の値が 1 に設定され、関数の計算はここで終了する。従って残りの

```
    if x%100 == 0: return 0
    if x%4 == 0: return 1
    return 0
```

は実行されない。if は条件を表す Pyhton の予約語である。また 条件式の中の == は等しい事を表す。(= ではない事に注意せよ。= は変数への代入を意味する。ちなみに、等しくない事は != で表される。) 他方、もしも

```
    x%400 == 0
```

が成立しない場合には、

```
    if x%400 == 0: return 1
```

に続く命令が実行される。

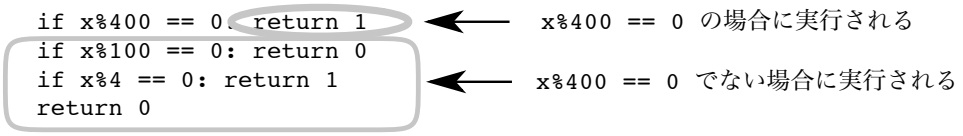


図 3.1: 条件式 x%400==0 の結果による実行される文の違い

プログラムは命令 (プログラミング言語では文と言う) の寄せ集めである。文には単純なものもあるし、if を使った文 (if 文) のように内部構造を持ち、他の文をその内部に含むような複合的な文もある。関数定義文 (def 文) もそうした複合的な文の 1 つである。

この Python のプログラムは

```
print 1996, leap(1996)
```

から実行される。これを単に

```
print leap(1996)
```

と書いていないのは、どの年の計算を行っているのか、計算結果を見て分るようにするためである。他の年についても同様である。

問 1 譜 3.9 では 4 で割り切れない場合の判定が最後に回されている。閏年でないケースが大半なのだからこの判定を最初に行う方が実行効率が良いと考えられる。この考えのもとにプログラム 3 の関数 `leap(x)` を書き直してみなさい。

注: この問題は難しい。そのような難しいプログラムの書き方を推奨しているのではない。

問 2 現在の暦 (グレゴリオ暦) は 1582 年 10 月 15 日から開始された [注 1]。つまり、現在の閏年の規則はこの時に制定された。この日は何曜日かを求めよ。(プログラムで解く事を要求していない)

ヒント: グレゴリオ暦は 1 年を 365.2425(= 365 + 1/4 − 1/100 + 1/400) 日と定義したことになる。従って 400 年間で [ ] 日になり、この日数は丁度 7 で割り切れる。従って 1600 年のカレンダーは 2000 年のカレンダーと同じである。

問 3 閏年を判定する関数 `leap(x)` を if 文を使わないで書くことが可能であると言えば驚きではないだろうか?

ヒント: 西暦 x 年までに発生した閏年の回数と $x-1$ 年までに発生した閏年の回数の差を計算します。(差だけが問題なので、グレゴリオ暦が実際にいつから始まったかを気にする必要はありません)

閏年豆知識

閏年は太陽年が 365.242191 日で[7]、1 日を単位として 1 年間を表した時に端数を持つ為に発生する。太陽年と言うのは地球から観測される太陽の動きを基に定めた 1 年の日数である。例えば北極点では夏至に太陽が最も高くなる。この夏至から夏至の周期が太陽年であると考えればよい。1 年を 365 日と定めると 4 年に一度 1 日程度の狂いが発生する。これが閏年である。1 年が丁度 365.250000 日ならば 4 年に一度の閏年の法則で問題はないが、実際にはそうではない。ではこの違いは何年に 1 回現われるか? 以下は Python の会話モードでの計算である。

```
>>> a=365.250000-365.242192
>>> a
0.007809
>>> 1/a
128.057369701481
```

[7]「アルゴリズム辞典」(共立出版、1998)

なんと 2 進数で区切りの良い 128 になるのである。100 年と 400 年で調整するのではなく 128 年で調整した方が良いのだ。

3.14　Zeller の公式

年月日から曜日を計算する方法として Zeller の公式がある。巧みな公式であり解説するつもりは無い。半ば理論的な、半ば試行錯誤によるもので、こんな巧い計算法があるものだと言う程度に受け取って貰えればよい。もちろんこの式は、1582 年 10 月 15 日以前には適用できない。なお以下のプログラムは「好ましいプログラムスタイル」ではない。1 ページに納めるために苦労しているのだから...

譜 3.10 Zeller の公式を使ったカレンダープログラム

```
def h(y,m,d):
    # h(y,m,d): day of week (Zeller's formula)
    #  奥村晴彦「コンピュータアルゴリズム辞典」(技術評論社,1989)
    # y: year, 0,1,2,..
    # m: month, 1,2,..,12
    # d: day of month, 1,2,..31
    # return: day of week, 0,1,2,..,6
    # example: h(2000,1,1) -> 6    # Saturday
    if m<3: y=y-1; m=m+12
    return (y+y/4-y/100+y/400+(13*m+8)/5+d)%7

def cal0(n,m):
    # cal0(n,m) は 1 つの月のカレンダーを作成する。
    # n は月の開始日の曜日を表す数字 (0,1,..,6) であり、0 が日曜日を意味する
    # m はその月の日数である。例えば 1 月は m=31 である
    # 従って 2000 年 1 月のカレンダーは cal0(6,31) で印刷される
    for x in ("sun","mon","tue","wed","thu","fri","sat"): print " ",x,
    print
    for x in range(0, n): print "  ---",
    for x in range(1, m+1):
        print "%5d"%x,
        if (x+n)%7==0 : print
    print

def cal(y,m):
    # y: year, 0,1,2,..
    # m: month, 1,2,..,12
    n=h(y,m,1)
    n1=h(y,m+1,1)
    r=(n1-n)%7
    d=28+r
    cal0(n,d)

cal(2000,1)
```

3.15 1752 年 9 月

ところで実は Python には calendar モジュールが備わっており、cal(y,m) に相当する関数 prmonth(y,m) が存在する。これを使って 1752 年 9 月のカレンダーを表示させると

```
   September 1752
Mo Tu We Th Fr Sa Su
             1  2  3
 4  5  6  7  8  9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28 29 30
```

他方 UNIX には cal コマンドが存在する。cal コマンドで 1752 年 9 月のカレンダを出すにはシェルから

```
cal 9 1752
```

を実行する。すると次の表示が得られる。

```
   September 1752
 S  M Tu  W Th  F  S
       1  2 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
```

変な結果のように見えるが、これはバグではない。UNIX の cal コマンドはイギリスの暦を表示している。計算だけではない、歴史をも考慮した、まともなカレンダーなのである。イギリスではこの年、この月にようやく暦が改訂された。(イギリスはローマ法王の影響力が弱いのである。)

問 1 グレゴリオ暦の前にはユリウス暦が使用されていた。ユリウス暦では単に 4 で割り切れる年を閏年と定めた。(つまり 1 年間を 365.25 日と考えたのである。) ユリウス暦は 1582 年 10 月 4 日 (木曜日) を最後にグレゴリオ暦に改められその翌日が 10 月 15 日 (金曜日) と定められた。従って Zeller の公式はユリウス暦の下では次の様に修正される。

```
def h0(y,m,d):
    if m<3: y=y-1; m=m+12
    return (y+y/4+(13*m-2)/5+d)%7
```

この下で 1582 年 9 月のカレンダーを表示しなさい。また、ユリウス暦とグレゴリオ暦が接続された 1582 年 10 月のカレンダーを表示しなさい。(ヒント: 接合月を表示する関数を柔軟にしてシンプルに設計すること。)

暦豆知識

読者は閏年が何故 2 月なのか不思議に思った事はないだろうか? 実はユリウス暦の前のローマ暦では March から新年が始まり、February は 1 年間の最後の月だったのである。(それで February で調整した。) この名残は月名に現れている:

9 月	September	sept-	7 の意味
10 月	October	oct-	8 の意味
11 月	November	nove(イタリア語)	9 の意味
12 月	December	decade	10 の意味

第4章　プログラムの作り方

プログラムとは人間が行っているデータ処理を自動化する仕組みである。自動化すると言う事は、処理の方法を知らなくても自動的に行ってくれると言う意味ではない。事態は全くその逆である。普段なにげなく行っている処理の手順を正確に分析し、それを計算機に理解できる形に言い表すと言う事である。従って計算機は何を理解し、何を行う能力を持っているかを知っている必要がある。

計算機ができる事をまとめるとそれ程多くはない事が分る。大雑把に言えば計算機ができる事はたった次の 4 つである。

1. 計算機はデータを記憶できる。
2. 計算機は簡単な計算ができる。
3. 計算機は簡単な論理判断ができる。
4. 計算機は以上の 3 つを順序立てて実行できる。

人間についてはどうであろうか? 人間は発見的な方法で問題を解決できる。これは計算機の持っていない能力である。しかしながら発見的な方法によって確実に問題が解決できる訳ではない。確実に問題を解決できる場合、解法があると言う。そして解法があれば、それを計算機に教える事ができる事が知られている。即ち、プログラム可能なのである。結局人間が確実に処理可能な問題は計算機と同様に上の 4 つの問題に帰着できるのである。

4.1　フィボナッチ数列

問題が与えられてから、プログラムを組むまでのプロセスを理解するために、例題としてフィボナッチ数列を採りあげよう。フィボナッチ数列とは最初に 2 つの数 1 と 1 から出発して残りの数はその前の 2 つの数を足して得られる数列である。

```
1    1    2    3    5    8    ...
```

フィボナッチ数列が何の役に立つかはここでは問題にしない。この数列の 8 以降にいかなる数字の列が続くかは、足し算さえ知っていれば誰でも理解できる。プログラムの初等的な練習問題として手頃であるから最初に採り挙げたのである。

問 1 フィボナッチ数列に現われる数 8 以降に続く 5 つの数を (手計算で) 求めよ。

計算がどのように行われていくかを分析しよう。そのために数字が生成されるプロセスを以下の様に簡潔に表現してみる。

```
1 + 1 → 2
1 + 2 → 3
2 + 3 → 5
3 + 5 → 8
```

計算のどのステップでも 3 つの数だけが関係している。この 3 つの数の入れ物 (変数) を準備する事とし、それに (左から順に) x、y、z と名前を付ける。即ち

```
x + y → z
```

と考える。(注意: 矢印記号 → はフローチャートを書く時に使用される代入記号であり、プログラミング言語で実際に使用される事はない。) 次の表は計算の各ステップで変数 x、y、z の値がどのように変化していくかを表している。

	x	y	z
STEP1	1	1	2
STEP2	1	2	3
STEP3	2	3	5
STEP4	3	5	8

ここで行われている計算を Python の言葉で表そう。各ステップで z は x と y から計算できる。

```
z=x+y
```

次のステップへの移行に伴って、x と y の値を再設定する必要がある。

```
x=y
y=z
```

そして、これを繰り返すのである。最初に x と y には 1 と 1 を設定する事を忘れてはならない。フィボナッチ数列を計算する手順は Python 風に表現すれば大体次のようなものであることがわかる。

```
x=1
y=1
繰り返す:
    z = x + y
    x=y
    y=z
```

出力はどのようにすべきであろうか? フィボナッチ数列を最初の数から順に表示していきたいのなら各ステップでの x の値を表示すればよいであろう。

```
x=1
y=1
繰り返す:
    print x
    z = x + y
    x=y
    y=z
```

と書けばよい事が分る。しかしまだプログラムは完成していない。「繰り返す」と書いた部分は Python は理解しないし、繰り返しの終了条件も与えていない。今仮に 10000 より小さい範囲でフィボナッチ数列を出力したいとせよ。この場合には単なる「繰り返す」ではなく「`x < 10000` の範囲で繰り返す」のである。これは Python では「`while x < 10000`」と表現する。こうして初めて Python にも理解できるプログラム

譜 4.1 フィボナッチ数列プログラム

```
x=1
y=1
while x < 10000:
    print x
    z = x + y
    x=y
    y=z
```

が完成するのである。

問 2 このプログラムを実行しなさい。もしも 30 番目のフィボナッチ数を知りたい場合には出力にどのような情報が含まれているのがよいかを考えよ。

問 3 プログラム 1 のプログラムに現われる

```
x=y
y=z
```

は実行順序に敏感である。これを

```
y=z
x=y
```

とした場合に正しく動作しない。このことを、STEP4 から STEP5 へ移る場合を列に採って説明せよ。

繰り返しの条件は様々な形が考えられる。例えば 100 項目まで出力し、何項目のフィボナッチ数であるかが分る様にしたい時にはどうするか? 次はそのようなプログラムの例である。

譜 4.2 for 文を使ったフィボナッチ数列のプログラム

```
x=1
y=1
for n in range(1,101):
    print n, x
    z = x + y
    x=y
    y=z
```

フィボナッチ数列の例題はプログラムが完成するまでに次の手順を踏む必要がある事を示している。

1. 必要な変数の個数と各々の役割を定める事
2. その下で計算手順を分析する事
3. 計算手順を計算機言語 (今の場合は Python) で表現する事

この内最も難しい部分は 1 と 2 であって、この部分はアルゴリズムの問題である。3 はアルゴリズムの計算機言語による表現の問題であり比較的易しい部分である。この部分をコーディング (coding) と言う。

問 4 最大公約数の計算は互除法が効率の良いアルゴリズムとして有名である。この方法によると、6201 と 11349 の最大公約数は以下の様に除算を繰り返して求まる。(右側に計算プロセスを記号的に示す。)

11349 を 6201 で割った剰余を計算して 5148 を得る。　`11349%6201 → 5148`
6201 を 5148 で割った剰余を計算して 1053 を得る。　`6201%5148 → 1053`
5148 を 1053 で割った剰余を計算して 936 を得る。　`5148%1053 → 936`
1053 を 936 で割った剰余を計算して 117 を得る。　`1053%936 → 117`
936 を 117 で割った剰余を計算して 0 を得る。　`936%117 → 0`
剰余が 0 になった所で計算を止め、117 を答とする。

この計算法を使用して 5442853 と 4555013 との最大公約数を求めなさい。

問 5 最大公約数を返す関数 `gcd(x,y)` を定義せよ。またその下で以下の計算が正しく行われているか否かを確認せよ。

`gcd(11349,6201)` , `gcd(6201,11349)`

また `x`, `y` の一方が 0 の時に `gcd(x,y)` は何を返すべきかを最大公約数の定義に基づいて考えよ。

問 6 家を買う為に、銀行から金利 5% で 3000 万円借りるとせよ。毎年 200 万円ずつ返済する。何年後にこのローンを返せるか? また最後の年にはいくら支払えばよいか?

4.2　数の総和

2000 年の 10 月のカレンダーを作成するに 10 月 1 日の曜日を知る必要がある。2000 年の 1 月 1 日が土曜日であると言う事実を利用すれば 10 月 1 日の曜日は計算できる。1 月から 9 月までの各月の日数の和を計算し (2000 年は閏年である事に注意せよ):

31 + 29 + 31 + 30 + 31 + 30 + 31 + 31 + 30

この計算結果を 7 で割り、余りを調べればよい。余りは 1 である。従って 10 月 1 日は (土曜日の次の) 日曜日である。

カレンダーの例を挙げるまでもなく、数の和の計算は日常生活至る所に転がっている有り触れた問題である。そこでこの節では数の和の計算について考える。

4.2.1 リストの要素の和

問い

リスト d=[31,29,31,30,31,30,31,31,30] の要素の和を計算するプログラムを作りなさい。

解答例

譜 4.3 リストの要素の和を求める

```
d=[31,29,31,30,31,30,31,31,30]
s = 0
for x in d:
    s = s + x
print s
```

解説

和の計算は次の様に行われている。

```
 0 + 31 → 31
31 + 29 → 60
60 + 31 → 91
...
```

従って計算の各々のステップを

```
x + y →  z
```

と考える。ここに y は各ステップにおける d の要素であり、次のステップに移行する際には

```
z →  x
```

が実行されている必要がある。以上の事をそのままプログラムに表すと、

```
d=[31,29,31,30,31,30,31,31,30]
x = 0
for y in d:
    z = x + y
    x=z
print x
```

となる。ところでこのプログラムにはまだ改善の余地がある。今の場合には

```
z=x+y
x=z
```

は

```
x=x+y
```

の 1 つで足りるのである。こうして、

```
d=[31,29,31,30,31,30,31,31,30]
x = 0
for y in d:
    x = x + y
print x
```

が得られる。このプログラムはプログラム 2 と同じである事に注意せよ。変数名は完全に任意に選んで構わないのだから。単に好みの問題である。変数名はその役割が連想されるものが好まれる。和は英語で sum であり、従って和の累積を記録する変数には屡々 s が使用される。

問い

リスト d=[31,29,31,30,31,30,31,31,30,31,30,31] の最初の 9 つの要素の和を求めなさい。

色々なプログラムの方法があるので順に解説する。

解答例 1

譜 4.4 リストの最初の 9 つの要素の和を求めるプログラム

```
d=[31,29,31,30,31,30,31,31,30,31,30,31]
s = 0
n=0
for x in d:
    s = s + x
    n = n + 1
    if n == 9: break
print s
```

このプログラムでは和をとった回数を調べ (そのために変数 n を導入している)、n が 9 になったら計算を終えている。break 文を用いると繰り返しから途中で抜け出す事ができる。

```
    n = n + 1
```

が実行されると n の値が 1 だけ増加する事に留意せよ。この書き方は回数を管理する時に非常によく使用される。

譜 4.5 リストの最初の 9 つの要素の和を求めるプログラム

```
d=[31,29,31,30,31,30,31,31,30,31,30,31]
s = 0
for n in range(0,9):
    s = s + d[n]
print s
```

解答例 2

このプログラムに現われる d[n] は d の n+1 番目の要素を表している。即ち Python では d の要素は順に

d[0], d[1], d[2], ..., d[11]

で表すのである。要素を表す [ ] の中の数字を d のインデックス (index) と言う。インデックスは 1 から始まらずに 0 から始まる事に注意せよ。我々の日常生活では数字は 1 から始まると考えている。しかし 0 から始まると考えた方が合理的である。Python では後者の立場を採っている。しかしながらプログラミングにおける数の始まりの考え方の違いは、日常生活的な考え方に慣れ切っている我々にとって思考ギャップをもたらすので注意が必要である。

解答例 3

譜 4.6 リストの最初の 9 つの要素の和を求めるプログラム

```
d=[31,29,31,30,31,30,31,31,30,31,30,31]
s = 0
for x in d[:9]:
    s = s + x
print s
```

d[:9] はリスト d の先頭の 9 個の要素から構成されるリストである。即ち

[31,29,31,30,31,30,31,31,30]

に等しい。従って d[:9] を使用すればこの問題は例題 1 に還元できる。

4.3　ちょっと変わった掛け算

公開鍵暗号にも応用されている数学の問題を扱おう。紙面の都合上もちろん全てを解説できない。ほんの入り口である。整数 x と y および正整数 m を与え x と y を掛けて m で割った余りの計算を行う。

図 4.1 は $x = 1, \cdots, 12$、$y = 1, \cdots, 12$、$m = 13$ の場合の表である。例えば $x = 3$、$y = 5$ の場合には $x \times y$ は 15 で、これを $m(= 13)$ で割った余は 2 であるから、この表の 3 行 5 列の交点は 2 になっている。

1	2	3	4	5	6	7	8	9	10	11	12
2	4	6	8	10	12	1	3	5	7	9	11
3	6	9	12	2	5	8	11	1	4	7	10
4	8	12	3	7	11	2	6	10	1	5	9
5	10	2	7	12	4	9	1	6	11	3	8
6	12	5	11	4	10	3	9	2	8	1	7
7	1	8	2	9	3	10	4	11	5	12	6
8	3	11	6	1	9	4	12	7	2	10	5
9	5	1	10	6	2	11	7	3	12	8	4
10	7	4	1	11	8	5	2	12	9	6	3
11	9	7	5	3	1	12	10	8	6	4	2
12	11	10	9	8	7	6	5	4	3	2	1

図 4.1: mod 13 による掛け算表

この表をよく鑑賞しよう。乱雑に数字が並んでいるように見えるが[1]、どの行にも 1 から 12 の数字が全て現れている。そしてどの列にも 1 から 12 の数字が全て現れている。まるで魔法陣のようだ。凄いではないか! どうしてこんな事が起こるのか? その探求は数学の話になるのでここでは行わない。興味ある読者は整数論の入門書に載っているので読んでみたら良い。

この表は次の譜 4.7 のプログラムから得られた。既に解説済みのカレンダーのプログラムを理解していれば、このプログラムは易しいであろう。

譜 4.7 剰余による掛け算表

```
m=13
for x in range(1,m):
    for y in range(1,m):
        print "%4d"%(x*y%m),
    print
```

[1]行や列が半ばぐらいになるとまるで乱数のように数字が散らばっている。規則性が読み取りにくいのである。この性質のために剰余の計算は初等的な乱数とか暗号などに利用される。

m を他の数字で確かめてみよう。そうすれば m が素数の場合には先に述べた特別の性質が成り立っていることが確認できるであろう。

我々は $0, \cdots, 12$ までの有限個の数字の世界に生きているとしよう。この世界においては 2 つの数 x と y は $x-y$ が m で割れきれる時に等しいと考える。そしてその事を

$$x \equiv y \mod m$$

と表す。例えば

$$3 \times 5 \equiv 2 \mod 13$$

である。m を固定して考えている場合には (mod m と書くのは面倒なので) 単に

$$3 \times 5 \equiv 2$$

とも書く。これはこの世界での新しい掛け算である。

この世界はどのような性質を持っているであろうか? 整数論はこのような問題を扱っている。以下、その香りを一端を紹介する。

この表から次の事が分かる。どの行どの列においても 1 が存在する。従って、どの x についても $x \times y \equiv 1$ となる y が存在する。例えば $x = 3$ の時には $y = 9$ である。この y を x の逆数と言い x^{-1} で表す。この変な掛け算の世界では足し算、引き算、掛け算について閉じているだけではなく割り算についても閉じているのである[2]。

この掛け算のやり方で $2, 2^2, 2^3, \cdots, 2^{12}$ を計算して行ってみよう。すると

```
 2   4   8   3   6  12  11   9   5  10   7   1
```

となる。最後の 1 に注目しよう。この数字は 2^{12} を $m(=13)$ で割った余りである。

Fermat の定理

m が素数のとき、$x = 1, \cdots, m-1$ のどの x に対しても

$$x^{m-1} \equiv 1 \mod m$$

この定理は Fermat の小定理とも言う。証明は省略する。

m が素数でない時には大抵はこの関係式が成立しない。例えば $m = 12$ の場合 $x = 2$ で

```
 2   4   8   4   8   4   8   4   8   4   8
```

となる。

公開鍵暗号では大きな素数を利用する。ある数が素数であるか否かの良い判定法があれば実用的に大きな価値を持つ。多くの数学者がこの問題に挑戦しているが決定的な方法は見つかっていない。この問題に Fermat の定理がどの程度役に立つか? この定理によって、ある数 m が素数ではないことの判定は容易になったが、素数である事の判定には使えない[3]。

[2] 閉じているとは、$0, \cdots, 12$ の中で演算ができる事を言う

[3] この問題に関連する話題が第 4.7 節に載っている

4.4　グラフの交点

関数 $y = \cos(x)$ と 関数 $y = x$ の交点を求めよう。いろいろな求め方があるが、ここでは (効率は悪いが) 初等的な方法を用いる。普通はニュートン法と言うもっと効率の良い方法が使用されるのだが、微分法の知識を必要とするので興味のある読者は他書を参照して頂きたい。

ここに紹介するのは二分法である。交点の x 座標が x_0 と x_1 の間に入るような x_0 と x_1 を最初に適当に選ぶ。但し x_0 と x_1 の間には交点は 1 つだけ存在しなくてはならない。また話を簡単にするために $x_0 < x_1$ とする。

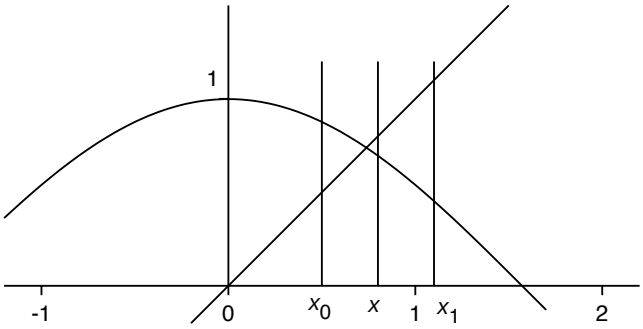


図 4.2: 二分法

この下で $x = (x_0 + x_1)/2$ が交点の右にあるのか、それとも左にあるのかを調べる。この図では右にあるので x_1 を x で置き換える。もしも左にあれば x_0 を x で置き換えればよい。この考えをプログラムで表すと次の様になる。

譜 4.8 2 分法を使った交点の計算

```
#
# root of a function
# Bisection method
#
from math import *
def f(x):
    return cos(x)-x
x0=0
x1=pi/2
while x1 - x0 > 1.0e-8:
    x = (x0+x1)/2
    y = f(x)
    print x,y,x0,x1
    if y > 0: x0=x
    elif y < 0: x1=x
    else: break
```

ここでは `f(x)` を $\cos(x) - 1$ と置いている。ここに現われる `elif` は Python の予約語で、`if` に続く条件 `y>0` が成立しない場合の条件を書く。(`elif` は else if から派生した造語である。)

予約語 else はどの条件も成立しない場合を受け持っている。この場合には y==0 の場合である。もしも y==0 となれば解が求まった事になり繰り返しから抜け出す必要がある。else に続く break はそのためにある。

さて while 文の条件 x1 - x0 > 1.0e-8 の中の小さな数 1.0e-8 は計算の精度を定めている。Python では実数の精度は 1.0e-16 程度なので、この数字を 1.0e-16 より小さくしても意味がない。また

```
print x,y, x0, x1
```

は計算の途中経過が分る様に入れられている。

他の関数について計算する場合には、このプログラムでは f(x0)>0 と f(x1)<0 が仮定されている事に注意すべきである。逆の場合についても同様に計算できる様にするためにはちょっとした工夫が必要である。

問 1 次の式で表される 2 つのグラフ

$$\begin{aligned} y &= -x^2 + 1 \\ y &= x \end{aligned}$$

の交点を求めなさい。(2 つとも求めなさい)

4.5 誕生日の怪

4.5.1 問題の提起

フランク先生はとても優しい先生で子供たちから慕われていた。ある日教室で「私の誕生日のパーティには必ず来てね」と生徒たちにせがまれ約束する事になった。フランク先生は約束した後でちょっと不安になった。誕生日が同じ子がいたら困るな.... 同じ日に 2 カ所には行けないよ.... でもフランク先生はこの不安を打ち消した。なあにクラスの子は 30 人だ。1 年は 365 日もあるのだ。ぶつかるなんてよほど運が悪くないと...

4.5.2 コンピュータシミュレーション

ここではこの問題をコンピュータシミュレーションによって調べてみよう。1 から 365 までの数字を乱数で 30 個発生させて誕生日の衝突が発生するか否かを調べるのである。

乱数の発生

まずは乱数の発生のさせ方を示す。

Python は乱数発生のモジュールをいくつか備えていて最初の行

```
from random import *
```

譜 4.9 乱数を使ったシミュレーション

```
from random import *
for i in range(0,30):
    x=int(365*random())+1
    print x
```

はそのうちの初等的な random モジュールを取り込むことを意味する。このモジュールの関数 random() は $0 \leq x < 1$ までの間の実数 x を片寄り無くランダムに取り出す[4]。従って 365*random() によって実数の区間は 0 から 365 にまで広がるが int によって少数以下が切り捨てられ整数になってしまうので int(365*random())+1 の範囲は 1 から 365 までの整数である。読者は実際にこのプログラムを実行し print x で表示される x の値が 1 から 365 までのランダムな値になっている事を確認するのが良い。

整列すれば見やすくなる

この 30 個のランダムな数字の中に同じ数字が含まれているか否かが問題である。30 個だからそのままでも何とか分かるが疲れる。もっと見つけやすくできないか? そう数字を小さい順に並べ替えれば良いのだ。並べ替えを行うにはリストを使うのが良い。次のプログラムがそれである。

譜 4.10 整列による譜 4.9 の改良版

```
from random import *
d=[]
for i in range(0,30):
    x=int(365*random())+1
    d.append(x)
print d
d.sort()
print d
```

このプログラムでは最初に空のリストを作り (d=[])、生成された乱数をそれに追加して行く (d.append(x))。d.sort() はリスト d を小さな順に並べ替える。2 つの print d があるが最初のは並べ替える前の結果を最後のは並べ替えた後のものを書き出す。最初のは単なる参考のためであり無くても構わない。

[4] これを区間 [0,1) の一様乱数と言う

重複した誕生日だけを書き出す

読者は誕生日を整列する (規則に従って並べ替える事) 事によって、同じ誕生日があるか否かが非常に分かりやすくなった事に気づくであろう。そう、隣り合った 2 つの数字に同じものがあるか否かを調べればよいのだ。そんなことは目で追わなくてもプログラムで簡単にやっていける。次に示すのは同じ誕生日があればそれを書き出すプログラムである。

譜 4.11 重なりだけを出力する譜 4.10 の更なる改良版

```
from random import *
d=[]
for i in range(0,30):
    x=int(365*random())+1
    d.append(x)
print d
d.sort()
print d
for i in range(0,29):
    if d[i]==d[i+1]: print d[i]
```

これを実行してみると誕生日の衝突が多い事に意外な感じを受けるのではなかろうか?

問 1 そこでフランク先生は誕生日の衝突の確率を理論的に見積もる事にしました。30 人の問題をいきなり扱うのは難しすぎるので、まずは 3 人から...

3 人で誕生日の衝突が発生しない確率 P は

$$P = \frac{365}{365} \times \frac{364}{365} \times \frac{363}{365}$$

で計算できます。これぐらいなら電卓でも計算できますが、30 人だとやっぱり計算機ですね。やってみましょう。また P が 5 割を切るのは何人でしょうね。

4.6 フェルマーの問題

ついに Fermat(フェルマー) の大定理が証明されたそうである。この定理は n が 3 以上の整数の時、

$$x^n + y^n = z^n$$

の自然数解 (x, y, z) は存在しない事を主張する[5]。Fermat[6]が残したメモの中にこの主張が語られている。Fermat は証明したと言うのだが、その証明は見つかっていない。従って「定理」

[5] $n = 2$ の場合には自然数解は存在する。例えば $(3, 4, 5)$ や $(1, 12, 13)$ などの組である。$n = 2$ の場合の解はピタゴラス数と呼ばれている。

[6] 1601-65、フランスの数学者

と呼ばずに「問題」とも呼ばれてきた。以来、多数の数学者がこの問題に挑戦してきた。その副産物として「整数論」と呼ばれる数学分野が大いに発展してきたのである。

ここでは Fermat の問題とよく似た問題を採り挙げよう。

例 11

$$x^3 + y^3 + z^3 = u^3$$

となる自然数の組を見つけよ。

計算機で様々な自然数の組 (x, y, z, u) を試すのだが、系統的に試す必要がある。効率よく捜すために

$$x \leq y \leq z$$

の条件を付けよう。その下で z について $1, 2, 3, \cdots$ と試していこう。そして、とりあえず 20 まで試すことにする。さらに u を試行錯誤で求めるような事はしないで、$x^3 + y^3 + z^3 = u^3$ を満たす u が存在するか否かは、$(x^3 + y^3 + z^3)^{1/3}$ を計算し、u の目星を付けて、その u を試すのが速い。そうすれば、次のようなプログラムができ上がる。

譜 4.12 整数解を求めるプログラム

```
for z in range(1,20):
    for y in range(1,z+1):
        for x in range(1,y+1):
            v = x**3 + y**3 + z**3
            u = int(v**(1.0/3)+0.5)
            if  u**3== v: print x,y,z,u
```

問 2

$$x^4 + y^4 + z^4 = u^4$$

となる自然数の組についてはどうだろうか?

4.7　再帰的アルゴリズム

ここでは再帰的アルゴリズムがどれほど有り難いかを示そう。具体的なテーマとして第 4.3 節に出て来た問題、すなわち、非常に大きな m に対して x^{m-1} を m で割った余りを計算する問題を取り扱う。例えば $x = 117$ に対して $m = 164445554499268607472268429 1497$ のようなケースである。

「簡単だね

```
x=117
m=164445554499268607472268429 1497
print x**(m-1) % m
```

を実行すればよいのだよ。」やってみるがよい。いつまでたっても君のパソコンは答えを出さないよ。

この方法が遅くなる原因は非常に大きな整数の計算に手間取っているからである。ならば

```
x=117
m=164445554499268607472268429 1497
y=1
for n in range(0,m-1):
    y=y*x %m
print y
```

でもこれは

```
Traceback (most recent call last):
  File "a.py", line 4, in ?
    for n in range(0,m-1):
OverflowError: range() result has too many items
```

と言って叱られる。range はメモリを食い、今のケースだとメモリに納まらなくなるのは目に見えている。メモリを食わない範囲指定として xrange(0,m-1) があるが、今度は

```
Traceback (most recent call last):
  File "a.py", line 4, in ?
    for n in xrange(0,m-1):
OverflowError: long int too large to convert to int
```

になる。こんなに大きな繰り返しは想定していないのである。では

```
x=117
m=164445554499268607472268429 1497
y=1
n=0
while n<m-1:
    y=y*x %m
    n=n+1
print y
```

はどうか? これはエラーにならないで実行してくれる。でもいつまでたっても答えは出てこないよ。発想が悪いのだ。計算機は素晴らしい計算能力を持っているけども、人間の知恵には及ばない事もある...

電卓を持っているとしたまえ。その下で 2^{16} の計算を考えてみよう。何回の掛け算が必要か? 15 回と答えると失格だ。次のようにたったの 4 回で計算できる。必要な計算量の差は m が大きい場合には凄まじい。

$$
\begin{aligned}
2^2 &\quad 2\times 2 \to 4\\
2^4 &\quad 4\times 4 \to 16\\
2^8 &\quad 16\times 16 \to 256\\
2^{16} &\quad 256\times 256 \to 65536
\end{aligned}
$$

こんなにうまく行くのは m が特殊な数だけか? そんなことはない。例えば $m = 17$ の場合には $2^{16} \times 2$ で計算できる。

この考え方は次のように表現できる。pow(x,n) が x^n を計算する関数である。

```
def pow(x,n):
    if n==0:
        return 1
    k=n/2
    y=pow(x,k)
    if n%2 == 0:
        return y*y
    return y*y*x
```

このプログラムでは x^n の計算は、$n = 2k$ で表せる場合には $x^n = (x^k)^2$ で、$n = 2k + 1$ で表せる場合には $x^n = x \cdot (x^k)^2$ で計算しなさいと語られている。そうすればベキ乗の計算を小さなベキ (約半分のベキ) の問題に還元できる。

大きな数の問題を小さな数の問題に還元する手法は素晴らしい効果を生む。最小公倍数を求めるプログラムもそうであった。ベキ乗の問題はもっと複雑であるが、それでも関数 pow の中に関数 pow を使う事によって、本当はもっと頭の中が混乱するような状況を上手に切り抜けている。関数 pow のように、その定義の内部に同じ関数を含む時に「再帰的関数」と言う[7]。再帰的に計算法が表現されている場合に「再帰的アルゴリズム」と言う。

Python の開発者たちはこのような旨いベキ乗の計算法があることはお見通しである。そして x**n の計算にはもう少し無駄の少ない計算法をとっているはずである。

譜 4.13 x^{m-1} を m で割った余りを計算するプログラム

```
def mpow(x,n,m):
    if n==0:
        return 1
    k=n/2
    y=mpow(x,k,m)
    if n%2 == 0:
        return y*y%m
    return y*y*x%m

m= 164445554499268607472268429149 7
x=117
print mpow(x,m-1,m)
```

これは一瞬の内に答えを出す。44822559907995903369197391 5232 だ。
つまり 164445554499268607472268429149 7 は素数ではない。

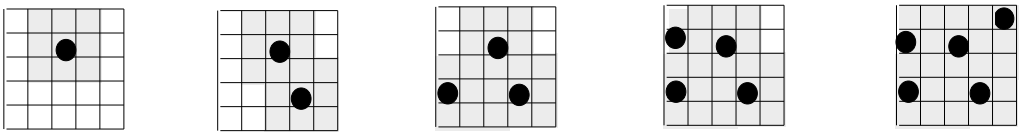
[7] ホア (Hoare) はこの考えを整列問題に適用し、クイックソート (quick sort) 法を編み出した。

実は、ここに述べた `mpow(x,n,m)` は Python で既にサポートされている。その使い方は単に `pow(x,n,m)` である。

4.8　置き石ゲーム

4.8.1　ゲームのルール

盤上に石を置いて争うゲームとしては囲碁が有名である。囲碁を採りあげるのは余りにも難しい問題が含まれているので、もっと単純なルールのゲームについて考察しよう。5 × 5 の升目の盤と碁石 (黒石だけでよい) を準備する[8]。2 人が交互に石を升目に置く。既に配置されている石の隣りには石は置けない。斜め隣もダメである。石を置ける場所はだんだん少なくなり、最後にはなくなる。石を置けなくなった方が負けである。ゲームの進行例を図に表す。



Bob　　Alice　　Bob　　Alice　　Bob

盤面には石を置けなくなった領域を影で示してある。盤の下に記されている Alice や Bob の名称は石を置いたプレーヤを表している。この例では Alice の負けである。

4.8.2　プログラムの考え方

ここでの課題はこのゲームに勝つ為のプログラムを作成する事である。その為に関数 next(d) を定義するのが目標である。この関数は盤面の情報 d を基に次の手を教えてくれる。盤面の情報としては石を置く事が許されている升目の集合を与えればよいであろう。ここではリストを集合の代わりに用いる事にしよう。最初に、

```
m=5
d=[]
for i in range(0,m):
    for j in range(0,m):
        d.append((i,j))
```

で升目の座標が [(0,0),(0,1),...,(4,4)] のようにリスト d の要素となる。盤面 d の升目 p に石を置くと、p の周囲の座標を除去する必要がある。q を p の周囲の座標の 1 つとすれば d.remove(q) で d から q が除去される。だが、除去する前に d の中に q が存在する事を確認しなくてはならない。従って、d に p を置いた後の盤面は次の様に計算される。

```
for i in -1,0,1:
    for j in -1,0,1:
        q=(p[0]+i,p[1]+j)
        if d.count(q): d.remove(q)
```

[8] ここでは盤の升目を 5 × 5 としたが、これは簡単のためである。一般に升目の大きさは任意で構わないし、形ですら任意で構わない。

最後に関数 next(d) の定義が残された。d に要素が存在しない時は次の手は存在しない。この場合は None を返す事にする。d に要素が存在する場合にはどのように次の手を見つけたらよいか? 次のの例え話が参考になるであろう。

> Frank は天才的な頭脳を持っており、一目で勝敗を読み取る。Alice は Frank と勝負をすることとなった。 Alice は先を読む能力が全く無いのだ。でも Alice は勝つ自身があった。Frank の弱点を知っていたからである。Frank はとても正直者で、感情が顔に現われるのである。
>
> ゲームが始まった。Alice が先に石を置く事になった。「ここがいいかしら…」とぶつぶつ独り言を言い、石を置くふりをしながら Alice は Frank の顔を見た。Frank がニコニコしたら、Alice は「じゃー、ここかな…」と別の場所に石を置くふりをした。Frank の顔が青ざめたのを Alice は見逃さなかった。…

Python では next(d) を定義するのに、盤面がより狭い時の next(d) を利用する事ができる。関数の再帰的な定義法である。再帰的な定義法が許されているので、Alice が行ったように 1 つ 1 つ石を置いてみて、Frank の反応を見るやり方を適用できる。以下にプログラムを載せる。

譜 4.14 置き石ゲームの次の手を見つけるプログラム

```
from copy import *
def put(d,p):
    for i in -1,0,1:
        for j in -1,0,1:
            q=(p[0]+i,p[1]+j)
            if d.count(q): d.remove(q)
def pr(d):
    for p in d: print p,
    print
def next(d):
    if len(d)==0: return None # No space to put a stone. I've lost.
    pr(d)
    for p in d:
        print p,":",
        e=copy(d)   # make a copy of d to protect from modification.
        put(e,p)              # put a stone
        if next(e)==None:     # and glance at Frank
            return p          # he looks pale. I will win.
    return None    # Frank is smiling for my all trials. He will win.

# initialize
m=5
d=[]
for i in range(0,m):
    for j in range(0,m):
        d.append((i,j))

print "RESULT:", next(d)
```

4.9 Boyce の壷の問題

4.9.1 問題の提起

ここに 1 つの壷と碁石がある。碁石は白が 4 個と黒が 6 個である。
「この碁石を壷に入れるよ。君が壷の中から白石を取り出せば君は 100 円もらえる。黒石だったら逆に君は 100 円払うんだよ。取った石は壷には戻さないで、君は壷の中の石がなくなるまでこのゲームを続ける事ができる。途中でやめてもよいし、最後まで続けてもよい。どこで止めるかは君次第だよ…」 で、君はこのゲームに参加するかね。

黒が白より多いので、分が悪そうだ。やめておこうか…

でもよく考えて見たまえ。白と黒が同じ場合に公平なゲームになっていると言うのは間違っているではないか? 白が 1 個、黒が 1 個の場合を考えてごらん。この場合には絶対に損はしないのだよ。最初に白が出れば、100 円もらって止めればよい。黒が出れば残りの 1 個をさらに取り出すさ。それは白に決まっているから。

このように考えると、黒の方が多い場合にも、このゲームに参加する価値がありそうだと言うことが分る。白石と黒石の個数を既知として、このゲームに上手に参加した時に期待される利益を計算する問題を「 Boyce の壷の問題」と呼ばれている。

4.9.2 プログラムの考え方

この問題は石の個数が少ない場合には以下のような考察で解決できる。以下で「期待利益」とは、もらえる 100 円硬貨の期待個数である。例えば期待利益が 1.33 の場合には、平均して 133 円もらえる事を意味する。黒石 m 個、白石 p 個の期待利益を $V(m,p)$ で表す。黒石 1 個、白石 0 個の場合: 石を取る価値がない。従って、$V(1,0) = 0$ 黒石 0 個、白石 1 個の場合: 石を取る価値がある。従って、$V(0,1) = 1$ 黒石 1 個、白石 1 個の場合: 最初に取った石が黒であれば残りは白石 1 個である。白であれば、残りは黒石 1 個である。いずれの場合にも、この後の判断は石が 1 個の問題に還元できる。従って、もしも最初に黒石を取り出せば $-1 + V(0,1)$ の利益、最初に白石を取り出せは $+1 + V(1,0)$ の利益であると考える事ができる。最初に黒石を取り出す確率は 1/2、白石を取り出す確率は 1/2 であるから、黒石 1 個白石 1 個の壷から石を取る選択をする場合の期待利益は

$$\frac{1}{2}(-1 + V(0,1)) + \frac{1}{2}(+1 + V(1,0)) = 0.5$$

となる。この値は正であるから石を取り出す価値がある。つまり $V(1,1) = 0.5$ である。この例で分るように $V(m,p)$ の計算は $V(m-1,p)$ と $V(m,p-1)$ の計算に還元でき、この計算式は

$$\begin{aligned} V(m,p) &= \max\left(0, \frac{m}{m+p}(-1 + V(m-1,p)) + \frac{p}{m+p}(+1 + V(m,p-1))\right) \\ V(0,p) &= p \\ V(m,0) &= 0 \end{aligned}$$

である。実際に $V(m,p)$ を求めるプログラムを作成しよう。まずは最初に問題の関数をそのまま表そう。

譜 4.15 Boyce の壷の問題を解くプログラム

```
def V(m,p):
    if m==0: return p
    if p==0: return 0
    return max(0,
        float(m)*(-1+V(m-1,p))/(m+p)+float(p)*(1+V(m,p-1))/(m+p))
for p in range(0,10):
    print p,
    for m in range(0,10): print "%5.2f"%V(m,p),
    print
```

このプログラムでは関数 $V(m,p)$ の定義の中に $V(m-1,p)$ と $V(m,p-1)$ が使用されている。関数の定義の中に、当の関数名が現われるような定義の方法を再帰的な定義と言う。問題の分析が再帰的に行われた為に関数の定義もこのようになったのである。これを実行すると $V(m,p)$ の表

```
0  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00
1  1.00  0.50  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00
2  2.00  1.33  0.67  0.20  0.00  0.00  0.00  0.00  0.00  0.00
3  3.00  2.25  1.50  0.85  0.34  0.00  0.00  0.00  0.00  0.00
4  4.00  3.20  2.40  1.66  1.00  0.44  0.07  0.00  0.00  0.00
5  5.00  4.17  3.33  2.54  1.79  1.12  0.55  0.15  0.00  0.00
6  6.00  5.14  4.29  3.45  2.66  1.91  1.23  0.66  0.23  0.00
7  7.00  6.12  5.25  4.39  3.56  2.76  2.01  1.34  0.75  0.30
8  8.00  7.11  6.22  5.35  4.49  3.66  2.86  2.11  1.43  0.84
9  9.00  8.10  7.20  6.31  5.43  4.58  3.75  2.95  2.21  1.52
```

が得られる。

左端の列の数字は p を表している。その右の列は m=0 の列である。そして順に m=1, m=2 の列が続く。従って、例えば 3 行目、3 列目の 1.33 は $V(2,1)$ の値である。この表から $V(6,4)$ は 0.07 であることが分る。つまり黒石 6 個、白石 4 個でもゲームに参加する価値があるのである。

4.9.3　譜 4.15 の改良

所でこのプログラムはひどく遅いのである。理由はひどく無駄な計算をしているからである。例えば $V(3,5)$ の計算をする時には $V(2,5)$ の計算が行われている。$V(2,6)$ の計算を行う時にも $V(2,5)$ の計算が行われている。すでに計算済の結果を使用しようとはしていないのである。

一度行った $V(m,p)$ の計算を記憶し、後で必要になったらその結果を利用するには Python では辞書を使用するのが簡単である。

```
u={}
```

で空の辞書 d が生成される。辞書は文字列だけではなくタプルを検索キーにする事ができる。従って V(m,p) の値を辞書に登録するには u[m,p] にこの値を代入すればよい。また辞書にキー (m,p) が登録されていれば (m,p) in u が True になる。従って次のプログラム (譜 4.16) が得られる。

譜 4.16 譜 4.15 の改良版

```
u={}
def V(m,p):
    if m==0: return p
    if p==0: return 0
    if (m,p) in u : return u[m,p]
    r=max(0,
        float(m)*(-1+V(m-1,p))/(m+p)+float(p)*(1+V(m,p-1))/(m+p))
    u[m,p]=r
    return r
for p in range(0,10):
    print p,
    for m in range(0,10): print "%5.2f"%V(m,p),
    print
```

このプログラムと先のプログラムの実行速度を比較してみるがよい。著しく速度が改善された事が実感できるであろう。

プログラムを作る時には、まずは堅実なプログラムを作成するがよい。誤りがないと言う事が最も大切である。実行速度を上げるためにプログラムを工夫すればするほど誤りが持ち込まれる可能性が高くなる。工夫したプログラムは堅実なプログラムと実行結果を比較する事によって誤りを回避する事が可能である。

第II部

Python 文法

第1章　定数

1.1　定数の分類

1.1.1　基本概念

データと定数　データとは計算機で操作の対象となるもの全てである。特定のデータをプログラムの中で表現したものを定数と言う。以下に定数の例を挙げる。

```
256            1.23           1.23E5                  "alice"
(2,3,"alice")  [2,3,"alice"]  {'bob':20, 'alice':16}
```

1 行目の定数は順に、整数 256 、実数 1.23、実数 123000、文字列 alice を表現している。2 行目の定数はいずれも 1 行目に挙げた種類の定数を寄せ集めて 1 個の定数を形成している。これらは各々、タプル、リスト、辞書と呼ばれる。プログラムの中で表現できる定数の種類と表現法はプログラミング言語毎に異なる。ここに挙げたのは Python が扱う定数である。

プログラムが計算機によって実行されると定数は評価される。即ち、計算機が扱いやすい表現に変換される。評価された結果を定数の値と言う。値は現実の計算機の能力による様々な制約を受ける。

データの型　データは幾つかの種類に分類される。Python ではデータを整数、実数、複素数、文字列、タプル、リスト、辞書、.... 等に分類している。これをデータの型と言う。プログラミング言語がデータを分類する理由は 2 つ挙げられる:
(1) 同じ型に属するデータは演算に対して共通の性質を持つ
(2) データの型により計算機が扱いやすい表現形式が異なる

先に挙げた定数の例の中で `1.23E5` を整数ではなく実数であると説明したのは (1) と (2) の理由による。数学の世界では整数は実数の一部であると考える。しかし計算機の世界では整数と実数を区別する。例えば `1` は整数であり、実数ではないと考え、実数としての 1 は `1.0` であると考える。このように区別するのは論理的な合理性と言うよりも計算機側の事情なのである。

数字と文字列　数字と文字列を区別する必要がある。例えば電話番号 809-1234 を考えよう。読者はここに現われる 1234 を数字だと考えるだろうか? 数字だと考える読者でも、ここに現われる '-' を引き算の記号だとは考えないであろう。では 1234 は何か? 数字だけで構成されてはいるが数としての意味は持っていない。本質的には、単に電話を区別するための、文字の羅列に過ぎないのであり、プログラミングにおいては文字列として扱うべきデータなのである。

名前　データに名前を付ける事が可能である。例えば

```
a=(2,3,"alice")
```

によってデータ `(2,3,"alice")` に対して名前 `a` が付けられる。

Python では如何なるデータに対しても名前を付ける事ができる。関数もデータの一種であり、既に名前を持っている。例えば絶対値を求める関数 `abs(x)` に対して

```
f=abs
```

で `abs` に別名 `f` を付ける事ができる。この下で `f(x)` は絶対値を求める関数として機能するのである。

シーケンス　文字列、タプル、リストをシーケンスと言う。シーケンスとはデータを並べたものを指す。従ってシーケンスには最初の要素とか最後の要素とか 5 番目の要素とか、あるいは要素の個数とかと言った共通の概念が存在するのである。Python ではシーケンスの持っているこうした特徴を統一した方法で扱う。要素は番号で指定される。そして番号は 0 から始まる。例えば `a` を `(2,3,"alice")` に付けられた名前とする時、`a[0]` は `2` を、`a[1]` は 3 を `a[2]` は `"alice"` を意味する。そして `a[2]` はシーケンスだから `a[2][0]` は ‘a’ を a[2][1] は ‘l’ を a[2][2] は ‘i’ を意味する。

可変と不変　データが可変であるとは、その要素が変更可能である事を言う。他方、不変であるとはその要素が変更不能である事を言う。リストや辞書は可変であるが、文字列やタプルは不変である。データの構成要素の変更法をリストを例にとって示そう。

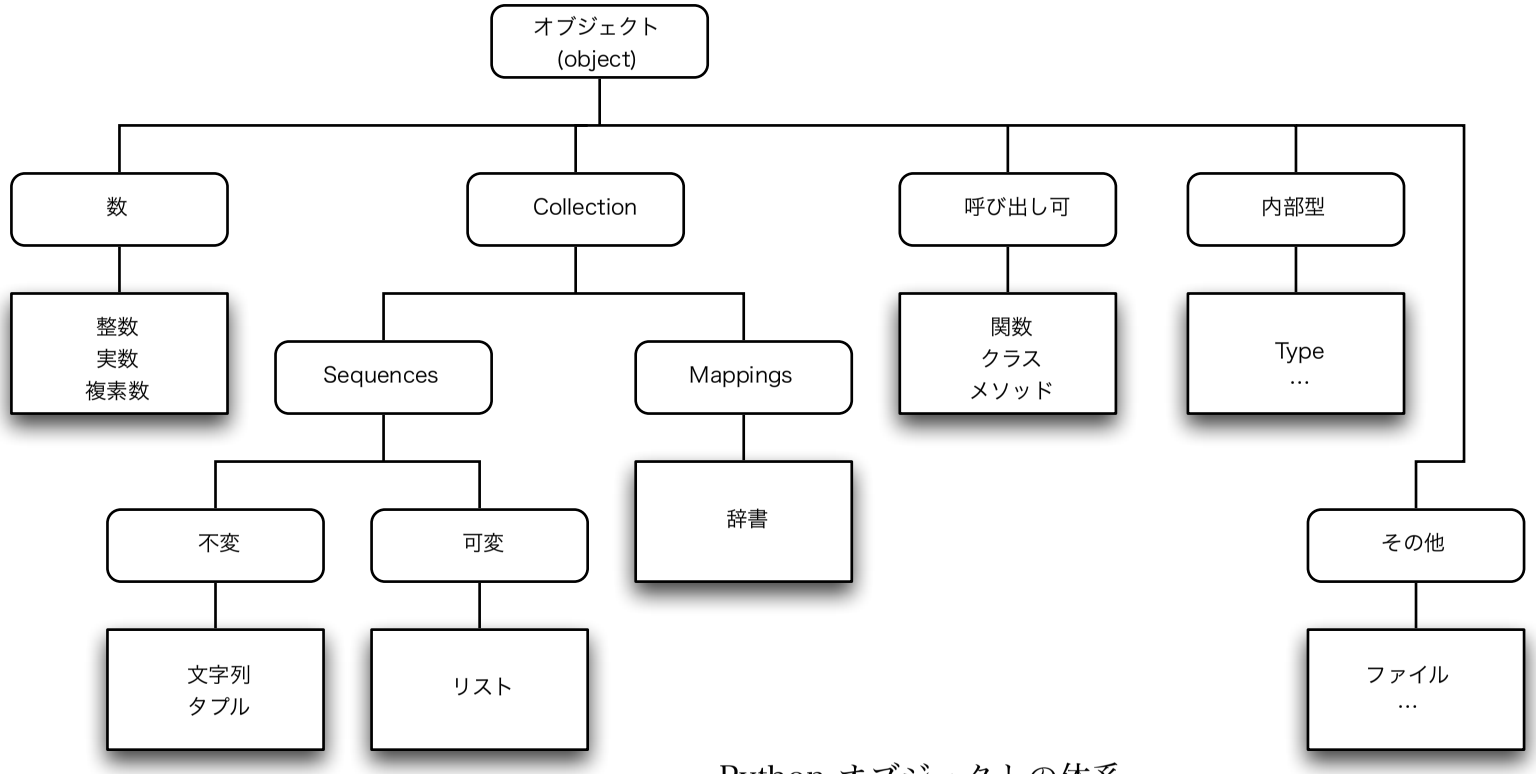
```
b=[2,3,"alice"]
b[2]="bob"
```

すると `b` は `[2,3,"bob"]` となる。

辞書　辞書の要素は番号だけではなく文字列、タプルなど様々なデータによって指定される。こうしたデータをキーと言う。リストや辞書のように可変なデータはキーにはなれない。タプルであっても `(2,3,[4,5])` のように、可変なデータを構成要素に含む場合にはキーにはなれない。

1.1.2　データの体系

次のページに Python が扱うデータの体系を載せる。ここではデータの事をオブジェクトと呼んでいる。この呼称にはデータと演算は一体のものであると言う思想が含まれている。Python はこの思想の下で一貫して設計されたプログラミング言語である。(そのために Python は素直で理解しやすい言語になっている。)



Python オブジェクトの体系

注釈
Collection: 寄せ集め。数学概念としての集合には和、差、積(共通部分)などの演算が定義されているが、
Python の Collection にはこうした概念は無い。
全てのオブジェクトが図示されてはいない。
このテキストでは整数、実数、複素数、文字列、タプル、リスト、辞書、関数、ファイルが解説されている。

1.2　数

Python では数の体系として、整数、実数、複素数の 3 つを扱うことができる[1]。数の間には次の算術演算が定義されている。

```
x + y, x - y, x * y, x / y, x % y, -x, +x
```

ここに * は乗算の意味である。また / は除算の商、% は除算の剰余を意味する。除算の剰余の意味に関しては第 6 章 3 節算術式を参照せよ。

整数　Python はメモリの許す限りの大きな整数を計算する。許される整数はとてつもなく大きい。

実数　例: `1.23`　　`3.12E-5`
`E-5` は $\times 10^{-5}$ を意味している。従って `3.12E-5` は `0.0000312` と同じである。`E` は小文字で書いてもよい。計算機においては実数は近似値に過ぎない。Python の実数は 10 進数 16 桁の精度を持つ。そのため、例えば `1.0/3` の計算結果を詳細に表示すると 0.3333333333333331 となる。非常に絶対値の大きな実数は 10^{308} まで、逆に非常に 0 に近い実数は 10^{-323} までが許される。

複素数　虚数単位は `j` である。単なる `j` は変数名と区別がつかない。そこで `j` の前には必ず数字を付ける必要がある。例えば `(1+j)` とは書かず `(1+1j)` と書く。
`z` を複素数とすると

```
z.real
z.imag
```

が `z` の実数部と虚数部であり、

```
z.conjugate()
```

で複素共役が得られる。

1.3　文字列 (string)

Python が扱う文字列は 1 バイト単位の任意データの任意の長さの並びである。文字列に含まれるデータは必ずしも我々が日常生活で言う「文字」である必要はないが、プログラムの中で文字列定数を使用する時は我々の日常の常識に従った (すなわち印字可能な) 短い文字の列である場合が殆どである。文字列は 1 行で表現できる場合もあるし、プログラムの数行を要求される場合もある。

[1] 実は Python では数のクラス (体系) を定義できて、それによって演算の意味を変更できる。その意味では Python が扱う数のクラスは 3 つだけではない。しかしこのテキストではクラスに関する解説は割愛する。

1.3.1 文字列定数

プログラムの行を跨がない文字列定数は二重引用符 (") あるいは単一引用符 (') で囲って表現する。例えば "alice" または 'alice' のように表現する。日常生活で文字列を表現する方法と似てはいるが重要な相違点がある[2]。プログラムの中で文字列定数を表現するために使用する引用符は始まりの引用符と終わりの引用符が同一である

改行の処理 文字列の中に改行が含まれる場合にはどのように表すのか? この場合には次の様に 3 個の引用符 (単一引用符、または二重引用符) を並べて表現する。

```
'''All in the golden afternoon
Full leisurely we glide;
For both our oars, with little skill,
By little arms are plied,
While little hands make vain pretence
Our wanderings to guide.'''
```

すると例えば afternoon の次にはちゃんと改行コードが含まれているので、この文字列を出力すれば、ここに書いた通り afternoon の直後で改行してくれる。長い長い、改行を含まない文字列はどのように表現するか? 次の様に行末にバックスラッシュ記号 (\) を入れると行末記号を取り消してくれる。

```
'''All in the golden afternoon\
Full leisurely we glide;\
For both our oars, with little skill,\
By little arms are plied,\
While little hands make vain pretence\
Our wanderings to guide.'''
```

即ちこの文字列の内部には行末記号が含まれない。出力するとあたかも長い長い 1 行の文字列であるかのように見えるであろう。

特殊文字の表現 行末にない \ 記号は特殊な文字を文字列の中に埋め込む時に使用される。

Python では 2 種類の引用符が使えるので

I don't know.

を print 文で表示するのに、

```
print "I don't know."
```

と書いても

```
print 'I don\'t know.'
```

と書いても構わない。(が、"I don't know." の方が分かりやすいであろう。)

Python の文字列にはこの他に引用符の前に R や U を付ける (r や u でもよい) 表現法があるが解説は省略する[3]。

[2] 正しい英文は始まりの引用符と終りの引用符が異なる。例えば 'alice' あるいは "alice" と書くが、プログラミングの世界ではこの書き方をしない。

[3] 前置文字 R は大ざっぱに言えばバックスラッシュ記号を特殊文字として扱わないためのものであるが、実際に

意味	文字列中での表現	意味	文字列中での表現
'	\'	Null	\0
"	\"	Vertical tab	\v
\	\\	Carriage return	\r
TAB	\t	Formfeed	\f
改行	\n	Escape	\e
Bell	\a	8 進コード	\0XX
Backspace	\b	16 進コード	\xXX

表 1.1: 特殊文字の表現

1.3.2 文字列に対する基本演算

文字列の長さ 文字列 s の長さとは文字列に含まれる文字の個数である。文字列 s の長さは関数 len(s) で与えられる。例えば s は 'alice' であるとせよ。その下で len(s) は 5 である。

部分文字列 文字列 s の n 番目の文字は s[n] で表す。n をインデックスと言う。n は 0 から始まる。従って s[0] は 'a' であり、 s[2] は 'i' である。 s[n:] は s[n] 以降の s の文字列を意味する。従って s[2:] は "ice" である。s[:m] は s[0] からの m 個の並びである。従って s[:4] は "alic" である。 s[n:m] は s[n:] と s[:m] の共通部分を意味する。従って s[2:4] は "ic" である。文字列から部分文字列を切り出す演算をスライス (slice) と言う。

[ ] の中の整数は負の数も許されている。その場合には文字位置が後ろから指定されたと解釈される。-1 が末尾の文字位置である。以下に例を示す。

```
s[-2]   'c'          s[-4:]    'lice'
s[:-2]  'ali'        s[-4:-2]  'li'
```

文字列の演算 文字列には + の演算と * の演算と % の演算が許される。例で示す。
'alice' + 'bob' は 'alicebob' を、
'alice'*3 は 'alice' + 'alice' + 'alice' 即ち 'alicealicealice' を、
'alice=%d'%x は x の値に応じて、例えば 18 の時には 'alice=18' を
を生成する。文字列に対する乗算は加算の帰結として定義されているので非負整数との乗算のみが定義されている。従って例えば文字列同士の乗算は定義されていない。最後の例、
'alice=%d'%x
では文字列 'alice=%d' に対して x が演算子 % によって作用しているのである。演算子 % の左の文字列 (この場合 'alice=%d') を書式指定文字列と言う。書式指定文字列の中に現われ

は正規表現と呼ばれる文字列パターンを扱いやすいように設計されている。U は文字列を Unicode 文字列として扱う場合に使用される。(Python では日本語の漢字などは Unicode として扱う。しかし単に文字列の中に日本語文字を書いてもうまく動作しないであろう。日本人の使用する文字コードの体系がたいていの場合 Unicode ではないからである。日本人は長らく日本の独自の文字コード体系の中でコンピュータ文化を発展させてきたために世界標準である Unicode への移行は容易ではない。)

る文字 % が埋め込みの場所を表している。書式指定に関しては、詳しくは付録の「書式」を見よ。

1.3.3 文字列の順序関係

文字列相互の順序関係が定義されている。文字列の順序関係は所謂「辞書式順序」に従う。辞書式順序とは文字の大小関係に基づいて、文字列を先頭の文字から順に比較し、最初に「小さな」文字が現われた方を小さい文字列であると考える順序付けである。もしもこの比較の中で一方が先に比較する文字が無くなったら、無くなった方を小さいと考える。

例えば "Alice" と "Anna" は 2 番目の文字が異なる。そして 'l' は 'n' よりも「小さい」と定義されている。それ故 "Alice" は "Anna" よりも「小さい」文字列である。他方 "An" と "Anna" では "An" が小さい。

文字コード 計算機で定義される文字の大小関係は文字コード表から知ることができる。文字コード表は文字に数字を対応させ、数字の大小関係によって文字の大小関係を定義する。表 1.2 にアメリカ合衆国の規格である ASCII 文字コード表を載せる。この規格は日本の JIS 規格にも採用されており、パソコンなどもこの規格を使用している。 この表によると、例え

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
20		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
30	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
40	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
50	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
60	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
70	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	

表 1.2: ASCII 文字コード表

ば文字 'A' の文字コードは、'A' を含む行の左端の 40 と、'A' を含む列の上端の 01 を加えて得られる。この場合は 41 になる。これは文字コードを 16 進数で表した数字である。

この表では 00 から 1F までの 32 個の「文字」は省略されている。これらの文字は形を持たないので印刷できないのである。これらは普通の意味での文字ではなく、制御文字と呼ばれている。なお空白は 20 である。7F は空白ではなく、DEL コードと呼ばれる制御文字である。

ASCII 文字コード表によって定義される文字の大小関係は、日常生活の辞書で使用されている文字の大小関係とは異なるので注意が必要である。文字コード表では英字の大文字は小文字よりも小さい。また 数字文字は英字より小さい。即ち、

'0' < '1' < ⋯ < '9' < 'A' < 'B' < ⋯ < 'Z' < 'a' < 'b' < ... < 'z'

の関係が成立している。

文字コード関数 Python では、文字 c を与えて文字コードを返す関数 ord(c) と、逆に文字コード n を与えて文字を返す関数 chr(n) が使える。参考のために、以下に表 1.2 を作成したプログラムを載せる。このプログラムでは関数 chr(n) が使用されている。

```
print "  ",
for n in range(0,16): print "%02X"%n,
for n in range(32,128):
    if n%16 ==0: print; print "%X"%n,
    print "%2s"%chr(n),
```

1.3.4 文字列に対する基本操作

Python では文字列はオブジェクトとして扱われる。この場合、文字列に対する操作の方法は「メソッド」と言われ、次のように書かれる。

```
s='alice'
print s.upper()      # ALICE を出力
print s.find('ic')   # 2 を出力
print s.find('iq')   # -1 を出力
```

すなわち操作の対象となる変数名 (あるいはデータ) に続いてピリオドを書いて、そのあとにメソッド名を書く[4]。

文字列の分解 文字列の中で与えたデータを使用して計算を行うにはリスト形式に変換するのがよい。以下に文字列

```
a='   alice=18   bob=20   '
```

を例に採って、a からリスト

```
['alice', '18']
```

がどのような操作で生成されていくかを示す。

```
a='   alice=18   bob=20   '   # 両端に余分な空白がある
b=a.strip()
print b    # alice=18   bob=20 を出力。両端の空白はない
c=b.split()
print c    # ['alice=18', 'bob=20'] を出力
x=c[0]
print x    # alice=18 を出力
d=x.split('=')
print d    # ['alice', '18'] を出力
```

分解と結合 文字列から、その中の 1 つ 1 つの文字を要素とするタプルを生成するには関数 tuple を使用する。またリストを生成するには関数 list を使用する。例を示す。# 記号の右が出力結果である。

```
a="alice"
print tuple(a) # ('a', 'l', 'i', 'c', 'e')
print list(a)  # ['a', 'l', 'i', 'c', 'e']
```

逆にタプルやリストの要素を結合して 1 個の文字列とするには join メソッドを使用する。以下に使い方を例で示す。(# 記号の右が出力結果である。)

[4]関数を用いた古い書き方も Python2 ではサポートされているが、その多くは Python3 では廃止されている。

```
a="alice"
b="bob"
print " ".join((a,b))       # alice bob
print "-".join((a,b))       # alice-bob
print "".join((a,b))        # alicebob
```

つまり、結合したいタプル (あるいはリスト) を t、それらを繋ぐ文字列を s として、

```
s.join(t)
```

で表す。

1.3.5 日本語の扱い

テキストエンコーディング

文字列の中に日本の文字を使う場合にはテキストのエンコーディング[5]を指定する必要がある。例えば次のように、プログラムの第一行目または第二行目に、コメントのようにして使用されている文字コードの種別を与える。

```
# coding: SHIFT-JIS
print "ありさわ"
```

この例は SHIFT-JIS の場合である。SHIFT-JIS は日本語 Windows の標準文字コードであるが、世界的には通用しない。世界的には世界中の文字を扱えるユニコード (Unicode) が標準文字コードであり日本でも普及しつつある。 UTF-8 はテキストデータや通信データにおけるユニコードの表現様式である。

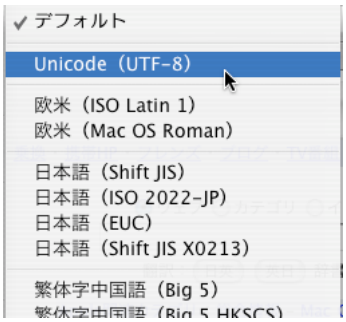


図 1.1: テキストエンコーディング

これは MacOSX の標準ウェブブラウザ Safari のテキストエンコーディングメニューである。様々な文字コードがサポートされているがユニコードが先頭に見える。ウェブの世界もユニコードに移行しつつある。

[5]世間では単に文字コードと呼ばれる事が多いが、他方では「文字コード」は文字を表す数字自体を意味するので混乱する。「テキストのエンコーディング」はテキストに現れる文字のコード化の規則であることを強調している。

実験

次の実験をしてみよう。(第一行の文字コードの指定は読者の環境に合わせて変更する事)

```
# coding: shift-jis
a="ありさわ"
print a, len(a)
b=u"ありさわ"
print b, len(b)
```

これを実行すると SHIFT-JIS の環境では

ありさわ 8
ありさわ 4

が出力される[6]。len(a) は変数 a の文字数、len(b) は変数 b の文字数の文字数を表している。

a="ありさわ"

は文字列データを 8 ビットの容器の列に入れたのに対して、

b=u"ありさわ"

はユニコードのサイズに合わせて 16 ビットの容器の列に入れたのである。(それによって正しい文字数 4 が得られた。) この違いは " の前の u に由来する (大文字の U でも可)。これはこの文字列をユニコードとして扱う事を指示しているのである。この例から、日本語文字を含む文字列の長さの計算や部分文字列などを計算する場合にはユニコードの指定が必要であることが分かる[7]。

コード変換

ASCII 文字列 (英文の文字列) s をユニコード文字列 u に変換するには
u=unicode(s)
逆にユニコード文字列 u を ASCII 文字列 s に変換するには
s=str(u)
とすればよい[8]。

コード変換の実際上のニーズは非 ASCII 文字列にある。例えば SHIFT-JIS で書かれた日本語の文字を含む文字列を読み取りユニコード文字列に変換するようなケースである。その場合には codecs モジュールを使用する。次のプログラムは codecs モジュールの代表的な関数である getdecoder と getencoder の使い方を示している。

[6] UTF-8 環境では平仮名は 3 バイトでコード化される。従って len(a) の値は 12 になる。
[7] 文字列の扱いは Python3 で根本的な変更を受けている。Python3 では文字列は全てユニコード文字列である。従って u を " の前に付けない。
[8] コード変換の方法も Python3 では異なる。

```
# coding: shift-jis
import codecs
s="ありさわ"
f=codecs.getdecoder("shift-jis")
print f(s) # f(s) is ("ありさわ", 8)

u=u"ありさわ"
f=codecs.getencoder("shift-jis")
print f(u) # f(u) is (u"ありさわ", 4)
```

1.4 タプル (tuple)

(6201,11349) の様に ( ) で囲まれ、コンマで区切られたデータの集まりをタプル (tuple) と言う。タプルは、('alice', 18) の様に異なった型のデータを要素として混在できる。任意の型のデータを要素にできるのである。タプルもまたタプルの要素となる。('alice', 18, ('bob', 3)) などである。() は空のタプルである。しかし (2) は数字の 2 でありタプルではない。要素数が 1 個、この場合には 2 だけのタプルは (2,) と書く。

1.4.1 タプルの基本演算

タプルの要素の個数 タプル t の要素の個数は len(t) で知ることができる。例えば

```
t=('alice', 18, ('bob', 3))
```

であれば len(t) は 3 である。

タプルの加算 タプルとタプルの加算が以下の例で示す様に定義されている。即ち:
(3,5)+(4,6,7) は (3,5,4,6,7) を生成する。

タプルへの代入 タプルを変数 a に代入するには

```
a=(2, 'alice', (abs, -5))
```

のようにする。この場合、

- a[0] は整数 2
- a[1] は文字列 'alice'
- a[2] はタプル (abs, -5)
- a[2][0] は関数 abs
- a[2][1] は整数 -5

である。関数もタプルの要素となる。従って、a[2][0](-7) は abs(-7) と同じであり 7 を与える。

以下の様にタプルを利用して複数の変数へ一度に代入できる。

```
(x,y)=(2,3)
```

この場合、x に 2 が、y に 3 が代入される。タプルは曖昧さが発生しない限り ( ) を外して使用できるので、これを次の様に書くこともできる:

```
x,y=2,3
```

記号 [ ] 記号 [ ] の使い方は基本的に文字列と同じである。特に [ ] の中に範囲を指定する記号 ':' を含める事ができる。具体的には第 3 節の文字列に関する解説を参照せよ。

タプルの要素は変更できない。例えば

```
a[0]=3
```

などのようにタプルの要素への代入はできない。もしも代入が必要な場合には後述のリストを使用する。

タプル相互の大小関係 タプル相互の大小関係が定義されている。タプルの大小関係は所謂「辞書式順序」に従う。辞書式順序とはタプルの要素の大小関係に基づいて、タプルを先頭の要素から順に比較し、最初に「小さな」要素が現われた方を小さいタプルであると考える順序付けである。もしもこの比較の中で一方が先に比較する要素が無くなったら、無くなった方を小さいと考える。(考え方は文字列の大小比較と基本的に同じである。)

1.4.2 任意個の引数を持つ関数

タプルの重要な応用は任意個の引数を持つ関数の定義である。関数定義文の中の * 記号に続く変数はタプルであり、関数を呼びだす時にはタプルで指定された位置には任意個の引数を書く事が可能である。次にその事を示す簡単な例を挙げる。

```
def f(a,*t):
    print t
f(1)        # () を出力
f(1,3,7)    # (3, 7) を出力
```

譜 1.1 に任意個の整数間の最大公約数を求める関数 gcd がどのように定義されるかを挙げる。

譜 1.1 最大公約数を求めるプログラム

```
#
# greatest common devision
# usage: gcd(x,y,z,...)
#
def gcd(x,*t):
    x=abs(x)
    if len(t) == 0:
        return x
    if len(t) == 1:
        y=abs(t[0])
        while y:
            z = x % y
            x,y = y,z
        return x
    else:
        u = gcd(x,t[0])
        for z in t[1:]:
            u = gcd(u,z)
        return u
print gcd(123)          # 123 を出力
print gcd(12857,21229)  # 299 を出力
print gcd(299, 27761)   #  23 を出力
print gcd(12857,21229,27761) #  23 を出力
```

1.5 リスト (list)

リスト (list) はタプルと似ているがタプルよりも柔軟性を備えた配列である。タプルで可能な事はリストでも可能である。リストは [3,7] の様に [ ] で囲み、要素をコンマで区切る。[] は空のリストを表す。

1.5.1 リストの演算

リストの要素の個数 リスト a の要素の個数はタプルと同様に 関数 len(a) で知る事ができる。

リストへの代入 代入は

```
a=[3,7]
```

のように行い、要素の参照はタプルと同様に a[0], a[1] の様に変数名の後に [ ] を付け、 [ ] の中に要素番号を書く。今の場合、a[0] は 3、a[1] は 7 である。

リストの要素への代入 リストはタプルと異なり、要素への代入が可能である。例えば

```
a[0] = 5
```

を実行できる。すると a の内容は [5,7] となる。

タプルと同様に要素のデータ型は混在していてもよい

```
a=[2,'alice',[abs,-5]]
```

すると、

a[0] は 2
a[1] は 'alice'
a[2] は [abs,-5]
a[2][0] は abs
a[2][1] は -5

である。関数もリストの要素となる。 従って a[2][0](-7) は abs(-7) と同じであり 7 を与える。

リストとリストの加算 リストとリストの加算が以下の例で示す様に定義されている:
[3,5]+[4,6,7] は [3,5,4,6,7] を生成する。
従ってリストへの追加は

```
a = a + [2,8,1]
```

のように + 記号を使えばよい。

append 関数 要素を 1 個ずつ追加する場合には

```
a = a + [2]
```

のようにしないで

```
a.append(2)
```

のように append を使ってもよい[9]。

a + [2] と a.append(2) の 2 つは似ているが行われている事は異なっている。+ 記号を使用した場合には新たなリストを生成するが、append は新たなリストを生成しない。従って

```
a=[3,7]
b=a
a=a+[2]
```

[9]変数 a の後にピリオド (.) を付けて、関数名を書くのはオブジェクト指向プログラミングのスタイルである。

を実行すると

b は [3,7]

a は [3,7,2]

となる。他方

```
a=[3,7]
b=a
a.append(2)
```

を実行した場合には a も b も [3,7,2] である。append は新たなリストを生成しない分だけ実行効率が良い。

リストの途中への要素の追加/削除 リストの途中に要素の並びを追加することもできる。例えばリスト a=[3,7,2,5] に対して

```
a[2:2] = [1,8,4]
```

を実行すると a は [3, 7, 1, 8, 4, 2, 5] となる。a[2] の前に要素の並びが追加された事が分る。a[2:2] は次の一般形の特殊な場合である。

```
a[n:m] = [1,8,4]
```

は a[n:m] を [1,8,4] に入れ替える。 これの特殊な場合として要素の削除が可能である。

```
a[n:m] = []
```

もっとも Python はリストの要素の削除の為に del 文を持っている。即ち

```
del a[n:m]
```

関数 range for 文における range は実はリストを返す関数である。例えば range(1,5) は [1,2,3,4] である。

1.5.2 リストへの作用関数

先に述べた a.append(2) の append はリスト a に働く作用関数 (method) である。append の他にも幾つかの作用関数が存在する。

プログラム例

以下にこれらをの作用関数の使い方を示すプログラム例を載せる。

作用関数	意味
a.append(x)	リスト a に要素 x を追加する
a.sort()	リスト a を整列する
a.sort(t)	リスト a を判定関数 t のもとに整列する
a.reverse()	リスト a を逆順にする
a.index(x)	リスト a の要素として x が現われるインデックスを返す。(なければエラーになる)
a.insert(i,x)	リスト a のインデックス i の位置に要素 x を挿入する
a.count(x)	リスト a の中に x と同じ要素が何個あるかを返す
a.remove(x)	リスト a の中に x と同じ要素があれば削除する。(なければエラーになる)

表 1.3: リストに定義されている作用関数

これらの関数はいずれもオブジェクト指向プログラミングの習慣に従って、関数が作用する変数名に続いてピリオドを書き、その後に関数名を書く。このように書く気持ちは、例えば a.sort() は sort と言う一般的な関数がありそれを使って a を整列するのではなく、変数 a の中に含まれている特殊な sort 関数を使用しているからである。

譜 1.2 リストへの作用関数の使用例

```
a=[8,2,1,3,4,0,9,5,3]
print a.count(3)        # 2 を出力
print a.index(0)        # 8 を出力
a.sort()
print a                 # [0, 1, 2, 3, 3, 4, 5, 8, 9] を出力
a.reverse()
print a                 # [9, 8, 5, 4, 3, 3, 2, 1, 0] を出力
a.remove(5)
print a                 # [9, 8, 4, 3, 3, 2, 1, 0] を出力
```

整列における判定関数

上のプログラム 1.2 では整列関数 sort は標準的な大小関係によって小さいデータから大きいデータの順に並べ替えているが、場合によってはもっと複雑な整理の仕方が要求される事もある。その場合には判定関数を導入する事になる。次のプログラムは年齢の大きな方から整理する場合の sort の判定関数の使い方を示している。

判定関数の引数 x, y はリストの要素を表している。例えば x=('alice',16) である。この場合 x[0] は 'alice' で x[1] は 16 である。y も同様である。関数 cmp(x,y) は

譜 1.3 整列における判定関数の使用例

```
def cmpf(x,y):
    return -cmp(x[1],y[1])
d=[('alice', 16), ('bob', 20), ('carol', 13)]
d.sort(cmpf)
print d         # [('bob', 20), ('alice', 16), ('carol', 13)] を出力
```

x < y の時に負、x > y の時に正 、x==y の時に 0 を返す Python の組み込み関数である。

1.5.3 2 次元の配列

2 次元の配列、即ち、行列はリストあるいは辞書として表現できる[10]。リストとして表現する場合には例えば行列

$$a = \begin{pmatrix} 3 & 1 & 5 \\ 2 & 4 & 8 \\ 7 & 6 & 1 \end{pmatrix}$$

は

```
a=[ [ 3, 1, 5],
    [ 2, 4, 8],
    [ 7, 6, 1]]
```

で表現される。リストのインデックスは 0 から始まるので、

a[0] が [3, 1, 5]
a[1] が [2, 4, 8]
a[2] が [7, 6, 1]
a[0][0] が 3、a[2][1] が 6

を表す。

行列の要素を表すのに a[2][1] のような表現法には抵抗があるかも知れない。数学の習慣により近い表現 a[2,1] を使用したい場合には辞書を利用できる。

リストで表現された行列の初期化には注意が必要である。要素が全て 0 の行列を作成するのに

```
a=[[0]*3]*3
```

とすると a の値は [[0, 0, 0], [0, 0, 0], [0, 0, 0]] となる。ここまでは当然であるが、しかし

```
a[0][0]=2
```

[10]数学的な問題では、array モジュールの array の方が使いやすい。もっと本格的な問題では numpy モジュールを利用した方がよい。ここには matrix がある。

で a の値は [[2, 0, 0], [2, 0, 0], [2, 0, 0]] となる。この不自然な結果は、Python が a=[[0]*3]*3 を処理するのに (内部で)

```
t=[0]*3
a=[t]*3
```

とやった為に発生したと考えられる。この問題は次のようにすれば回避できる。

```
   a=[0]*3
a[0]=[0]*3
a[1]=[0]*3
a[2]=[0]*3
```

1.5.4 プログラム例 (素因数分解)

リストを使用した実用的なプログラム例を挙げる。関数 factors(x) は引数 x を素因数に分解し、その構成要素を返す。例えば 36000 を引数として与えた場合には

```
[2, 2, 2, 2, 2, 3, 3, 5, 5, 5]
```

を返す。

譜 1.4 素因数分解のプログラム

```
def factors(x):
    "factor decomposition"
    f=[]
    while x % 2 == 0:
        f=f+[2]
        x = x / 2
    n=3
    while x >= n * n:
        while x % n == 0:
            f=f+[n]
            x = x / n
        n = n + 2
    if x == 1: return f
    return f + [x]
print factors(36000)
```

このプログラムの素因数分解法は初等的である。14443 の素因数を見つけるのに

$$2 \quad 3 \quad 5 \quad 7 \quad 9 \quad 11 \quad 13 \quad 15 \quad \cdots$$

で割り算を試し、剰余の有無を確かめていく。割る数が素数であるか否かの判定も行わない。3 以上の偶数の割り算は明らかに無駄であるから奇数のみを試している。今の場合 11

で割り切れる。そこで次は 1313(= 14443/11) の割り算を 11 から試す。すると 13 で割り切れる。そこで次は 101(= 1313/13) の割り算を 13 から試す。ところが 13 以上の数は $13 \times 13 > 101$ 故 101 の素因数にはなり得ない (なぜなら 13 以下の数は素因数でない事が確認されている) のでここで終了する。このプログラムに現われる 3 つの行

```
f=f+[2]
f=f+[n]
return f + [x]
```

の部分は append を使用して各々

```
f.append(2)
f.append(n)
return f.append(x)
```

と書き換えてもよい。そしてその方が実行効率が良い。(もっともここで扱ったような小さなリストでは実行効率の向上は期待できないであろう。)

1.6 辞書 (dictionary)

1.6.1 辞書の基本操作

キー タプルやリストでは変数 a の要素は番号によって指定され、従って

```
a[0], a[1], a[2], ...
```

で参照されるのに対して Pyhton の辞書 (他の言語では連想配列とも呼ばれる) を使用すると要素は名前によって指定され[11]、従って

```
a['alice'], a['bob']
```

のようにして参照される。ここでの 'alice' や 'bob' のような辞書の要素の名称をキーと呼ぶ。

辞書の生成 代入文

```
a={}
```

によって空の辞書 a が生成される。また、

```
a={'alice':13, 'bob':15}
```

によっても辞書が生成される。この辞書は 2 つのキー 'alice' と 'bob' を持ち、各々のキーによって指定された要素の値は 13 と 15 である。辞書の要素は任意のデータ型を混在できる。

キーは辞書が生成された後からも追加できる。即ち、辞書 a にキー 'carol' を生成し、それに 17 を設定する場合には、

```
a['carol'] = 17
```

を実行する。設定された値は a['carol'] で参照される。

[11] 実は文字列だけではなく様々なデータ型を基底にできる。しかしながらここでは説明を簡単にする為に文字列を基底にした場合を主に扱う。

1.6.2 辞書のメソッド

辞書を対象とする幾つかのメソッドが定義されている。以下に例で示す。

```
a={'alice':13, 'bob':15, 'carol': 17}
```

とすれば、
a.get('alice') は 13
a.keys() は ['bob', 'alice', 'carol']
a.values() は [15, 13, 17]
a.items() は [('bob', 15), ('alice', 13), ('carol', 17)]
を返す。keys や values や items の返すリストの項目の順序は必ずしもキーの辞書式順序ではない事に注意する。(この順序は実行環境によって異なる。) キーが登録されているか否かの判別が屡々必要となる。そのためには演算子 in が使える。例えば上の辞書では、

```
'alice' in a    は True
'david' in a    は False
```

となる。

キーの抹消 キーの登録を抹消するには del 文を使用する。

```
del a['alice']
```

でキー 'alice' が抹消される。

辞書の内容の消去 以上の他に、辞書に対するメソッドとして、a.clear() は辞書の内容を消去する

辞書のコピー b=a.copy() は辞書 a のコピー b を作成する。

1.6.3 プログラム例 (伝票の処理)

伝票の処理を考えて見よう。伝票にはキーが与えられ、発生した金額等の情報が書かれている。そこで、実際の伝票を単純化した以下のようなデータの処理を考えよう。

```
alice  23
bob    52
alice  81
carol  37
carol  53
```

1 つの行が 1 つの伝票であると考え、キーと金額が各行に記載されている。このようなデータが与えられた時に、キー毎の金額の和をとり、結果を表示するプログラムを作成するのがここでの問題である。ここで与えられたデータ例であれば

```
alice 104
bob 52
carol 90
```

を表示するプログラムを作りたいのである。

このような問題では各キーの金額の和を記憶する変数が必要になってくる。しかるにプログラムを作成する時点では予めどのようなキーが必要になるか分らない。この場合には辞書は有力な武器になる。この問題を処理する為には次の 2 つの関数を定義するのが基本的であろう。

1. 辞書 d の内容を表示する関数 pr(d)
2. 辞書 d の中の キー k の金額に、伝票の金額 v を加算する関数 put(d,k,v) 。この関数は、キー k が d に存在しなければキー k を作成し、金額を v に設定するものとする。

譜 1.5 に pr(d) と put(d,k,v) の定義と、サンプルプログラムを載せる。

譜 1.5 辞書を使った伝票の処理

```
def put(d,k,v):
    if k in d: d[k]=d[k]+v
    else: d[k]=v
def pr(d):
    for k in d: print k,d[k]

# test data
s=[ "alice 23",
    "bob 52",
    "alice  81",
    "carol  37",
    "carol  53" ]

# test program
d={}
for x in s:
    k,v=x.split()
    v=int(v)
    put(d,k,v)
pr(d)
```

この例ではデータはプログラムの中に、文字列を要素とするリストとして埋め込まれている。実用的なプログラムではそのような作り方はあり得ない事である。データはプログラムから独立して、ファイルの中で与えられるのが普通である。ファイルからのデータの読み取りに関しては第 4 章 1 節が参考になる。

1.6.4 キーとして許されるデータの種類

どのようなデータが辞書のキーになるかを調べよう。以下の例では、キーの値として全て 3 を設定しているが本質的ではない。この部分は何でもよいのだ。

```
a={}              # 空の辞書を生成する
a[1]=3            # 整数は OK
a[1.2]=3          # 実数は OK
a['alice']=3      # 文字列はは OK
a[(1,4)]=3        # タプルは OK
a[1,4]=3          # これも OK。[ ] の中の 1,4 はタプルであることを思い起こそう
a[1,'alice']=3    # 混在も OK
```

この他、関数などもキーになる。他方リストを含むデータはキーになれない。間接的なリストの参照、例えばタプルの中にリストが含まれている場合もダメである。辞書も辞書のキーとしては使用できない。

タプルが辞書のキーとして許されているために、辞書を 2 次元あるいは多次元の配列として利用できる。例えば行列

$$a = \begin{pmatrix} 3 & 1 & 5 \\ 2 & 4 & 8 \\ 7 & 6 & 1 \end{pmatrix}$$

は (面倒ではあるが)

```
a={(1,1):3, (1,2):1, (1,3):5,
   (2,1):2, (2,2):4, (2,3):8,
   (3,1):7, (3,2):6, (3,3):1}
```

で表現される。リストのインデックスと異なり、辞書の場合には 0 から始める必要はない。a[1,1] が 3 、 a[3,2] が 6 を表す。

1.6.5 関数のキーワード引数

辞書を使用するとキーワードを関数の引数に与える事ができる。即ち

```
f(alice=18, carol=10)
```

のように、関数引数の中に、代入式を書くことができるのである。この書き方は関数 f にオプション値を与える時に便利である。例えば GUI インターフェースを与える Tkinter モジュールの関数は、ボタンの色とか、使用フォントなどの多数のオプションを指定でき、その際にここに述べたキーワード引数によってオプション値を与えている。以下にキーワード引数を許す関数の定義法を示す簡単なプログラム例を挙げる。

関数定義文の引数並びの ** 記号に続く変数は辞書である。この辞書に関数呼び出し側で与えた代入式形式の情報が渡される。例えば

```
f(alice=18, carol=10)
```

譜 1.6 キーワード引数を使った関数の例

```
def f(**k):
    print k          # {'carol': 10, 'alice': 18} を出力
    d={'alice':16, 'bob':20, 'carol':12}
    for x in k.items():
        d[x[0]]=x[1]
    print d          # {'alice': 18, 'bob': 20, 'carol': 10} を出力
f(alice=18, carol=10)
```

が実行されると、辞書 k には k={'alice':18, 'carol':10} が代入される仕組みになっているのである。オプションの省略時の値を予め辞書として持っていれば、この辞書を実行時に与えたオプション値に基づいて修正する必要があろう。この事は

```
    for x in k.items():
        d[x[0]]=x[1]
```

で行われている。

1.7 集合

Python2.4 からは集合がサポートされた。集合の扱いは辞書と似ているのだが、要素は(辞書の場合と異なり)順序付けられていない。

```
names=set(['alice','bob','carol'])
print 'alice' in names      # True を出力
print not 'alice' in names  # False を出力
for a in names : print a    # carol  bob  alice を一行毎に出力
```

コメント欄には print による出力結果が載せてある。この最後の例では、実行環境によっては必ずしも carol bob alice の順に表示されないかも知れない。集合の要素は順序付けられていないからである。

集合と集合の間の演算に関しては、第 2 章 2.6 の集合演算を見よ。

第2章　変数と式

変数とはデータに付けられた名前である[1] 。名前として任意の文字列が許される訳ではない。数字や記号と区別する必要があるからである。そこで**名付け規則**と呼ばれるものがある。**名前**とは:

> 英字または下線記号 (_) で始まり、その後に (必要なら) 英字または下線記号または数字が続く文字列である。

ここで言う英字とは英語のアルファベットの事で大文字・小文字のいずれでも構わない。例えば x、s、alice、Alice、a0x、_01 等はいずれも名前になる資格を持っているが、数字で始まっている 123、123.45、1E-3、3+4 等や、記号文字や空白を含む L.Carroll や Lewis Carroll 等はプログラムで言う所の名前ではないのでデータに付ける名前としては使用できない。大文字と小文字は区別される。従って alice と Alice は異なった名前として扱われる。

以下の単語は Python では特殊な意味に使用されており名前として使用できない。

access	and	break	class	continue	def	del
elif	else	except	exec	finally	for	from
global	if	import	in	is	lambda	not
or	pass	print	raise	return	try	while

表 2.1: 予約語一覧

データに名前を設定するには

```
x=5
```

の様に記号‘=’の左に名前を書き、右にデータを書く。このデータの事を変数 x が指しているデータ、あるいは簡単に変数 x の値と言う。記号‘=’の右には 5 の様な単体のデータだけではなく、式が許される。その場合には式の計算結果に対して名前が付けられる。

[1]注 1: Python における変数の考え方は古典的なプログラミング言語 (FORTRAN, BASIC, PASCAL, C など) とは次の点で異なる: 古典的なプログラミング言語では変数とはデータを格納する容器である。この容器には名前が付けられ、この名前を変数名と呼んでいる。そして格納するデータの種類 (データの型) に容器の型を合わせる必要がある。

古典的な言語ではデータ型に関する情報は単にコンパイラのために存在するに過ぎない。ソースプログラムが一旦機械語に翻訳されると、個々のデータの型情報など演算を正しく遂行する為に必要な情報は失われ、その結果、全く不合理な計算が行われる場合もある。

Python ではデータ型など演算に必要な情報がデータの内容とともにパッケージになっている (これをオブジェクトと呼んでいる)。そして Python の変数にはオブジェクトの置かれている場所情報 (メモリのアドレス) が格納される。(言語 C の言葉で言えば Python の変数はポインタ変数なのである。) Python では (そして一般にオブジェクト指向言語では) データの型情報など演算に必要な情報がプログラムの実行時においても存在しているからこそ、それらの情報に基づいて理に適った計算を行う事ができるのである。Python の変数は全てポインタ変数であることから発生する Python の変数の特性を易しく理解するには、 Python では変数を容器とは考えないで、データに付けられた名前であると考えればよい。(詳しくは第 3 章 3 節を見よ)

式の中に現われる変数は、その変数の値と置き換えて式を解釈する。例えば x の値が 5 であれば

```
y=x*(x+1)
```

は

```
y=5*(5+1)
```

であると解釈する。そしてこの計算結果 30 に対して y と名付けられるのである。

変数が指すデータは固定的ではなく変化し得る。例えば n が 3 の時に

```
n=n+1
```

を実行すると、n の指すデータは 3 から 4 に変化する。同名の変数が様々な型のデータを指す事が可能である。例えは次の様に、整数 5 を表していた変数 x に、文字列 "alice" を設定してもよい。

```
x=5
...
x="alice"
```

しかしながら、このような書き方はプログラムを分りにくくするので避けるべきである。記号 '=' の右に変数が来ると、この変数が指すデータを記号 '=' の左に書いた変数と共有する。例えば、

```
a=['apple',1]
b=['apple',1]
c=b
```

において、変数 a と b は、内容が同じではあるが独立したデータを指すが、c と b は同一のデータを共有する。(この問題に関して、詳しい解説は第 3 章 3 節を見よ)

2.1　式

数学用語の式とは異なり、プログラミング用語としての式は必ず値を持つ。値を持つと言う事は、計算を進めていけば必ず具体的な数値や文字列あるいはそれらの寄せ集めに還元できることを意味している。式と言うのは値を計算する規則でもある。例えば次の式を考えて見よう。

```
a*x*(x+b)
```

この中には変数 a, b, x が現われている。式の値が求まる為には、これらの変数は値が前もって設定されていなければならない。プログラムに誤りが無ければ設定されているはずである。式の中に関数が含まれていても、その関数は値を持つようになつているはずである。

式の中に現われる記号、例えば '+'、'*'、'('、')' 等は計算の方法や規則を指示している。'+' と ' *' は演算子である。演算子は同じ優先度で評価されるのではなく、優先順位を持っている (表 2.2 を参照せよ)。乗算の方が加算よりも先に評価される。他方 '( )' の記号は、この部分の優先度がさらに高いことを表している。

x or y, lambda	論理演算子	論理和、ラムダ式
x and y	論理演算子	論理積
not x	論理演算子	否定
<, <=, >, >=, ==, !=, is, is not, in, not in	関係演算子	大きい、小さい 等しい、等しくない
x | y	ビット演算子	ビット OR
x ^ y	ビット演算子	ビット XOR
x & y	ビット演算子	ビット AND
x << n, x >> n	ビット演算子	ビットシフト
x + y, x - y	算術演算子	和、差
x * y, x / y, x % y	算術演算子	乗算、除算、剰余
-x, +x, ~x	単項演算子	符号反転、ビット反転
x[i], x[i:j], x.y, x(···)		添字、関数等
(···), [···], {···}, ‘···‘		タプル、リスト等

表 2.2: 演算の優先順位

この表では演算の優先順位の高い方が下方に書かれている。同じ優先度を持つ演算子は左から右へと評価される。この表の中の x や y は定数や変数や関数や式である。

定数、変数、関数、タプル、リスト、辞書はそれだけで式を形成する。さらにこれらを基本要素としてもっと複雑な式を構成できる。演算方法を指定する幾つかの記号 (演算子と言う) と演算の優先順位を示す丸い括弧がその場合に利用される。式が評価可能である為には、式の中に現われる全ての基本要素の値が既に定まっているか、あるいは評価可能である必要がある。

ここで述べた事を念頭に置いてもう一つの例を考えて見よう。

```
for x in [1996,1999,1900,2000]:
    print x, leap(x):
```

をプログラムの一部分とせよ。ここに現われる leap(x) は関数である。関数は式を構成する基本要素の 1 つであり、式の特殊なケースである。Python は for 文の最初の繰り返しで

```
print 1996, leap(1996)
```

に出会う。Python は leap(1996) の評価を開始する。しかし評価できるためには Python は関数 leap(x) の計算法を知っている必要がある。第 1 部第 3 章例題 3 のプログラムを添えてやれば Python は leap(1996) の計算法を知る事が可能である。この関数は引数に値を与えた時に評価可能なようにプログラムされていたのである。

関数引数の値を定めれば (正しくプログラムされた) 関数は評価可能である。

2.2　算術式

算術式は数字を基に数字を算出する式である。計算の基本となるのは四則であり、そのために 5 つの記号

```
+    -    *    /    %
```

を使用できる。

整数の間に次の算術演算が定義され、結果は整数となる。

```
x + y, x - y, x * y, x / y, x % y, -x, +x
```

ここに * は乗算の意味である。また / は除算の商、% は除算の剰余を意味する。例えば 23/5 は 4、23%5 は 3 である。しかし -23/5 は −5、-23%5 は 2 となる。これらが −4 や −3 にならないのは Python の単項演算子 (-) が二項演算子 (/や %) よりも計算の優先順位が高く -23/5 は (-23)/5 と、 -23%5 は (-23)%5 と見なされ、さらに Python では整数の商と剰余を数学者好みに定義しているからである。即ち、 y が正の時 x を y で割った時の剰余 r は、正または負の x に対して次の数学の式

$$x = py + r \qquad (\ 0 \le r < y\ )$$

を満たす r である。(大抵のプログラミング言語は負の x に対して負の剰余を出すが Python の剰余は数学における剰余の定義に忠実で、常に正である。) y が負の時 x を y で割った時の剰余 r は、正または負の x に対して次の数学の式

$$x = py + r \qquad (\ 0 \ge r > y\ )$$

を満たす r である。

実数の間にも整数と同様

```
x + y, x - y, x * y, x / y, x % y, -x, +x
```

の算術演算が定義されて、何れも実数を生成する。但し除算の意味が整数とは異なる。除算の商の計算は実数の範囲で行う。これは数学で普通に行う計算法である。実数における除算の剰余とは耳慣れない概念であるが次の様に考えれば分りやすい。x の長さの線分を y の大きさで刻むと最後に半端が残る。残った半端の長さは y より小さい。これが実数 x を実数 y で割った時の剰余である。Python の実数における除算の剰余 x%y は以下の手順で求めた r と同じである。(x, y が負の場合にも成立する)

```
p=floor(x/y)
r=x-p*y
```

実数と整数の演算は実数を生成する。複素数に対してもまた

```
x + y, x - y, x * y, x / y, x % y, -x, +x
```

の算術演算が任意の複素数 x と y に対して定義されている。 Python の複素数の剰余は不自然な結果を示す。従って将来は変更されるかもしれないので解説を省略する。

2.3 関係式

関係式は 2 つ値を比較し真偽を定める。
例えば x の値が 100 の時に x < 10000 は「真」である。

次に示す記号が比較のために使用できる。等号が‘=’ではなく、‘==’で表される事に注意せよ。

関係式	意味
X == *Y*	*X* と *Y* は等しい
X != *Y*	*X* と *Y* は等しくない
X < *Y*	*X* は *Y* より小さい
X > *Y*	*X* は *Y* より大きい
X <= *Y*	*X* は *Y* より小さいか等しい
X >= *Y*	*X* は *Y* より大きいか等しい
X is *Y*	*X* は *Y* である
X is not *Y*	*X* は *Y* でない
X in *Y*	*X* は *Y* の要素である
X not in *Y*	*X* は *Y* の要素ではない

表 2.3: 関係演算子 (ここに *X*, *Y* は式)

最後の 2 つでは、*Y* はタプルまたはリストである。

関係式は真の時に True 、偽の時に False の値を返す[2]。データの表現では真を 1 、偽を 0 と同一視される事が多い。Python 2.3 でも

```
True == 1
```

は True として扱われている。

True や False は数字ではないが 1 や 0 は数字である。そしてしばしば 1 や 0 で真偽を表す方が好都合な事がある。そのために True を 1 に False を 0 に翻訳するのに関数 int を使えるようになっている。例えば int(True) は 1 である。

Python の関係式は

```
0 < X < Y == Z
```

のような 3 個以上の式の比較を許す。この時の解釈は

```
0 < X and X < Y and Y==Z
```

である。

[2]Python 2.2 までは関係式は 1 と 0 を返していた。

X==*Y* と *X* is *Y* との違い

Python の変数はデータに付けられた名前である。*X*==*Y* は *X* と *Y* が指すデータの内容が同じである事を言う。他方 *X* is *Y* は *X* と *Y* が同じデータに付けられた名前である事を言う。

2.4　論理式

論理式とは論理値 (True=真または False=偽の値) から論理演算によって論理値を定める式である。論理演算には and、or、not を使用する。これらの意味を表 2.4 に示す。

X and *Y*	*X* と *Y* は共に真である
X or *Y*	*X* または *Y* は真である (共に真でもよい)
not *X*	*X* ではない

表 2.4: 論理演算子
X、*Y* は論理値である

例えば x の値が 50 、y の値が 200 の時に

```
x < 100 and y < 100
```

は「偽」である。

論理値 True と False は実際には 0 または 1 の数値である。従って表 2.3 の演算結果から 0 または 1 が得られる。表 2.4 において *X* と *Y* が 0 または 1 であれば結果は 0 または 1 となる。次に真と偽を各々 1 と 0 で代用して、論理積 (and) と論理和 (or) の演算表を示す[3]。

X and *Y*	*Y*=0	*Y*=1
X=0	0	0
X=1	0	1

表 2.5: 論理積

X or *Y*	*Y*=0	*Y*=1
X=0	0	1
X=1	1	1

表 2.6: 論理和

[3] True と False を数字で表してみると、何故論理「積」と言うのか良く分かると思う。

論理積と論理和において X と Y は 0 または 1 以外の数も許される。その場合、0 が偽、0 でなければ真と見なされる。その場合、演算結果が真の場合、 1 が返される保証は無いが 0 以外の数が返されることは保証されている。

2.5 ビット演算

整数データに対してビット演算が定義されている。整数の 2 進数表現の各々の桁は 0 または 1 であり、これに対する演算が、ここで扱っている整数のビット演算である。異なる位置にある桁は演算に対して独立している。(算術演算の場合には繰り上がりが発生するので独立していない。) ビット間には表 2.7 に示す 3 つの二項演算が定義されている。この表で x と y はビットを表す。

$x \mid y$	$y=0$	$y=1$
$x=0$	0	1
$x=1$	1	1

ビット OR

$x \hat{} y$	$y=0$	$y=1$
$x=0$	0	1
$x=1$	1	0

ビット XOR

$x\&y$	$y=0$	$y=1$
$x=0$	0	0
$x=1$	0	1

ビット AND

表 2.7: ビット演算子

例

以下では整数の 2 進数による表現を $(B)_2$ のように、ビット列 (0 と 1 から構成される数字の列) B を丸い括弧で括り、2 を添えて表す。例えば

$3 = (11)_2$

$5 = (101)_2$

である。従ってビット OR, ビット XOR, ビット AND の結果は次の様になる。

演算	演算結果の 2 進数表現	演算の結果の 10 進数表現
3|5	$(111)_2$	7
3^5	$(110)_2$	6
3&5	$(1)_2$	1

2 つの整数間のビット演算を考える時には、桁を揃える必要があり、必要ならば上位の桁に 0 を補う。3 と 5 のケースでは $(011)_2$ と $(101)_2$ とのビット演算を考える事になる。

整数の -1 の 2 進数による表現は

$$(\cdots 11111111111111111111111111111111)_2$$

であると定義する。ここに $\cdots$ の部分は無限に 1 が続くことを意味する。このように定義する理由は、これに 2 進数の 1 を加えると、繰り上がりが無限の桁まで連鎖的に発生し、そ

の結果、全ての桁が 0 になるとの解釈が可能だからである。その解釈は $(-1)+1=0$ の関係を表している。

ビット反転の単項演算子 ~ について説明しよう。この演算子は整数の 2 進数表現の各桁を反転する。すなわち、0 ならば 1 に、1 ならば 0 にする。この定義によれば、例えば、~5 の 2 進数の表現は

$$(\cdots 11111111111111111111111111111010)_2$$

である。特に ~0 の 2 進数の表現は

$$(\cdots 11111111111111111111111111111111)_2$$

となり、10 進数との関係では　~0 = −1 であることが分かる。

任意の整数 n に対して $\tilde{}n+n=-1$ の関係が成り立つので、Python のビット反転演算子 ~ は

$$\tilde{}n=-(n+1)$$

で定義される事もある[4]。

以上の他にビットシフト演算子が存在する。即ち整数データ I に対して

$$I << n$$

で I を構成するビットが左に n 個だけシフトする。その時、右には n 個の 0 が補われる。例えば

5<<2　=　$(10100)_2$　=　20

である。同様に右シフト演算子が存在し、

5>>2　=　$(1)_2$　=　1

となる。シフトの結果、はみ出したビットは消える。

2.6　集合演算

集合から集合を生成する演算に関してはビット演算の演算子が代用される。以下に例を示す。

```
names1=set(['alice','bob','carol'])
print names1            #  set(['carol', 'bob', 'alice']) を出力
names2=set(['carol','david'])
print  names1 | names2  # 和集合、set(['carol', 'david', 'bob', 'alice']) を出力
print  names1 - names2  # 差集合、set(['bob', 'alice'])
print  names1 & names2  # 積集合、set(['carol']) を出力
print  names1 ^ names2  # 排他的和集合、set(['bob', 'alice', 'david']) を出力
```

[4]むしろ、これが公式の定義である

ここでは print の結果がコメントに示されている。

集合の比較 (包含関係等) には、次の演算子:

```
==      !=      <       <=      >       >=
```

が使用される[5]。結果は論理型である。

```
names1=set(['alice','bob','carol'])
names2=set(['carol','david'])
names3=set(['bob','carol'])
print names1 > names3       # True を出力
print names1 > names2       # False を出力
print names1 <= names3      # False を出力
print names1 == names3      # False を出力
```

2.7 lambda 式

lambda 式は簡単な関数を定義するのに使用される。ここで言う簡単な関数とは数式の様に単に式を与えればすむような関数で

```
def f(引数並び): return 式
```

の形で記述可能な関数の事を言う。例えば次のプログラムに現れる関数を想像すればよい。

```
def f(x,y): return x+y
print f(2,3)     # 5 を出力
```

これと同じ事を行う関数は lambda 式を使用して次の様に書くことができる。

```
f=lambda x,y: x+y
print f(2,3)     # 5 を出力
```

lambda 式の一般形は次の通りである。

```
lambda 引数並び: 式
```

lambda 式は関数を返す。def 文による関数定義と同様に、式の評価は呼びだされるまで行われない。lambda 式を使用すると 1 つの式の中で関数が定義され、かつそれを実行できる。

```
print (lambda x,y: x+y)(2,3) # 5 を出力
```

def 文ではこのような事はできない事に注意する。

次のプログラムは lambda 式を利用して関数を繰り返し文の中で定義して、定義した関数を配列に入れている。

```
f=[0]*5
for n in 1,2,3,4: f[n]=eval("lambda x: x**%d"%n)
print f[1](2) # 2 を出力
print f[2](2) # 4 を出力
print f[3](2) # 8 を出力
print f[4](2) # 16 を出力
```

def 文を用いても同様な事は可能であるが、lambda 式の方がいくらか簡潔である。

> 注意: lambda 式は好んで使うような式ではない。こんな式は知らなくても困らない。

[5] 他の書き方もあるが省略する

第3章　プログラムの構成

3.1　基本概念

3.1.1　文

プログラムは文 (statement) と呼ばれる単位から構成される。英語の‘statement’は命令文を意味している。文は何かを実行するよう計算機に命令を下しているのである。例えば代入文

```
x = 5
```

はデータ 5 に名前 x を付けるよう命令している。

文は単純文 (simple statement) と複合文 (compound statement) に分類できる。print 文、代入文、式だけの文などが単純文の例として挙げられる。複合文はその構成要素として他の文を含む。例えば条件文、繰り返し文、関数定義文、try 文などが複合文である。複合文の範囲は字下げによって示される。

プログラムは文の集まりであり、文と文の区切りには記号‘;’を使用するか、または、行を改める。プログラムの文は原則として英文と同じ順序で実行される。即ち、上から下へ、左から右に向かって実行される。

関数定義文 (def 文) とクラス定義文 (class 文) はこの規則の例外である。これらは必要に応じて実行される。

計算機はプログラムに記述された文を 1 つ 1 つ順を追って実行する。一般的に言えば実行の順序が異なれば異なった結果を得る。従って文の実行順序を正しく記述する事は重要である。

3.1.2　コメント

文字列の外にある記号‘#’をコメント記号と言う。この記号の右側に何が書かれていようとプログラムの実行に影響を与えない。コメントはプログラミングのメモとして使用される。コメントは文と見なさない。また文字列だけの文も実行結果に影響を与えないのでコメントとして利用できる。

3.1.3　行

一つの行は

1.　字下げ空白 (インデント)

2. 文の並び
3. コメント

の順に構成される。これらのどれも必須ではない。以下に例で図示する。

```
                x=5 ; a="#"    # ............
字下げ空白      文      文        コメント
```

図 3.1: 行の構成

3.1.4 字下げ

字下げ空白に使用する空白文字の個数を字下げの大きさと言う。(字下げの深さとも言う)

> 注釈:　文を行の先頭からではなく、いくらか空白を置いて書き始める事を段落付けすると言う。上手に段落付けすると文の論理的な構造が直感的に分る。そのためにプログラマは段落付けを好む。 Python 以外のプログラミング言語では、段落付けは単にプログラムの可読性 (人にとっての読みやすさ) を高める為の便法であって、プログラムの解釈とは無関係であった。Python は段落付けをプログラムの解釈と結びつけたのである。

文を含まない行は捨てられるので字下げの大きさは定義されない。

次のプログラムの #1 から #3 までの行は字下げの大きさが 0 であり、#4 から #7 までの行は字下げの大きさは 4 である。

譜 3.1 プログラム例 1

```
x=1                     #1
y=1                     #2
while x < 10000:        #3
    print x             #4
    z = x + y           #5
    x=y                 #6
    y=z                 #7
```

3.1.5 ブロック

2 つの行 A と B は、字下げの大きさが同じで、かつ、A と B の間に (A または B よりも) 字下げの小さい行が存在しない時、同じブロックに属すると言う。譜 3.1 のプログラム例 1 では、#1 から #3 は同じブロックに属する。また #4 から #7 も同じブロックに属する。

譜 3.2 のプログラム についても[1]同様に行を類別すると次の様に 4 つのブロックに分けら

譜 3.2 プログラム例 2

```
def sim(x,p,r):                                                    #1
    n=0                                                            #2
    while x > 0:                                                   #3
        y=x*(1+r)                                                  #4
        z=y-p                                                      #5
        n=n+1                                                      #6
        # print "%3d %10.4f %10.4f %10.4f" % (n,x,y,z)            #7
        x=z                                                        #8
    return (n,y)                                                   #9
                                                                   #10
for x in range(0,11):                                              #11
    print "%3d %10.4f" % sim(3000,200+10*x,0.05)                   #12
```

れる。

```
#1  #11
#2  #3  #9
#4  #5  #6  #8
#12
```

#7 及び #10 は読み捨てられる。また#12 と #9 は字下げの大きさは同じではあるが間に字下げの小さな #11 があるので同じブロックには属していない。

頭部

記号 ‘:’ の左 (字下げ空白を含まない) をプロックの頭部と言う。このプログラムでは

```
def sim(x,p,r)
while x > 0
for x in range(0,11)
```

の 3 つが頭部である。(それぞれ def 文の頭部、while 文の頭部、for 文の頭部と言う)

プログラムに現れる任意のプロックの頭部の 1 つを H とする。頭部 H に続く行で、H よりも字下げの大きな行を H に伴う行と言う。例えば #1 の

```
def sim(x,p,r)
```

に伴う行は #2 から #9 までの全ての行である。

[1] プログラム 2 は、銀行から金利 5% で 3000 万円借りるとして、毎年の支払いの幾つかのケース (200 万円、210 万円、…、300 万円) について、返済に必要な金額と最後の年に支払う金額を求めている。ここに現われる sim(x,p,r) は x 万円の借入を行い、金利 r の下に毎年 p 万円ずつ返済する場合に、完済するに必要な年数と最後の年に支払う金額を返す。関数 sim(x,p,r) は返済不可能な場合の処理を行っていない事に注意せよ。

本体

頭部と ‘:’ と頭部に伴う行は 1 つの文を形成している。プログラム 2 では #3 から #8 は while 文、#1 から #9 までが def 文である。頭部に伴う行をそれらの文の本体と言う。プログラムを簡潔に書く為に頭部に伴う文が他の頭部を含まない時、頭部と同じ行に書くことが許されている。

3.1.6　特殊記号と空白

プログラムのコメントを除いた部分に現われる以下の記号を特殊記号と言う。

```
+   -   *   /   %   |   ^   &
,   .   :   ;   ‘   ’   "
=   ==  !=   <   <=  >   >=  <<  >>
(  )  [  ]  {  }
```

図 3.2: 特殊記号

以下にいくつかの注意事項を挙げる。

■　字下げの大きさが変化しない限り、特殊記号の左右に任意の個数の空白を置いてもプログラムの解釈に影響を与えない。逆に、字下げの大きさが変化しない限り、特殊記号の左右の空白は除去可能である。従って、例えば

```
z = x + y
```

は

```
z=x+y
```

と同じ意味である。但し以下の 2 つは例外である。

1. 数の一部である小数点、例えば 1.2 の中に含まれるピリオドの左右には空白を入れてはならない。
2. 文字列の一部としての空白を変更すると別の文字列となる。

■　1 個の空白が許される個所には任意個の空白が許される。例えば、

```
def sim(x,p,r):
```

の def と sim の間には 1 個の空白が許される。従って、ここには空白を何個書いてもよい。例えば

```
def     sim(x,p,r):
```

と書いてもよい。

■ 名前を構成する文字 (英字と数字と下線記号) の間にある空白は除去できない。例えば

```
def sim(x,p,r):
```

の def と sim の間の空白は除去できない。除去すると defsim と言う別の名前になるからである。

3.2 print 文

print 文は式の値を出力する。複数個の式を指定する場合には式をコンマで区切る。出力された式の値は 1 個の空白で区切られる。改行動作を伴う print 文は次のように書く[2]。

```
print
print 式
print 式，式，…，式
```

図 3.3: 改行動作を伴う print 文の形式

式が指定されていない場合には単に改行する。改行動作を伴わない print 文は次のように、式に続けてコンマを書く[3]。出力の整形は文字列に作用し文字列を生成する演算子 % に

```
print 式,
print 式，式，…，式,
```

図 3.4: 改行動作を伴わない print 文の形式

よって行う。つまり Python では出力書式は print 文の一部と見なされていない。しかしながら実際には出力書式は殆どの場合 print 文と共に使用される。従って以下に書式の指定の方法について述べる。

3.2.1 書式

文字列に対する演算子 % はデータを文字列の中に埋め込み、同時に埋め込んだデータを指定された書式に基づいて整形する。例えば

```
"name=%s age=%d" % ('alice', 18)
```

はリスト ('alice', 18) を演算子 % によって文字列 "name=%s age=%d" に作用させ、その結果

[2]Python3 では print は文ではなく関数になった。そのために print(式，式，…，式) のように、() で括らなくてはならない。

[3]Python3 では改行動作を伴わない場合には print(式，式，…，式，end="") のように、end="" を添える。

```
"name=alice age=18"
```

を生成する。従って例えば、変数 name の値を "alice"、変数 age の値を 18 とすると、

```
print "name=%s age=%d" % (name, age)
```

を実行すれば

```
name=alice age=18
```

が表示される。

この例では文字列 "name=%s age=%d" の中の % の後に s や d の文字が続く、これを変換文字という。変換文字は変換の方法を指定する。例えば "x=%X"%x はこの文字列の %X の部分を x の 16 進数表現で置き換える事を意味している。x の値が 12 であれば結果は 'x=C' となる。

次の表 3.1 に Python で使用できる変換文字を表に掲げる。これ以外に %i や %u もあるが筆者には必要性は感じられないので載せていない。

変換文字	意味	変換文字	意味
s	文字列	x, X	16 進数
c	文字	e, E	指数表示
d	整数	f	固定小数点表示
o	8 進整数	g, G	指数表示/固定小数点表示

表 3.1: 変換文字

出力幅を指定するには % と変換文字との間に出力する幅を表す数字を入れる。例えば "%5d"%18 は文字列 '　 18' を生成する。ここで 18 の前には 3 個の空白が挿入され文字列の全体の長さが 5 になる。もしも -5 を指定すれば左に詰められたであろうし、"%05d"%18 では '00018' のように 0 が埋められる。

実数の表示精度の指定は表示幅数の次にピリオドを書き、その後に小数点以下の出力桁数を指定する。例えば、実数の出力形式を、小数点以下 5 桁まで表示し、全体の表示幅を 10 にするには "%10.5f" で書式を指定すればよい。もしも x に (1.0/3) が設定されていれば "%10.5f"%x は

```
'   0.33333'
```

を生成する。

指数表示も同様である。例えば "%12.4E"%(1.0/3) は

```
' 3.3333E-01'
```

を与える。E を小文字で書けば小文字の表示が得られる。G の指定は、固定小数点表示で間に合う場合には固定小数点表示で、そうでなければ自動的に指数表示に切り替えてくれる。Python の書式指定は言語 C の書式指定と良く似ているが、以下の例で示すように %s でほとんど間に合ってしまう。(# 記号の右側に出力結果を示してある。)

```
s="alice"
print "---%s---"%s      # ---alice---
print "---%10s---"%s    # ---     alice---
print "---%-10s---"%s   # ---alice     ---
n=123
print "---%s---"%n      # ---123---
print "---%10s---"%n    # ---       123---
print "---%-10s---"%n   # ---123       ---
x=10.0/3
print "---%s---"%x      # ---3.333333333333---
print "---%10.5s---"%x  # ---     3.333---
print "---%-10.5s---"%x # ---3.333     ---
```

Python は辞書を扱う為に拡張された書式を持っているが、それについては解説を省略する。

3.3 代入文

代入文によって変数に値が設定される (これを「代入する」と言う)。 変数名を記号 '=' の左に書き、'=' の右には式を書く。複数の変数への代入も可能である [注 1]。

図 3.5: 代入文の形式

変数には式が評価された結果が代入される。(Python における代入文の意味は、式の値に対して名前を割り当てているのであり、割り当て文と呼ぶべきである。しかし我々は古典的なプログラミング言語の習慣に従って代入文と呼ぶことにしよう。)

注 1. Python では以下の様に複数の変数に代入できる。

```
X=Y=3
```

この書き方は言語 C と似てはいるが、これを根拠に Python の代入文が C のように値を持つと誤解してはならない。代入文の書き方の 1 つとして許されているだけである。

Python におけるデータへの名前の割り当てかたは次の図 3.6 に示す様に、名札を付けていると理解すればよい。

この考え方で次のプログラムの動作について考えて見よう。print 文による出力結果は右側にコメントとして示されている。

```
a=['apple',1]
b=['apple',1]
c=b
```

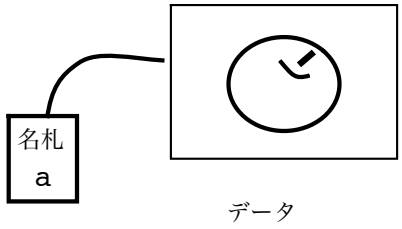
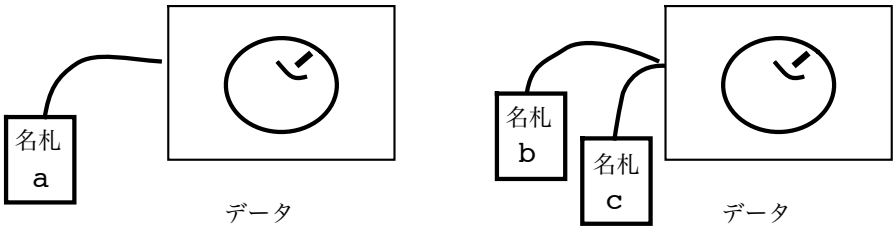


図 3.6: Python における代入の意味

```
b[0]='orange'
print a     # ['apple', 1]   を出力
print b     # ['orange', 1]  を出力
print c     # ['orange', 1]  を出力
```

この結果は何を意味しているか?
a および b の指すデータ ["apple",1] は内容的には同一であるが、独立したデータと見なされていることが分る。代入文 c=b によって b を同じデータに名前 c が付けられたことが分る。ここまでの結果を絵に書くと以下の様になる。



この後 b[0]="orange" によって b の指すデータの内容が変更された。その結果 c の指すデータも変更された。

名札の指すデータは 1 つであるが、1 つのデータに異なる名札が付く事がある。つまり同じデータをいろいろな名前で呼ぶ事は可能なのである。データの内容がたまたま同じであると言う事と、同じデータを表していると言う事は区別する必要がある。

X , Y を変数名とすると、関係式 X is Y によって X と Y が同じデータを指しているか否かを判定できる。この値が 1 なら同じデータを指しており、0 なら異なるデータを指している。

```
a=['apple',1]
b=['apple',1]
c=b
print a is b     # 0  を出力
print c is b     # 1  を出力
```

さて、再び

```
a=['apple',1]
b=['apple',1]
```

について考えて見よう。何故 同じ内容のデータを独立したデータとみなす必要があったのだろうか? リスト形式のデータの要素は変更可能だからである。つまりこの時点で同じ内容であっても、この後どのように変化していくか分らないので安全の為に独立したデータが割り当てられたのである。

問 1 次のプログラムの実行結果は 1 になる。

```
a='apple'
b='apple'
print a is b     # 1  を出力
```

つまり a と b はデータを共有している。これで問題が発生しない理由について考えよ。

問 2 次のプログラムの実行結果は 0 になる。

```
a=('apple',1)
b=('apple',1)
print a is b     # 0  を出力
```

つまり a と b はデータを共有していない。タプル形式の要素への代入が禁じられているにもかかわらず安全策が採られているのである。何故か? もしも要素の性格に関わらずタプル形式のデータを共有した場合に、どのような問題が発生するかを具体例で示せ。

3.4　式

式は文としての資格を持つ。その場合には式は評価されるがその値は無視される。

■　第 3 章の最後のプログラム (Zeller の公式) は多くのコメントを含んでいる。これを次の様に文字列として処理する事が可能である。# 記号を多数並べるよりもこの方がエレガントである。

譜 3.3 文字列はコメントの代わりになる

```
"""
h(y,m,d): day of week (Zeller's formula)
奥村晴彦「コンピュータアルゴリズム辞典」(技術評論社,1989)
y: year, 0,1,2,..
m: month, 1,2,..,12
d: day of month, 1,2,..31
return: day of week, 0,1,2,..,6
example: h(2000,1,1) -> 6    # Saturday
"""
def h(y,m,d):
    if m<3: y=y-1; m=m+12
    return (y+y/4-y/100+y/400+(13*m+8)/5+d)%7
```

■　Python の関数は常に値を持つ[4]。この値を利用する場合には関数は式の一部として使用されるが、利用しない事もできる。その場合にはその関数は評価される (即ち実行される) が戻り値は捨てられる。あたかも単独の命令文のように機能するのである。

[4] この点も言語 C と Python では異なる。詳しくは 6 節関数定義文を見よ。

3.5 条件文

ある条件が成立している時にだけ何かを実行したい場合が発生する。もっと一般的には、場合分けを行い、それぞれに応じて実行内容を決定したい場合もある。こうした事は条件文によって実現される。

Python の条件文は次の形をとる。

```
if 条件:
    本体
elif 条件:
    本体
elif 条件:
    本体
...
else:
    本体
```

図 3.7: if 文の形式 1

「条件」の部分は本体が実行される条件を書く。条件は関係式や論理式で与えられるのが普通であるが、Python は任意の式を許す。式の値が True であれば (あるいは 0 でなければ) 条件が成立したとみなされる。

条件文は場合分けになっており、条件は順に調べられる。成立していなければ次の elif の条件が調べられる。成立した場合にはその本体が実行され、if 文から抜け出す。else はどの条件も成立しない場合を受け持っている。(elif は 'else if' から派生した造語である。)

最初の if 文以外は必要に応じて書けばよい。従って一番簡単な場合には次の形をとる。

```
if 条件:
    本体
```

図 3.8: if 文の形式 2

例 12 次に示すのは場合分けを行うプログラム例である。このプログラムは 0 と 1 と 2 をランダムに 200 個生成する。 0 と 1 と 2 の生成比率は 5 対 2 対 3 に設定されている。生成比率を定めるために 0 と 1 の間の一様な実数値乱数が利用されている。

譜 3.4 場合分けを行うプログラム例

```
from random import *
n=0
while n < 200:
    x=random()
    if x < 0.5: print 0,
    elif x < 0.7: print 1,
    else: print 2,
    n=n+1
```

3.6　繰り返し文

3.6.1　while と for

Python は 2 つの繰り返しの表現を持っている。それらは次の形をとる。

```
while 条件:
    本体
```

図 3.9: while 文の形式

```
for 制御変数 in 範囲:
    本体
```

図 3.10: for 文の形式

while 文は条件が成立する間、本体の実行を繰り返す。例えば

```
x=0
while x < 5:
    print x
    x=x+1
```

は条件 x<5 の条件の下に

```
print x
x=x+1
```

を実行する。(従って 0, 1, 2, 3, 4 を出力するはずである。)

for 文は制御変数が与えられた範囲の全ての要素の値を採りながら本体が実行される。そして範囲はシーケンスを使って与えられる。例えば

```
for x in [5,2,3,6]: print x
```

は

```
print 5
print 2
print 3
print 6
```

が実行されるのと同じ効果を持つ。シーケンス部分は文字列、タプル、リストのどれでもよい。

3.6.2 range

for 文の応用では制御変数は単純な規則に従って変化する事が多く、シーケンスは関数 range で与えられる事が多い。range は 1 個から 3 個の引数をとる。

```
range(b)
range(a,b)
range(a,b,c)
```

range の中の a、b、c は各々 x の初期値、x の実行範囲 (x < b) 、ステップを表す。即ち x は a から始まり x < b の条件の下で c ずつ増加しながら本体部分が繰り返し実行される。c が 1 の場合には range(a,b) と書く事ができる。また a が省略された場合には 0 が仮定される。従って、例えば

```
for x in range(3,13,4):
```

であれば、x は 3, 7, 11 の値を順にとる。

range はリストを生成する関数である。以下に幾つかのケースに付いて range が生成するリストを挙げる。

```
range(13)        [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
range(3,13)      [3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
range(3,13,4)    [3, 7, 11]
```

range はリストを実際に生成するのでリストの大きさに比例してメモリが使用される。非常に大きなリストが生成される場合には range の代わりに xrange を使うが良い[5]

3.6.3 continue と break

繰り返し文の実行中に、何か特殊な条件の下に処理の流れの変更が要求される場合がある。その場合 continue 文と break 文を利用できる。繰り返し中に continue 文に出会うと

[5]古い Python では xrange はタプルを生成する関数であったたが、最近の版では xrange は仕様が変更された。xrange は xrange 型と言う独自のデータを返す。この型のデータは使い方が強く制限され、実際上は for 文の中でしか使用されない。その代わり大きな範囲を指定しても一定量のメモリしか食わない。なお for 文の中での xrange の使い方は range と同じである。

本体の残りの実行を省略して繰り返しの次のステップへ進む。また break 文に出会うとそのまま繰り返し文の実行を終了する。これらは if 文の中で使用される。

次のプログラム 3.5 とその出力結果はは continue と break によって繰り返し文の本体部分のどこが実行されたか (あるいは実行されなかったか) を見せている。

譜 3.5 continue 文と break 文による制御

```
for x in [3,4,5,6,7]:
    print x
    if x==5: continue
    print "alice"
    if x==6: break
    print "bob"
```

このプログラムの出力結果は次のようになる。

```
3
alice
bob
4
alice
bob
5
6
alice
```

x が 5 で print "alice" 以下は実行されていない。また x が 6 で print "bob" を実行することなく繰り返しが終了している。

3.7　関数定義文

3.7.1　def

関数定義文 (def 文) は次の形をとる。

```
def 関数名 (引数並び):
    本体
```

図 3.11: def 文の形式

def 文が実行されると関数名が登録される。関数本体の実行は関数を呼びだした時点で行われる。「引数 (ひきすう) 並び」は関数が呼びだされる時に値を受け取る変数の一覧である。例えば図 3.6 のプログラムでは

```
sim(3000,200,0.05)
```

によって関数 sim が呼びだされている。この時、引数並びの x, p, r に 3000, 200, 0.05 が、この順で代入される。(そして関数本体が実行される。)

譜 3.6 ローン返済シミュレーション

```
def sim(x,p,r):
    # x: amount of loan
    # p: every year payment to return the loan
    # r: rate
    # return: number of years to finish and the last payment
    if x*r >= p: # unable to return the loan
        return None
    n=0
    while x > 0:
        y=x*(1+r)
        z=y-p
        n=n+1
        x=z
    return (n,y)

print "%3d %10.4f" % sim(3000,150,0.05)
```

3.7.2 return

我々が普通の意味で関数と言う時には、関数の引数に値を与え、引数の値に応じた関数の値が存在する事を期待する。関数の値は return 文によって定まる。 return 文は一般に次の形をとり、予約語 return に続く式の値が関数の値となる。

```
return 式
```

図 3.12: return 文の形式

従って関数の実行を終了する時には原則として return 文で終了する必要がある。return に続く式が省略されたり、return 文で終了しない場合には、関数は None と言う特殊な値をとる。None は演算の対象にはならない。従って常に None を返す関数、例えば return 文を含まない関数は式の中で使う事はできない。そのような関数は単独の命令文として使用される。(このような関数を「**手続き**」とも言う)

3.7.3　引数並び

引数並びを持たない関数も許される。その場合関数定義文は

```
def 関数名 ():
    本体
```

の形をとる。関数の名前は命名規則の範囲内で任意に付ける事ができる。

引数並びを持たない関数を呼びだすには

```
関数名 ()
```

のように関数名の次に () を付け、() の中には何も書かない。

3.7.4　局所変数と大域変数

関数の中で生成された変数は実行が終わり次第捨てられる。従って関数の外からは参照できない。また他の関数からも参照できない。このような変数を局所変数と言う。関数 sim では x, p, r, n, y, z が局所変数である。

> 何故関数の外から隠されている変数が必要なのか?　関数はプログラミングにおける部品である。電化製品を例に採ると、電球が切れた場合、代わりの電球を買い替えるに多くの知識を要しない。ソケットに合い、ワット数さえ同じであれば基本的に問題は無いからである。電球の作り方に多くの工夫を凝らされているであろうが、使い方が電球設計の細部に依存していると難しくてとても素人には扱えなくなる。同様な事は関数にも当てはまる。関数設計の細部がプログラムの他の部分と干渉を引き起こさないで使用できる事、言い換えれば関数の仕様 (使い方) さえ分っていればその関数を誤り無く利用できる事が大切であり、その為には関数の内部で一時的に使用されているにすぎない変数は外部から隠す必要があるのである。

他方譜 3.7 に示すように、関数の中からは関数の外で定義された変数が参照可能である。関数の内部で参照される関数の外で定義された変数を大域変数と言う。大域変数を使用した例は第 2 章に載っている:

但しコメント行は省略されている。このプログラムでは wday が大域変数であり、どの関数の中からも参照可能である。そして実際に cal0(n,m) の中の行

```
for x in wday: print " ",x,
```

で wday を参照している。

関数の中から大域変数を変更するのは好ましくはないが可能である[6]。 その場合には global 宣言を添えなくてはならない。global 宣言の効果を譜 3.8 の簡単なプログラム例で説明する。

譜 3.8 のプログラムでは global 宣言によって、関数 f(x) と関数 g() は変数 a を共有している。f(3) が実行される事によって変数 a が 3 に設定される。g() が実行される事によって変数 a の値が出力される。そして実際に 3 が出力された。global 宣言された

[6]時には必要な場合もある

譜 3.7 関数の中から関数の外の変数を参照するプログラム例

```
wday=("sun","mon","tue","wed","thu","fri","sat")
def cal0(n,m):
    for x in wday: print " ",x,
    print
    for x in range(0, n): print "  ---",
    for x in range(1, m+1):
        print "%5d"%x,
        if (x+n)%7==0 : print
cal0(6,31)
```

譜 3.8 global 宣言を使ったプログラム例

```
def f(x):
    global a
    a=x
def g():
    global a
    print a
f(3)
g() # 3 を出力
print a # 3 を出力
a=4
g() # 4 を出力
```

変数は関数の外部からも参照と変更が可能である。このプログラムの最後の 3 つの行はそのことを示している。

2 つ以上の変数を global 宣言する場合には

```
global a,b,c
```

のように変数をコンマで区切る。

関数の中で関数を定義する事も可能であるが推奨できない。

3.7.5 いろいろな関数引数

Python では様々な種類の関数引数 (ひきすう) がサポートされている。図 3.9 に簡単な例を挙げて、実行結果に基づいて使い方を説明する。

実行結果

```
2 3 13 20 () {}
```

譜 3.9 いろいろな関数引数の使用例

```
def f(x,y,a=13,b=20,*t,**k):
    print x,y,a,b,t,k

f(2,3)
f(2,3,alice=18, carol=10)
f(2,3,5,alice=18,carol=10)
f(2,3,5,7,8,9,alice=18,carol=10)
```

```
2 3 13 20 () {'carol': 10, 'alice': 18}
2 3 5 20 () {'carol': 10, 'alice': 18}
2 3 5 7 (8, 9) {'carol': 10, 'alice': 18}
```

引数並び変数の 4 つの種類

関数定義文の引数並びの変数は以下の 4 つの種類に分類できる。

1. 省略時の値が与えられていない変数関数 f の引数並びの最初の 2 つ x と y はこれまで扱ってきた普通の引数である。変数名が単独で現われている事に注意する。関数を呼びだす時にはこの 2 つの変数には必ず値を与えなくてはならない。従って f の呼び出しには少なくとも 2 個の値を渡す必要がある。
2. 省略時の値が与えられでいる変数引数並びの変数で a と b は省略時の値が与えられている。省略時の値は代入式形式で指定する。関数を呼びだす時に第 3 引数が存在すればその値が a に、第 4 引数が存在すればその値が b に渡される。
3. タプル引数並びの記号 * に続く変数はタプルである。ここでは変数 t はタプルである事を意味している。関数を呼びだす時に引数が 4 個よりも多く存在すれば、残りの引数の値がタプルとして変数 t に渡される。
4. 辞書引数並びの記号 ** に続く変数は辞書である。ここでは変数 k は辞書である事を意味している。関数を呼びだす時の引数の中に、代入形式の引数があれば、辞書にキーとその値として渡される。

関数定義文の中でも、関数呼び出しにおいても上記 1 から 4 の順序を守らなくてはならない。なお、第 1 章 4 節および第 1 章 6 節のタプルと辞書の解説も参照せよ。

3.8　クラス定義文

3.8.1　class

class 定義文 (あるいはクラス宣言文) は新しいオブジェクトのクラスを定義する。Python がオブジェクト指向言語であると言われるのは、この文が存在するからである。クラス生成

に関する全てを解説するのは紙面の関係でできない。そこで簡単な場合に関してのみ解説を加える。

次のような問題を考えよう。Python のタプルは数学で扱うベクトルに近い表現形式を持っている。即ち数学で $(2,3,7)$ と書けば屡々ベクトルを表し、ベクトルの間には固有の演算方法が存在する。その演算方法は要素ごとの和であり、$(2,3,7)+(5,1,4)$ は $(7,4,11)$ である。ところが Python では (2,3,7) はタプルであり、2 つのタプルの和はタプルの結合として定義されてしまっている。即ち (2,3,7)+(5,1,4) は (2,3,7,5,1,4) である。ベクトルの計算をベクトルらしい表現で行いたいのだが何とかならないであろうか?

オブジェクト指向プログラミング言語はこのような要請に対する 1 つの方法を提供する。クラスと言われる理由は、クラスに属するオブジェクト (これをクラスのインスタンスと言う) はクラスに固有な内部データを持ち、そして固有の演算規則に従うからである。従ってクラスを定義する場合には、そのクラスのインスタンスがどのような内部データを持ち、それらのデータがどのような演算でどのような変化を受けるのかを指定する必要がある。

3.8.2 プログラム例

以下の譜 3.10 にプログラム例を示す。この例では以下の仕様に従う Vector と言う名称のクラスを定義している。(クラス名は大文字で始めるのが習慣である。)

1. v=Vector(2,3,7) でベクトル (2,3,7) が生成され、それに名前 v が付けられる。
2. Vector はタプルを Vector に変換する機能をも持つ
3. Vector で生成されたデータはタプルとの混合演算を許し、その結果の型は Vector となる。
4. Vector で生成されたデータをタプルへ変換できる。

こうした仕様がプログラムの中にどのように反映されているかを見ていこう。

class Vector: --- これによりクラスの名称が与えられる。

v=Vector(2,3,7) が実行されるとクラス定義文の中の

```
def __init__(self,*t):      ①
    if len(t)==1: t=t[0]    ②
    self.v=t                ③
```

が実行される。関数名 __init__ は要素を生成する時に最初に実行される関数名として予約されているのである。

Vector(2,3,7) の引数 (2,3,7) は __init__ の t に渡される。今の場合には t=(2,3,7) である。しかし Vector の引数が 1 個のタプルとして与えられた場合には例えば t=((2,3,7)) のようになる。条件文②はこの時の対策である。

譜 3.10 クラス定義文の例

```
class Vector:
    def __init__(self,*t):
        if len(t)==1: t=t[0]
        self.v=t
    def value(self):
        return self.v
    def __add__(x,y):
        u=[]
        if hasattr(x,"v"): x=x.v
        if hasattr(y,"v"): y=y.v
        for n in range(0,len(x)):  u.append(x[n]+y[n])
        return Vector(tuple(u))
    def __sub__(x,y):
        u=[]
        if hasattr(x,"v"): x=x.v
        if hasattr(y,"v"): y=y.v
        for n in range(0,len(x)):  u.append(x[n]-y[n])
        return Vector(tuple(u))

v=Vector(2,3,7)
u=Vector(5,1,4)
s=Vector(8,2,6)
print (u+v+s).value()    # (15, 6, 17) を出力
print (v+u-s).value()    # (-1, 2, 5) を出力
z=Vector(0,0,0)
print (z + (2,3,7) + (5,1,4) + (8,2,6)).value()  # (15, 6, 17) を出力
print (z + (2,3,7) + (5,1,4) - (8,2,6)).value()  # (-1, 2, 5) を出力
```

3.8.3 self

代入文 ③ に現れる予約語 self はこれから生成するオブジェクトを表している。そのオブジェクトの内部変数 v に t を代入するのが ③ の文である。関数 __init__ の第 1 引数は self でなければならない。内部変数の名前は好きなように付けれて構わない。

x を Vector のインスタンスとせよ。その下で x.value() を実行すると次の関数が実行される。

```
    def value(self):
        return self.v
```

ここに現れる self は x を意味している。従って x.value() で内部データ v が返されるのである。関数の呼びだし方が value(x) ではない事に注意せよ。x の内部で定義されている変数 v は x.v と書くのであるから、x の内部で定義されている関数 value は x.value と表現するのが自然であると考えてこのような書き方をするのである。クラス定義文の中で定義されている関数をメソッドと言う。クラスの内部変数とメソッドをクラスの属性と言う。

3.8.4 クラス定義で使用可能な関数

加法演算が実行されると

```
def __add__(x,y):
    u=[]
    if hasattr(x,"v"): x=x.v
    if hasattr(y,"v"): y=y.v
    for n in range(0,len(x)):  u.append(x[n]+y[n])
    return Vector(tuple(u))
```

が実行される。ここに __add__ は予約語である。ここに現れる 2 つの if 文は、x, y として受け取ったデータをタプル形式に還元するために置かれている。こうする事によってプログラムがいくらか簡潔になり、同時にタプルとの混合計算を可能にしている。そして最後の

```
return Vector(tuple(u))
```

によって混合計算の結果を再び Vector として返している。

さて、このプログラムに現れた __add__ と __sub__ の他にも演算に関する予約関数が多数存在する。以下に代表的な例を挙げる。

x+y	__add__
x-y	__sub__
x*y	__mul__
x/y	__div__
x%y	__mod__
-x	__neg__
+x	__pos__
x**y	__pow__
x or y	__or__
x xor y	__xor__
x and y	__and__
x<<y	__lshift__
x>>y	__rshift__
^x	__invert__

オブジェクトの属性に関して以下の関数を利用できる。

getattr(*obj*,*name*)	属性の値を返す
hasattr(*obj*,*name*)	属性を持っているか否かを調べる
setattr(*obj*,*name*,*value*)	属性に値を設定する
delattr(*obj*,*name*)	属性を消去する

ここに *obj* はオブジェクトの名称、*name* は属性の名称 (文字列で与える)、*value* は値である。

3.9　エラーの捕捉

3.9.1　try

多くのプログラミング言語では goto 文は必要悪として認めている[7]。しかしながら Python は goto 文を持っていない。その代わりにエラー処理に関わる goto 文の機能を例外 (exception) 処理で対応している。次に示すのはエラーを発生する簡単なプログラムである。

譜 3.11 エラーを発生させるプログラム

```
def func1():
   func2(0)
def func2(x):
   y=1/x
func1()
```

このファイル名を e1.py とする。これを実行すると、

```
bash$ python e1.py
Traceback (innermost last):
  File "e1.py", line 5, in ?
    func1()
  File "e1.py", line 2, in func1
    func2(0)
  File "e1.py", line 4, in func2
    y=1/x
ZeroDivisionError: integer division or modulo
```

となり、どのような経路でエラーが発生したかを表示してくれる。

[7] goto の後に、次に実行するプログラムの場所を指定する命令を goto 文と言う。極めて強力な命令で、プログラムの持つ論理構造を破壊する。そのためにスパゲティプログラムと揶揄される分かりにくいプログラムが生成される。

3.9.2 raise

これまでのプログラミング言語ではエラー発生時のトレースバック機能はデバッガの機能であり、プログラミングとは独立していた。ところが Python ではトレースバックはプログラミングの一部なのである。その事を確認しよう。

譜 3.12 exception "Hello"

```
def func1():
   func2(0)
def func2(x):
   raise "Hello"
func1()
```

このプログラムでは raise が例外 (exception) と呼ばれる特殊なエラーを発生させている。例外の名称は "Hello" である。例外 "Hello" を受け止めるようにプログラムが構成されていない場合にはトレースバックが自動的に行われる。このファイル名を e2.py とする。これを実行すると、

```
bash$ python e2.py
Traceback (innermost last):
  File "e2.py", line 5, in ?
    func1()
  File "e2.py", line 2, in func1
    func2(0)
  File "e2.py", line 4, in func2
    raise "Hello"
Hello
```

3.9.3 except

次の譜 3.13 は例外 "Hello" を受け止めるように構成されている。このプログラムを実行すると input() がデータ入力を要求する。その時、

1 を与えると Alice が、
0 を与えると Bob が、そして
2 を与えると
0.5
Carol

が出力される。

プログラムの中の

```
    try:
```

譜 3.13 例外 (exception) を受け止めるプログラム

```
"""
exception demonstration code
keywords: raise, try, except, else
"""
def func1(x):
    print func2(x)
def func2(x):
    if x == 1:
        raise "Hello"
    y = 1.0 / x
    return y
try:
    func1(input())
except "Hello":
    print "Alice"
except:
    print "Bob"
else:
    print "Carol"
```

に続くブロック即ち、

```
    func1(input())
```

内で発生した例外は `except` で捕捉される。

例外 `"Hello"` は

```
    except "Hello":
```

で捕捉され、それ以外の例外は

```
    except:
```

で捕捉される。

例外が発生しない場合には

```
    else:
```

で処理される[8]。

3.9.4　組み込みの例外名

以下に Python のシステムに組込みの例外名を参考の為に挙げておく。(解説は省略する)

[8] この他に全ての場合について最後に実行される `finally` がある。

```
ArithmeticError
AssertionError
AttributeError
EOFError
FloatingPointError
IOError
ImportError
IndexError
KeyError
KeyboardInterrupt
LookupError
MemoryError
NameError
OverflowError
RuntimeError
StandardError
SyntaxError
SystemError
SystemExit
TypeError
ValueError
ZeroDivisionError
```

3.10 import 文 (モジュール)

実際に何かの目的の為にプログラムを書いていくと、自分の作成したある関数がとても利用価値が高くて色々なプログラムの中で利用されている場合がある。多分最初のうちはその都度、その関数をコピーして他のプログラムの中に取り込まれているだろう。しかし十分に完成度の高い関数はライブラリとして使用できる。そうすればコピーしなくても import 文で取り込めるのだ。

たとえば整数問題の好きな Bob (数学者かも知れない) は素因数分解を行う関数や最大公約数を求める関数をとてもよく使う。ところが Python にはこのような特殊な問題を扱うのに必要な関数は今のところ準備されていない[9]。Bob は幾つかの有用な関数を作成し、それらをファイル foo.py にまとめている。

ファイルには拡張子 '.py' を付けておく事が大切だ。そうすれば Bob は

```
from foo import *
```

を実行するだけで foo.py の中の全ての関数を使用することができる[10]。(foo.py を foo モジュールと言う。)

Python はハードディスク中の膨大なファイルの中から如何にして foo.py を見つけるのであろうか? これを見つけるメカニズムが必要な事は明らかである。Python は OS の環境変数 PYTHONPATH をそのために使用する。PYTHONPATH には foo.py が置かれているディ

[9] このような特殊な関数を初めから準備しているプログラミング言語は筆者の知る限り存在しない!

[10] ここではファイルの名前を foo.py として解説したが、これは単なる便宜のためである。もっとファイルの内容に即した相応しい名前を付けるべきである。

譜 3.14 最大公約数を求めるプログラム

```
#
# Number functions
#
def gcd(x,y):
    x=abs(x); y=abs(y)
    while y:
        x,y=y,x%y
    return x
```

レクトリを設定する。もしもこの環境変数を設定していないならば Bob は foo.py を、これを import するプログラムと同じディレクトリに置くか、あるいは配布された Python のライブラリと一緒に置くしかない。foo.py が汎用的な価値を持っていれば何れも好ましい方法ではないだろう。

Alice は Bob の友人であり、やはり数学が好きで Python で作成した多数の関数を持っている。Bob は Alice からファイル bar.py を貰った。Bob は

```
from foo import *
from bar import *
```

を実行した。そして Bob は何かがおかしい事に気付いた。原因は foo.py と bar.py に同一の名前の関数 gcd が存在し、その 2 つが異なる事を行っている事にあった。

異なる開発者が独立してプログラムを開発すればこのような問題が発生する。この問題に対処する為に Python は他に 2 つの import の方法を提供している。一つは簡単な

```
import bar
```

である。この場合には Bob が ファイル bar の関数を使用する時には

```
bar.gcd
```

のようにモジュールの名称を関数名の前に添える。モジュール名と関数名の間には区切り記号としてコンマを使用する。 この方法は安全ではあるが面倒でもある。もう一つの方法はモジュールから取り込む関数を個別に指定する方法である。

```
from bar import factors, prime
```

ではモジュール bar の中の関数 factors と primes を取り込むよう指定している。こうすれば Bob は factors と primes に関してはモジュール名 bar を添える煩わしさから解放される。bar の中の他の関数は bar を付けて利用することになる。

配布されている Pyhon には膨大な量のライブラリ (モジュール) が付属している。それらの一つ一つを解説する事は不可能に近い。このうち使用頻度が高い math モジュールと string モジュールだけを付録で紹介する。

第4章　ファイル

4.1　ファイルの中の数の総和

データはファイルに蓄えられている場合が多い。またデータはプログラムから独立してファイルに蓄えられるべきである。例えば次のデータが入っているファイルを考えて見よう。このファイルでは最初の一行は単なるコメントであり、計算の対象となるデータは 2 行目から始まる。データの各行には、年齢、男子人口、女子人口が書かれている。(ここでは 10 才までのデータを例示したが実際にはもっと多い。)

```
1980 91M3069  IMAI TAKAFUMI
 0    813741   772763
 1    839734   796501
 2    873852   832203
 3    896520   852643
 4    942253   895206
 5    987894   939188
 6   1043386   990115
 7   1057899  1009314
 8   1038900   986149
 9   1014408   964781
10    986691   934427
```

図 4.1: 人口調査データ (ファイル: 1980.txt)

我々の目標は、男子人口の総和と、女子人口の総和を求める事である。

ファイルの名称を 1980.txt としよう[1]。 目標が達成されるまでのプロセスは大きく 3 つに分かれる。

1. ファイルの内容をともかく読み取る事
2. データをその性格ごとに適切な変数に受け取る事
3. 和を求める事

最初の問題から解決しよう。ファイルにアクセスする方法はプログラミング言語ごとに大きく異なる。Python ではファイル 1980.txt を読み取り、そのまま表示するには次の様に行う。

[1] このファイルは Python のプログラムが置かれているディレクトリと同じディレクトリに置かれていると仮定する。さらに、そのディレクトリにおいてプログラムが実行されると仮定する。

```
f=open('1980.txt')
for s in f.readlines():
    print s,
f.close()
```

f.readlines() の f は open で返された f である。そしてこの関数はファイルの全体を行単位で区切ってリストとして返す。このリストの各要素は改行文字で終わる文字列である。上の例では f.readlines() は

```
['1980 91M3069  IMAI TAKAFUMI\n', ' 0    813741   772763\n',' 1    839734   796501\n',  ... ]
```

を返す。これはファイルの行を要素とするリストである事に注意する。\n は改行文字である。for 文によって s にはリストの中の文字列が順に代入される。もしも

```
print s
```

とすれば、s に含まれる改行文字による改行の他に、print 文自体による改行が発生する。print 文が引き起こす改行を押さえるには ‘,’ を付ける。我々の問題では第一行を読み捨てる必要がある。その為にはファイルの全てを一挙に読む readlines ではなく、一行だけを読む関数が必要である。readline がそれを行ってくれる。それを使うと、

```
f=open('1980.txt')
s = f.readline()
for s in f.readlines():
    print s,
f.close()
```

こうすれば今度は f.readlines() は

```
[' 0    813741   772763',' 1    839734   796501', ...]
```

を返す。

次に必要な事は、文字列に含まれる 3 つの数値データを取り出す作業である。この作業は Python ではちょっとやっかいである。原因はこのファイルのデータが Python で処理しやすい形式 (Python は ‘,’ で区切られたデータ形式を好む) になっていないからである。まず文字列を 3 つに分解する。string メソッドの split がこれを行ってくれる。

```
t = s.split()
```

すると t には例えば、文字列のリスト ['0','813741','772763'] が入る。しかし得られたものは、まだ文字列であって数字ではないのだ (つまり数字としての計算の対象にはならない)。数字に変換するには int を用いる。受け取る変数名を age、male、female とすると、

```
age=int(t[0])
male = int(t[1])
female=int(t[2])
```

譜 4.1 ファイルから年齢、男子人口、女子人口を読み取るプログラム

```
f=open('1980.txt')
s = f.readline()
for s in f.readlines():
    t = s.split()
    age=int(t[0])
    male = int(t[1])
    female=int(t[2])
    print age,male,female
f.close()
```

ここで t[0] はリストの最初の要素であり、ここでは '0' を意味する。t[1] は '813741'、t[2] は '772763' である。

以上の結果をまとめると、

ここまでくれば和を求めるのは簡単である。男子和と女子和を入れるための変数を準備する。これを sum_male と sum_female とせよ。すると、

譜 4.2 人口調査データを読み取り、男女別人口の和を求める

```
f=open('1980.txt')
s = f.readline()
sum_male=0
sum_female=0
for s in f.readlines():
    t = s.split()
    age=int(t[0])
    male = int(t[1])
    female=int(t[2])
    print age,male,female
    sum_male=sum_male+male
    sum_female=sum_female+female
f.close()
print sum_male, sum_female
```

問 1 実際に和を求めて見よ。

問 2 ファイルの第一行の最初の数字は調査年度を表している。この数字も和と一緒に表示せよ。

4.2 ファイルの基本操作

我々はプログラムやデータをファイルに保存する。ファイルは外部記憶装置の中に存在し、名前で参照する。ファイルの名付け規則はオペレーティングシステム毎に異なるのでここでは解説しない。現代的なオペレーティングシステムではファイルの内容は 1 バイト単位のデータの並びである。Python の文字列も 1 バイト単位のデータの並びである事を思い出そう。従ってファイルの内容は Python の文字列にそのままコピーすることができる。次のプログラムはファイル foo の内容を文字列 s に読み取り、ファイル bar を作成し s の内容を書き込んでいる。つまり foo のコピーである bar を作成している。

譜 4.3 ファイルを読み取る基本的なプログラム 1

```
f=open("foo","r")
s=f.read()
f.close()
# you can change s here if you need
f=open("bar","w")
f.write(s)
f.close()
```

多くの応用では単なるコピーではなく、何らかの変更を加えた結果を保存したいのである。このプログラムのコメント部分

```
# you can change s here if you need
```

の代わりに、必要ならば s に対する処理をつけ加える事ができる。ファイルの読み書きは open で始まり close で終わる。プログラムが終了した時にはプログラムの中で開いていたファイルは自動的に閉じられるので close 文は必ずしも必要はないが、用が終わり次第 close を実行するのは好ましいスタイルである。(何故なら、1 つのプログラムの中で同時に開く事のできるファイルの個数はあまり多くはない。またプログラムが並列に動作している環境では長時間ファイルを開いたままにしておくと弊害が発生することがある。) open の第二引数は動作モードを表しており、以下のモードが使用できる。

DOS 系の OS(Microsoft Windows の系列) では注意が必要である。この OS のテキストファイルは改行を 2 つの連続した文字コード ('\015' と '\012') で表現している。モード"r" でファイルを開いた場合には read で '\015' が読み捨てられる。逆に write で '\015' が '\012' の前に追加される。この規則をバイナリファイルに適用すれば問題が発生する事は明らかである。そこで DOS 系の OS ではテキストファイルとバイナリファイルを区別する必要がある。

テキストファイルとは人に対する可読性を考慮して作成されたファイルである。通常はテキストエディタを通じて作成されるがプログラムの処理の結果として作成されることもある。テキストファイルに含まれる文字は基本的に印字可能な文字である。制御文字は可読性を失わないもの (TAB と改行文字) しか含まない。テキストファイルは行の集まりである。処理

動作モード	意味
"r"	読み込みのみ。(省略可能)
"w"	新たに作成する。書き込みのみ。
"a"	追加書き込み。
"r+"	読み書き。
"w+"	読み書き。ファイルが既にあればその内容は捨てられる。
"a+"	読み書き。オープン直後のファイル位置はファイル終端になる。
"rb","wb","ab", "r+b","w+b","a+b"	DOS 向けのバイナリオプション。

表 4.1: open のモード一覧

は行を単位として進められる事が多い。テキストファイルを扱う場合には、譜 4.3 のプログラム 1 ではなく、行単位に読み書きする次のプログラムの方が参考になる。

譜 4.4 ファイルを読み取る基本的なプログラム 2

```
f=open("foo")
x=f.readlines()
f.close()
f=open("bar","w")
for s in x:
    # you can change s here if you need
    f.write(s)
f.close()
```

以下に Python で定義されているファイルに関する操作を表で整理する。

関数	意味
f=open(*name*)	ファイルを開く
f=open(*name*,*mode*)	*name*:　ファイルの名前
f=open(*name*,*mode*,*bufsize*)	*mode*:　動作モード (省略すれば 'r' が仮定される) *bufsize*:　バッファサイズ
f.close()	ファイルを閉じる
s=*f*.read()	ファイルを文字列 *s* に読む
s=*f*.read(*size*)	*size*:　読み取りサイズ
f.write(*s*)	ファイルに文字列 *s* を書き込む
f.seek(*offset*,*whence*)	ファイルの読み書きの位置を変更する *offset*:オフセット。この意味は次の *whence* の値で定める 0:　先頭からの位置 1:　現在の位置 2:　ファイルの終端 (EOF) からの位置
f.flush()	変更の内容をディスク上のファイルに反映させる
f.tell()	ファイルの読み書きの位置を知る
s=*f*.readline()	テキストファイルから 1 行だけ文字列 *s* に読み取る *s* の末尾は改行文字である [注 1]
x=*f*.readlines()	テキストファイルから全ての行をリスト *x* に読み取る
f.writelines(*x*)	リスト *x* をファイルに書き込む
f.isatty()	会話型ファイル [注 2] であるか否かを判定する

表 4.2: ファイルに関する関数とその使い方の一覧

注 1. テキストファイルは改行文字で終わるのが由緒正しいテキストファイルの作り方である。テキストファイルを扱うアプリケーションはこの事を前提に作成される事が多い。しかしテキストエディタ自体は必ずしもこれを強制していない。従って改行文字で終わっていないテキストファイルが存在する。この場合 readline あるいは readlines が読み取る最後の行は改行文字が末尾に付かない。

注 2. 現代的な OS ではファイル概念が抽象化され、必ずしも記録装置との関係では定義されていない。プログラミングの立場から見れば、何かに書き込まれ、あるいは何かから読み取られることだけが問題であり、書き込まれたデータが恒久的に保存されるか否かを問題にする必要はないのである。そこでファイルを広く入出力の関係で理解する。例えば会話型のプログラムは利用者がキーボードから打ち込んだ要求を文字列として読み取るが、それは会話型ファイルから読み取っていると考えるのである。

4.3 sys.stdin

sys モジュールで定義されている sys.stdin を使用するとファイル変数に対して標準入力を割り当てることができる。標準入力とはプログラムの実行の方法によって定まる入力先の事で、時にはキーボードから、また時にはファイルから、また時には他のプログラムからデータを受け取ることができる[2] 便利な機能である。標準入力からデータを読み取る様に作成されたプログラムは、プログラムの修正なしにデータの受け取り先を変更することができる。

例えば次のような問題を考えて見よう。英文では文字 e の出現頻度が高いと言われる。ポーの有名な推理小説「黄金虫」は英文の持っている文字の出現頻度の癖を基に暗号が見事に解読されていく物語である。このことを実際の英文でコンピュータの助けを借りて調べようと言うわけである。

譜 4.5 プログラム 1. count.py

```
import sys
f=sys.stdin
c=[0]*256
def count(c,s):
    for x in s: c[ord(x)]=c[ord(x)]+1
def pr(c):
    for x in range(32,127): print chr(x),c[x]
f=sys.stdin
while 1:
    s=f.readline()
    if len(s)==0: break
    count(c,s)
pr(c)
```

このプログラムは

```
f=sys.stdin
```

によって f を標準入力に割り当てている。このプログラムは

```
python count.py
```

を実行する事によって[3]。キーボードからデータを読み取る。従って、このプログラムが完成するまでは、キーボードから簡単なデータを打ち込み、プログラムが正しく文字の出現数をカウントしているか否かを確認する事ができる。一旦プログラムが完成すると、英文の入っているファイル (これを a.txt とせよ) を実行時に次の様に指定する。

[2] プログラムからデータを受け取る仕組みをパイプと言う。UNIX にはパイプの使用を前提にした豊富なツールが揃っている。Windows にも一応はパイプはある。

[3] コマンドによってプログラムを実行するには、Windows の場合にはコマンドプロンプトを使う。しかし Windows の場合には Python をインストールしても、コマンドとして実行できるように設定する事が困難である。Mac や UNIX の場合には特段の設定無しに python コマンドが使える。

```
python count.py < a.txt
```

すると、このプログラムは今度はファイル a.txt からデータを読み取ってくれる。ところでこのプログラムは大文字と小文字を別々にカウントしている。大文字を小文字に変換してその出力を count.py に渡すには (Mac や UNIX であれば)、

```
tr '[A-Z]' [a-z]' < a.txt | python count.py
```

を実行すればよい。UNIX には既に文字を変換するツール tr が存在し、それを使えば

```
tr '[A-Z]' [a-z]' < a.txt
```

でファイル a.txt から読み取った大文字を全て小文字に変換してくれる。そして記号「|」がパイプを表わしている。これによって tr によって変換されたその結果が

```
python count.py
```

に渡るのである。

第5章　基本関数

ここでは Python の組み込み関数 (基本関数) を紹介する。他の章で紹介済のものは省略した。

5.1 input

用法 input(*prompt*?)

意味 キーボードからデータを読みとる。入力を Python の式であると解釈し、その評価結果を返す。

実行時にデータを入力し、それに基づいて計算する時の Python の標準的な方法は

```
input()
```

を使用する。( ) の中には必要ならばプロンプト文字列を与えることができる。*prompt* に付けられた '?' は、これがオプションに過ぎない事を表している。BASIC(初心者向けのプログラミング言語の一つ) にも同名の命令が存在し、Python の input はそれとよく似ているが以下の点で異なる。

1. Python の input は命令ではなく関数である。
2. Python の input は入力を Python の式であると解釈し、その評価結果を返す。

例

以下に input() の使い方を示す簡単なプログラム例を挙げる。

```
while True:
    s=input("> ")
    print s
```

これを実行すると

```
>
```

が表示される。この記号は input の引数によって指定された文字列である。この文字列は利用者に対しデータ入力を促す (prompt = 促す) ために使用される。

```
> 3
3
```

即ち入力されたデータが変数 s に渡され、それが出力されている。しかし次の結果を見ると input はそれ以上の能力を備えている事がわかる。

```
> 2*3
6
```

入力された式を計算するのだ。入力は数値データだけではなく任意のデータを許す。

```
> 'alice'
alice
> ('alice',16)
('alice', 16)
> ['bob',20]
['bob', 20]
> 'alice',16
('alice', 16)
```

さらに関数も使用できる。

```
> len('alice')
5
```

但し、数学関数を使用するにはこの例のプログラムで math モジュールを予め取り込んでおく必要がある。関数 input の入力対象となるのは式だけである。例えば、代入文は文であり式ではないので

```
> a=3
Traceback (innermost last):
  File "x", line 2, in ?
    s=input("> ")
  File "<string>", line 1
    a=3
     ^
SyntaxError: invalid syntax
```

のようにエラーになる。

プログラムの中の input にデータを与える時に参照可能である。例えば次のプログラムの実行結果を見よ。

```
a=5; b=3
while True:
    s=input("> ")
    print s
```

実行結果

```
> a
5
> a*4
20
> a*b
15
```

5.2 raw_input

用法 raw_input(*prompt*?)

意味 キーボードからデータを読みとる。入力をそのまま返す。

注意 Python3 では廃止されている。

関数 input は式を評価するが、関数 raw_input は入力された文字列をそのまま返す。引数としてプロンプト文字列を与える事ができる。

例

```
while True:
    s=raw_input("> ")
    print s
```

以下にプログラムの実行例を挙げる。

```
> 2
2
> 2*3
2*3
> alice
alice
```

入力は評価されずにそのまま書き出されているのが観察されるであろう。

参考: Python には文字列として与えられた式を評価する関数 eval が存在する。eval は例えば以下の様に使用する。

```
a=5
s=eval("a*3")
```

すると s には a*3 が評価された値である 15 が代入される。関数 input() の動作は eval(raw_input()) の動作と等価である。

5.3 rstr, repr, ‘...‘

用法 str(*x*), repr(*x*), ‘*x*‘

意味 オブジェクト *x* の文字列表現

repr(*x*) と ‘*x*‘ は同じ意味である。str(*x*) は引用符を簡略化している。

例

以下は会話モードでの実行である。

```
>>> str(123)
'123'
>>> `123`
'123'
>>> str('123')
'123'
>>> `'123'`
"'123'"
>>> f
<function f at 1f81d0>
>>> str(f)
'<function f at 1f81d0>'
>>> str('<function f at 1f81d0>')
'<function f at 1f81d0>'
>>> `f`
'<function f at 1f81d0>'
>>> `'<function f at 1f81d0>'`
"'<function f at 1f81d0>'"
```

5.4 filter

用法 filter(*func*,*s*)

意味 シーケンス *s* の要素を関数 *func* によって篩い分ける。

例

```
def f(x): return x%2
v=[1,3,4,2,3,5,6]
print filter(f,v) # [1, 3, 3, 5] を出力
```

組み込み関数 filter は次の定義と等価である。

```
def filter(f,v):
    r=[]
    for x in v:
        if func(x): r=r+[x]
    return r
```

5.5 map

用法 map(*func*,*s*)

意味 シーケンス *s* の要素を関数 *func* に作用させてリストを作る。

例

```
def f(x): return x%2
v=[1,3,4,2,3,5,6]
print map(f,v) # [1, 1, 0, 0, 1, 1, 0] を出力
```

組み込み関数 map は次の定義と等価である。

```
def map(f,v):
    r=[]
    for x in v:
        r=r+[func(x)]
    return r
```

5.6 reduce

用法 reduce(*func*, *s*, *init*?)

意味 2 変数関数 *func* で指定された演算によってシーケンス *s* を計算する。*init* は演算に初期値であり、省略されれば *s*[0] が仮定される。

注意 Python3 では reduce は基本関数から外され、functools モジュールに移動している。

組み込み関数 reduce は次の定義と等価である。初期値が与えられない場合:

```
def reduce(f,s):
    t=s[0]
    for x in s[1:]: t=f(t,x)
    return t
```

初期値が与えらた場合:

```
def reduce(f,s,i):
    t=i
    for x in s: t=f(t,x)
    return t
```

例 1

```
def f(x,y): return x+y
v=[1,3,4,2,3,5,6]
print reduce(f,v) # 24 を出力。これは v の要素の和である。
```

例 2

```
T=(type(""),type((1,)),type([1]))
def g(x,y):
    if type(x) in T: x=reduce(g,x,0)
    if type(y) in T: y=reduce(g,y,0)
    return x+y
def sum(s):
    return reduce(g,s,0)
print sum([2,3,[4,5]]) # 14 を出力
print sum([[4,5]]) #  9 を出力
print sum([2]) #  2 を出力
print sum([]) #  0 を出力
```

5.7 eval

用法 eval(*code*)

意味 Python の式が書かれた文字列 *code* を評価するして式の値を求める。

例

```
a=3
print eval('a*7') # 21 を出力
```

入力関数 input() は eval(raw_input()) と等価である。

5.8 exec

用法 exec(*code*)

意味 Python のコード *code* を実行する。*code* はプログラムが書かれた文字列もしくは関数 compile によって生成されたされたコード。

例

```
exec('a=3')
print a # 3 を出力
```

5.9 execfile

用法 execfile(*file*)

意味 Python のファイル *file* を実行する。

注意 Python3 では廃止されている。

例

```
execfile('a1.py')      # ファイル a1.py を実行
```

5.10 compile

用法 compile(*code*, *label*, *kind*)

意味 Python のプログラムが書かれた文字列をコンパイルする。*label* はこのコードが実行されエラーが発生した時にエラーメッセージに表示される疑似ファイル名。*kind* は "eval" もしくは "exec"。

例

```
s="""
#
# Fibonacci series
#
x=1;y=1
n=1
while n < 100:
    print n, x
    z=x+y
    x=y
    y=z
    n=n+1
"""
c=compile(s,'Fibo','exec')
exec(c)
```

この結果は次のようになる。

```
1 1
2 1
3 2
```

```
4 3
[中略]
44 701408733
45 1134903170
Traceback (innermost last):
  File "a1.py", line 15, in ?
    exec(c)
  File "Fibo", line 9, in ?
OverflowError: integer addition
```

エラーメッセージ中の "a1.py" はこのプログラムが入ったファイル名であり、"Fibo" が関数 compile の第 2 引数である。なお、Python のソースファイルをコンパイルしてオブジェクトファイルを作成する場合には py_compile モジュールの関数 compile を使用する。

5.11 id

用法 id(*obj*)

意味 オブジェクト *obj* の ID を返す。(実際には *obj* が置かれているメモリアドレスである)

5.12 type

用法 type(*obj*)

意味 *obj* のデータ型を返す。

例

```
>>> type("")
<type 'string'>
>>> type(0)
<type 'int'>
```

5.13 hash

用法 hash(*obj*)

意味 オブジェクト *obj* のハッシュ値を返す。

5.14 callable

用法 callable(*obj*)

意味 オブジェクト *obj* が呼び出し可能[1]なら 1、そうでなければ 0 を返す。

5.15 dir

用法 dir(obj?)

意味 オブジェクト obj の属性名をリスト形式で返す。 obj が省略された場合には dir を実行した場所で参照可能な名前一覧 (変数名の一覧) を返す。

実行例

```
>>> a=[2,3]
>>> x=5
>>> dir()
['__builtins__', '__doc__', '__name__', 'a', 'x']
>>> dir(__builtins__)
['ArithmeticError', 'AssertionError', 'AttributeError', 'EOFError',
'Ellipsis','Exception', 'FloatingPointError', 'IOError', 'ImportError',
'IndexError', 'KeyError', 'KeyboardInterrupt', 'LookupError', 'MemoryError',
'NameError', 'None', 'OverflowError', 'RuntimeError', 'StandardError',
'SyntaxError', 'SystemError', 'SystemExit', 'TypeError', 'ValueError',
'ZeroDivisionError', '_', '__debug__', '__doc__', '__import__', '__name__',
'abs', 'apply', 'callable', 'chr', 'cmp', 'coerce', 'compile', 'complex',
'delattr', 'dir', 'divmod', 'eval', 'execfile', 'filter', 'float',
'getattr', 'globals', 'hasattr', 'hash', 'hex', 'id', 'input', 'int', 'intern',
'isinstance', 'issubclass', 'len', 'list', 'locals', 'long', 'map', 'max',
'min', 'oct', 'open', 'ord', 'pow', 'range', 'raw_input', 'reduce', 'reload',
'repr', 'round', 'setattr', 'slice', 'str', 'tuple', 'type', 'vars', 'xrange']
>>> dir(a)
['append', 'count', 'index', 'insert', 'remove', 'reverse', 'sort']
>>> dir(x)
[]
```

5.16 vars

用法 vars(*obj*?)

意味 オブジェクト *obj* の属性辞書を返す。 *obj* が省略された場合には vars を実行した場所での名前辞書 (local scope dictionary [注 1]) を返す。

注 1: 参照可能な名前の一覧とその値が納められているシステム辞書

[1] 関数の事

実行例

vars を使った場合の名前辞書の辿り方を示す。

```
>>> a=[2,3]
>>> x=5
>>> d=vars()
>>> print d
{'x': 5, '__doc__': None, '__name__': '__main__',
'__builtins__': <module '__builtin__'>, 'a': [2, 3]}
>>> e=d['__builtins__']
>>> print e
<module '__builtin__'>
>>> vars(e)
{'cmp': <built-in function cmp>, 'dir': <built-in function dir>,
'round': <built-in function round>,
'AttributeError': <class exceptions.AttributeError at 1f684
[以下省略]
```

関数 vars の方が関数 dir より多くの情報を返してくれている。

5.17 locals

用法 locals()

意味 locals を実行した場所での名前辞書 (local scope dictionary) を返す。

5.18 globals

用法 globals()

意味 大域的に参照可能な名前辞書 (global scope dictionary) を返す。

第III部

Python によるグラフィックス

第1章　グラフィックスの基礎

Python で線分や円等の図形を表示するには Canvas クラスを使用する。Python の Canvas クラスはキャンバスに線分や円等の幾何学図形を描くためのクラスである。キャンバスに描いた絵は eps 形式のファイルに出力できる。描画結果はプリンタの解像度で印刷され、美しい仕上がりを得る。　Python の Canvas クラスに関しては今のところ解説は皆無である[1]。しかしながら Python の Canvas クラスは Tcl/Tk を利用しているので Tcl/Tk の解説書 [4][5] から Python の Canvas クラスの使用法を推測する事ができる。また Python の Canvas の記述のシンタックスは以下の Python のソースライブラリを参照すれば判明する。

python/Lib/lib-tk/Canvas.py
python/Lib/lib-tk/Tkinter.py

さらに Python の Canvas の使用法は UNIX 版 Python のソースプログラムに付属する 2 つのデモプログラム:

Python/Demo/tkinter/Guido/
Python/Demo/tkinter/Matt/

からも窺い知ることができる。この論稿はこうした断片的な情報を基に Python におけるグラフィックスの使い方を体系的に解説している。

以下に参照を容易にする為に、この解説の構成を記す。

第 1 節 サンプルプログラム
第 2 節 キャンバス
第 3 節 座標系
第 4 節 基本図形
第 5 節 オプション (補足)
第 6 節 印刷
参考文献
付録

1.1　サンプルプログラム

Python における Canvas クラスの使用法を見るために Canvas を使用したサンプルプログラムを 1 つ紹介し、それを解説する[2]。

[1] この章は 1999 年の筆者の記事 [8] を基に最小限の修正を加えている。現在ではこの言い方は当たらない。
[2] このサンプルプログラムを Python3 で動かすには最初の行の Tkinter を tkinter と変更すればよい。

譜 1.1 プログラム sample.py

```
from Tkinter import *
import sys
def pr():
    d=c.postscript(file="a.eps")
def quit():
    tk.destroy()
tk=Tk()
f=Frame(tk)
b1=Button(f,text='write canvas to a.eps', command=pr)
b1.pack(side='left',expand=YES,fill='x')
b2=Button(f,text='quit',command=quit)
b2.pack(side='right',expand=YES,fill='x')
f.pack(fill='x')

# --- Canvas starts from this line ---
c=Canvas(width=400,height=300,background='cyan')
c.pack()

c.create_rectangle(2,2,401,301);# The maximum rectangle we can draw.
c.create_line(230,170,350,170)
c.create_rectangle(75, 30, 175,80,outline='blue')
c.create_rectangle(100, 50, 200,100,fill='#F00')
c.create_polygon(300,200,350,250,250,250,fill='yellow',width=2,outline='black')
c.create_oval(100, 200, 200,250,outline='green',width=20)
c.create_arc(250,30,350,80,start=90,extent=145,width=5,fill='pink')
c.create_text(250, 100,text='ABC',font='Times 30 bold italic',anchor='nw')
c.create_text(10,10,text='0',font='Courier 12')
c.create_line(20,10,360,10,arrow='last')
c.create_text(370,10,text='x',font='Courier 12')
c.create_line(10,20,10,270,arrow='last')
c.create_text(10,280,text='y',font='Courier 12')
c.create_line(100,150,130,180,160,150,190,180,smooth=YES,width=5,fill='blue',
    arrow='last')
for n in range(0,35):
    c.create_line(30,100+5*n,70,100+5*n)
for n in range(0,13):
    c.create_text(100+20*n,280,text=n,font="Times 12",anchor="n")

tk.mainloop()
```

Python のプログラムは任意のテキストエディタで作成し、ファイル拡張子を ‘py’ とする。以下の説明では上のプログラムはファイル sample.py に保存されているとする。sample.py を単にマウスでダブルクリックするだけで、この内容が Python インタープリタに渡され、ファイルに書かれたプログラムが実行される。

しかしながら Python のプログラムの開発過程でこの実行方法をとるのは賢明ではない。プログラムにエラーが含まれている場合には、そのエラーの所在と性質を発見できないからである。プログラム作成の過程では、初心者は Python の開発環境である IDLE を使うのが無難である。

さて、このプログラムを実行すると次の図形が表示される。

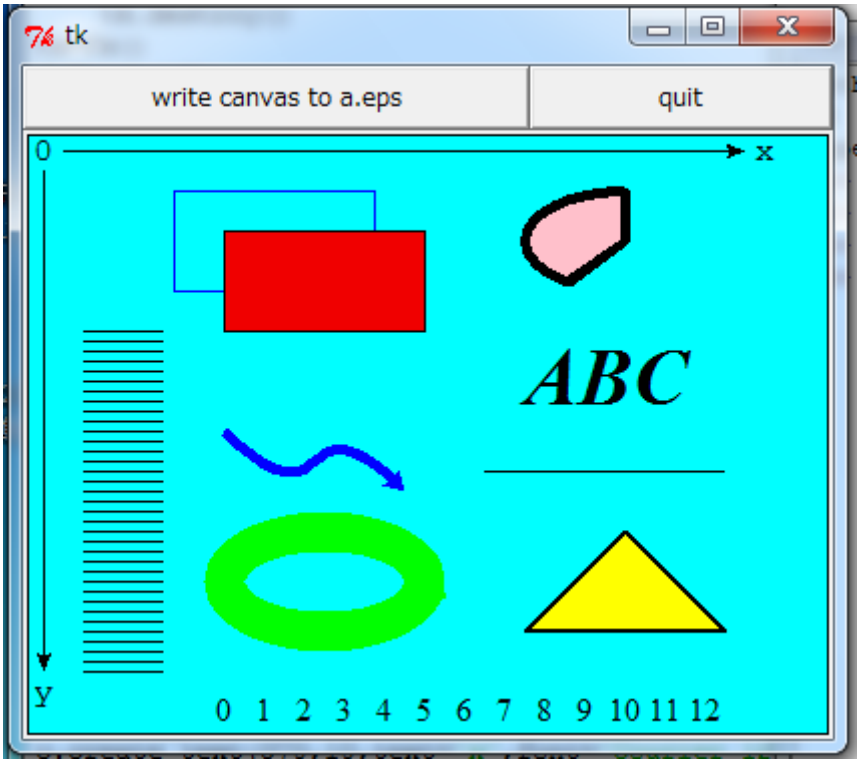


図 1.1: sample.py の実行結果

このプログラムの 19 行目

```
# --- Canvas starts from this line ---
```

まではボタンの構成と、ボタンが押された時の動作を記述している。このプログラムでは ‘write canvas to a.eps’ と書かれたボタンをマウスでクリックすればキャンバスの描画結果をファイル a.eps に書き出される。(このファイルは印刷に使用される予定である。) さらに ‘quit’ と書かれたボタンをクリックするとプログラムの実行が終了する。

しかしながらここではこの問題に関しては解説しない。従ってこの行から上の部分は所与のものとみなして欲しい。(ボタン等のユーザーズ・インターフェースの構築に関しては参考文献 [1],[2],[3] が詳しく解説している。)

1.2　キャンバス

解説は

```
# --- Canvas starts from this line ---
```

以下から始まる。Python では記号 '#' で始まる行はコメントである。コメントはプログラムの実行に関しては影響を与えない。この次の行、

```
c=Canvas(width=400,height=300,background='cyan')
```

はキャンバスを生成する。width と height によってキャンバスのサイズを指定する。サイズの単位は画素 (pixcel) である。即ち画像を構成する色を持った点の個数である。キャンバスの背景色は backgroud によって指定する。これらの指定は省略する事もできる。その場合には暗黙に仮定された値が使用される。

生成されたキャンバスは変数 c によって参照される。キャンバス c に対する操作は

```
c.pack()
```

のように、c の次にピリオドを書き、その後に操作名を書く。ここでは c を pack している。この操作によって c が張りつけられる。(従って見えるようになる。) ここでは追加の方法を指定していない。この場合にはキャンバスを (ボタンの) 下に追加する。

後に見る様にキャンバスに描画する際には必ず Canvas によって生成された変数を指定する事になる。この変数はキャンバスの名前なのである。名前を指定してキャンバスに描画するという事は、Python では一つのプログラムの中で複数のキャンバスを扱えるという事である。

Canvas には width, height, background 以外にも多数のオプションが指定できる。ここではキャンバスの見栄えを変更する為のオプション relief だけを紹介する。relief は borderwidth と一緒に使用される。

```
c=Canvas(width=400,height=300,background='cyan')
```

を

```
c=Canvas(width=400,height=300,background='cyan',relief='raised', borderwidth=6)
```

に置き換えてプログラムを実行してみよう。キャンバスが浮き上がったように見えるはずである。relief には 'raised' の他に、'sunken', 'groove', 'ridge', 'flat' が指定できる。次の図は、これらの指定よってキャンバスの境界がどのように見えるかを示す。

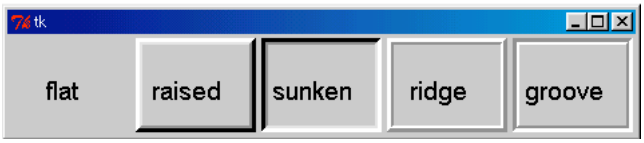


図 1.2: いろいろなレリーフ
レリーフ (relief) の指定によるキャンバスの境界の違いが図に示されている。
(順に、flat, raised, sunken, groove,ridge)

relief を指定しない場合には 'flat' が仮定される。borderwidth は適度に調整する。

サンプルプログラムの最後の行の

```
tk.mainloop()
```

は Canvas を使用したプログラムでは必須である[3]。もしもこの行が存在しないならプログラムは直ちに終了し、結果を鑑賞する暇は無いであろう。このプログラムはこの行が存在する事によってユーザ側のマウスによるアクションを待っているのである。

1.3 座標系

キャンバスに図形を描く場合には描画位置を指定する必要がある。座標は左から右方向に x 軸、上から下方向に y 軸である。座標原点はキャンバスの左上隅である。しかしこの言い方は厳密ではない。実際にはキャンバスの左上隅の座標は (2,2) となっている。プログラム 1 では、

```
c.create_rectangle(2,2,401,301)
```

によってこの事実を確認している。

rectangle とは矩形 (くけい) のことである。矩形とは水平線と垂直線によって囲まれた特殊な長方形の事である。そして Python では create_rectangle が矩形を描く命令文となっている。この例では create_rectangle の引数に現われる (2,2) は矩形の左上隅の座標、(401,301) は矩形の右下隅の座標である。そしてこれがこのキャンバスに描ける一番大きな矩形である。

読者は何故キャンバスの左上隅を座標 (0,0) として定義しないのか不思議に思うであろう。そもそもここに現われる 2 と言う数字は何を意味しているのか? この疑問に解決するには

```
c=Canvas(width=400,height=300,background='cyan')
```

を

```
c=Canvas(width=400,height=300,background='cyan',highlightthickness=10)
```

に置き換えてプログラムを実行してみるがよい。2 の数字の意味が分かるであろう。

紙に印刷する場合には画素をサイズの単位にするのは不便である。画素の間隔はディスプレイに依存しており、実際の印刷イメージを掴みにくい。画素の代わりにセンチメートルを単位として指定する事もできる。その場合には、

```
width="8c"
```

の様に、数字の後に c を付け、引用符で括る。同様にミリメートル、インチ、ポイントで指定する事もできる。各々

```
width="80m", width="3.15i", width="234p"
```

のように指定する。なお 1 インチは 2.54 センチメートルである。また 1 ポイント は 1/72 インチであり、この単位は文字フォントを指定する場面で使用されている。

Windows の場合、1 インチは 96 画素である[4]。Canvas で width をインチ単位で与えて、

[3] ここに現れる tk はプログラムの最初の方に現れた tk=Tk() の左辺である。

[4] Mac や UNIX では 72 画素である。なお Windows 7 からは 1 インチの画素数はユーザの好みで設定できるようになった。詳しくは次を見よ。
http://windows.microsoft.com/ja-JP/windows7/DPI-setting

```
print c.cget('width')
```

を実行させればキャンバスの幅を 画素単位で出力してくれる。ここに c はキャンバスである。キャンバスの幅を 10 インチに設定しこの実験を行えば 1 インチ当たりの画素数を得る。

技術の発展によって、ディスプレイは高密度化する傾向がある。そのために Python で指定したサイズが実際にディスプレイ上でどのような大きさに見えるかはディスプレイ毎に異なる。従って 5 センチメートルの長さを指定してもディスプレイ上の長さは 5 センチメートルから大きく離れている場合が多いのである。(但し印刷の場合には正確に 5 センチメートルで描く。)

数学的な問題にはキャンバスの左上隅を座標原点とする座標系は扱いにくい。実は Python は長さの単位、座標原点、座標の正の方向を変更する機能を持っている。これに関しては第 5 節オプション (補足) の tags で述べる。

1.4　基本図形

キャンバスには以下の基本図形を描くことができる。

create_line	線分、折れ線、曲線
create_rectangle	矩形 (くけい)
create_polygon	多角形
create_oval	楕円
create_arc	円弧
create_text	文字列
create_bitmap	画像 (2 色)
create_image	画像 (カラー)
create_window	窓

キャンバスに図形を描くにはキャンバスの名称、図形の位置、さらに大きさ等の形状を特徴付ける情報を与える必要がある。以下にこれらの図形を描く命令を個別に解説する。(但しここでは create_bitmap と create_image と create_window に関しては解説を省略する。)

1.4.1　create_line

create_line は線分、折れ線、曲線を描く。単に 2 つの座標点を指定すると create_line は線分を描く。例えば、

```
c.create_line(230,170,350,170)
```

は座標 (230,170) と (350,170) を結ぶ線分をキャンバス c に描く。色を指定していないので黒色で描くことになる。描画オプションを付ける事もできる。例えば、

```
c.create_line(20,10,360,10,arrow='last')
```

は座標 (20,10) と (360,10) を結ぶ線分に矢印を付けて描く。矢印の指定は

```
'none', 'first', 'last', 'both'
```

の 4 つが可能である。'none' の場合には矢印を付けない。arrow の指定が無い場合には 'none' が仮定される。次の図は arrow の指定の意味を示している。何れも左から右に向かって描いた時の直線である。

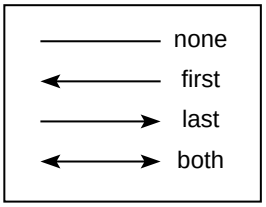


図 1.3: arrow の指定

create_line は線分を描くだけではない。折れ線や曲線も描く事ができる。例えば

```
c.create_line(100,150,130,180,160,150,190,180,smooth=YES,width=5,
    fill='blue', arrow='last')
```

は create_line の複雑な使い方である。もしも単に

```
c.create_line(100,150,130,180,160,150,190,180)
```

となっていれば、これは 4 つの座標 (100,150) と (130,180) と (160,150) と (190,180) とを結ぶ折れ線を表している。width=5 によってこの折れ線の線幅は 5 となる。fill='blue' によってこの折れ線は青色に描かれる。smooth=YES とすることによって線分を滑らかにした図形、即ち曲線を描く。

次の図は smooth の効果を示している。この図の曲線図形は単に create_line で描いた折れ線図形に smooth を指定しただけである。基になった折れ線も参考の為に合わせて示してある。

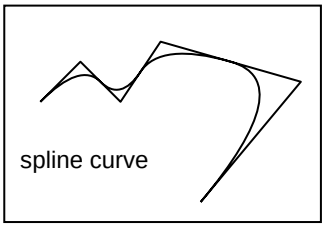


図 1.4: smooth 指定による図形の変化

smooth の指定によって生成される曲線は

1. 端点は通る
2. 中間の折れ線に関しては、その中点を通る
3. 折れ線には接する

4.　接点での曲率 (曲がり具合) は 0 である。
の性質を持っている事が分る。

smooth=YES を指定した場合には平滑の程度を splinesteps で指定できる。しかしながら splinesteps の指定は印刷結果には反映されない。従ってこの解説は省略する。

まとめ　create_line で指定可能なオプションの一覧

arrow	'none', 'first', 'last', 'both'
capstyle	'butt', 'projecting', 'round'
fill	'red', 'green', ...　などの色
joinstyle	'bevel', 'miter', 'round'
smooth	YES, NO
splinestep	1, 2, 3, ..
stipple	'gray12', 'gray25', 'gray50', 'gray75'
width	1, 2, 3, ..
tags	(第 5 節の tags を参照せよ)

1.4.2　create_rectangle

create_rectangle は矩形 (くけい) を描く。矩形とは、2 本の水平線と 2 本の垂直線によって囲まれる長方形の事である。矩形を描くには対角の 2 つの座標点を指定する。例えば、

```
c.create_rectangle(75, 30, 175,80,outline='blue')
```

によって座標 (75,30) と (175,80) を対角座標とする矩形が描かれる。

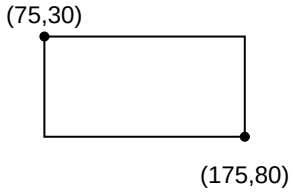


図 1.5: c.create_rectangle(75,30,175,80)

outline='blue' によってこの矩形は青色の輪郭で囲まれる。

```
c.create_rectangle(100, 50, 200,100,fill='#F00')
```

は内部を赤で塗り潰している。fill の指定が塗り潰し指定である。#F00 は赤を意味している。

大抵の印刷機は現状では色を出せない。その場合でも印刷機は色を明暗に従って適度なグレーで表現してくれる。塗り潰しを色で指定する代わりに塗り潰しのパターンを指定することもできる。その為には、

```
c.create_rectangle(100,50,200,100,fill='black',stipple='gray12')
```

などとする。stipple によって塗り潰しパターンが指定されている。指定可能なパターンとそれらの見え方については付録に解説されている。

まとめ create_rectangle で指定可能なオプションの一覧

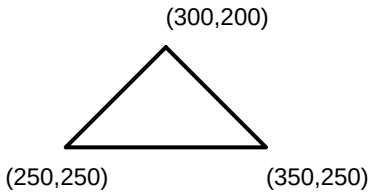
fill	'red', 'green', ... などの色 (塗り潰しの色)
outline	'red', 'green', ... などの色 (外枠の色)
stipple	'gray12', 'gray25', 'gray50', 'gray75'
width	1, 2, 3, ... (外枠の幅)
tags	第 5 節の tags を参照せよ

1.4.3 create_polygon

create_polygon は多角形あるいは閉曲線を描く。多角形を描くには頂点の座標を指定する。例えば、

```
c.create_polygon(300,200,350,250,250,250,fill='yellow',width=2,
outline='black')
```

によって 3 つの座標点 (300,200) と (350,250) と (250,250) を結ぶ多角形が描かれる。ここでは輪郭の太さが width=2 で指定されている。




```
c.create_polygon(300,200,350,250,250,250,
      fill='yellow',width=2,outline='black')
```

図 1.6: create_polygon での座標の指定の方法

5 角形を描きたい場合には座標点の個数を 5 つに増やす。この様にして任意の多角形を描くことができる。

閉曲線を描くには create_line で行った様に smooth=YES を指定する。(この下で splinestep も有効になる。) 次の図は先の三角形に対して smooth=YES を指定したものである。基になった三角形も参考の為に合わせて示してある。

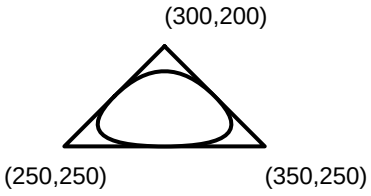


図 1.7: smooth の効果

まとめ　create_polygon で指定可能なオプションの一覧

fill	'red', 'green', ...　などの色 (塗り潰しの色)
outline	'red', 'green', ...　などの色 (外枠の色)
smooth	YES, NO
splinestep	1, 2, 3, ..
stipple	'gray12', 'gray25', 'gray50', 'gray75'
width	1, 2, 3, ...　(外枠の幅)
tags	第 5 節の tags を参照せよ

1.4.4　create_oval

create_oval は楕円を描く。円は楕円の特殊な場合として扱われる。create_oval で描かれる楕円は長径と短径が x 軸あるいは y 軸に平行である。楕円の大きさと位置を定める情報は楕円に外接する矩形を指定する事によって与える。例えば、

```
c.create_oval(100, 200, 200,250,outline='green',width=20)
```

によって楕円が描かれる。この楕円は

```
c.create_rectangle(100,200,200,250)
```

に内接する。

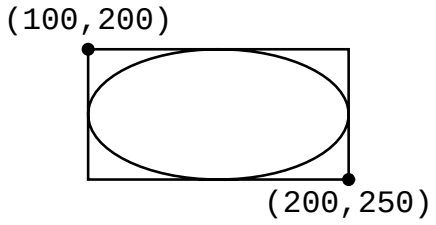


図 1.8: c.create_oval(100,200,200,250)

まとめ　create_oval で指定可能なオプションの一覧)

fill	'red', 'green', ... などの色 (塗り潰しの色)
outline	'red', 'green', ... などの色 (外枠の色)
stipple	'gray12', 'gray25', 'gray50', 'gray75'
width	1, 2, 3, ... (外枠の幅)
tags	第 5 節の tags を参照せよ

1.4.5 create_arc

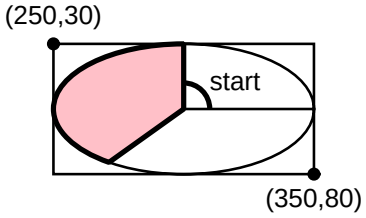
create_arc は扇形または扇形の弧を描く。create_arc が描く扇形は Oval を基にしている。即ち、Oval の中心を通る 2 つの直線で Oval を切り取った図形が Arc である。例えば、

```
c.create_arc(250,30,350,80,start=90,extent=145,width=5,fill='pink')
```

によって扇形が描かれる。この扇形の弧は

```
c.create_oval(100, 200, 200,250)
```

の上にあり、弧の開始角と弧の開き角は start と extent で与えられている。角度の単位は度である。



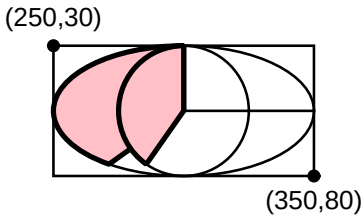
c.create_arc(250,30,350,80,start=90,extent=145,width=5,fill='pink')

図 1.9: create_arc で指定する座標の意味

図をよく見て見よう。注意深い読者は、この図で弧の開きが本当に 145 度にはなってはいない事に気付くであろう。弧の開始角は正しく 90 度になっている。実は開始角が正しく表示されたのは偶然なのである。この図の扇型は楕円を基礎にして作成されている。この楕円を横方向に一様に圧縮すると図 1.10 に示す円ができ上がる。図 1.10 ではこの円を上側に示している。

この図を見て分るように、create_arc の start と extent で示される開始角と弧の開き角は、円に矯正された扇型に対して、正しい数値を表す。

create_arc は style を指定する事によって図 1.11 に示す図形を生成する。




```
c.create_arc(250,30,350,80,start=90,extent=145)
c.create_arc(275,30,325,80,start=90,extent=145)
```

図 1.10: 円との比較で見る角度の指定

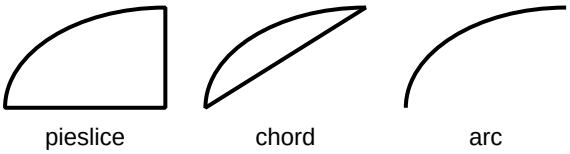


図 1.11: style の効果

まとめ　create_arc で指定可能なオプションの一覧

extent	数字 (開きの角度)
fill	'red', 'green', ...　などの色 (塗り潰しの色)
outline	'red', 'green', ...　などの色 (外枠の色)
start	数字 (開始角)
stipple	'gray12', 'gray25', 'gray50', 'gray75'
style	'pieslice', 'chord', 'arc'
width	1, 2, 3, ...　(外枠の幅)
tags	第 5 節の tags を参照せよ

1.4.6　create_text

create_text は文字列を表示する。例えば、

```
c.create_text(250, 100,text='ABC',font='Times 30 bold italic',
    anchor='nw')
```

は文字列 ABC を座標 (250,100) に書く。font は文字の書体と大きさを指定する。ここでは anchor='nw' に注意しよう。文字列 ABC を (250,100) の位置に書くといっても色々な意味に解釈できる。例えば点 (250,100) を表示文字列 ABC の中央に来る様に書くのか、

左端に来る様に書くのか等の解釈の多様性である。Python ではこの問題を anchor の指定によって解決する。anchor には

'n', 'ne', 'nw', 's', 'se', 'sw', 'e', 'w','center'

を指定できる。指定が無い場合には 'center' が採用される。'center' 以外に現われる e, w, s, n は東西南北の意味である。これらは座標が、表示される文字列に対してどの位置なのかを指定している。その際、地図と同様に、上方向を北、右方向を東と考えている。同一の座標を指定し、 anchor の指定を変化させた場合に結果がどのように変化するかを図 1.12 に示す。ディスプレイ上での結果と、eps ファイルを通じて印刷した結果とは微妙に異なっている。図では左がディスプレイでの結果であり、右が印刷結果である。指定した座標点は各々の図の中央部の太い点で示されている。但しこれは Windows に固有な問題であり、MacOSX や UNIX ではこのような問題は発生しない。

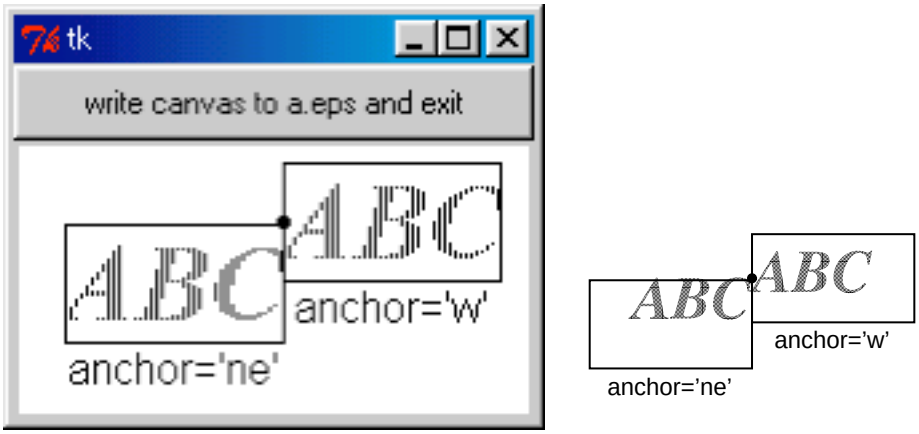


図 1.12: anchor の指定による文字列の表示位置の違い

anchor='ne' と anchor='w' で同一の座標が指定されている。見え方はディスプレイでの表示と eps ファイルを通じて印刷した場合では異なる。右が左のディスプレイの表示を印刷した結果である。

キャンバスに複数行に渡る文字列を書きたい場合がある。その場合にはその文字列を一旦変数に格納した方が分りやすい。Python では複数行に渡る文字列は普通の引用符 (')、または二重引用符 (") を 3 つ並べて実現する事ができる。例えば、

```
s='''My name is Alice.
I am waiting for your mail.
Thank you.'''
c.create_text(250,100,text=s,font='Times 12',anchor=nw,justify='center')
```

等のようにすればよい。ここに現われている justify は行揃えを指定している。justify の値は 'center' の他、'left' と 'right' を指定できる。justify による指定を省略した場合には 'left' が仮定される。次の図は justify の効果を示している。

My name is Alice.
I am waiting for your mail.
Thank you.

My name is Alice.
I am waiting for your mail.
Thank you.

My name is Alice.
I am waiting for your mail.
Thank you.

図 1.13: justify の効果
上から順に left, center, right

まとめ create_text で指定可能なオプションの一覧

anchor	'nw', 'n', 'ne', 'w', 'center', 'e', 'sw', 's', 'se'
fill	'red', 'green', ... などの色 (塗り潰しの色)
font	'Times 12', ... など書体と大きさの指定
justify	'left', 'right', 'center'
stipple	'gray12', 'gray25', 'gray50', 'gray75'
text	'ABC' など表示したい文字列
width	100 など、一行のサイズ (このサイズで折り畳まれる。0 で折畳なし)
tags	第 5 節の tags を参照せよ

1.5　オプション (補足)

1.5.1　font

さてプログラム 1 の

```
c.create_text(250, 100,text='ABC',font='Times 30 bold italic',anchor='nw')
```

では書体が

```
font='Times 30 bold italic'
```

によって指定されている。実はこのフォントの指定の仕方はちょっと古いのである。Python の最近の配布ファイルに付属のプログラム例では

```
font=('Times', 30, 'bold', 'italic')
```

のようにタプルを使う書き方になっている。でも筆者はこの書き方は面倒で嫌いである。

font の指定には、書体の名称、文字サイズをこの順に指定し、その後に太字なら bold を、斜体なら italic を指定する。文字サイズの数値はポイントである。1 ポイントは 1/72 インチであり、キャンバスを構成する場合に使用される画素を単位としたピクセル値とは異なることに注意する。PDF 形式の文書で使われている代表的なフォントは

```
Times    Helvetica    Courier
```

である。使用可能なフォントはシステム毎に異なる。実際にどのようなフォントが使用可能であるかはワープロなどのフォントメニューを調べれば分かる (図 1.14)。

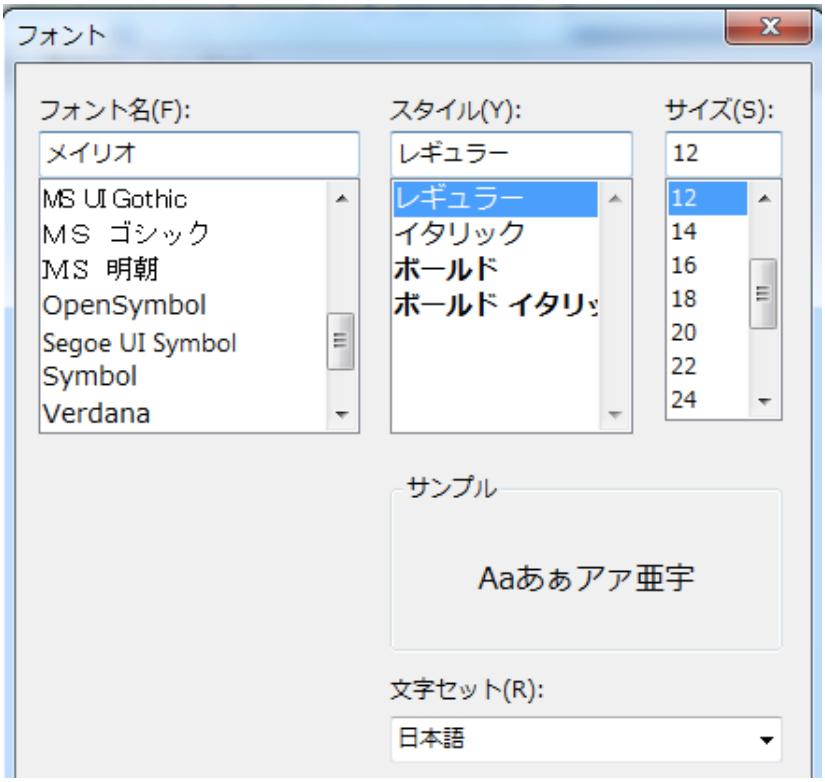


図 1.14: Window 7 のフォントメニュー

Windows のフォントメニューを見るとフォント名称の中に空白文字を含むものがある。例えば

```
Times New Roman
ＭＳ 明朝
```

などである。このような場合に

```
font='ＭＳ 明朝'
```

と指定すると実行に失敗するであろう。何故なら Python は「ＭＳ」 をフォント名称であると解釈し、その次には文字サイズを表す数字が来ると期待するが、その期待が裏切られるからである。フォント名称に空白が含まれる場合には空白を

```
font='Times-New-Roman'
font='ＭＳ-明朝'
```

のようにハイフンで置き換えれば認識してくれる。

フォントはビットマップフォントとアウトラインフォントに分類される。ビットマップフォントはディスプレイに小さな文字を表示させたい場合には良くデザインされているが、大きな文字を表示する場合やプリンタへの出力では印字品質が落ちる。従って可能な限りアウトラインフォントを指定した方が良い。Windows ではアウトラインフォントは TrueType フォントと呼ばれ、フォントメニューでは T_T で示されている。

印刷を必要とする場合にはフォントの選択にさらに注意が必要である。Python 自体は印刷機能を持っていない。しかし表示結果を EPS 形式のファイルに落とす事ができる。EPS 形式は PostScript 形式の親戚である。従って PostScript 形式のフォントが使用され、これは TrueType フォントと同じではない。Windows に備わっているフォントが必ずしも効力を持っているとは限らないのである。

印刷をも視野に入れた時に安全なフォントは限られてくる。現状では Times、Helvetica、Courier だけであると割り切った方がよい。その場合には残念ながら日本語は使えない。

1.5.2 color

Python では以下の色名称が使用できる。

```
black    darkgray    gray    lightgray    white    snow    seashell
AntiqueWhite    bisque    PeachPuff    NavajoWhite    LemonChiffon
cornsilk    ivory    honeydew    LavenderBlush    MistyRose    azure
SlateBlue    RoyalBlue    blue    DodgerBlue    SteelBlue    DeepSkyBlue
SkyBlue    LightSkyBlue    LightSteelBlue    LightBlue    LightCyan
PaleTurquoise    CadetBlue    turquoise    cyan    SlateGray
DarkSlateGray    aquamarine    DarkSeaGreen    SeaGreen    PaleGreen
SpringGreengreen    chartreuse    OliveDrab    DarkOliveGreen    khaki
LightGoldenrod    LightYellow    yellow    gold    goldenrod
DarkGoldenrod    RosyBrown    IndianRed    sienna    burlywood    wheat
tan    chocolate    firebrick    brown    salmon    LightSalmon
orange    DarkOrange    coral    tomato    OrangeRed    red    DeepPink
HotPinkpink    LightPink    PaleVioletRed    maroon    VioletRed
magentaorchid    plum    MediumOrchid    DarkOrchid    purple
MediumPurple    thistle
```

gray に関しては gray30 のように数字で暗さを指定できる。gray0 で黒、gray100 で白である。gray 系列ではない他の色に関しては 1, 2, 3, 4 の数字を色名の後に付ける事が可能である。

さらに Pyhton では 16 進数文字 (0, 1, 2, ···, 9, A, B, ···, F) で色を指定する事もできる。光の三原色は赤、緑、青である。次の、

```
c.create_rectangle(100, 50, 200,100,fill='#F00')
```

の fill='#F00' に見られる F と 0 と 0 はこの順に色を構成する赤と緑と青の光の強さを表している。ここでは各々の色の強さが 1 桁の 16 進数で表されている。F が一番強く、0 は一番弱い。一番弱いと言う事はその色が含まれていないことを意味している。従って #F00 の色は赤色であり、しかも一番明るい赤色である。ここでは各々の色が 16 諧調で表現されている。キャンバスに図形を描く場合にはこれで十分な事も多いが、実は Python は微妙な色合いを出すのにもっと木目細かく諧調を表現する事もできる。例えばオレンジ色は #FFA400 である。ここでは 6 個の 16 進文字が並んでいる事に注意する。この場合には各々の色の強度は 2 桁の 16 進数で表現されており、ここでは赤が FF、緑が A4、青が 00 である。Python は各々の色を 3 桁の 16 進数で表現する事もできる。その場合には # の後に 9 個の 16 進文字が並ぶ。

指定した色が実際にどのように見えるかを知る為には Python のプログラムを作って実際に色を表示してみるのがよい。付録の color1.py あるいは color2.py を実行すれば色名称と実際の見え方の対応が分かる。

Python の色指定は Tk に従っている。 Tk はまた UNIX の X ウィンドウに従っている。X ウィンドウの色指定は文献 [6] に解説されている。

1.5.3 stipple

Python では塗り潰しのパターンを stipple によって指定できる。パターンとしては

```
'gray12'
'gray25'
'gray50'
'gray75'
```

が指定できる。パターンは

```
c.create_rectangle(100,50,200,100,fill='black',stipple='gray12')
```

の様に、文字列として与える。

次の図は各々のパターンの印刷結果である。

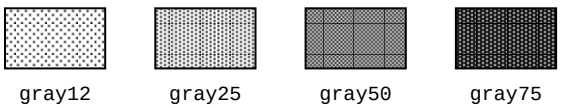


図 1.15: stipple の指定による塗り潰しパターンの違い

1.5.4　capstyle

capstyle は線の端点の形状を表す。capstyle は create_line におけるオプションであり、

```
c.create_line(230,170,350,170,width=20,capstyle='round')
```

の様に、文字列として指定する。形状としては次のものが指定できる。

```
'butt'
'projecting'
'round'
```

以下のそれらの違いを示す。

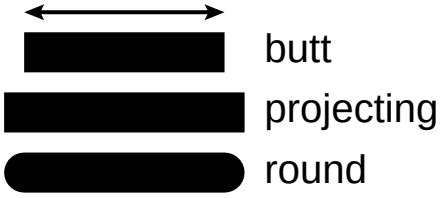


図 1.16: capstyle の指定による線端の違い

butt と projecting はよく似ている。butt の切り取り位置は projecting よりも線幅の半分だけ短い。capstyle は太い線でのみ実際上の意味を持つ。

1.5.5　joinstyle

joinstyle は線の接続点での形状を表す。capstyle と同様に形状を文字列として指定する。指定できる文字列は

```
'bevel'
'miter'
'round'
```

の 3 つである。次の図は指定の違いによる形状の違いを示している。

1.5.6　tags

Python ではキャンバス上の図形の幾つかを纏めてグループ化する機能を持っている。グループ化された図形は、あたかも一個の図形のように、大きさを変えたり、移動したり、削除したりできる。Python のグループ化は構成要素にタグ (札) を付けることによって行われる。同じタグを付けられた図形は同一のグループに属していると見なされるのである。例え

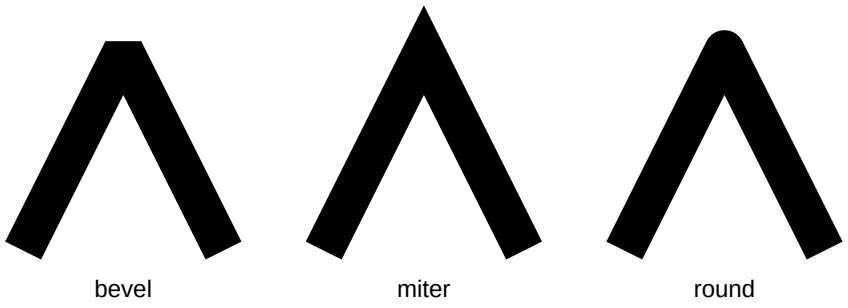


図 1.17: joinstyle の指定による接合点の違い

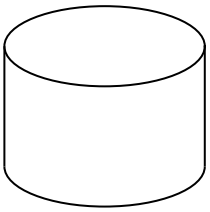


図 1.18: 描きたいドラム図形

ば次のドラム形の図形は、 1 個の oval 、 1 個の arc 、それと 2 個の line から構成されている。これらが纏まって 1 個の図形が構成されているのである。

この図は例えば次のように作成できる。(これはプログラムの全てではない。sample.py の「# --- Canvas starts from this line ---」以下の行である。)

```
c=Canvas(width=200,height=150,background='cyan')
c.pack()
c.create_oval(-1,1,1,0.2,tags='drum')
c.create_arc(-1,-1,1,-0.2,start=180,extent=180,style='arc',tags='drum')
c.create_line(-1,-0.6,-1,0.6,tags='drum')
c.create_line(1,-0.6,1,0.6,tags='drum')
c.scale('drum',0,0,50,-50)
c.move('drum',100,75)
tk.mainloop()
```

ここではこの図形には分りやすく 'drum' と言うタグを付けているが、この名称に拘る必要はない。このプログラムではドラムをデザインするに当たって、次の図 1.19 に示す図案を頭の中に描いている。

注目して欲しい点が幾つかある。この図案では

1. 上向きの方向が y の正の方向である。

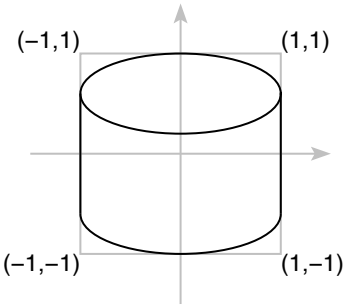


図 1.19: ドラム図形の図案

2.　原点 (0, 0) を中心にデザインされている。
3.　長さの単位が画素数ではなく、抽象化された大きさになっている。

即ち、キャンバス本来の原始的な座標系よりも洗練され、扱い易くなっているのである。このトリックを可能にしたのは

```
c.scale('drum',0,0,50,-50)    --- [a]
c.move('drum',100,75)         --- [b]
```

の 2 つである。

scale は座標系を定める。抽象的な 1 の長さは [a] によって 50 画素 のサイズであると定められている。[a] に現われる 4 つの数字の内、最後の 2 つ $(50, -50)$ が各々 x 方向、y 方向の拡大率を表しているのである。y 方向の拡大率を負の値に設定する事によって y 方向の向きが反転する。従って上方向を y の正の方向と考えてデザインする事ができる。

[a] の最初の 2 つの数字 $(0, 0)$ は座標の原点を定めている。そして $(0, 0)$ の場合には原点の移動は発生しない。$(0, 0)$ でない場合の解説は後に回す。

原点を中心に描かれた図形は [b] によってキャンバス上 $(100, 75)$ だけ移動される。この時の座標はキャンバス本来の座標である。それ故に下方向が y の正の方向となっている。

今の問題では [a] と [b] を次の 1 個に纏める事が可能である。

```
c.scale('drum',-2,1.5,50,-50)    --- [c]
```

ここに現われる scale の最初の 2 つの数字 $(-2, 1.5)$ はキャンバスの左上隅の座標を表している。この座標の数字は図案 (図 1.19) を基にしている。

上のプログラム例では個々の図形にタグを付けている。そのためにコーディングが面倒になっている。Python では様々な方法でタグを付けることができる。ここでは二つの方法を紹介する。

一つは矩形 (くけい) 領域を指定しその中に含まれる全ての図形にタグを付ける方法である。それには addtag_enclosed を使用する。

```
c.create_oval(-1,1,1,0.2)
c.create_arc(-1,-1,1,-0.2,start=180,extent=180,style='arc')
c.create_line(-1,-0.6,-1,0.6)
c.create_line(1,-0.6,1,0.6)
c.addtag_enclosed('drum', -1.1,-1.1,1.1,1.1)
```

ここではドラムが完全に含まれる様に矩形領域を少し大きめに採っている。

定義済タグ 'all' が存在する。このタグは既に全ての図形に対して付けられている。タグ 'all' は関数のグラフのような数学的な問題を扱う場合に便利である。

1.6　印刷

Python では background で指定されたキャンバスの色は印刷されない。また、印刷に Ghostscript を借用しているために[5]、ディスプレイに表示できるフォントが必ずしも使用できない。さらに文字サイズの考え方が、ディスプレイに表示する時と、印刷する時とでは異なっている[6]。この違いを図に示す。

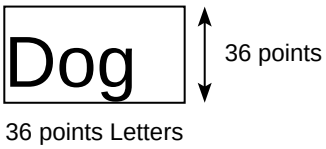


図 1.20: フォントで指定した文字の大きさと印刷結果の比較
ディスプレイ上では図に示した長方形の大きさに見える。
(図 1.12 も同様な問題を扱っているので参照せよ。)

この図は印刷結果である。OS に Windows を使用した場合には、ディスプレイでの表示では Dog の文字は丁度図の四角の枠ぐらいの大きさだったのである。

サンプルプログラムの描画結果を印刷するには以下の手順をとる。

1. 'write to a.eps' と書かれたボタンをマウスでクリックする。するとサンプルプログラムが置かれているディレクトリに、ファイル a.eps が生成される。
2. ファイル a.eps を Ghostview にドラッグする。すると Ghostview にキャンバスの図形が再び表示されるであろう。
3. Ghostview から印刷メニューを選ぶ。するとプリンタ一覧が表示されるであろう。ここで実装されているプリンタに合わせてプリンタを選ぶ。もしも希望するプリンタ名称がこの一覧に載っていない場合には、そのプリンタと同じ系列のプリンタ名称を捜すのがよい。印刷可能な場合がある。例えば Canon の BJC-80v は bjc800 で印刷可能である。

[5]MacOSX の印刷環境はとてもよい。Ghostscript 使わずに済むのだ。
[6]これは Windows での現象である。MacOSX や UNIX ではこのような問題は発生しない。

関連図書

[1] Mark Lutz 著、飯坂剛一監訳、村山敏夫、戸田英子共訳『Python 入門』(O'Reilly Japan, 1998)

[2] Mark Lutz 著、飯坂剛一監訳、村山敏夫、戸田英子共訳『Python プログラミング』(O'Reilly Japan, 1998)

[3] Mark Lutz ''Programming Python'' (O'Reilly & Associates,Inc. 1996)

[4] John K.Ousterhout 著/ 西中芳幸、石曾根信共訳『Tcl&Tk ツールキット』(ソフトバンク、1995)

[5] 宮田重明、芳賀敏彦『Tcl/Tk プログラミング入門』(オーム社、1995)

[6] V.Quercia,T.O'Reily/ 大木敦雄監訳『X ウィンドウ・システム・ユーザ・ガイド』(ソフトバンク、1993)

[7] 江口庄英『Ghostscript Another Manual』(ソフトバンク、1997)

[8] 有澤健治「Python によるグラフィックス」(「Com」Vol.10,No.2、愛知大学情報処理センター、1999 年 9 月)

付 録A　基本演算と基本関数

A.1　演算子と算術関数

演算	説明
x+*y*, *x*-*y*	加算、減算
*x***y*, *x*/*y*, x%y	乗算、除算、剰余
-*x*,+*x*	符号
x|*y*, *x*^*y*, x&*y*	ビット演算。順に OR, XOR,AND 演算
x<<*n*, x>>*n*	ビットシフト演算
~*x*	ビット反転
abs(*x*)	絶対値
int(*x*)	*x* の整数部分あるいは整数への変換
float(*x*)	実数への変換
round(*x*)	実数 *x* を丸める (四捨五入する事)
round(*x*,*n*)	実数 *x* を小数点 *n* 桁で丸める
divmod(*x*,*y*)	(*x*/*y*, *x*%*y*) 除算、剰余のペア計算
*x****y*, pow(*x*,*y*)	ベキ乗計算 (どちらも同じ意味)
pow(*x*,*y*,*z*)	pow(*x*,*y*)%*z*

表 A.1: 演算子と算術関数

A.2 基本関数

演算	関数の機能と返される値
chr(*i*)	文字コードが *i* の文字 (1.3)
ord(*c*)	文字 *c* のコード (1.3)
int(*s*,*base*?)	文字列 *s* を整数に変換する。 *base* は基数であるが、省略されると 10 進数で変換する。
max(*s*)	シーケンス *s* の最大値
max(x_1,x_2,...)	x_1, x_2, ... の最大値
min(*s*)	シーケンス *s* の最小値
min(x_1,x_2,...)	x_1, x_2, ... の最大値
str(*x*)	簡略化された *x* の文字列変換
repr(*x*)	*x* の文字列変換。‘*x*‘ に同じ
oct(*x*)	*x* の 8 進数表現
hex(*x*)	*x* の 16 進数表現
range(*n*?,*m*,*i*?)	リストを生成する (3.6)
tuple(*s*)	シーケンス *s* をタプルへ変換する
list(*s*)	シーケンス *s* をリストへ変換する
cmp(*x*,*y*)	*x*<*y*, *x*==*y*, *x*>*y* に応じて -1, 0, 1 を返す (1.5) 。注 1
input(*prompt*?)	キーボードからデータを読み取る (5.1)
raw_input(*prompt*?)	キーボードからデータを読み取る (5.2)。注 1
open(*name*,*mode*?,*bufsize*?)	ファイルを開く (4.2)
filter(*func*,*s*)	シーケンス *s* の要素を関数 *func* によって篩い分ける (5.4)
map(*func*,*s*)	シーケンス *s* の要素に関数 *func* を作用させリストを作る (5.5)
reduce(*func*,*s*,*init*)	2 変数関数 *func* によってシーケンス *s* を計算する (5.6)。注 2
apply(*func*,*t*,*d*)	関数 *func* に引数情報をタプルと辞書の形式で渡して実行させる (5.7) 。注 1

表 A.2: 基本関数 (次頁へ続く)

? の付いた引数は無くてもよい。必要に応じて与える。

( ) 内の数字は第 II 部の章と節を表す。例えば (1.3) は第 II 部 1 章 3 節を意味する。

注 1: Python3 では廃止される。

注 2: Python3 では reduce は functools モジュールに移された。

演算	関数の機能と返される値
eval(*code*)	値を評価する (5.8)
exec(*code*)	文を実行する (5.9)
compile(*code*,*label* ,*kind*)	Python のプログラムが書かれた文字列をコンパイルする (5.11)
id(*obj*)	オブジェクト *obj* の ID を返す (5.12)
type(*obj*)	オブジェクト *obj* のデータ型を返す (5.13)
hash(*obj*)	オブジェクト *obj* のハッシュ値を返す (5.14)
callable(*obj*)	オブジェクト *obj* が呼び出し可能か否かを調べる (5.15)
dir(*obj*?)	*obj* の属性名を返す／参照可能な名前一覧を返す (5.16)
vars(*obj*?)	*obj* の属性辞書を返す／局所的な名前辞書を返す (5.17)
locals()	locals を実行した場所での名前辞書を返す (5.18)
globals()	大域的に参照可能な名前辞書を返す (5.19)
getattr(*obj*,*name*)	属性の値を返す (3.8)
hasattr(*obj*,*name*)	属性を持っているか否かを調べる (3.8)
setattr(*obj*,*name*,*value*)	属性に値を設定する (3.8)
delattr(*obj*,*name*)	属性を消去する (3.8)
reload(*module*)	モジュールを再ロードする (注 1)

表 A.3: 基本関数 (続き)

? の付いた引数は無くてもよい。必要に応じて与える。
( ) 内の数字は第 II 部の章と節を表す。例えば (5.8) は第 II 部 5 章 8 節を意味する。

注 1: reload について本書の中での解説を省略する。参考文献 [1] に詳しい解説があるので、必要ならこれを参考にしてほしい。

付 録B　基本モジュール

Python は膨大なモジュールを持っている。その一覧とモジュールに含まれる関数の仕様は次のアドレスに載っている[1]。
http://docs.python.org/library/
ここではその内の 2 つだけを紹介する。

B.1　math モジュール

math モジュールの関数を使用するには

```
import math
```

あるいは

```
from math import *
```

を実行しておく必要がある。両者の使い方の違いについては第 II 部第 3 章 10 節を参照せよ。math の複素関数版は cmath モジュールに含まれている。

[1] 各々のモジュールの使い方の解説が含まれていれば有り難いのであるが、残念ながら見つからない。

Python	数学	意味
`pi`	π	円周率 (3.1415$\cdots$)
`e`	e	自然対数 (2.71828$\cdots$)
`acos(`x`)`	arccos(x)	逆余弦関数 (0 から $+\pi$ の値をとる) cos(y) が x となる y の値
`asin(`x`)`	arcsin(x)	逆正弦関数 ($-\pi/2$ から $+\pi/2$ の値をとる) sin(y) が x となる y の値
`atan(`x`)`	arctan(x)	逆正接関数 ($-\pi/2$ から $+\pi/2$ の値をとる) tan(y) が x となる y の値 この値はベクトル $(1,x)$ とベクトル $(1,0)$ の交角
`atan2(`x`,`y`)`	arctan(y/x)	逆正接関数 ($-\pi$ から $+\pi$ の値をとる) tan(z) が y/x となる z の値 この値はベクトル (x,y) とベクトル $(1,0)$ の交角
`ceil(`x`)`		x 以上の整数の内の最小のもの 例: `ceil(2.3)` は 3.0、`ceil(2)` は 2.0
`cos(`x`)`	cos(x)	余弦関数 (角度の単位はラジアン)
`cosh(`x`)`	cosh(x)	双曲線余弦関数 $(e^x + e^{-x})/2$
`exp(`x`)`	e^x または exp(x)	指数関数
`floor(`x`)`	[x] で表現する事がある (Gauss の記号)	x 以下の整数の内の最大のもの 例:`floor(2.3)` は `2.0`、`floor(2)` は `2.0`
`fmod(`x`,`y`)`		`x-int(x/y)*y` この値は `x>0` の時には `x%y` と同じであるが `x<0` の場合には `fmod(x,y)` は負になる。
`frexp(`x`)`		$a \times 2^n$ が x に等しくなる (a,n) を返す。 ここに n は整数であり、$0 \le a < 1$
`hypot(`x`,`y`)`	$\sqrt{x^2+y^2}$	`sqrt(x*x + y*y)`
`ldexp(`x`,`y`)`		`x*2.0**int(y)`
`log(`x`)`	$\log_e(x)$ または $\ln(x)$	自然対数 e^y が x に等しくなる `y` の値
`log10(`x`)`	log(x) または $\log_{10}(x)$	10 の対数 10^y が x に等しくなる y の値
`modf(`x`)`		x の少数部分と整数部分を返す 例: `modf(3.14)` は $(0.14, 3)$、 `modf(-3.14)` は $(-0.14, -3)$ を返す
`sin(`x`)`	sin(x)	正弦関数 (角度の単位はラジアン)
`sinh(`x`)`	sinh(x)	双曲線正弦関数 $(e^x - e^{-x})/2$
`sqrt(`x`)`	$\sqrt{x}$	`x` の平方根
`tan(`x`)`	tan(x)	正接関数 (角度の単位はラジアン)
`tanh(`x`)`	tanh(x)	双曲線正接関数 sinh(x)/ cosh(x)

表 B.1: 数学定数と数学関数

B.2 文字列のメソッド

以下の表で s は全て文字列である。その他については意味欄と解説を見よ。

Python	意味
s.expandtabs(*tabsize*)	*s* の中のタブを空白に変換する
s.find(*sub*,*start*?)	*s* の中に含まれている文字列 *sub* の位置を捜す
s.rfind(*sub*,*start*?)	*s* の中に含まれている文字列 *sub* の位置を逆方向から捜す
s.index(*sub*,*start*?)	*s* の中に含まれている文字列 *sub* の位置を捜す
s.rindex(*sub*,*start*?)	*s* の中に含まれている文字列 *sub* の位置を逆方向から捜す
s.replace(*old*,*new*,*max*?)	*s* の中に含まれている文字列 *old* を *new* に置き換える。(*max* には置換の最大個数を指定する)
s.count(*sub*, *start*?)	*s* の中に含まれている文字列 *sub* の出現回数を返す
s.split()	空白文字を区切り文字として *s* をリストに分割する
s.split(*sep*)	*sep* で与えた文字列を区切り文字として *s* をリストに分割する
sep.join(*x*)	文字列を要素とするタプルあるいはリスト *x* を結合した文字列を返す。*sep* が指定された場合は *sep* を間に入れる。*sep* が省略された場合には 1 個の空白を間に入れる。
s.strip()	文字列の始まり部分と終わり部分の空白文字を取り除く
maketrans(*from*,*to*)	translate の為の変換テーブルを作成する
s.translate(*table*,*delete*?)	文字列 *s* の中の文字を変換テーブルに基づいて変換する。(削除も可能である。その場合 *delete* で削除文字を指定する)

次頁へ続く

注意: maketranse は関数であり、メソッドではない。これを利用するには string モジュールをインポートする必要がある。なお Python3 では文字列のメソッドとして扱われる。

前頁から続く

s.swapcase()	*s* の中の小文字と大文字に、大文字を小文字に変換する
s.upper()	*s* の中の小文字を大文字に変換する
s.lower()	*s* の中の大文字を小文字に変換する
s.ljust(*width*)	長さ *width* の文字列の中に文字列 *s* を左寄せで埋め込む
s.rjust(*width*)	長さ *width* の文字列の中に文字列 *s* を右寄せで埋め込む
s.center(*width*)	長さ *width* の文字列の中に文字列 *s* を中央寄せで埋め込む
s.zfill(*width*)	長さ *width* の文字列の中に文字列を右寄せで埋め込む。残りは 0 を埋める。

表 B.2: 文字列関数

解説

この表で *s* は全て文字列である。?の付いた引数は無くてもよい。必要に応じて与える。

find や rfind、index、rindex において *start?* は探索のスタート位置を意味する。省略されれば 0 が指定されたと見なされる。find, rfind は文字列が見つからない時に −1 を返す。他方 index, rindex はエラーを発生させる。

translate の実行に必要な変換テーブルは maketrans で作成する。例えば

```
t=maketrans("abc", "xyz")
```

とすると、'a' を 'x' に、'b' を 'y' に、'c' を 'z' に変換するテーブル t が作成される。その下で

```
print "alice".translate(t,"le")
```

を実行すると

```
xiz
```

が出力される。translate の第 2 引数 "le" は削除する文字の一覧である。

付 録C　colors1.py

次のプログラムは Python の色名称とディスプレイ上での実際の見え方を比較するためのものである。プログラムを打ち込むのが面倒であれば筆者のサーバから採ってくるがよい。http://ar.aichi-u.ac.jp/netlib/Python/samples/ に置かれている。(ただし、最近の Python2 に合わせた修正が行われている)

```
--------------------------- colors1.py -------------------------
#
# colors
# coded by Kenar (Kenji Arisawa)
# REF: Tcl/lib/tk8.0/demos/colors.tcl
#
from Canvas import *
import sys
colors='''
black darkgray gray lightgray
white snow seashell AntiqueWhite bisque
PeachPuff NavajoWhite LemonChiffon cornsilk
ivory honeydew LavenderBlush MistyRose azure
SlateBlue RoyalBlue blue DodgerBlue SteelBlue
DeepSkyBlue SkyBlue LightSkyBlue LightSteelBlue
LightBlue LightCyan PaleTurquoise CadetBlue
turquoise cyan SlateGray DarkSlateGray aquamarine
DarkSeaGreen SeaGreen PaleGreen SpringGreen
green chartreuse OliveDrab DarkOliveGreen
khaki LightGoldenrod LightYellow yellow
gold goldenrod DarkGoldenrod RosyBrown
IndianRed sienna burlywood wheat tan
chocolate firebrick brown salmon LightSalmon
orange DarkOrange coral tomato
OrangeRed red DeepPink HotPink pink LightPink
PaleVioletRed maroon VioletRed magenta
orchid plum MediumOrchid DarkOrchid
purple MediumPurple thistle
'''

colorlist=colors.split()
numcolor=len(colorlist)

def pr(event):
    global t
    n,m=event.x/20,event.y/20
    num=16*m+n
    if num >= numcolor: return
    c.delete(t)
    color=colorlist[num]
```

```
    red,green,blue=c.winfo_rgb(color)
    red=red/256
    green=green/256
    blue=blue/256
    t=c.create_text(150,120, text="%s   #%02X%02X%02X"%(color,red,green,blue),anchor="w")

c=Canvas(width=320,height=140,background='white')
c.pack()
c.bind("<Button-1>",pr)

n,m=0,0
for color in colorlist:
    c.create_rectangle(20*n, 20*m, 20*n+20, 20*m+20, fill=color)
    n=n+1
    if n == 16: n=0; m=m+1
t=c.create_text(150,420,text="")
c.mainloop()
-----------------------------------------------------------
```

付 録D　colors2.py

次のプログラムは Python の色名称とディスプレイ上での実際の見え方を比較するためのものである。プログラムを打ち込むのが面倒であれば筆者のサーバから採ってくるがよい。http://ar.aichi-u.ac.jp/netlib/Python/samples/ に置かれている。(ただし、最近の Python2 に合わせた修正が行われている)

```
------------------------ colors2.py ---------------------
#
# colors
# coded by Kenar (Kenji Arisawa)
# REF: Tcl/lib/tk8.0/demos/colors.tcl
#
from Canvas import *
import sys
colors='''
    gray60 gray70 gray80 gray85 gray90 gray95
    snow1 snow2 snow3 snow4 seashell1 seashell2
    seashell3 seashell4 AntiqueWhite1 AntiqueWhite2 AntiqueWhite3
    AntiqueWhite4 bisque1 bisque2 bisque3 bisque4 PeachPuff1
    PeachPuff2 PeachPuff3 PeachPuff4 NavajoWhite1 NavajoWhite2
    NavajoWhite3 NavajoWhite4 LemonChiffon1 LemonChiffon2
    LemonChiffon3 LemonChiffon4 cornsilk1 cornsilk2 cornsilk3
    cornsilk4 ivory1 ivory2 ivory3 ivory4 honeydew1 honeydew2
    honeydew3 honeydew4 LavenderBlush1 LavenderBlush2
    LavenderBlush3 LavenderBlush4 MistyRose1 MistyRose2
    MistyRose3 MistyRose4 azure1 azure2 azure3 azure4
    SlateBlue1 SlateBlue2 SlateBlue3 SlateBlue4 RoyalBlue1
    RoyalBlue2 RoyalBlue3 RoyalBlue4 blue1 blue2 blue3 blue4
    DodgerBlue1 DodgerBlue2 DodgerBlue3 DodgerBlue4 SteelBlue1
    SteelBlue2 SteelBlue3 SteelBlue4 DeepSkyBlue1 DeepSkyBlue2
    DeepSkyBlue3 DeepSkyBlue4 SkyBlue1 SkyBlue2 SkyBlue3
    SkyBlue4 LightSkyBlue1 LightSkyBlue2 LightSkyBlue3
    LightSkyBlue4 SlateGray1 SlateGray2 SlateGray3 SlateGray4
    LightSteelBlue1 LightSteelBlue2 LightSteelBlue3
    LightSteelBlue4 LightBlue1 LightBlue2 LightBlue3
    LightBlue4 LightCyan1 LightCyan2 LightCyan3 LightCyan4
    PaleTurquoise1 PaleTurquoise2 PaleTurquoise3 PaleTurquoise4
    CadetBlue1 CadetBlue2 CadetBlue3 CadetBlue4 turquoise1
    turquoise2 turquoise3 turquoise4 cyan1 cyan2 cyan3 cyan4
    DarkSlateGray1 DarkSlateGray2 DarkSlateGray3
    DarkSlateGray4 aquamarine1 aquamarine2 aquamarine3
    aquamarine4 DarkSeaGreen1 DarkSeaGreen2 DarkSeaGreen3
    DarkSeaGreen4 SeaGreen1 SeaGreen2 SeaGreen3 SeaGreen4
    PaleGreen1 PaleGreen2 PaleGreen3 PaleGreen4 SpringGreen1
    SpringGreen2 SpringGreen3 SpringGreen4 green1 green2
    green3 green4 chartreuse1 chartreuse2 chartreuse3
```

```
    chartreuse4 OliveDrab1 OliveDrab2 OliveDrab3 OliveDrab4
    DarkOliveGreen1 DarkOliveGreen2 DarkOliveGreen3
    DarkOliveGreen4 khaki1 khaki2 khaki3 khaki4
    LightGoldenrod1 LightGoldenrod2 LightGoldenrod3
    LightGoldenrod4 LightYellow1 LightYellow2 LightYellow3
    LightYellow4 yellow1 yellow2 yellow3 yellow4 gold1 gold2
    gold3 gold4 goldenrod1 goldenrod2 goldenrod3 goldenrod4
    DarkGoldenrod1 DarkGoldenrod2 DarkGoldenrod3 DarkGoldenrod4
    RosyBrown1 RosyBrown2 RosyBrown3 RosyBrown4 IndianRed1
    IndianRed2 IndianRed3 IndianRed4 sienna1 sienna2 sienna3
    sienna4 burlywood1 burlywood2 burlywood3 burlywood4 wheat1
    wheat2 wheat3 wheat4 tan1 tan2 tan3 tan4 chocolate1
    chocolate2 chocolate3 chocolate4 firebrick1 firebrick2
    firebrick3 firebrick4 brown1 brown2 brown3 brown4 salmon1
    salmon2 salmon3 salmon4 LightSalmon1 LightSalmon2
    LightSalmon3 LightSalmon4 orange1 orange2 orange3 orange4
    DarkOrange1 DarkOrange2 DarkOrange3 DarkOrange4 coral1
    coral2 coral3 coral4 tomato1 tomato2 tomato3 tomato4
    OrangeRed1 OrangeRed2 OrangeRed3 OrangeRed4 red1 red2 red3
    red4 DeepPink1 DeepPink2 DeepPink3 DeepPink4 HotPink1
    HotPink2 HotPink3 HotPink4 pink1 pink2 pink3 pink4
    LightPink1 LightPink2 LightPink3 LightPink4 PaleVioletRed1
    PaleVioletRed2 PaleVioletRed3 PaleVioletRed4 maroon1
    maroon2 maroon3 maroon4 VioletRed1 VioletRed2 VioletRed3
    VioletRed4 magenta1 magenta2 magenta3 magenta4 orchid1
    orchid2 orchid3 orchid4 plum1 plum2 plum3 plum4
    MediumOrchid1 MediumOrchid2 MediumOrchid3 MediumOrchid4
    DarkOrchid1 DarkOrchid2 DarkOrchid3 DarkOrchid4 purple1
    purple2 purple3 purple4 MediumPurple1 MediumPurple2
    MediumPurple3 MediumPurple4 thistle1 thistle2 thistle3
    thistle4
'''

colorlist=colors.split()
numcolor=len(colorlist)

def pr(event):
    global t
    n,m=event.x/20,event.y/20
    num=16*m+n
    if num >= numcolor: return
    c.delete(t)
    color=colorlist[num]
    red,green,blue=c.winfo_rgb(color)
    red=red/256
    green=green/256
    blue=blue/256
    t=c.create_text(150,420, text="%s     #%02X%02X%02X"%(color,red,green,blue))

c=Canvas(width=320,height=440,background='white')
c.pack()
c.bind("<Button-1>",pr)

n,m=0,0
```

```
for color in colorlist:
    c.create_rectangle(20*n, 20*m, 20*n+20, 20*m+20, fill=color)
    n=n+1
    if n == 16: n=0; m=m+1
t=c.create_text(150,420,text="")
c.mainloop()
-----------------------------------------------------------------
```

なお、このプログラムに現われる色名称は Tk 付属のサンプルプログラムから借用されている。