

**Python による
プログラミング入門**

第 5 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。日本では Ruby に押されまだ知名度は低いですが、外国では人気の高いプログラミング言語である。最近では Python のテキストもかなり揃いだした。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。最新版(2008年3月現在)は 2.5.2 である。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

- 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)

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。

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から手に入るので興味があれば読んでみるがよい。

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。

2008 年 3 月 10 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	7
1.5 練習問題	7
第 2 章 プログラムの実行法	9
2.1 準備	9
2.2 例 1	9
2.3 例 2	11
2.3.1 打ち込みの注意点	11
2.3.2 ファイル名	12
2.3.3 実行結果	13
2.3.4 もしも間違いがあれば...	13
2.4 雑談	14
第 3 章 サンプルプログラムの解説	15
3.1 表示する	15
3.2 よく使う命令をまとめる	16
3.3 条件に応じて処理を変える	17
3.4 何回も繰り返す	20
3.5 範囲を指定する	22
3.6 書式を整える	23
3.7 第 2 章のプログラム 1 の動作の説明	25
3.8 Zeller の公式	27
3.9 1752 年 9 月	28
第 4 章 プログラムの作り方	31
4.1 フィボナッチ数列	31
4.2 数の総和	34
4.2.1 リストの要素の和	35

4.3	ちょっと変わった掛け算	38
4.4	グラフの交点	40
4.5	誕生日の怪	41
4.5.1	問題の提起	41
4.5.2	コンピュータシミュレーション	41
4.6	フェルマーの問題	43
4.7	再帰的アルゴリズム	44
4.8	置き石ゲーム	47
4.8.1	ゲームのルール	47
4.8.2	プログラムの考え方	47
4.9	Boyce の壺の問題	50
4.9.1	問題の提起	50
4.9.2	プログラムの考え方	50
4.9.3	譜 4.15 の改良	51

第 II 部 Python 文法 53

第 1 章 定数 55

1.1	定数の分類	55
1.1.1	基本概念	55
1.1.2	データの体系	56
1.2	数	58
1.3	文字列 (string)	58
1.3.1	文字列定数	59
1.3.2	文字列に対する基本演算	60
1.3.3	文字列の順序関係	61
1.3.4	string モジュール	62
1.3.5	日本語の扱い	63
1.4	タプル (tuple)	65
1.4.1	タプルの基本演算	65
1.4.2	任意個の引数を持つ関数	66
1.5	リスト (list)	67
1.5.1	リストの演算	67
1.5.2	リストへの作用関数	69
1.5.3	2次元の配列	70
1.5.4	プログラム例 (素因数分解)	72
1.6	辞書 (dictionary)	73
1.6.1	辞書の基本操作	73
1.6.2	辞書の作用関数	73
1.6.3	プログラム例 (伝票の処理)	74

1.6.4	キーとして許されるデータの種類	75
1.6.5	関数のキーワード引数	76
第2章	変数と式	79
2.1	式	80
2.2	算術式	82
2.3	関係式	83
2.4	論理式	84
2.5	ビット演算	85
2.6	lambda 式	87
第3章	プログラムの構成	89
3.1	基本概念	89
3.1.1	文	89
3.1.2	コメント	89
3.1.3	行	89
3.1.4	字下げ	90
3.1.5	ブロック	90
3.1.6	特殊記号と空白	92
3.2	print 文	93
3.2.1	書式	93
3.3	代入文	95
3.4	式	98
3.5	条件文	99
3.6	繰り返し文	100
3.6.1	while と for	100
3.6.2	range	101
3.6.3	continue と break	101
3.7	関数定義文	102
3.7.1	def	102
3.7.2	return	103
3.7.3	引数並び	104
3.7.4	局所変数と大域変数	104
3.7.5	いろいろな関数引数	105
3.8	クラス定義文	106
3.8.1	class	106
3.8.2	プログラム例	107
3.8.3	self	108
3.8.4	クラス定義で使用可能な関数	109
3.9	エラーの捕捉	110

3.9.1	<code>try</code>	110
3.9.2	<code>raise</code>	110
3.9.3	<code>except</code>	111
3.9.4	組み込みの例外名	112
3.10	<code>import</code> 文 (モジュール)	113
第 4 章	ファイル	115
4.1	ファイルの中の数の総和	115
4.2	ファイルの基本操作	118
4.3	<code>sys.stdin</code>	121
第 5 章	基本関数	123
5.1	<code>input</code>	123
5.2	<code>raw_input</code>	125
5.3	<code>rstr</code> , <code>repr</code> , <code>'...'</code>	126
5.4	<code>filter</code>	126
5.5	<code>map</code>	127
5.6	<code>reduce</code>	127
5.7	<code>eval</code>	128
5.8	<code>exec</code>	129
5.9	<code>execfile</code>	129
5.10	<code>compile</code>	129
5.11	<code>id</code>	130
5.12	<code>type</code>	131
5.13	<code>hash</code>	131
5.14	<code>callable</code>	131
5.15	<code>dir</code>	131
5.16	<code>vars</code>	132
5.17	<code>locals</code>	132
5.18	<code>globals</code>	133
第 III 部	Python によるグラフィックス	135
第 1 章	グラフィックスの基礎	137
1.1	サンプルプログラム	137
1.2	キャンバス	140
1.3	座標系	141
1.4	基本図形	142
1.4.1	<code>create_line</code>	142
1.4.2	<code>create_rectangle</code>	144

1.4.3	create_polygon	145
1.4.4	create_oval	146
1.4.5	create_arc	147
1.4.6	create_text	148
1.5	オプション (補足)	150
1.5.1	font	150
1.5.2	color	152
1.5.3	stipple	153
1.5.4	capstyle	153
1.5.5	joinstyle	154
1.5.6	tags	155
1.6	印刷	157
付録 A 基本演算と基本関数		161
A.1	演算子と算術関数	161
A.2	基本関数	162
付録 B 基本モジュール		165
B.1	math モジュール	165
B.2	string モジュール	167
付録 C colors1.py		169
付録 D colors2.py		171

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。

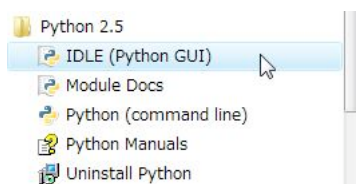


図 1.1: Python IDLE

すると図 1.2 の画面が生成される。

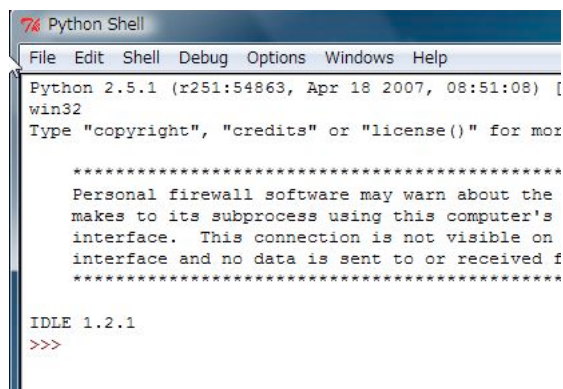


図 1.2: Python Shell

¹Windows 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。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。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²。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
```

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
```

²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

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³。

```
>>> 2**100
12676506000228229401496703205376
```

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する事ができる。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 を付ける。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

```
>>> 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。

³Python 2.2 以前は `2L**100` のように大きな整数を計算する場合には数字の後に L を付ける必要がある。`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。

```
>>> (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⁴。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)
>>> s,c
(0.841470984808, 0.540302305868)
>>> s/c
1.557407724655
```

ここで π 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前もって定義されている。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 $()$ の中に関数に与えるデータを書く。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。(360度で 2π ラジアンである。)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。

⁴。Python は言語 C と同程度の数学関数をサポートしており、数学関数の名称は C に準じている。詳しくはの付録の `math` モジュールを見よ。

1.3 変数

第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` とい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 [注 1]。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。

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。

注 1: Python の変数に関する詳しい解説は第6章第1節と第7章第2節を見よ。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。

数学に慣れてしていると `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余り推薦できない。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。従って関数の定義と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
```

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。

$(3*(2+5)-1)*2$

$17/3*2$

$3*(2+3)**3$

$1+2**3$

$-2**3+1$

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。但し割り算については 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 次の16進数は10進数では幾つですか? また、大きい順に番号を書きなさい。

16進数	10進数	順位
F7		
7F		
FF		
77		

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。

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⁵。ここに x は $1, 2, \dots, m-1$ 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。

問7 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実際上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。

⁵ 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 Vista 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

自分のホームフォルダの中に Python とい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。そのためにはホームフォルダを開き「整理」メニューから「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」という名のフォルダが作成されるので、それを「Python」に変更する。

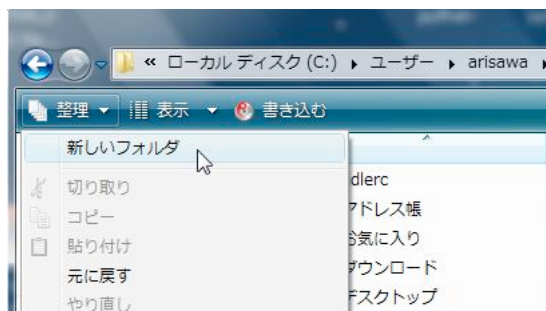


図 2.1: フォルダの作成

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。

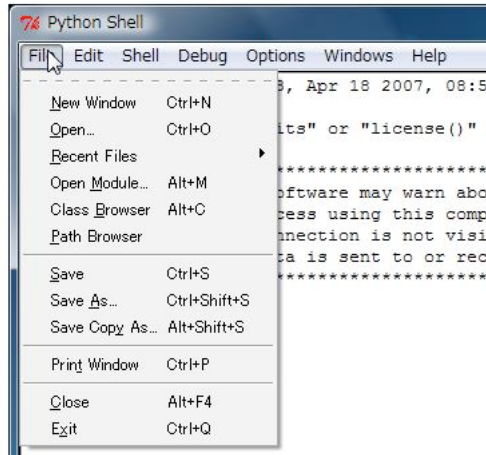


図 2.2: Python Editor の起動

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。

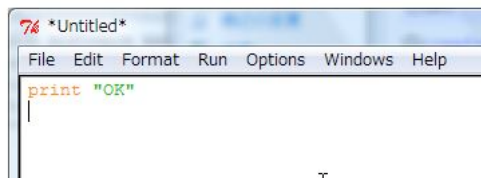


図 2.3: OK を表示するプログラム

「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。

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)。

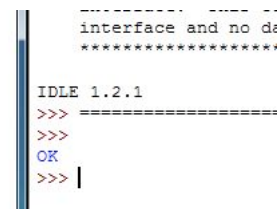


図 2.4: プログラムの実行結果

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

```
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。(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¹。例えば Windows 環境であればプログラムは

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

で始める事。

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。

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。

¹文字コードとして UTF-8 が使用される場合には文字コード情報は省略できる。

```

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。

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²。

²ファイル名には空白文字や日本の文字は使わない方がよい。トラブルの原因になる。

2.3.3 実行結果

打ち間違い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的な正確な知識を手に入れる事です...)

2.3.4 もしも間違いがあれば ...

もしも間違いがあれば、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icolon (;) 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icolon 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)。

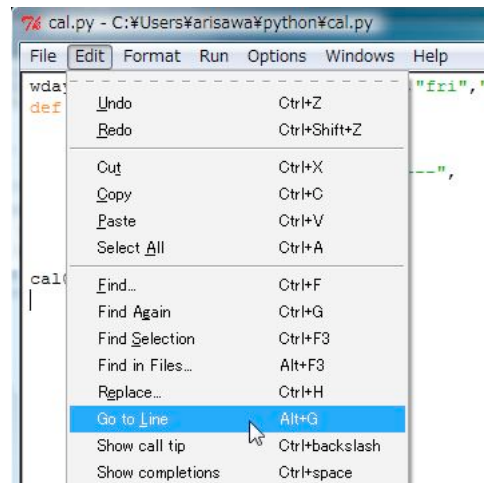


図 2.6: 行番号を指定して行を探す

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 環境はあくまで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。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、プログラムさえ作れば人手を解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。

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

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ことを 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。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, float(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 float(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 よく使う命令をまとめる

def, return

例 2 球の体積を次の3つの半径の各々について計算しなさい。

3, 4, 5

譜 3.2 簡単な関数を定義したプログラム

```
from math import *  
def v(r): return 4*pi*r*r*r/3  
print v(3)  
print v(4)  
print v(5)
```

解説

この問題では球の体積を何回か求めている。もしも球の体積を計算してくれる関数があればプログラムが単純化される。ところが Python には球の体積を求める関数は存在しない。

Python だけではなく汎用的なプログラミング言語では必要性の高い基本的な関数しか備わっていない。特殊な用途の関数は自分で作りなさいと言う考え方を採っているのである。そこで、この例題では極簡単な関数の作り方を解説している。

円周率 π が必要になる為にこのプログラムでは

```
from math import *
```

を予め実行している。そして

```
def v(r): return 4*pi*r*r*r/3
```

が半径から球の体積を求める関数を定義している。ここに現われる `def` が Python において関数を定義する時に使用される予約語 [注釈] である。そしてこの後に関数を使用される時の形を書き、その後にコロン (`:`) が続く。`return` もまた予約語である。`return` の後の計算式が関数の値となる。ここでは半径として文字 `r` を使用しているが、この文字に拘る必要はない。(何でも構わないのである。) 関数名の `v` についても同様である。

予約語: 予め意味が定義された単語。予約語は変数名や関数名としては使用できない

問 1 関数 $f(x) = 1/(x^2 + 1)$ の値を x が 0.5, 1.0, 1.5, 2.0 の 4 つの値について求めなさい。また

```
from math import *
```

がこの場合にも必要か否かについて考えなさい。さらにまた

```
print f(1)
```

に対して、正しい答 0.5 が得られる様に関数の定義を工夫しなさい。

3.3 条件に応じて処理を変える

if

例 3 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 3.3 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 `leap(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` を実行しなさいという意味である。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 Python 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 != あるいは <> 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。

```
if x%400 == 0: return 1
if x%100 == 0: return 0
if x%4 == 0: return 1
return 0
```

← x%400 == 0 の場合に実行される

← x%400 == 0 でない場合に実行される

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なるためである。他の年についても同様である。

問 2 プログラム 3 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良くと考えられる。この考えのもとにプログラム 3 の関数 `leap(x)` を改善しなさい。

問 3 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。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4 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れ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¹、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
```

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。

3.4 何回も繰り返す

for, in

例題 3 のプログラム 3 は次の様に書くこともできる。

譜 3.4 繰り返すプログラム

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

即ち

¹ 「アルゴリズム辞典」(共立出版、1998)

```
print 1996, leap(1996)
print 1999, leap(1999)
print 1900, leap(1900)
print 2000, leap(2000)
```

は

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

に置き換え可能である。この部分は

`x` が 1996 と 1999 と 1900 と 2000 の場合に対して `print x, leap(x)` を計算しなさい

と言う意味である。(計算は 1996, 1999, 1900, 2000 の順に行われる。)

ここに現われた `for` は Python の予約語であり、予約語 `in` と組み合わせて、変数に指定されたデータを自動的に順に設定して何かの命令を実行させる場合に使用される。`for` 文を使うことによってどのようなメリットがあるのだろうか? この例題では 4 つのデータしか扱っていないが、多数のデータを扱う場合にはプログラムが簡単になりそうだという事は直ちに気がつくであろう。しかしその他に(そしてもっと重要な事は) `for` 文を使用する事によって `x` の値の集合を柔軟に(何かの計算の結果として)指定できるのである。

[1996, 1999, 1900, 2000] のように、データをコンマで区切って [] の中に並べたものを Python ではリストと呼んでいる。この例ではリストの要素は整数であるが、実はあらゆるデータがリストの要素になる事が可能で、そのために極めて複雑な構造を持ったリストも存在し得る。

譜 3.4 の `for` 文は

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

あるいは簡単に

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

と書くこともできる。

(1996, 1999, 1900, 2000) のようにデータをコンマで区切って () の中に並べたものをタプルと言う。リストと同様にタプルも要素のデータ型に関して制約がない。タプルとリストはよく似ているのであるが、リストは要素を変更できるがタプルはできない。その代わりにタプルの方がリストに比べて処理速度が速い。

タプルは曖昧性が発生しない限り丸い括弧を省略できる。

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

の中で使用されているのは実はタプルである。なお、ここでは `for` 文による繰り返し実行の対象となる文「`print x, leap(x)`」が `for` と同じ行に書かれているが、普通は次の様に改行して書く。

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

字下げを行っている事に注意せよ。for 文の実行対象が複数個あれば、それらはみな同じだけ字下げをする。Python は字下げによって実行対象の範囲を定めているのである。実行対象となる文に複合文が含まれない場合には、それらをコロン(:)に続けて同じ行に書くことが許される²。この事は全ての複合文に対して成立する。なお、複合文とは for 文、if 文、def 文などのように、他の文を構成要素として含む文の事を言う。Python のどの複合文もコロン(:)を持ち、その後に対象となる文が続く。

問5 例題2のプログラムを for 文を用いて書き換えなさい。

問6 次のプログラムは第1節のプログラム1の中から取り出したものである。(この3行だけで独立に動作するプログラムとなっている。)

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

これを実行して見なさい。ここに現われる

```
print " ",x,
```

の行末のコンマ(,)はどのような意味を持っていますか。(ヒント:コンマを外して実行してみなさい。)また、なぜ最後に print を実行するのか考えなさい。

3.5 範囲を指定する

range

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すことができる必要がある。次のプログラムは単に1から31までの数字を書き出している。

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

解説

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ようにするために関数 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]
```

²1つの行に複数の文を書く時にはセミコロン(;)で文を区切る。詳しくは第II部第3章第1節を見よ。

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7 1から31の数を横方向に書きなさい。(ヒント: 例題4の問題5を参考にしなさい。)

問8 関数 $f(x) = 1/(x^2 + 1)$ の値を x について

0.00、0.01、0.02、…、0.98、0.99、1.00

の合計101個の値に対して求めなさい。(ヒント: $x=0.01*n$ とし、 n について0から100まで計算しなさい。)

3.6 書式を整える

書式

例4 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

```

これを書いてくれるプログラムを作りなさい。

```

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

```

このプログラムには2つの新しい問題が含まれる。1つは土曜日の所で改行する事、2つ目は数字を揃える事である。

改行の問題は比較的易しい。改行のタイミングは今の問題では日付を表す変数 x の値が7の倍数で発生する。そして改行するには単に `print` を実行すればよい。従って日付 x を書いた後で

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

を実行すれば問題は解決する。第2の問題、即ち、奇麗に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

```
wday=("sun","mon","tue","wed","thu","fri","sat")
for x in wday: 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wday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。書式の指定方法は非常に豊富な内容を持っているためにここで解説するのは相応しくない。従って、ここではこれ以上の解説を省略する。詳しくは第II部第3章2節print文を見よ。

問9 次の出力が得られるプログラムを書きなさい。

```
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
```

問10 次の出力が得られるプログラムを書きなさい。

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

問11 以下のprint文を実行し書式文字列中の%の役割について考えなさい。

```
x=18
y=x+2
print "alice=%5d,bob=%d"%(x,y)
```

3.7 第2章のプログラム1の動作の説明

例5 第2章のプログラム1の動作を説明しなさい。

第2章のプログラムから実行に無関係なコメントを除去すると

```
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)
```

となる。このプログラムの中の

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

に続く字下げされている部分は新しい命令 `cal0` の定義部分である。この部分の実行は後回しにされる。従って最初に実行されるのは字下げが終了した `cal0(6,31)` からである。`cal0(6,31)` が実行されると、ここに現われる6と31の2つの数字は `def cal0(n,m)` で指定された2つの変数 `n` と `m` に代入される。6の位置には変数 `n` が書かれているので `n` には6が代入される。31の位置には `m` が書かれているので `m` には31が代入される。従って `cal0(6,31)` の実行によって何が行われるかを調べるには

```
wday=("sun","mon","tue","wed","thu","fri","sat")
n=6
m=31
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
```

の動作を調べていけばよい。既に例題4の問題4で見たように

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

は

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

を書き出す。そして

```
n=6
for x in range(0, n): print "  ---",
```

は

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

を書き出す。(改行はしない。) n と m の値を考慮すると残された部分は

```
for x in range(1, 32):
    print "%5d"%x,
    if (x+6)%7==0 : print
```

である。これは例4のプログラムとよく似ているが改行のタイミング異なる。今度は x が 1, 8, 15, 22, 29 で `if` の条件が成立して改行される。以上のようにして

```
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
```

が出力されるのである。

問12 例5のプログラムにおいて

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

は

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

と書きたくなるのだが、その場合には正しく動作しない事を確認しなさい。

3.8 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。

譜 3.5 Zeller の公式を使ったカレンダープログラム

```
# 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

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 (0,1,..,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 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

def cal(y,m):
    n=h(y,m,1)
    n1=h(y,m+1,1)
    r=(n1-n)%7
    d=28+r
    cal0(n,d)

cal(2000,1)
```

3.9 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 月のカレンダー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 13 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。自動化するという事は、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以降にいく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。	$11349 \div 6201$	→	5148
6201 を 5148 で割った剰余を計算して 1053 を得る。	$6201 \div 5148$	→	1053
5148 を 1053 で割った剰余を計算して 936 を得る。	$5148 \div 1053$	→	936
1053 を 936 で割った剰余を計算して 117 を得る。	$1053 \div 936$	→	117
936 を 117 で割った剰余を計算して 0 を得る。	$936 \div 117$	→	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 $\text{gcd}(x, y)$ 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。

$\text{gcd}(11349, 6201)$, $\text{gcd}(6201, 11349)$

また x, y の一方が 0 の時に $\text{gcd}(x, y)$ 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を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 $[]$ 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 $x = 1, \dots, 12$, $y = 1, \dots, 12$, $m = 13$ 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 $0, \dots, 12$ までの有限個の数字の世界に生きておきましょう。この世界においては 2 つの数 x と y は $x - y$ が m で割れきれぬ時に等しいと考える。そしてその事を

$$x \equiv y \pmod{m}$$

と表す。例えば

$$3 \times 5 \equiv 2 \pmod{13}$$

である。 m を固定して考えている場合には $(\text{mod } m)$ 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 $2, 2^2, 2^3, \dots, 2^{12}$ 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、 $x = 1, \dots, m-1$ のどの x に対しても

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

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、 $0, \dots, 12$ 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。

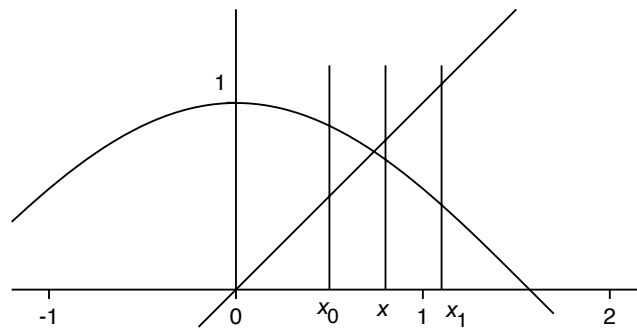


図 4.2: 二分法

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ようになる。

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か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ようにするためにはちょっとした工夫が必要である。

問7 次の式で表される2つのグラフ

$$y = -x^2 + 1$$

$$y = x$$

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。なあとクラスの子は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 *
```

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

⁴これを区間 [0,1) の一様乱数と言う

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
```

`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。

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いていける。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らうか?

問 8 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。Fermat⁶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」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 6

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

となる自然数の組を見つけよ。

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。

⁶1601-65、フランスの数学者

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。効率よく探すために

$$x \leq y \leq z$$

の条件を付けよう。その下で z について $1, 2, 3, \dots$ 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 9

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

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 $m = 1644455544992686074722684291497$ のようなケースである。

「簡単だね

```
x=117
m=1644455544992686074722684291497
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=1644455544992686074722684291497
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=1644455544992686074722684291497
```

```
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。

$$\begin{array}{ll} 2^2 & 2 \times 2 \rightarrow 4 \\ 2^4 & 4 \times 4 \rightarrow 16 \\ 2^8 & 16 \times 16 \rightarrow 256 \\ 2^{16} & 256 \times 256 \rightarrow 65536 \end{array}$$

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。そうすればべき乗の計算を小さなべき (約半分のべき) 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。べき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べき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。ただこの考えを剰余計算に適用する関数をサービスしていないだけである。

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= 1644455544992686074722684291497
```

```
x=117
```

```
print pow(x,m-1,m)
```

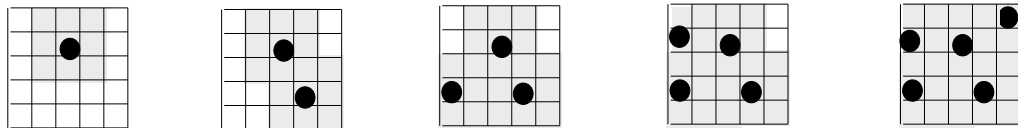
これは一瞬の内に答えを出す。448225599079959033691973915232 だ。
つまり 1644455544992686074722684291497 は素数ではない。

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。

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 $[(0,0), (0,1), \dots, (4,4)]$ 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)
```

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。

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。

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の壺の問題

4.9.1 問題の提起

ここに1つの壺と碁石がある。碁石は白が4個と黒が6個である。

「この碁石を壺に入れるよ。君が壺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。取った石は壺には戻さないで、君は壺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。

一見すると、黒が白より多いので、分が悪そうだな。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のではないか? 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個をさりに取り出すさ。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なと言うことが分る。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の壺の問題」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ば $+1 + V(1, 0)$ の利益であると考え事ができる。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個の壺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

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

$$V(0, p) = p$$

$$V(m, 0) = 0$$

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 の壺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。誤りが無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られる。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 であると考えられる。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。関数もデータの種類であり、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。

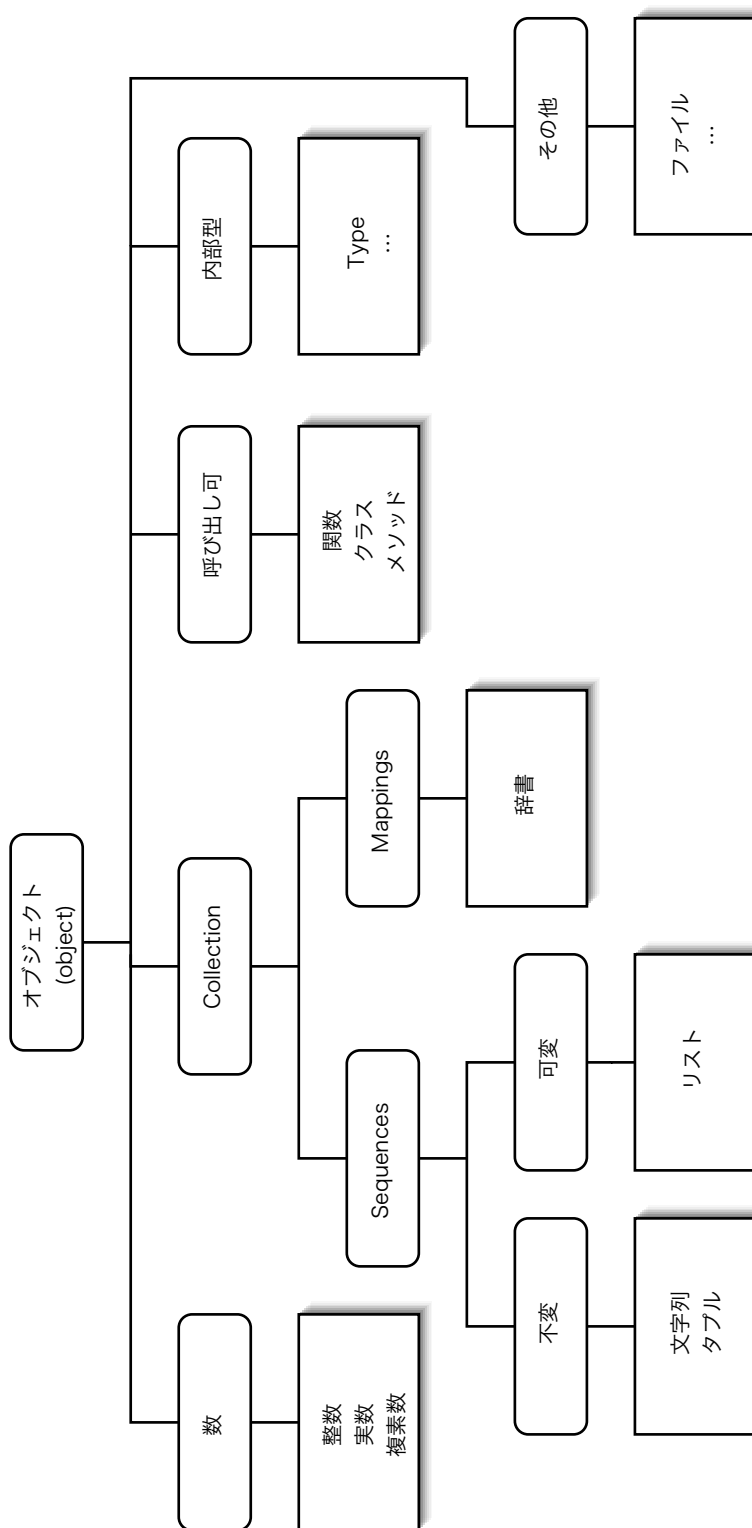
```
b=[2,3,"alice"]
b[2]="bob"
```

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。

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。

[] 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發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する順序付けである。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、文字を与えて文字コードを返す関数 `ord(c)` と、逆に文字コード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 string モジュール

文字列を対象とした多数の関数が `string` モジュールの中で定義されている。`string` モジュールの一覧は付録「`string` モジュール」に載せてある。以下にそれらの関数を用いた簡単な例を載せる。(プログラムのコメント部分に出力結果を参考の為に示してある。)

```
from string import *
s='alice'
print upper(s)      # ALICE を出力
print find(s,'ic') # 2 を出力
print find(s,'iq') # -1 を出力
```

`upper` は小文字を大文字に変換する。文字変換関数としては他に `lower`(小文字に変換)、`swapcase`(大文字と小文字の入れ替え)がある。`find` は文字列を見つけ出し、その位置を返す。よく似た関数に `index` があるが、`index` とは見つからない時の振る舞いが異なる。

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。

```
from string import *
a='    alice=18    bob=20    ' # 両端に余分な空白がある
b=strip(a)
print b      # alice=18    bob=20 を出力。両端の空白はない
c=split(b)
print c     # ['alice=18', 'bob=20'] を出力
x=c[0]
print x    # alice=18 を出力
d=splitfields(x,'=')
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 `string` モジュールの関数 `join(t,s)` を使用する。ここに `t` はタプルまたはリストである。また `s` は結合時に間に入れる文字列である。`s` を省略すると1個の空白を入れる。以下に例を示す。`#` 記号の右が出力結果である。

```
from string import *
a="alice"
b="bob"
print join((a,b))      # alice bob
print join((a,b),"-") # alice-bob
print join((a,b),"")  # alicebob
```

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⁴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なっ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。

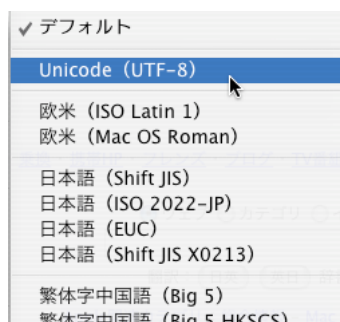


図 1.1: テキストエンコーディング

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。

⁴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

次の実験を試みよう。(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⁵。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。

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。

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 codes モジュールを使用する。次のプログラムは codes 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。

⁵ユニコード文字列 b が print で出力できたのは筆者の使用した Python が 2.3jp(SJIS enhanced) すなわち日本語の SJIS 環境に特別に仕立てられているからであろう。MacOSX 版ではエラーになる。しかし将来は同様に print で直接に出力できるようになると思える。

```
# 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。要素が 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。

以下のよう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する順序付けである。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、関数を呼び出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 deivision
# 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]
```

のようにならないで

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

のように `append` を使ってもよい⁶。

`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]
```

⁶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

以下にこれらを用いた作用関数の使い方を示すプログラム例を載せる。

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

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

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 `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。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]
```

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

```

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

```

```

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
```

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 3 5 7 9 11 13 15 ...

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 Python 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⁷、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。

1.6.2 辞書の作用関数

辞書を対象とする幾つかの関数が定義されている。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

⁷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。

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屢々必要となる。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、辞書に対する演算として、`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 辞書を使った伝票の処理

```

from string import *
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=split(x)
    v=int(v)
    put(d,k,v)
pr(d)

```

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。

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 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 インターフェイスを与える 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。

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)
```

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)
```

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るだろう。この事は

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

で行われている。

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。

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

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。例えば次の様に、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っているはずである。

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。'+', '*' 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。

<code>x or y, lambda</code>	論理演算子	論理和、ラムダ式
<code>x and y</code>	論理演算子	論理積
<code>not x</code>	論理演算子	否定
<code><, <=, >, >=, ==, <>, !=, is, is not, in, not in</code>	関係演算子	大きい、小さい 等しい、等しくない
<code>x y</code>	ビット演算子	ビット OR
<code>x ^ y</code>	ビット演算子	ビット XOR
<code>x & y</code>	ビット演算子	ビット AND
<code>x << y, x >> y</code>	ビット演算子	ビットシフト
<code>x + y, x - y</code>	算術演算子	和、差
<code>x * y, x / y, x % y</code>	算術演算子	乗算、除算、剰余
<code>-x, +x, ~x</code>	単項演算子	符号反転、ビット反転
<code>x[i], x[i:j], x.y, x(...)</code>		添字、関数等
<code>(...), [...], {...}, '...'</code>		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、式の特異なケースである。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の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

$$x = py + r \quad (0 \leq r < y)$$

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。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

$$x = py + r \quad (0 \geq r > y)$$

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かりやすい。`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
$X <> Y$	$X != Y$ に同じ
$X < Y$	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 より大きい等しい
$X \text{ is } Y$	X は Y である
$X \text{ is not } Y$	X は Y でない
$X \text{ in } Y$	X は Y の要素である
$X \text{ not in } Y$	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 で真偽を表す方が都合がある。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。

²Python 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。

<code>X and Y</code>	<code>X</code> と <code>Y</code> は共に真である
<code>X or Y</code>	<code>X</code> または <code>Y</code> は真である (共に真でもよい)
<code>not X</code>	<code>X</code> 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。

<code>X and Y</code>	<code>Y=0</code>	<code>Y=1</code>
<code>X=0</code>	<code>0</code>	<code>0</code>
<code>X=1</code>	<code>0</code>	<code>1</code>

表 2.5: 論理積

<code>X or Y</code>	<code>Y=0</code>	<code>Y=1</code>
<code>X=0</code>	<code>0</code>	<code>1</code>
<code>X=1</code>	<code>1</code>	<code>1</code>

表 2.6: 論理和

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。

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。

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) をブ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ブロックの頭部の 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 $\text{sim}(x,p,r)$ 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 $\text{sim}(x,p,r)$ 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` とい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。

```
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。出力の整形は文字列に作用し文字列を生成する演算子 `%` によって

```
print 式,
print 式, 式, ..., 式,
```

図 3.4: 改行動作を伴わない `print` 文の形式

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

```
"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]。



変数 = 式

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
b[0]='orange'
print a # ['apple', 1] を出力
print b # ['orange', 1] を出力

```

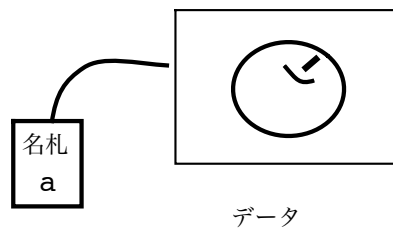
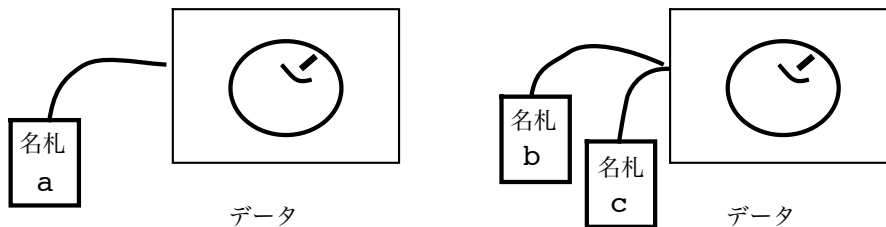


図 3.6: Python における代入の意味

```
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か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²。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。

²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」の部分に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 7 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

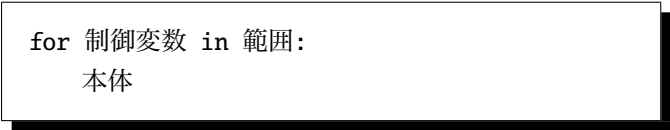
3.6.1 while と for

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³

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` 文に出会うと本体

³古い Python では `xrange` はタプルを生成する関数であったが、最近の版では `xrange` は仕様に変更された。`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。

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 `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。

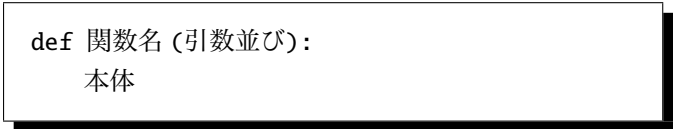
```
3
alice
bob
4
alice
bob
5
6
alice
```

`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出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` という特殊な値をとる。`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出すには

```
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利用できる事、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⁴。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` 宣言された変数は関数

⁴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
```

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。関数を呼び出す時にはこの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て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。関数を呼び出す時に第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。関数を呼び出す時に引数が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。関数を呼び出す時の引数の中に、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 $(2, 3, 7)$ と書けば屢々ベクトルを表し、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** とい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。

<code>x+y</code>	<code>__add__</code>
<code>x-y</code>	<code>__sub__</code>
<code>x*y</code>	<code>__mul__</code>
<code>x/y</code>	<code>__div__</code>
<code>x%y</code>	<code>__mod__</code>
<code>-x</code>	<code>__neg__</code>
<code>+x</code>	<code>__pos__</code>
<code>x**y</code>	<code>__pow__</code>
<code>x or y</code>	<code>__or__</code>
<code>x xor y</code>	<code>__xor__</code>
<code>x and y</code>	<code>__and__</code>
<code>x<<y</code>	<code>__lshift__</code>
<code>x>>y</code>	<code>__rshift__</code>
<code>^x</code>	<code>__invert__</code>

オブジェクトの属性に関して以下の関数を利用できる。

<code>getattr(obj,name)</code>	属性の値を返す
<code>hasattr(obj,name)</code>	属性を持っているか否かを調べる
<code>setattr(obj,name,value)</code>	属性に値を設定する
<code>delattr(obj,name)</code>	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。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。

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:
```

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":
```

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"
```

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⁵。

3.9.4 組み込みの例外名

以下に Python のシステムに組み込みの例外名を参考の為に挙げておく。(解説は省略する)

```
ArithmeticError
AssertionError
AttributeError
EOFError
FloatingPointError
IOError
ImportError
```

⁵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。

```

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。所が 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⁶。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。

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

```

最大公約数を求める関数が書かれたファイル `foo.py`

このファイルにはまだ他に関数が定義されていてもよい。

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 *

```

⁶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!

を実行するだけで `foo.py` 中の全ての関数を使用することができる⁷。(`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であった。

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` 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` 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 Python 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。

⁷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。

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: p.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 p.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 p.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('p.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', ... ]
```

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` は改行記号である。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('p.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` モジュールの関数 `split` がこれを行ってくれる。

```
t = split(s)
```

すると `t` には例えば、`['0','813741','772763']` が入る。しかしこれら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])
```

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。

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

```
from string import *
f=open('p.txt')
s = f.readline()
for s in f.readlines():
    t = split(s)
    age=int(t[0])
    male = int(t[1])
    female=int(t[2])
    print age,male,female
f.close()
```

以上の結果をまとめると、

ここで

```
from string import *
```

の行は `string` モジュールから全ての関数を取り込んでいる。これを行うのは関数 `split` を利用するためである。

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

```
from string import *
f=open('p.txt')
s = f.readline()
sum_male=0
sum_female=0
for s in f.readlines():
    t = split(s)
    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、名前を参照する。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は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。

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

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
<code>f=open(name)</code>	ファイルを開く
<code>f=open(name,mode)</code>	<i>name</i> : ファイルの名前
<code>f=open(name,mode,bufsize)</code>	<i>mode</i> : 動作モード (省略すれば 'r' が仮定される) <i>bufsize</i> : バッファサイズ
<code>f.close()</code>	ファイルを閉じる
<code>s=f.read()</code>	ファイルを文字列 <i>s</i> に読む
<code>s=f.read(size)</code>	<i>size</i> : 読み取りサイズ
<code>f.write(s)</code>	ファイルに文字列 <i>s</i> を書き込む
<code>f.seek(offset,whence)</code>	ファイルの読み書きの位置を変更する <i>offset</i> : オフセット。この意味は次の <i>whence</i> 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
<code>f.flush()</code>	変更の内容をディスク上のファイルに反映させる
<code>f.tell()</code>	ファイルの読み書きの位置を知る
<code>s=f.readline()</code>	テキストファイルから 1 行だけ文字列 <i>s</i> に読み取る <i>s</i> の末尾は改行文字である [注 1]
<code>x=f.readlines()</code>	テキストファイルから全ての行をリスト <i>x</i> に読み取る
<code>f.writelines(x)</code>	リスト <i>x</i> をファイルに書き込む
<code>f.isatty()</code>	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` を標準入りに割り当てている。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。

```
python count.py < a.txt
```

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。

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 (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 1:
    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 1:
    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。

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 1:
    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

組み込み関数 `filter` は次の定義と等価である。

```
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
```

```
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。

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
```

初期値が与えられた場合:

```
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

入力関数 `input()` は `eval(raw_input())` と等価である。

```
a=3
print eval('a*7') # 21 を出力
```

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。

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

この結果は次のようになる。

```
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)
```

```
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',
```

¹関数の事

```
'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

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。

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。

譜 1.1 プログラム sample.py

```
from Tkinter import *
from Canvas import *
import sys
def pr():
    d=c.postscript(file="a.eps")
def quit():
    sys.exit()
f=Frame()
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')
for n in range(0,35):
    c.create_line(30,100+5*n,70,100+5*n)
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')

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。開発中のプログラムはコマンドによって実行するのがよい。Windows であれば「DOS プロンプト」を立ち上げ、`cd` コマンドを使って `sample.py` の置かれたディレクトリに移り、

```
python sample.py
```

を実行する。

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。

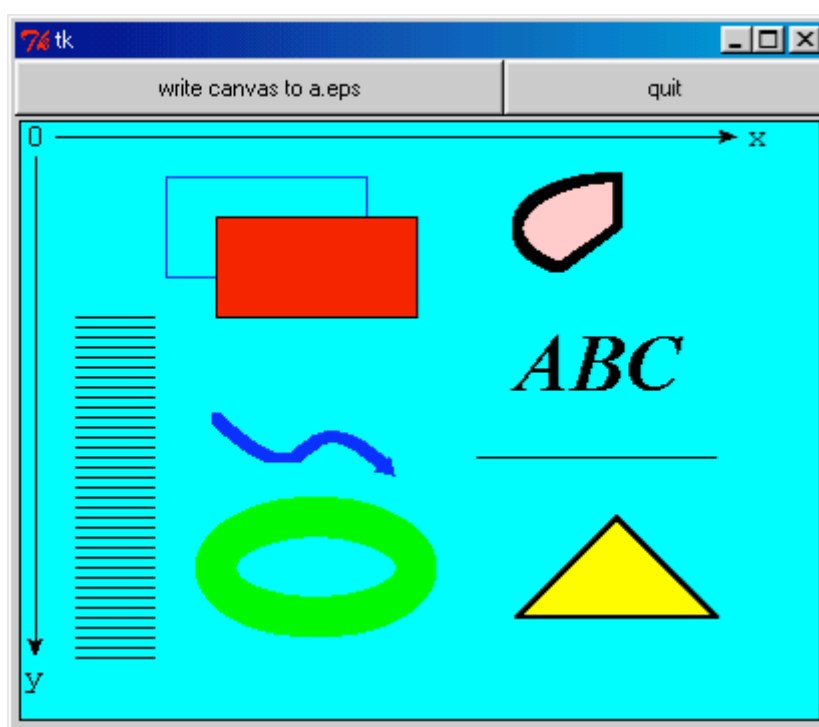


図 1.1: `sample.py` の実行結果

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のみなして欲しい。(ボタン等のユーザズ・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 (pixel) 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 background 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、これらの指定によってキャンバスの境界がどのように見えるかを示す。

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

```
mainloop()
```

は Canvas を使用したプログラムでは必須である。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マウスによるユーザ側のアクションを待っているのである。

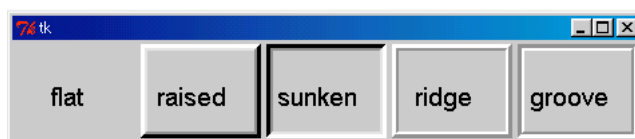


図 1.2: いろいろなレリーフ

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)

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 とい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。

Python ではサイズを単に数字で指定する事が多い。即ち画素の個数で指定する事が多いのである。Python で指定した 1 インチは約 96 画素であると考えて構わない。Canvas で width をインチ単位で与えて、

```
print c.cget('width')
```

を実行させれば画素単位のキャンバスの幅を出力してくれる。ここに `c` はキャンバスである。ディスプレイの基本的な3つの解像度 640 x 480、800 x 600 及び 1024 x 768 の下でこの実験を行えば共に 96 ± 1 の値を得る。(これは Windows の下での実験である) 不思議な事に Python には長さの単位を変換する関数が見当たらない。1 インチの画素数を知る為に実験に頼ったのはこのためである。

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。

<code>create_line</code>	線分、折れ線、曲線
<code>create_rectangle</code>	矩形(くけい)
<code>create_polygon</code>	多角形
<code>create_oval</code>	楕円
<code>create_arc</code>	円弧
<code>create_text</code>	文字列
<code>create_bitmap</code>	画像(2色)
<code>create_image</code>	画像(カラー)
<code>create_window</code>	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。

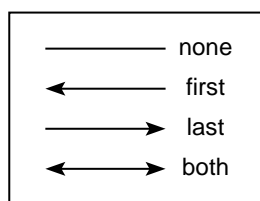


図 1.3: arrow の指定

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いる。

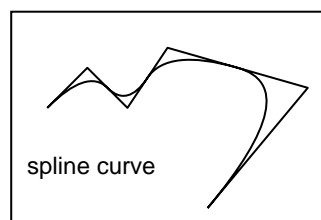


図 1.4: smooth 指定による図形の変化

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。

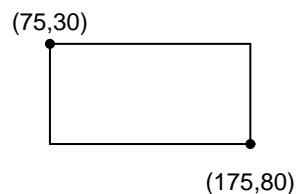


図 1.5: c.create_rectangle(75,30,175,80)

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。

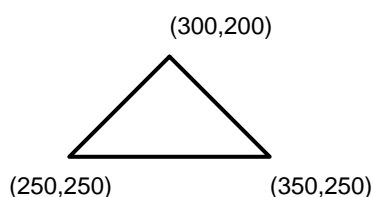


図 1.6: c.create_polygon(300,200,350,250,250,250,fill='yellow',width=2,outline='black')

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。

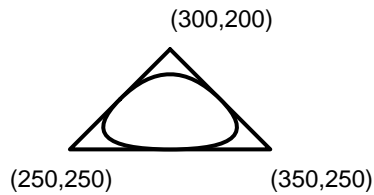


図 1.7: smooth の効果

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	
splinesstep	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。

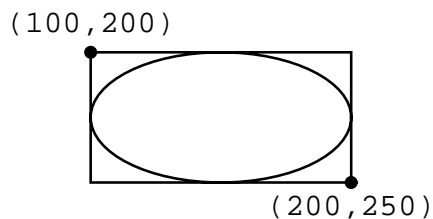


図 1.8: c.create_oval(100,200,200,250)

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)
```

の上であり、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。

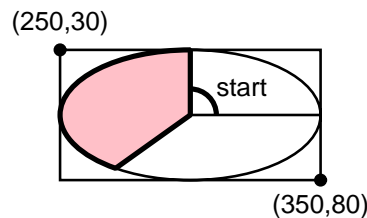


図 1.9: `c.create_arc(250,30,350,80,start=90,extent=145,width=5,fill='pink')`

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。この図の扇形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樣に圧縮すると図 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。

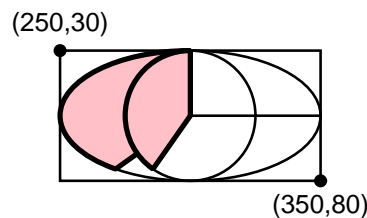


図 1.10: 円との比較で見る角度の指定

```

c.create_arc(250,30,350,80,start=90,extent=145) と
c.create_arc(275,30,325,80,start=90,extent=145) との比較

```

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次の図に示す図形を生成する。

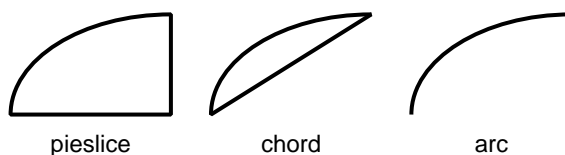


図 1.11: style の効果

まとめ `create_arc` で指定可能なオプションの一覧

<code>extent</code>	数字 (開きの角度)
<code>fill</code>	'red', 'green', ... などの色 (塗り潰しの色)
<code>outline</code>	'red', 'green', ... などの色 (外枠の色)
<code>start</code>	数字 (開始角)
<code>stipple</code>	'gray12', 'gray25', 'gray50', 'gray75'
<code>style</code>	'pieslice', 'chord', 'arc'
<code>width</code>	1, 2, 3, ... (外枠の幅)
<code>tags</code>	第5節の <code>tags</code> 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 の中央に来る様を書くのか、左端に来る様を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。

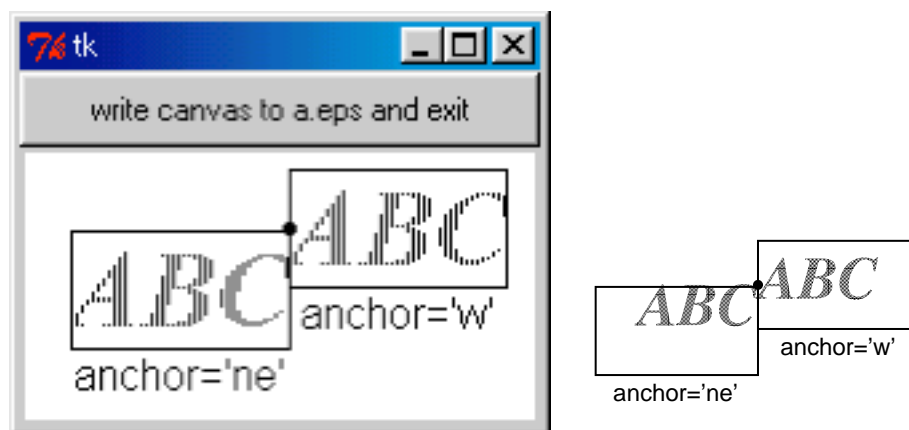


図 1.12: `anchor` の指定による文字列の表示位置の違い

`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かりやすい。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。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。例えば Windows であれば以下のようなフォントが含まれている。

Times Helvetica Bookman Courier Lucida Script Century

この他に日本語フォントを含め、多数のフォントが含まれている。



図 1.14: Window95 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

M S 明朝

などである。このような場合に

```
font=' M S 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「M S」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よい。Windows ではアウトラインフォントは TrueType フォントと呼ばれている。TrueType フォントは図 1.14 では TT 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 Python 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 階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もっと木目細かく階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。

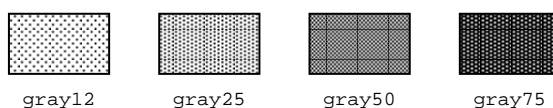


図 1.15: stipple の指定による塗り潰しパターンの違い

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。

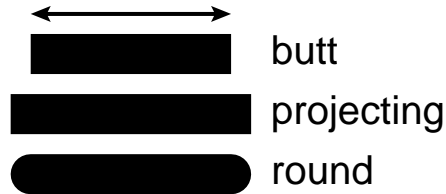


図 1.16: capstyle の指定による線端の違い

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。

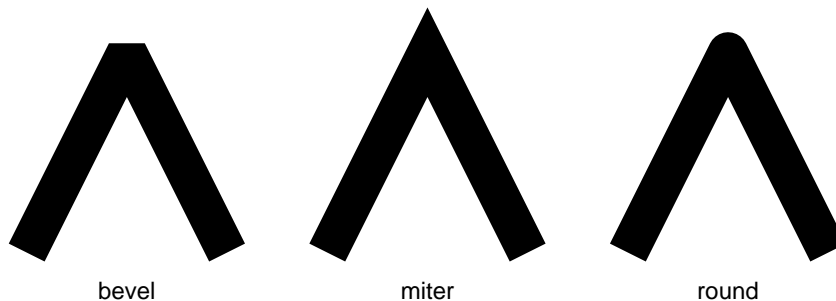


図 1.17: joinstyle の指定による接合点の違い

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、あたかも一つの図形のように、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。

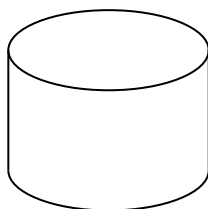


図 1.18: 描きたいドラム図形

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)
mainloop()
```

ここではこの図形には分りやすく 'drum' とい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。

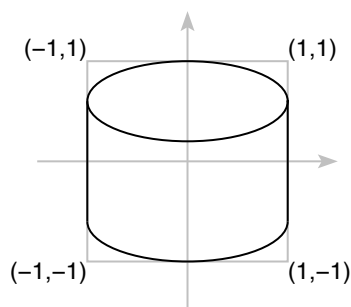


図 1.19: ドラム図形の図案

注目して欲しい点が幾つかある。この図案では

1. 上向きの方が y の正の方向である。
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²、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、印刷する時とは異なっている³。この違いを図に示す。

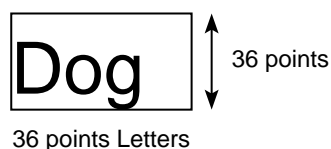


図 1.20: フォントで指定した文字の大きさと印刷結果の比較
ディスプレイ上では図に示した長方形の大きさに見える。
(図 1.12 も同様な問題を扱っているので参照せよ。)

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。するとプリンター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。

²MacOSX の印刷環境はとてもよい。Ghostscript 使わずに済むのだ。

³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'Reilly/ 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
$x y, x\wedge y, x\&y$	ビット演算。順に OR, XOR, AND 演算
$x\ll n, x\gg n$	ビットシフト演算
x	ビット反転
$\text{abs}(x)$	絶対値
$\text{int}(x)$	x の整数部分あるいは整数への変換
$\text{long}(x)$	長い整数への変換
$\text{float}(x)$	実数への変換
$\text{round}(x)$	実数 x を丸める (四捨五入する事)
$\text{round}(x, n)$	実数 x を小数点 n 桁で丸める
$\text{divmod}(x, y)$	$(x/y, x\%y)$ 除算、剰余のペア計算
$x**y, \text{pow}(x, y)$	べき乗計算 (どちらも同じ意味)
$\text{pow}(x, y, z)$	$\text{pow}(x, y)\%z$
$\text{coerce}(x, y)$	混合演算 (例えば +) を行う為に共通の型に揃える

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
<code>chr(i)</code>	文字コードが i の文字 (1.3)
<code>ord(c)</code>	文字 c のコード (1.3)
<code>max(s)</code>	シーケンス s の最大値
<code>max(x₁, x₂, ...)</code>	x_1, x_2, \dots の最大値
<code>min(s)</code>	シーケンス s の最小値
<code>min(x₁, x₂, ...)</code>	x_1, x_2, \dots の最大値
<code>str(x)</code>	簡略化された x の文字列変換
<code>repr(x)</code>	x の文字列変換。'x' に同じ
<code>oct(x)</code>	x の 8 進数表現
<code>hex(x)</code>	x の 16 進数表現
<code>range(n?, m, i?)</code>	リストを生成する (3.6)
<code>xrange(n, m, i)</code>	<code>range</code> と使い方は良く似ている (3.6)
<code>tuple(s)</code>	シーケンス s をタプルへ変換する
<code>list(s)</code>	シーケンス s をリストへ変換する
<code>cmp(x, y)</code>	$x < y$, $x == y$, $x > y$ に応じて -1, 0, 1 を返す (1.5)
<code>input(prompt?)</code>	キーボードからデータを読み取る (5.1)
<code>raw_input(prompt?)</code>	キーボードからデータを読み取る (5.2)
<code>open(name, mode?, bufsize?)</code>	ファイルを開く (4.2)
<code>filter(func, s)</code>	シーケンス s の要素を関数 $func$ によって篩い分ける (5.4)
<code>map(func, s)</code>	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)
<code>reduce(func, s, init)</code>	2 変数関数 $func$ によってシーケンス s を計算する (5.6)
<code>apply(func, t, d)</code>	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)

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。

() 内の数字は章と節を表す。例えば (1.3) は 1 章 3 節を意味する。

演算	関数の機能と返される値
<code>eval(code)</code>	値を評価する (5.8)
<code>exec(code)</code>	文を実行する (5.9)
<code>compile(code, label, kind)</code>	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)
<code>execfile(file)</code>	Python のファイルを実行する (5.10)
<code>id(obj)</code>	オブジェクト <i>obj</i> の ID を返す (5.12)
<code>type(obj)</code>	オブジェクト <i>obj</i> のデータ型を返す (5.13)
<code>hash(obj)</code>	オブジェクト <i>obj</i> のハッシュ値を返す (5.14)
<code>callable(obj)</code>	オブジェクト <i>obj</i> が呼び出し可能か否かを調べる (5.15)
<code>dir(obj?)</code>	<i>obj</i> の属性名を返す／参照可能な名前一覧を返す (5.16)
<code>vars(obj?)</code>	<i>obj</i> の属性辞書を返す／局所的な名前辞書を返す (5.17)
<code>locals()</code>	<code>locals</code> を実行した場所での名前辞書を返す (5.18)
<code>globals()</code>	大域的に参照可能な名前辞書を返す (5.19)
<code>getattr(obj, name)</code>	属性の値を返す (3.8)
<code>hasattr(obj, name)</code>	属性を持っているか否かを調べる (3.8)
<code>setattr(obj, name, value)</code>	属性に値を設定する (3.8)
<code>delattr(obj, name)</code>	属性を消去する (3.8)
<code>reload(module)</code>	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。

() 内の数字は章と節を表す。例えば (5.8) は 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/lib/modindex.html>

ここではその内の2つのモジュールだけを紹介する。

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...)
e	e	自然対数 (2.71828...)
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
fabs(x)	x	絶対値
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)		$x - \text{int}(x/y) * y$ 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、 $0 \leq a < 1$
hypot(x, y)	$\sqrt{x^2 + y^2}$	sqrt(x*x + y*y)
ldexp(x, y)		$x * 2.0^{**\text{int}(y)}$
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
pow(x, y)	x^y	$x ** y$ 即ち、x の y 乗
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 string モジュール

string モジュールの関数を使用するには

```
import string
```

あるいは

```
from string 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 3 章 10 節を参照せよ。

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
<code>atof(s)</code>	<i>s</i> を実数に変換する。 (<code>float(s)</code> の使用が推奨される)
<code>atoi(s, base?)</code>	<i>s</i> を整数に変換する。 (<code>int(s, base?)</code> の使用が推奨される。)
<code>expandtabs(s, tabsize)</code>	<i>s</i> の中のタブを空白に変換する
<code>find(s, sub, start?)</code>	<i>s</i> の中に含まれている文字列 <i>sub</i> の位置を捜す
<code>rfind(s, sub, start?)</code>	<i>s</i> の中に含まれている文字列 <i>sub</i> の位置を逆方向から捜す
<code>index(s, sub, start?)</code>	<i>s</i> の中に含まれている文字列 <i>sub</i> の位置を捜す
<code>rindex(s, sub, start?)</code>	<i>s</i> の中に含まれている文字列 <i>sub</i> の位置を逆方向から捜す
<code>replace(s, old, new, max?)</code>	<i>s</i> の中に含まれている文字列 <i>old</i> を <i>new</i> に置き換える。(<i>max</i> には置換の最大個数を指定する)
<code>count(s, sub, start?)</code>	<i>s</i> の中に含まれている文字列 <i>sub</i> の出現回数を返す
<code>split(s)</code>	空白文字を区切り文字として <i>s</i> をリストに分割する
<code>splitfields(s, sep)</code>	<i>sep</i> で与えた文字列を区切り文字として <i>s</i> をリストに分割する
<code>join(x, sep?)</code>	文字列を要素とするタプルあるいはリスト <i>x</i> を結合した文字列を返す。 <i>sep</i> が指定された場合は <i>sep</i> を間に入れる。 <i>sep</i> が省略された場合には 1 個の空白を間に入れる。
<code>joinfields(x, sep?)</code>	<code>join</code> に同じ
<code>strip(s)</code>	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
<code>maketrans(from, to)</code>	<code>translate</code> の為の変換テーブルを作成する

次頁へ続く

前頁から続く

<code>translate(s,table,delete?)</code>	文字列 <i>s</i> 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 <i>delete</i> で削除文字を指定する)
<code>swapcase(s)</code>	<i>s</i> の中の小文字と大文字に、大文字を小文字に変換する
<code>upper(s)</code>	<i>s</i> の中の小文字を大文字に変換する
<code>lower(s)</code>	<i>s</i> の中の大文字を小文字に変換する
<code>ljust(s,width)</code>	長さ <i>width</i> の文字列の中に文字列 <i>s</i> を左寄せで埋め込む
<code>rjust(s,width)</code>	長さ <i>width</i> の文字列の中に文字列 <i>s</i> を右寄せで埋め込む
<code>center(s,width)</code>	長さ <i>width</i> の文字列の中に文字列 <i>s</i> を中央寄せで埋め込む
<code>zfill(s,width)</code>	長さ <i>width</i> の文字列の中に文字列を右寄せで埋め込む。残りは <code>0</code> 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。例えば表では

```
atoi(s,base?)
```

となっている。*base* が省略されると 10 進数であるとして変換する。例えば `atoi("100")` は 100 である。`atoi` と `atol` は与えられた文字列が 10 進数の場合だけではなく、16 進数などの場合も整数に変換してくれる。例えば `atoi("100",16)` は 16 進数の "100" を整数に変換する意味であり 256 を返す。

`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 translate("alice",t,"le")
```

を実行すると

```
xiz
```

が出力される。`translate` の第三引数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。

```

----- colors1.py -----
#
# colors
# coded by Kenar
# REF: Tcl/lib/tk8.0/demos/colors.tcl
#
from Canvas import *
import sys, string
from Tkinter import *
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=string.split(colors)
numcolor=len(colorlist)

def pr(event):
    global t
    n,m=event.x/20,event.y/20
    num=16*m+n
    if num >= numcolor: return
    t.delete()
    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。

```
----- colors2.py -----
#
# colors
# coded by Kenar
# REF: Tcl/lib/tk8.0/demos/colors.tcl
#
from Canvas import *
import sys, string
from Tkinter import *
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=string.split(colors)
numcolor=len(colorlist)

def pr(event):
    global t
    n,m=event.x/20,event.y/20
    num=16*m+n
    if num >= numcolor: return
    t.delete()
    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。