1

はじめての SAGE

by Ted Kosan

翻訳：横田博史

Table of Contents

1 はじめに**.....9**
1.1 献辞.....9
1.1 ご了承.....9
1.2 支援団体.....9

2 はじめに**.....10**
2.1 数学計算環境とはなに?.....10
2.2 SAGE はなに?.....11
2.3 WEB サービスとして SAGE を利用.....13
2.3.1 WEB サービスとして SAGE を利用　手法その１.....14
2.4 SAGE のセルにソースコードを入力.....17

3 SAGE でプログラムをする上での基礎**.....21**
3.1 対象、変数、そして式.....21
3.2 演算子.....22
3.3 演算子の優先度.....23
3.4 式で演算の順番を変更する事.....24
3.5 変数.....25
3.6 文.....26
3.6.1 print 文.....26
3.7 文字列(string).....28
3.8 注釈(comment).....28
3.9 条件演算子.....29
3.10 if 文による判断.....32
3.11 論理演算子 and,or,not.....34
3.12 while 文による反復処理.....35
3.13 長い反復処理, 無限反復処理, そして処理の中断.....38
3.14 ワークシートのセルの挿入と削除.....39
3.15 より高度な対象の型について.....39
3.15.1 有理数.....39
3.15.2 実数.....40
3.15.3 他の対象の列を包含する対象: リストとタプル.....41
3.15.3.1 多変数の取りまとめと取り崩し.....42
3.16 リストとタプルを使った while 反復処理の利用.....43
3.17 in 演算子.....44
3.18 for 文を使った反復処理.....44
3.19 函数.....45
3.20 def 文を用いた函数の定義.....45
3.21 SAGE 内部の函数の一部.....47
3.22 SAGE 函数の情報を得る.....55

3.23 利用者が入力した函数についても情報が使える....56
3.24 SAGE に含まれる函数を用いた例....56
3.25 for 文で srange()と zip()を利用....58
3.26 リスト内包(list comprehension)....59

4 オブジェクト指向プログラミング**....61**
4.1 オブジェクト指向な心の書換....61
4.2 属性と振舞....61
4.3 類 (対象を生成する為の設計図）....62
4.4 オブジェクト指向プログラムは必要に応じて対象の生成と破壊を行う....62
4.5 オブジェクト指向プログラム例....63
4.5.1 Hellos オブジェクト指向プログラム例 (無注釈)....64
4.5.2 Hellos オブジェクト指向プログラム例 (注釈付き)....65
4.6 SAGE 類(Class)と対象(Object)....68
4.7 SAGE の対象について情報を得ること....69
4.8 対象のメソッドのリスト....71
4.9 継承による類の拡張....72
4.10 対象類、dir() 函数、組込メソッド....73
4.11 Sage.rings.integer.Integer 類の継承階層....75
4.12 "は一つの"関係....76
4.13 こんがらがったかな?....76

5 いろいろなこと**....77**
5.1 前回の処理結果の参照....77
5.2 例外処理....77
5.3 数値結果を得る....78
5.4 式の表記指南....79
5.5 組込定数....80
5.6 根....81
5.7 記号変数....81
5.8 記号変数....83
5.9 展開と因子分解....84
5.10 いろいろな記号式の例....84
5.11 記号式への値の引き渡し....85
5.12 記号方程式と solve()函数....85
5.13 記号数学函数....86
5.14 グラフを使った根の検出と find_root()メソッドを利用した数値的な根の検出....88
5.15 伝統的な書式で数学対象を表示....89
5.15.1 伝統的な数式で対象を表示する為に LaTeX を利用....89
5.16 集合....90

6 2D 描画**....91**
6.1 plot()と show()函数....91
6.1.1 描画の結合と描画の色の変更....93

6.1.2 グラフィックス対象とグラフィックスの結合....94
6.2 matplotlib による進んだ描画....96
6.2.1 網目と軸のラベルを持ったリストデータの描画....96
6.2.2 対数目盛の Y 軸を持った描画....97
6.2.3 描画の中にラベル付きの二つのグラフ....98

7 SAGE の書式....100
7.1 スピード書式....100
7.2 オープンオフィスプレゼン書式....100

8 高校数学の問題 (大半がまだ著作中)....101
8.1 Pre-Algebra....101
8.1.1 方程式....101
8.1.2 式....101
8.1.3 幾何学....101
8.1.4 不等式....101
8.1.5 線形函数....101
8.1.6 Measurement....101
8.1.7 非線形方程式....102
8.1.8 Number Sense And Operations....102
8.1.8.1 分数の約分....102
8.1.9 多項式函数....103
8.2 代数....103
8.2.1 絶対値函数....103
8.2.2 複素数....103
8.2.3 合成函数....104
8.2.4 Conics....104
8.2.5 データ解析....104
9 離散数学：初等的数とグラフ理論 104
9.1.1 方程式....104
9.1.1.1 記号分数の約分....104
9.1.1.2 二つの記号分数の積を計算....106
9.1.1.3 x について線型方程式を解く....107
9.1.1.4 分数を持つ線型方程式の解法....108
9.1.2 指数函数....110
9.1.3 冪....110
9.1.4 式....110
9.1.5 不等式....110
9.1.6 逆函数....110
9.1.7 線型方程式と函数....111
9.1.8 線型プログラミング....111
9.1.9 対数函数....111
9.1.10 兵站函数....111
9.1.11 行列....111
9.1.12 Parametric Equations....111
9.1.13 区分函数....111

9.1.14 多項式函数....112
9.1.15 冪級数函数....112
9.1.16 Quadratic Functions....112
9.1.17 Radical Functions....112
9.1.18 有理函数....112
9.1.19 列....112
9.1.20 級数....112
9.1.21 方程式系....113
9.1.22 変換....113
9.1.23 三角函数....113
9.2 Precalculus And Trigonometry....113
9.2.1 二項定理....113
9.2.2 複素数....113
9.2.3 合成函数....113
9.2.4 Conics....114
9.2.5 データ解析....114
10 離散数学: Elementary Number とグラフ理論....114
10.1.1 方程式....114
10.1.2 指数函数....114
10.1.3 逆函数....114
10.1.4 対数函数....114
10.1.5 兵站函数....114
10.1.6 行列と行列代数....115
10.1.7 数学的解析....115
10.1.8 Parametric Equations....115
10.1.9 区分函数....115
10.1.10 Polar Equations....115
10.1.11 多項式函数....115
10.1.12 冪関数....115
10.1.13 多項式函数....116
10.1.14 Radical Functions....116
10.1.15 有理函数....116
10.1.16 実数....116
10.1.17 列....116
10.1.18 級数....116
10.1.19 集合....116
10.1.20 方程式系....117
10.1.21 変換....117
10.1.22 三角函数....117
10.1.23 ベクトル....117
10.2 解析....117
10.2.1 微分....117
10.2.2 積分....117
10.2.3 極限....117
10.2.4 多項式近似と級数....118

10.3 統計.....118
10.3.1 データ解析.....118
10.3.2 Inferential Statistics.....118
10.3.3 標準分布.....118
10.3.4 １変数解析.....118
10.3.5 確率と試行.....118
10.3.6 ２変数解析.....119

11 高校生の科学の問題.....120
11.1 物理学.....120
11.1.1 原子物理.....120
11.1.2 円運動.....120
11.1.3 力学.....120
11.1.4 電磁界.....120
11.1.5 流体.....120
11.1.6 運動学.....121
11.1.7 光.....121
11.1.8 光学.....121
11.1.9 相対性理論.....121
11.1.10 回転運動.....121
11.1.11 音響.....121
11.1.12 波.....121
11.1.13 熱力学.....121
11.1.14 仕事.....122
11.1.15 エネルギー.....122
11.1.16 モーメント.....122
11.1.17 Boiling.....122
11.1.18 浮力.....122
11.1.19 Convection.....122
11.1.20 密度.....122
11.1.21 Diffusion.....123
11.1.22 Freezing.....123
11.1.23 摩擦.....123
11.1.24 熱伝導.....123
11.1.25 Insulation.....123
11.1.26 Newton の法則.....123
11.1.27 圧力.....123
11.1.28 プーリー.....124

12 計算の基礎.....125
12.1 計算機ってなあに.....125
12.2 記号(symbol)って何?.....125
12.3 記号としてビットの並びを使う計算機.....126
12.4 文脈的意味.....130
12.5 変項.....130

12.6 モデル....131
12.7 機械語....133
12.8 コンパイラとインタプリタ....136
12.9 アルゴリズム....137
12.1 計算....139
12.2 アルゴリズムの記録に図式が使える....141
12.3 1 から１０までの数の総和を計算....141
12.4 数学計算系の数学の部分....143

13 SAGE サーバーの立ち上げ**....144**
13.1 インターネットに基づく技術への入門....144
13.1.1 複数の計算機はどの様にして互いに通信するの?....144
13.1.2 TCP/IP プロトコルについて....145
13.1.3 クライアントとサーバー....147
13.1.4 DHCP....147
13.1.5 DNS....148
13.1.6 プロセスとポート....149
13.1.7 良く知られたポート, 登録されたポートと動的ポート,....153
13.1.7.1 著名なポート (0 - 1023)....154
13.1.7.2 登録されたポート (1024 - 49151)....156
13.1.7.3 動的/個人ポート (49152 - 65535)....156
13.1.8 SSH (Secure SHell) サービス....156
13.1.9 ネットワーク上の計算機間のファイルを複写する為の scp の利用について.157
13.2 SAGE の構成 (まだ)....157
13.3 Linux に基づく SAGE 版....159
13.4 SAGE の VMware 仮想計算機版(大半の Windows 利用者向け)....159

1 はじめに

1.1 献辞

この本はSteve Yeggeと彼のblog”Math Every Day(数学徒然）”に捧げます（ http://steve.yegge.googlepages.com/math-every-day).

1.1 ご了承

次の方々からは、この本についての意見を頂きました（もし、このリストの中に貴方の名前が抜け落ちていたら、どうか私宛（ted.kosan at gmail.com)に電子メイルを下さい:

Dave Dobbs
David Joyner
Greg Landweber
Jeremy Pedersen
William Stein
Steve Vonn
Joe Wetherell

1.2 支援団体

この本の面倒を見ているのは**sage-support**で、次で対処出来ます:

http://groups.google.com/group/sage-support ． このグループに電子メイルを送る時には必ず表題に ”[Newbies book]”と入れて下さい。

2 はじめに

SAGE は記号代数と数値計算の為のオープンソースの数学計算環境(MCE)です。数学計算環境は複雑なので、それに習熟する為に莫大な時間と労力を必要とします。ひとつの数学計算環境を使えるようにする為に費やされるこの莫大な労力は、それを学ぶ為に必要となる労力に値します。 初心者が SAGE 使いの専門家になるには時間が暫く掛かりますが、幸運な事に、問題を解く為に SAGE の専門家になる必要はありません。

2.1 数学計算環境とはなに?

数学計算環境は幅広い数学計算アルゴリズムを自動処理する事の出来る計算プログラムの集合です。計算アルゴリズムは数学の殆ど全ての領域に存在し、そして、新しいアルゴリズムは日々発展し続けています。

１９６０年代より、膨大な数の数学計算機環境が生成され、そして、次野リストには最も人気のある物が含まれています：

http://en.wikipedia.org/wiki/Comparison_of_computer_algebra_systems

幾つかの環境は高度に特殊化している一方で、ある物は汎用のものです。幾つかのものは伝統的な書式（これは数学の教科書で見られるものです）を表示したり、入力出来たり、ある物は伝統的な数式を表示出来ても、入力はテキストでなければならなかったり、それから、幾つかは数式出力も入力もテキストだけが可能であったりします。

伝統的な数式とテキスト書式の違いの例として、ここでは伝統的な書式の式を挙げておきます：

$$A = x^2 + 4 \cdot h \cdot x$$

それから、テキスト書式で同じ数式を示しておきましょう：

```
A == x^2 + 4*h*x
```

殆どの数学計算環境はある種の数学向け高水準プログラム言語を保有しています。

これによって、その環境が有する数学のアルゴリズムをプログラムで利用出来る様になります。幾つかのこれらの数学向けプログラム言語は、既存のプログラム言語の周囲で構築され、それらが動作する特殊な環境向けに構築されています。

ある数学計算環境は商用で購入しなければなりませんが、その他のものはオープンソースなの自由に利用出来ます。双方の環境は似た核となる機能を持っていますが、他の領域では通常異なっています。

商用の環境はオープンソース環境よりもより洗練されており、それらは往々にして GUI を持ち、数式を伝統的な書式で入力したり操作する事が比較的容易になっています。その一方で商用環境は欠点も持っています。一つの欠点はソフトウエアを保持する会社が撤退する機会が常に付き纏い、それによって環境を将来利用出来なくなる恐れがある事です。別の欠点は商用環境を拡張する事が出来ない事で、その環境のソースコードが利用者が利用出来る様になっていないからです。

オープンソースの数学計算環境は通常 GUI を持っていませんが、それらのユーザーインターフェイスは汎用性があり、その環境のソースコードを望む人に対しては何時でも使えます。これは人が興味を持てば使える環境で、望むままに拡張も出来る事を意味するのです。

2.2 SAGE はなに?

SAGE はオープンソースの数学計算環境で、数式をテキスト形式で入力し、テキスト形式、あるいは伝統的な書式で表示するものです。大半の数学計算環境が自分自身を包含するものですが、SAGE は他の数学計算環境を包含する事で、それ自体があるアルゴリズムを提供する傘の様な手法を取っています。この戦略によって SAGE は簡単に将来の需要を取り込む事が容易な構造で、複数の数学計算環境の力を提供する事が出きるのです。

SAGE は強力でとても人気のある Python プログラム言語で記述されており、SAGE の利用者が扱える数学指向のプログラム言語は Python の拡張です。これは SAGE の熟練利用者は Python プログラムの熟練者でなければならないことを意味します。Python プログラム言語の知識がまさに SAGE を上手に使いこなせるかどうかの指標となり、その SAGE の熟練度を決定する手助けとなる利用者の Python の知識もあります（表 1 参照)。

水準	知識
SAGE 習熟者	Python も SAGE もよく分かっている
SAGE 知見者	Python は知っているけど、SAGE だけを少々かじった程度
SAGE 新米	Python を知らないいけど、最低一つのプログラム言語を挙げる事が出来る
プログラム作成新米	計算機がどのように動作するか知らないし、一度もプログラムを組んだ事が無い

表 1: SAGE 利用者の経験水準

この本は SGAGE 新米の為のものです。読者は最低でも一つのプログラム言語が挙げられるものの Python でのプログラム経験は皆無だと仮定しています(もし、貴方が計算機のプログラムがどの様に動作するのかを再確認しておく必要があれば、この本の計算の基礎 の節を通読すると良いでしょう。) この本は SAGE を使って問題を解くために十分な Python の事を教示します。貴方が SAGE 知見者となる為の手助けをしますが、SAGE 熟練者になる前に、この本に関連する本から Python を学ばなければなりません。

もし、貴方がプログラムの初心者であれば、この本はおそらく貴方に取っては余りにも上級過ぎるかもしれません。私は *The Professor and Pat Programming Series*(http://professorandpat.org) という無料本を書いており、それらはプログラム初心者向けになっています。もし、貴方がプログラム初心者であって、SAGE を使って学ぶ事に興味があるのなら、*The Professor and Pat Programming* 本から進める事を第一にすべきで、それらを済ませてから、この本に戻ってくるべきです。

SAGE のウエブサイト（sagemath.org）には他の SAGE の情報源に従った SAGE の沢山の情報があります。

2.3　WEB サービスとして SAGE を利用

SAGE はその構造上可能な限り、色々な方法に柔軟に対処可能です。SAGE の超新米さんは WEB ブラウザから WEB サービスとして SAGE を使うことになるでしょう。SAGE の任意の複製はこの WEB サービスが行えるように設定されています。　図 2.1 に三つの SAGE の WEB サービスの手法を示しておきましょう:

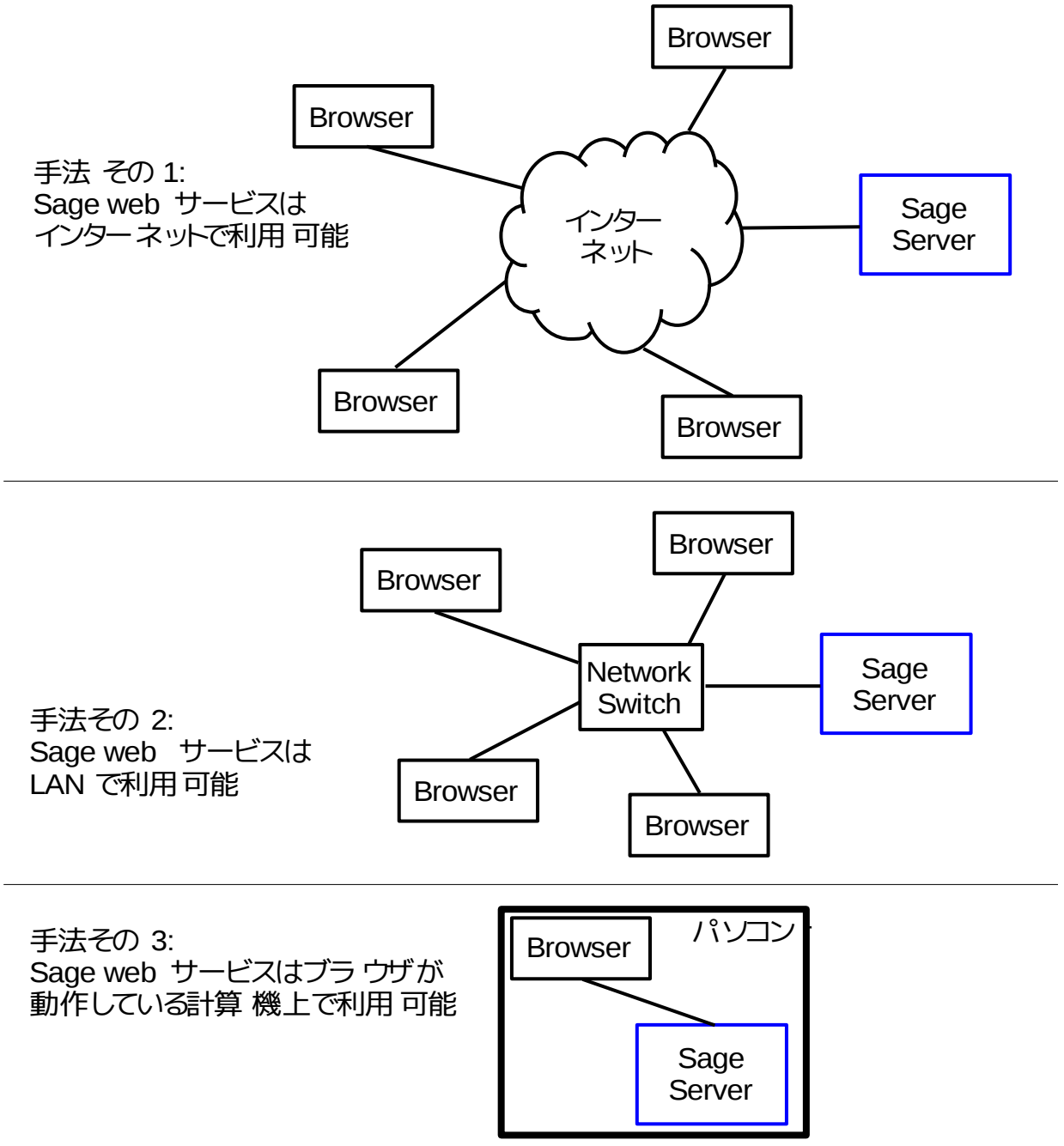


図 *2.1: 三通りの Web サービスによるセッション*

2.3.1 WEB サービスとして SAGE を利用　手法その 1

SAGE は現在 Firefox ウエブブラウザが動作に最適なので、もしも、貴方の計算機に Firefox を入れていなければ、 http://mozilla.com/firefox.で入手しておきましょう。

SAGE 開発チームは公開 SAGE ウエブサービスを(http://sagenb.com) で提供しており、このサービスは SAGE のホームページのトップからも利用出来ます。この公開 SAGE ウエブサービス向けのアカウントを取得する為には登録が必要なので、手順をここで解説しておきましょう。

Firefox ブラウザのウィンドウを開いて、下記を URL バーに書き込みます:

```
http://sagenb.com
```

すると Welcome ページが表示されます（ 図 2.2 を参照してください）

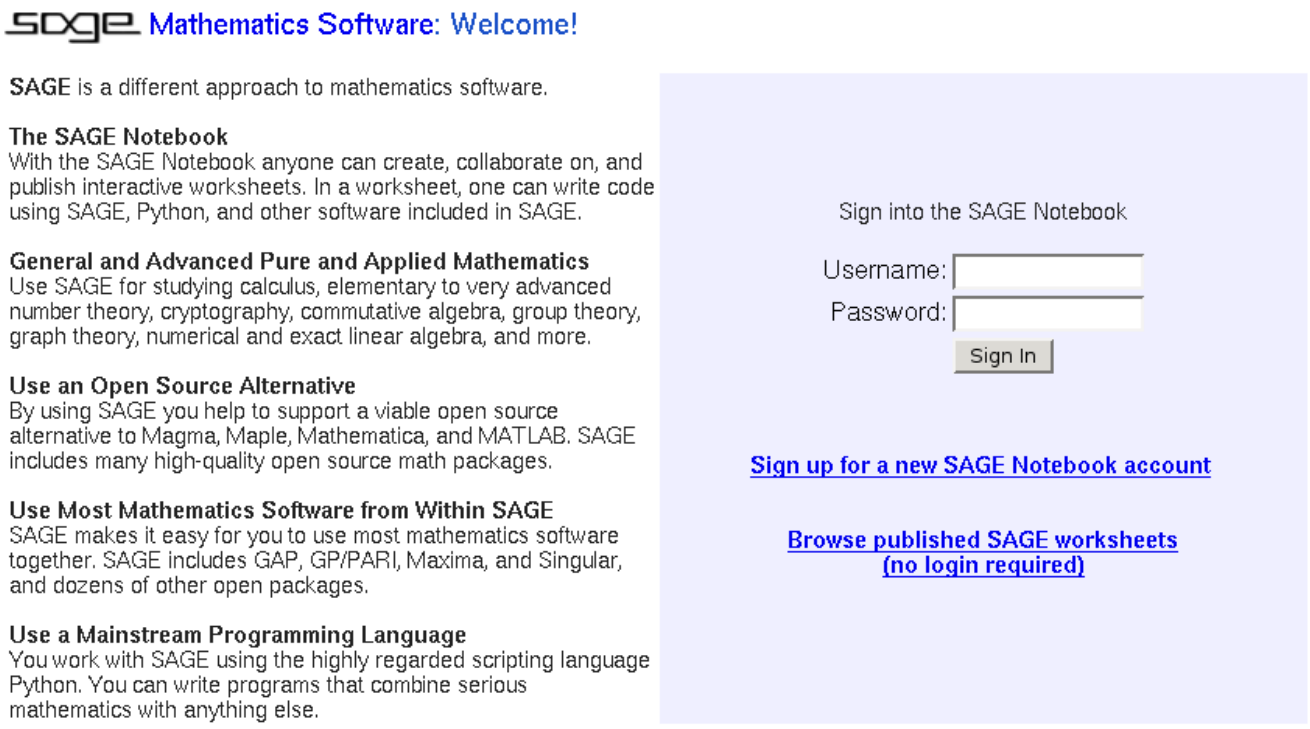


図 *2.2: SAGE Welcome screen.*

SAGE ウエブサービスは SAGE **ノート**（Notebook）と呼ばれています。何故かというと、数学

者が数学の計算を実行する為に伝統的に用いているノートを模擬しているからです。ノートが使えるようになる前に、貴方はノートのアカウントの為に最初に登録をしなければなりません。

Sign up for a new SAGE Notebook account リンクを選択すると登録ページが表示されます(図 2.3 を参照して下さい)

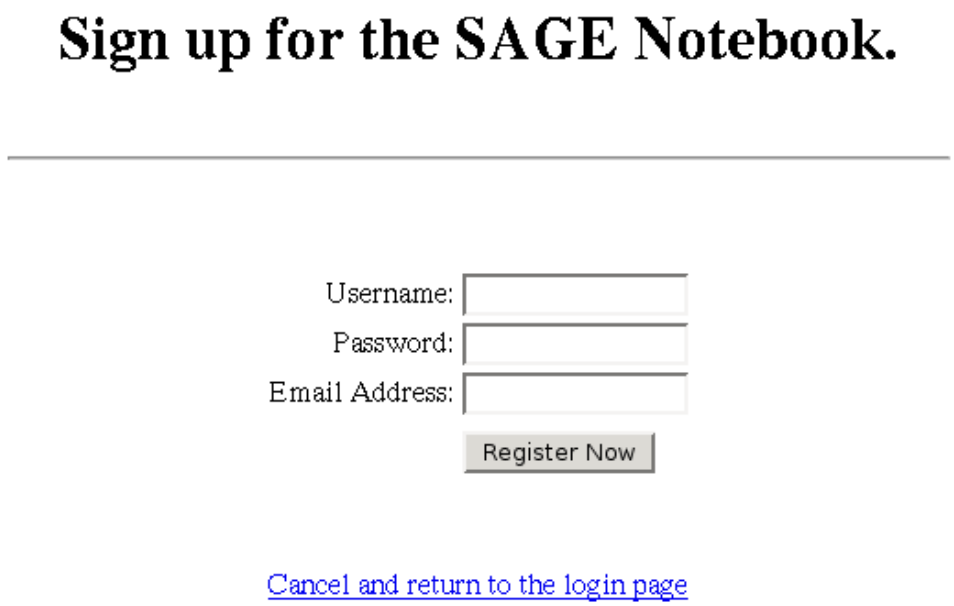


図 *2.3: Signup page.*

利用者名 (Username)とパスワード(Password)を Username と Password の入力欄に入れ、それから Register Now ボタンを押します. すると、別のページが表示され、そこには登録情報が受け取られた事と、承認メッセージが貴方が指定した電子メイルアドレスに送られた事が書かれています。

このメイルを開いて、その中に有るリンクを開きます。これで登録は完了し、いよいよ、貴方はノートの Welcome ページに戻って、ログインが出来ます。

貴方のノートのアカウントでログインが成功すると、 worksheet management ページが現れます（ 図 2.4 を参照して下さい)

図 *2.4: Worksheet management* ページ

112 物理的な数学ノートはワークシートを含んでいるので、SAGE の仮想ノートもワークシートを
113 持っています。ワークシート管理ページはワークシートの生成、削除、インターネット上で
114 印刷等の事が出来ます。これは新規に生成されたノートなので、未だ何もワークシートがあ
115 りません。

116 New Worksheet リンクを選択すると新しいワークシートを生成します。ワークシートは伝
117 統的な書式で数式を表示する為の特別な数学フォント、或いはこれらのフォントの画像が使
118 えます。もし、貴方が作業している計算樹に数学フォントが入っていなければ、ワークシー
119 トは代わりに組込みの画像フォントを用いる様に指示するメッセージが表示されます（ 図
120 2.5 参照）

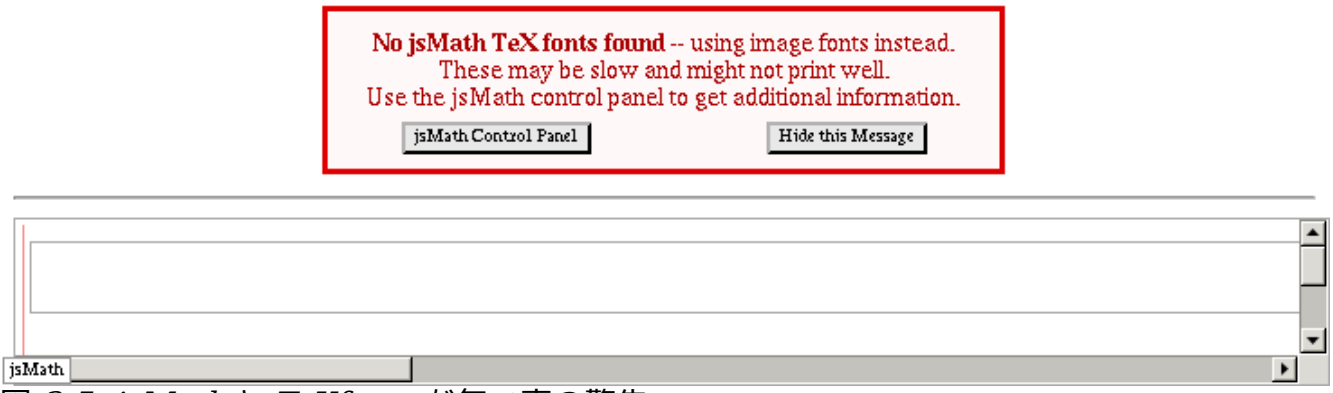


図 *2.5: jsMath* と *TeXfonts* が無い事の警告

121 画像フォンとは通常の数学フォントとしてはあまり明瞭ではありませんが、殆どの事には使
122 えます。
123 望めば、あとで数学フォントを貴方の計算機に入れられますが、ここでは取り敢えず Hide

this Message ボタンを押します。すると、真っ新のワークシートを含むページが現れます（ 図 2.6 参照）

図 *2.6: まっさらのワークシート*

ワークシートは 1 つ、或いはそれ以上のセル(cell)を含みます。ここでセルは SAGE によって実行されるソースコードに用います。セルは図 6 に示す様に長方形で、その中に沢山書込んでゆくにしたがって大きくなります。ワークシートが最初に生成された時には最初のセルはそのワークシートの上端にあり、これが貴方が通常式を書き込む場所になります。

2.4 SAGE のセルにソースコードを入力

早速、SAGE を簡単な計算樹として使って試してみましょう。貴方のマウスカーソルを貴方のワークシートの上端にあるセルの中に置いて下さい。カーソルが自動的に新しいセルの左端に置かれた事に注意して下さい。貴方は SAGE のソースコードの新しい行をセルの左端から改行を入れずに開始しなければなりません（貴方が他の事をするように指図されるまでは）。

次の式を入力しますが、Enter キーは押さないで:

```
2 + 3
```

貴方のワークシートは 図 2.7 の様になっている筈です.

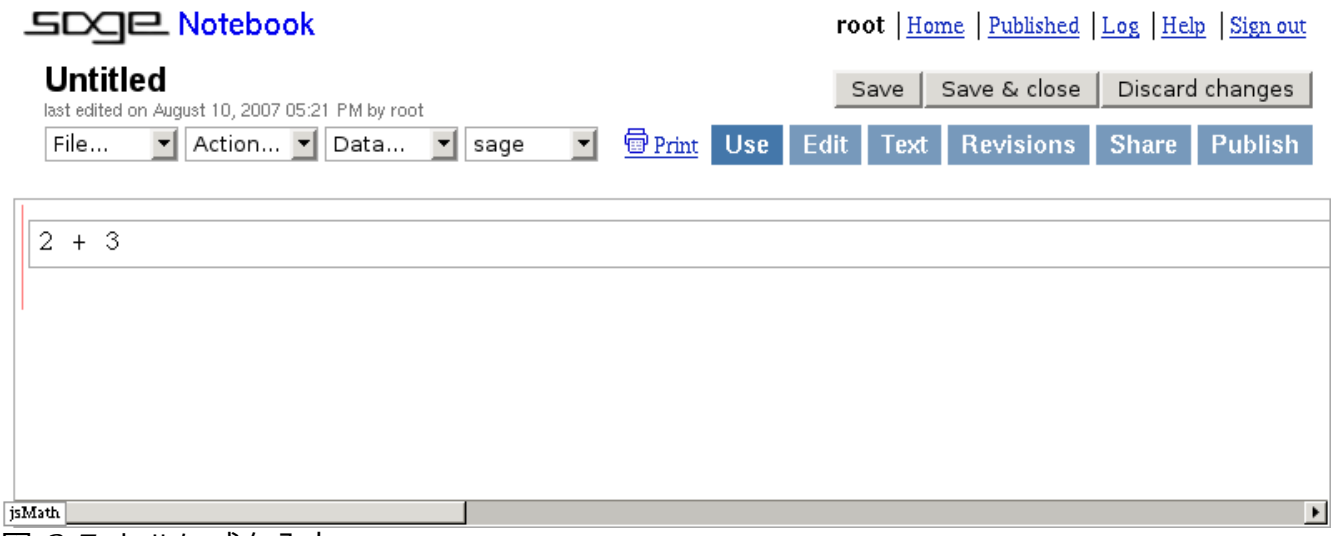


図 *2.7:* セルに式を入力

この時点で貴方には二つの選択肢があります。Enter キー<enter>を押すか、Shift キーを押しながら Enter キー<shift><enter>を押すかの何れかです。単純に Enter キーを押せば、セルは展開されてカーソルは次の行に移動して、ソースコードの入力が続けられます。

Shift と **Enter** を押すとワークシートはセルに入力されたすべてのソースコードを取り込んでネットワークを通じて SAGE サーバーにそのコードを**処理出来る様に**送り付けます。SAGE に処理すべきソースコードが与えられると最初に SAGE 前処理機と呼ばれるソフトウエアを用いた処理になります。この前処理機では SAGE ソースコードを Python ソースコードに変換して SAGE を構築した Python 環境を用いて処理出来る様にします。

変換されたソースコードは Python 環境を通過しますが、そこで、**Python バイトコード**と呼ばれる特殊な書式の機械言語に翻訳されます。このバイトコードはハードウエア CPU を模擬するプログラムで処理されますが、このプログラムの事を **Python 翻訳機**と呼びます。

サーバがそのコードを効率良く処理出来る事もあれば、時間が掛かる事もあります。そのコードがサーバーで実行されている間、件のワークシートは小さな緑の縦棒を図 2.8. に示す様にセルの下側、ウィンドウの左端に表示します。

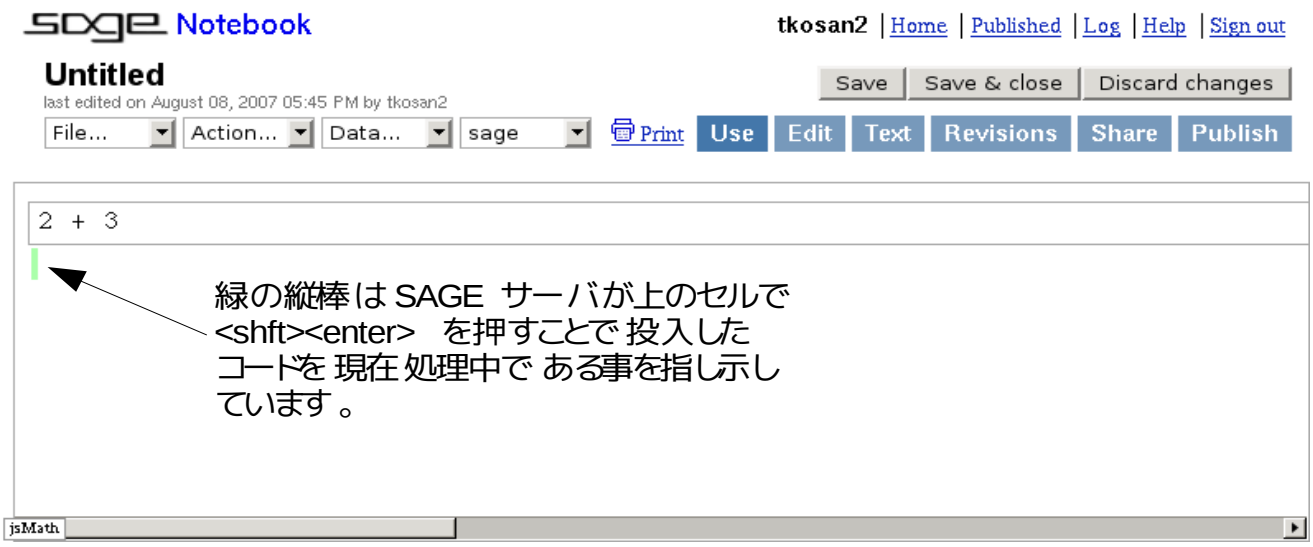


図 *2.8:* セルの式を処理中

152 サーバーがソースコードの処理を終えると緑の棒が消えます。もし、表示可能な結果が生成
153 されていれば、この結果は件のワークシートに返送され、ワークシートはその結果を式を投
154 入したセルの真下の領域に表示します。

155 ここで shift と enter を貴方のセルで押して、ちょっと待てば 図 2.9 に示すような結果が得
156 られます。

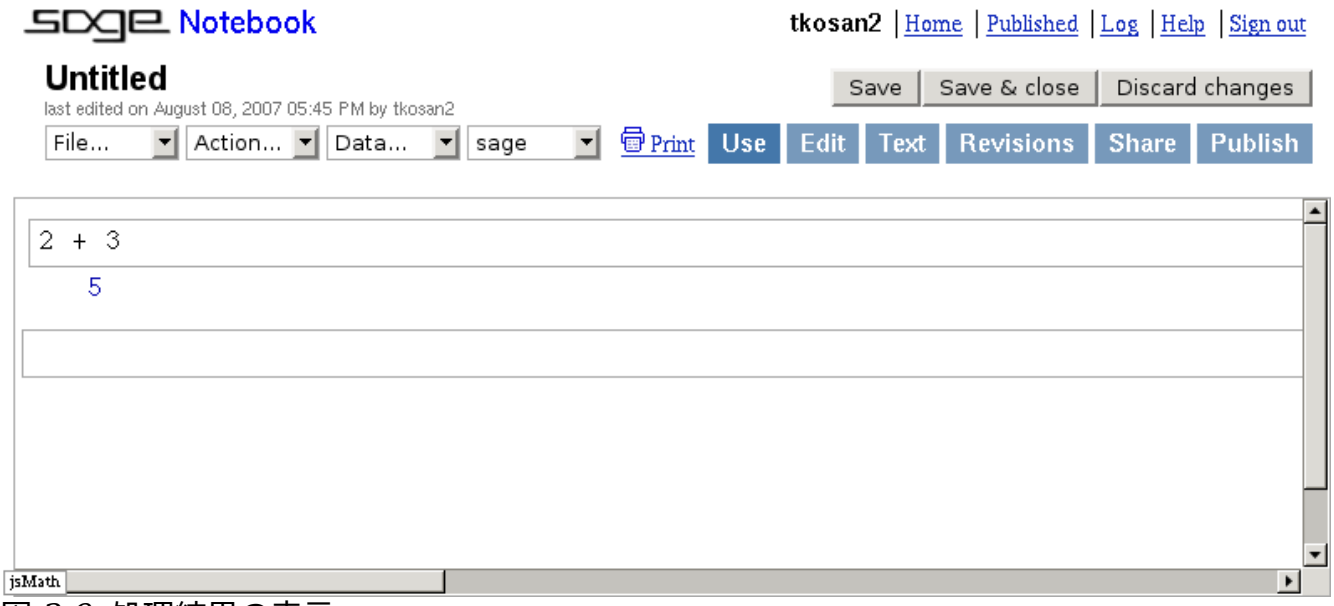


図 *2.9:* 処理結果の表示

157 ノートの下のセルから処理させるためにコードを投入すると、そのコードの処理をサーバー
158 が終えると自動的にこのセルの下に真っ新のセルが付加されます。

159 そこで、 図 2.10 の 2 番目のセルにあるソースコードを入力して処理しましょう。

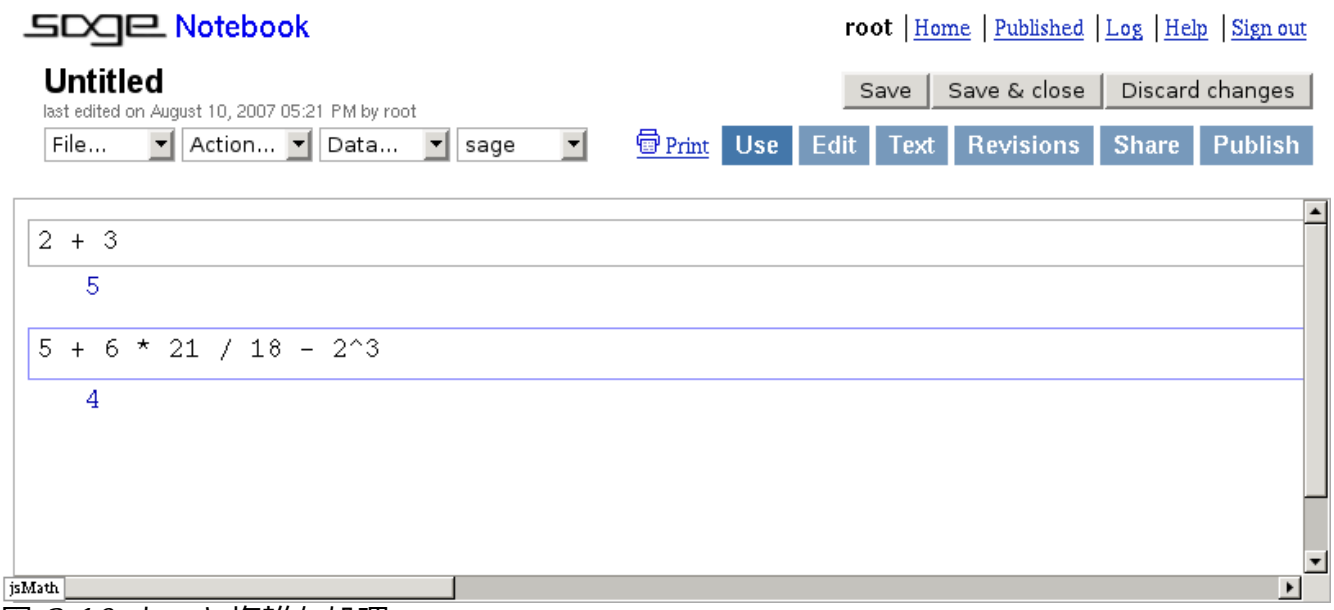


図 *2.10:* もっと複雑な処理

3 SAGE でプログラムをする上での基礎

3.1 対象、変数、そして式

ソースコード行

```
2 + 3
```

と

```
5 + 6*21/18 - 2^3
```

の双方を式(expression)と呼び、次を何が式であるかの定義とします:

プログラム言語での**式(expression)**とは、値、変数、演算子、函数の組合せであり、既存の特定の規則や、それらを計算したり別の値で置き換える特定のプログラム言語の介在で解釈（評価）されるものです。式を評価するとはその値の事を指します。数学で式はそれ自体の評価された値です（あるいは持つと言っても良いでしょう）;式はその値の表現です。(http://en.wikipedia.org/wiki/Expression_(programming))

計算機では、**値 (value)**は一つ、或いはそれ以上のメモリでのビットの並びで、与えられた文脈(context)を使って解釈される時の何者かを意味します。SAGE では意味を持ったメモリでのビットの並びを**対象(object)**と呼びます。SAGE 自体は対象で構成されており、その SAGE プログラムが生成する与件もまた対象として表現されます。対象は第 4 章 でより詳細に解説します。

上の式で,2, 3, 5, 6, 21,と 18 は対象で、**sage.rings.integer.Integer** 文脈と呼ばれる文脈を用いて解釈されます。文脈は 型(type)と呼ばれる対象に関連付けられ、型 sage.rings.integer.Integer は整数(integer)の表現で用いられる型の対象です。SAGE には type()と呼ばれる命令があり、これは引数として与えた任意の対象の型を返します。type()命令を使って、対象 3 と 21 の型が何であるか、次のコードを実行させてみましょう:(註:この点からソースコードはセルに入れるべきで、さらに表示されるべき任意の結果は GUI を使わないものにすべきです。)

```
type(3)
```

```
    <type 'sage.rings.integer.Integer'>
```

```
type(21)
|
    <type 'sage.rings.integer.Integer'>
```

対象の型の情報がなにであるかを type()に尋ねる方法は、'type'の右側の括弧の中にその対象を置くことです。

3.2 演算子

上の式の文字+,-,*,/,^は**演算子**と呼ばれ、それらの目的は SAGE に式中の対象に対して実行すべき操作が何であるかを伝える事です。例えば、式 2+3 では**和**の演算子+は SAGE に整数 2 と 3 の和を計算し、その結果を返却すべき事を伝えています。ここで対象 2 と 3 は sage.ring.integer.Integer 型なので、それらの和から得られた結果は共に sage.rings.integer.Integer 型の対象になります。

差の演算子は-で、**積**の演算子は*,/は**商**の演算子で、%は**剰余**の演算子、そして、^は**冪**の演算子です。SAGE はこれらに加え、他にも沢山の演算子を持っており、それらに関する詳細な情報は Python の文書で見つけることが可能です。

次の例では、-,*,/,%と^演算子が用いられています:

```
5 - 2
|
   3
```

```
3*4
|
   12
```

```
30/3
|
   10
```

```
8%5
|
   3
```

```
2^3
|
   8
```

文字-は負の数を示す為にも用いられます:

```
-3
|
    -3
```

負の数の差は正数になります:

```
- -3
|
   3
```

3.3 演算子の優先度

式が一つ以上の演算子を含む場合、SAGE は**演算子の優先度**と呼ばれる規則の集合を用いて、その式の対象に作用させる演算子の順位を定めます。演算子はまた**演算子の順位**も参照します。より高い優先度の演算子は優先度の低い演算子よりも先に評価されます。次の表ではSAGE の優先度の規則の一部を示し、高い優先度の演算子が表の上に来る様にしています:

^　　指数は右から左に評価されます。

*, %, /　積、剰余、そして商演算子については左から右に評価されます。T

+, -　最後に、和と差は左から右に評価されます。

沢山の演算子を含む式を手早く扱える様に、これらの優先度の規則を実際に人力で試してみましょう。ここで、ソースコードの式は:

```
5 + 6*21/18 - 2^3
```

すると、伝統的な書式はこうなります:

$$5 + \frac{6 \cdot 21}{18} - 2^3$$

この優先度規則に従うと、これが式の演算子を SAGE が評価する際の順番になります:

```
5 + 6*21/18 - 2^3
5 + 6*21/18 - 8
5 + 126/18 - 8
5 + 7 - 8
12 - 8
4
```

最初の式から始めると SAGE はその下の式の 8 を結果とする^演算子を最初に評価します。2 番目の式では*演算子が次に実行され、以降、順番に実行されて、最後の式は全ての演算子が評価された後の最後の結果として 4 を示しています。

3.4 式で演算の順番を変更する事

式に含まれる演算子の順序は括弧の中に式の一部を纏める事で変更する事が可能です。括弧の中に式を入れる事で他の演算子が評価されるよりも、その式が強制的に評価されます。例えば、式 2+4*5 は通常の優先度規則を用いて評価すると 22 になります:

```
2 + 4*5
|
    22
```

もし、4+5 の回りに括弧を置くと、積の前に強制的に和の評価が行われるので結果は 30 になります:

```
(2 + 4)*5
|
    30
```

括弧は入れ子にする事が可能で、最も内側の括弧がその外側よりも先に評価されます:

```
((2 + 4)*3)*5
|
    90
```

括弧は他の任意の演算子よりも先に評価されるので、これらを優先度表の一番上に置いておきます:

() 括弧はその内側から先に評価される。

^ 指数は右から左に評価される。

*, %, / 積、剰余、そして商演算子については左から右に評価されます。

+, - 最後に、和と差は左から右に評価されます。

3.5 変数

変数は人間が**数**の代わりにメモリの中のビットの羅列記号が参照出来る様にメモリの番地に関連させられた**名前**です。SAGE で変数を生成する一つの方法は**割当**を通じて行う事で、この場合は記号'='の左側に貴方が生成したい変数の名前を置き、記号の右側に式を置きます。式が対象を返すと対象はその変数に割当てられています・

次の例では、box という変数を生成し、数 7 をそれに割り当てます:

```
box = 7
```

以前の例とは異なり、結果がワークシートに帰っていない事に注意してください。何故なら、その結果は変数 box に置かれたからです。box の内容を見たければ真っ新のセルにその名前を入れてセルを評価します:

```
box
```

```
7
```

この例で見ることが出来る様にワークシート上の、とあるセルで生成された変数はワークシートの他のセルでも利用可能です。変数はワークシートが開かれている限り存在しますが、ワークシートを閉じると、その変数は失われます。ワークシートを再び開くと変数は割当が行われたセルを評価する事で再び生成する必要があります。変数はワークシートを閉じる前に保存する事も可能で、ワークシートを再び開いた時に読込む事も可能ですが、これは後で解説するより上級の項目になります。

SAGE 変数は大文字と小文字を区別します。これは二つ、あるいはそれ以上の変数名が同じ変数かそうでないかどうかを決定する時に、SAGE が変数名に現れる個々の文字を調べ上げる事を意味します。例えば、変数名 Box と変数名 box は同じ変数ではありません。何故なら、最初の変数名は大文字'B'で始まっていますが、二番目の変数名は小文字の'b'で開始しているからです。

プログラムは 1 変数以上持つ事が可能です。ここでは 3 変数のより洗練した例を示します:

```
a = 2
```

```
b = 3
```

```
a + b
|
    5

answer = a + b
|

answer
|
    5
```

記号'='の式の右側にある部分は常に最初に評価され、その結果は等号記号の左側の変数に割り当てられます。

変数を type()命令に通すと、変数に割り当てられた対象の型が返されます:

```
a = 4
type(a)
|
    <type 'sage.rings.integer.Integer'>
```

与件型と type 命令は後でその詳細を解説します。

3.6 文

文はアルゴリズム(algorithm)の記述で用いられるプログラム言語の一部分です。式とは違って、文は対象を返却せず、それらが作り出せる様々な効果の為に用いられます。文は式と個々の文を含み、プログラムは文の列を用いて構成されます。

3.6.1 print 文

セルの一つ以上の式が表示可能な結果を生成すると、セルは頭の式からだけ結果を表示します。例えば、このプログラムは 3 変数を生成し、これらの変数の内容を表示しようとします:

```
a = 1
b = 2
c = 3
a
b
c
|
    3
```

SAGE では、プログラムは一度にその一行を処理し、その際にコードの一番上の行の処理を開始し、そして、そこから下側の処理を行います。この例では、a = 1 の行が最初に実行されると b = 2 の行が実行され等々となります。しかしながら、全ての 3 変数に何が割当てられているかを見たくても、最後の変数に割当てられた物だけが表示される事に注意してください。

SAGE は print と呼ばれる文を持ち、それはセルに置かれた式であれば、その結果を表示する事を許容するものです。この例は前の例と一つの例外を除いて同様です。つまり、print 文は全ての 3 変数を表示する為に用いられている点です:

```
a = 1
b = 2
c = 3
print a
print b
print c
```

```
    1
    2
    3
```

print 文は、式と式の間にコンマが置かれたものが引き渡されると、同じ行に複数の結果を表示します:

```
a = 1
b = 2
c = 3*6
print a,b,c
```

```
  1 2 18
```

コンマは print 文に引き渡される変数や対象の後ろに置かれると、print 文に表示を終えたあとも次の行にカーソルを落とさない事を指示します。それ故に、print 文が実行されたあとには、その前の print 文の出力と同じ行にその出力が置かれます。

一つのセルから複数の結果を表示させる別の方法はセミコロン';'を使うことです。SAGE で

はセミコロンは文の末尾にオプションの行の末尾を示す文字として置く事が出来ますが、大半は同じ行に複数の文を置く為だけに用いられています。次の例ではセミコロンは変数 a,b と c を一行で初期化する為に用いられていることを示すものです:

```
a=1;b=2;c=3
print a,b,c
|
   1 2 3
```

次の例は、セミコロンがセルからの複数の結果を出力する為にも、どのように用いられているかを示すものです:

```
a = 1
b = 2
c = 3*6
a;b;c
|
   1
   2
   18
```

3.7 文字列(string)

文字列(string)は**文字に基づく**情報を保つ為に用いる対象の型です。文字列の対象を生成する為に使われる典型的な式は**二重引用符**或いは**単引用符**で囲まれた文書で出来ています。文字列は丁度数の様に変数を用いて参照することが可能で、print 文を使って文字列を表示する事も出来ます。次の例では文字列対象を変数'a'に割り当て、文字列対象を'a'を参照する事で表示し、それから、その型も表示させています。

```
a = "Hello, I am a string."
print a
type(a)
|
   Hello, I am a string.
   <type 'str'>
```

3.8 注釈(comment)

ソースコードは往々にして理解し難いもので、それ故に、全てのプログラム言語ではコードの中に注釈を書込めます。注釈はその近くにあるコードが何を遂行する物であるかを説明する為に用いられ、ソースコードを眺めている人間が読むものと意味づけられています。注釈

はプログラムが実行される時には無視されます。

SAGE でソースコードに注釈を入れる方法が二つあります。第一の方法は記号'#'を任意の文書の左側に置いて、サーバーに注釈である事を意味付けておきます。記号'#'から行末尾迄の文書は注釈として扱われます。ここで、記号'#'を使った注釈を含むプログラムを示しておきます:

```
#これは注釈
x = 2 #変数 x に 2 を割り当て
print x
|
   2
```

このプログラムを実行すると、記号'#'で開始する文書は無視されます。

SAGE プログラムに注釈を入れる第二の方法は 3 個の引用符で注釈を囲む事です。このオプションは注釈が長過ぎて一行に収まらない時に便利です。このプログラムは 3 個の引用符を用いた注釈を示しています:

```
"""
これは長い注釈で一行よりも長いものに使います。
次のコードでは変数 x に 3 を割り当てて、x を印字します。
"""

x = 3
print x
|
   3
```

3.9 条件演算子

条件演算子は二つの対象を比較する為に用いられる演算子です。条件演算子を含む式は論理値 (boolean) 対象を返し、論理値対象は True か False の何れか一つになります. 表 2 に SAGE で用いる条件演算子を示しておきます:

演算子	概要
x == y	二つの対象が同値であれば True を返し、それらが同値でなければ False を返します。==は比較を実行し、=の様な意味を持たないことに注意してください
x <> y	対象が同値でなければ True を返し、同値であれば False 。
x != y	同値でなければ True を返し、同値ならば False 。
x < y	左側の対象が右側よりも小さければ True を返し、左側の対象が右側よりも小さくなければ False 。
x <= y	左側の対象が右側の対象以下であれば True を返し、左側の対象が右側の対象以下でなければ False 。
x > y	左側の対象が右側の対象よりも大であれば True を返し、左側の対象が右側の対象よりも大でなければ False。
x >= y	左側の対象が右側の対象以上であれば True を返し、左側の対象が右側の対象以上でなければ False 。

表 2: 条件演算子

次の例では、 表 2 の各条件演算子が変数 x と y に置かれた対象の比較で用いられている様子を示すものです:

```
# Example 1.
x = 2
y = 3

print x, "==", y, ":", x == y
print x, "<>", y, ":", x <> y
print x, "!=", y, ":", x != y
print x, "<", y, ":", x < y
print x, "<=", y, ":", x <= y
print x, ">", y, ":", x > y
print x, ">=", y, ":", x >= y
```

```
    2 == 3 : False
    2 <> 3 : True
    2 != 3 : True
    2 < 3 : True
    2 <= 3 : True
    2 > 3 : False
    2 >= 3 : False
```

```
# Example 2.
```

```
x = 2
y = 2

print x, "==", y, ":", x == y
print x, "<>", y, ":", x <> y
print x, "!=", y, ":", x != y
print x, "<", y, ":", x < y
print x, "<=", y, ":", x <= y
print x, ">", y, ":", x > y
print x, ">=", y, ":", x >= y
|
   2 == 2 : True
   2 <> 2 : False
   2 != 2 : False
   2 < 2 : False
   2 <= 2 : True
   2 > 2 : False
   2 >= 2 : True

# Example 3.
x = 3
y = 2

print x, "==", y, ":", x == y
print x, "<>", y, ":", x <> y
print x, "!=", y, ":", x != y
print x, "<", y, ":", x < y
print x, "<=", y, ":", x <= y
print x, ">", y, ":", x > y
print x, ">=", y, ":", x >= y
|
   3 == 2 : False
   3 <> 2 : True
   3 != 2 : True
   3 < 2 : False
   3 <= 2 : False
   3 > 2 : True
   3 >= 2 : True
```

条件演算子は今迄解説してきた他の演算子と比較して低い優先度に置かれています:

()　　括弧はその内側から先に評価される。

^　　指数は右から左に評価される。

*,%,/　積、剰余、そして商演算子については左から右に評価されます。

+, -　それから和と差は左から右に評価されます。

==,<>,!=,<,<=,>,>=　最後に条件演算子が評価されます。

3.10 if 文による判断

全てのプログラム言語には判断を行う機能が提供されており、SAGE で最も一般的に用いられる判断を行う文として **if** 文があります。

If 文の単純化した構文は次のようになります:

```
if <式>:
    <文>
    <文>
    <文>
    .
    .
    .
```

if 文の動作は、その if のすぐ右に置かれた式を評価して、それから、返却された対象を調べます。ここで、この対象が"真"であれば **if** 文内部の文が実行されます。対象が"偽"であれば、**if** 文内部の文は実行されません。

SAGE では対象が**非零**、或いは**非空**であれば"真"で、**零**、あるいは**空**になれば"偽"となります。一つ、あるいはそれ以上の条件演算子を含む式は**論理値**(boolean)対象の **True** か **False** を返却します。

if 文内部に文を置く方法は、if 文の先頭部分の末尾にコロン':'を置いて、一つ、あるいはそれ以上の文をその下に置きます。if 文内側に置く文は、if 文の行に続けて、一つあるいはそれ以上の **tab** や **space** を使って、if 文の行の左側の先頭から字下(indent)をしていなければなりません。その上、全ての字下をされた文は同じ方法で、しかも、同じ分量だけ字下を行っていなければなりません。この様に字下げされた一つ、あるいはそれ以上の文はコードの一塊として参照されます。

次のプログラムでは **if** 文を変数 x に割り当てられた数値が 5 よりも大であるかどうかを決定させる為に用いています。 もし、x が 5 よりも大であれば、プログラムは"Greater"を表示

し、それから”End of program”を表示します。

```
x = 6

print x > 5

if x > 5:
    print x
    print "Greater"

print "End of program"
|
    True
    6
    Greater
    End of program
```

このプログラムの x は 6 が割り当てられており、それ故に x > 5 は真となります。この式が表示された時、 6 が 5 よりも大なので論理値対象 True を表示します。

if 文で式を評価し、それが True の時、if 文はその内側の print 文を実行し、それから変数 x に割り当てられた対象を文字列”Greater”の前に表示します。もし、追加の文が if 文の内側に置く必要があれば、その追加の文は print 文と同じ字下位置で、その下に追加します。

最後に、終に表示した文字列”End of program”は if 文が何をしようと無関係です。
ここで 6 の代わりに x が 4 を割り当てられていた場合は:

```
x = 4

print x > 5

if x > 5:
    print x
    print "Greater."

print "End of program."
|
    False
    End of program.
```

ここで、式 x > 4 は対象 False を返し、これは if 文がその内側の文の実行を行わない原因となります。

3.11 論理演算子 and, or, not

二つ、あるいはそれ以上の式が全て真であるかどうかを検証したければ and 演算子を使いましょう：

```
a = 7
b = 9
print a < 5 and b < 10
print a > 5 and b > 10
print a < 5 and b > 10
print a > 5 and b < 10
if a > 5 and b < 10:
    print "These expressions are both true."
|
    False
    False
    False
    True
    These expressions are both true.
```

また一群の式のうち、少なくとも一つが真であるかどうかを決定したければ or 演算子を使いましょう：

```
a = 7
b = 9
print a < 5 or b < 10
print a > 5 or b > 10
print a > 5 or b < 10
print a < 5 or b > 10

if a < 5 or b < 10:
    print "At least one of these expressions is true."
|
    True
    True
    True
    False
    At least one of these expressions is true.
```

最後に、not 演算子で True の結果を False に、あるいは False の結果を True に変更することに使えます：

```
a = 7
print a > 5
print not a > 5
```

```
|
   True
   False
```

ブール演算子は他の演算子と比較してより低い優先度にしています：

()　括弧は中から外へと評価します。

^　それから冪は右から左と評価します。

*,%,/　それから、積、剰余、商演算子が左から右へと評価されます。

+, -　そして、加法と減算が左から右へと評価されます。

==,<>,!=,<,<=,>,>=　それから条件演算子が評価されます。
not　ブール演算子が最後に評価されます。

and

or

3.12 while 文による反復処理

計算機を含め、多彩な機械は繰り返される一連の動作という原理から、それらの力の大半を導き出しています。

SAGE はプログラムに於ける反復処理を実装する為に、いくつかの手法を提供しますが、それらの手法は単刀直入のものからまどろっこしいものまであります。

単刀直入的な **while** 文から開始する事で、SAGE での反復処理の解説を始めましょう

While 文の構文指定は次のようなものです:

```
while <式>:
    <文>
    <文>
    <文>
    .
```

599 .
600 .

601 while 文は if 文に似ていますが、その頭の右側にある式が真である限り、その while 文が含
602 む文を反復して実行する事が異なります。その式が False の値を持つ対象になると、 **while**
603 文は包含する文の実行を飛ばして **while** 文に続く文（存在していれば)の処理を継続して実
604 行します。
605
606 次の例題は while を用いた 1 から１０までの整数を表示する反復処理です:

607 ```
１から１０までの整数を表示
```

608 ```
x = 1  #Initialize a counting variable to 1 outside of the loop.
```

609 ```
while x <= 10:
610   print x
611   x = x + 1 #Increment x by 1.
612 |
613   1
614   2
615   3
616   4
617   5
618   6
619   7
620   8
621   9
622   10
```

623 このプログラムでは、 **x** という名前の一つの変数が生成されます。その変数は **print** 文に
624 どの整数を印字すべきかを伝え、そしてまた **while** による反復処理を継続すべきかどうかを
625 決定することにも用いられます。

626 プログラムが実行されると、 1 は x に入れられて、それから **while** 文に入ります。式 x<=10
627 は 1<=10 になって、 1 が１０以下なので論理対象が True を包含しているとその式を繰り返し
628 ます。

629 while 文はその式が真となる対象が返されるので、その文に含まれるすべての文を実行しま
630 す。

631 print 文でその時点の x に含まれる値(それは 1 ）を印字し、それから x=x+1 が実行されま
```

す。

式 x=x+1 は多くのプログラミング言語で用いられる標準的な式です。この書式の式が評価される度に、変数が含む値に 1 を加えます。この式の x に与える効果は別途、x に 1 増分するとも言います。

この場合では、x は 1 を包含し、その式の評価の後には x は 2 を包含します。

while 文内部の最後の文が実行されると、**while** 文は while の左側にある式の再評価を行って、反復を継続するかどうかを決定します。この時点では x が 2 なので、その式は **True** を返すので、コード内部の **while** 文は再び実行されます。この反復処理は x が１１に達して、その式が **False** を返すまで実行されます。

このプログラムは別の結果が得られる様に調整する事ができます。　例えば、次のプログラムは 1 から１００迄の整数を表示する様に while の右にある 10 を１００に書き換えたものです。そして、コンマを出力がウィンドウの右側に当たるまで同じ行に表示される様に print 文の後ろに置きます。

```
#  1から１００迄の整数を印字

x = 1

while x <= 100:
    print x,
    x = x + 1  #Increment x by 1.
|
  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 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51
  52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75
  76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99
  100
```

次のプログラムは増分を 1 から 2 に変更する事で 1 から９９迄の奇数を印字します:

```
# 1から９９迄の奇数を印字

x = 1

while x <= 100:
    print x,
    x = x + 2  #Increment x by 2.
|
```

```
  1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43 45 47 49 51
  53 55 57 59 61 63 65 67 69 71 73 75 77 79 81 83 85 87 89 91 93 95 97 99
```

最後に、このプログラムは 1 から１００迄の数を逆の順序で印字します:

```
#  1から１００迄の整数を逆順で印字

x = 100

while x >= 1:
    print x,
    x = x - 1  #Decrement x by 1.
|
  100 99 98 97 96 95 94 93 92 91 90 89 88 87 86 85 84 83 82 81 80 79 78 77
  76 75 74 73 72 71 70 69 68 67 66 65 64 63 62 61 60 59 58 57 56 55 54 53
  52 51 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36 35 34 33 32 31 30 29
  28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2
  1
```

この結果に達する為に、x の初期値を１００にするので、反復処理を継続する為に **x が 1 以上**(x>=1)である事を検証し、 1 を加えるのではなく x から 1 を引く様にしています。

3.13 長い反復処理，無限反復処理，そして処理の中断

膨大な時間を費やしたり、更には、無限の時間を費やす様な反復処理を行わせる事は、まっとうな事でも錯誤からであったも簡単な事です。無限反復処理を含むプログラムを実行してしまったとき、
SAGE にその実行を中断する事を伝えない限り処理が継続します。これはワークシートの左上側の Action メニューを選択して Interrupt メニュー項目を選択する事で行われます。長く実行されている反復処理を持つプログラムはこの方法でも中断する事ができます。双方で、緑の縦棒がプログラムが実行中である事を示しており、プログラムを中断すれば緑の棒が消えます。

このプログラムは無限反復処理を包含します:

```
#無限反復処理の例

x = 1
while x < 10:
    answer = x + 1
|
```

反復処理内部の x の包含する値が決して変化しないので、式 x<10 は常に True と判断されるので、反復処理が継続するのです。

このプログラムを実行し、ワークシートの Interrupt 命令を使って中断します。時にはワークシートが
実行を止めるのに十分でない事があれば、Action->Restart worksheet を選択しましょう。
ワークシートを再起動する時には、全ての変数がその初期状態に戻されるので、それらの変数に値を指定したセルを再び実行する必要があるでしょう。

3.14 ワークシートのセルの挿入と削除

新しいワークシートのセルが必要になると、貴方のマウスカーソルをセルを挿入する上下の二つのセルの間に移動すると**水平の青色の棒**が現れます。この青い棒を押すと新しいセルがワークシートのその箇所に挿入されます。

もしもセルを削除したければ、セルのすべての文書を削除して、それを空にします。カーソルを空のセルに置いたことを確認してからキーボードのバックスペースキーを押します。するとセルが削除されます。

3.15 より高度な対象の型について

この点について、我々は単に'sage.rings.integer.Integer'と'str'という対象の型しか扱っていません。しかし、SAGE は多様な目的に応じて使える莫大な数の数学的、或いは非数学的対象の型を保有します。次の節では、新たに二つの数学的対象と二つの非数学的対象を紹介しましょう。

3.15.1 有理数

有理数は sage.rings.rational,Rational 型の対象に属します。次の例で、有理数 1/2 の型を表示し、変数 x に 1/2 を割り当て、それから x が参照する対象の型を表示させています:

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

```
    <type 'sage.rings.rational.Rational'>
    1/2
    <type 'sage.rings.rational.Rational'>
```

次のプログラムは前のプログラムを実行した後に、そのワークシートの別のセルに入力したものです。二つの有理数同士の和とその結果を示すもので、結果も有理数で、変数 y に割り当てられています:

```
y = x + 3/4
print y
type(y)
|
    5/4
    <type 'sage.rings.rational.Rational'>
```

有理数と整数の和を行うと、その結果は sage.rings.rational.Rational 型の対象になります。

```
x = 1 + 1/2
print x
type(x)
|
    3/2
    <type 'sage.rings.rational.Rational'>
```

3.15.2 実数

実数は **sage.rings.real_mpfr.RealNumber** 型の対象に属します。 次の例題では、実数.5 の型を表示し、変数 x に.5 を割り当てて変数 x を表示し、それから x が参照する対象の型を表示させています:

```
print type(.5)
x = .5
print x
type(x)
|
    <type 'sage.rings.real_mpfr.RealNumber'>
    0.500000000000000
    <type 'sage.rings.real_mpfr.RealNumber'>
```

次の処理は前の処理を実行した後で、そのワークシートの別のセルに入力したものです。二つの実数同士の和とその結果を示し、その結果もまた実数で、変数 y に割り当てています:

```
y = x + .75
print y
type(y)
|
```

```
    1.25000000000000
    <type 'sage.rings.real_mpfr.RealNumber'>
```

実数と有理数との和により、その結果は sage.rings.real_mpfr.RealNumber 型の対象となります:

```
x = 1/2 + .75
print x
type(x)
|
    1.25000000000000
    <type 'sage.rings.real_mpfr.RealNumber'>
```

3.15.3 他の対象の列を包含する対象: リストとタプル

list 対象の型は順序がある対象、即ち、**列(sequence)**を保持するために設計されています。List はとても柔軟性があって、SAGE で最も頻繁に用いられる型の一つです。list は任意の型の対象を保持する事が可能で、必要に応じて大きくしたり小さくしたりする事が出来れば、入れ子にする事も出来ます。list に含まれる対象は list に於けるそれらの位置で取り出す事や他の対象で置き換える事も出来ます。list を大きくしたり小さくしたりする事や、その包含する内容を変えられる能力により**可変(mutable)**対象型となります。大括弧の対の替わりに 0 あるいはそれ以上の対象や式を置く事でリストを生成する手法があります。次のプログラムではリストの型を印字する事から開始します。それから、それは数 50, 51, 52 と 53 を含むリストを生成し、変数 x に割り当てて x を印字します。次に 0 と 3 の位置にある対象を印字し、3 の位置にある 53 を 100 で置き換えて x を再度印字し、それから最後に x の対象の型を印字します:

```
print type([])
x = [50,51,52,53]
print x
print x[0]
print x[3]
x[3] = 100
print x
type(x)
|
    <type 'list'>
    [50, 51, 52, 53]
    50
    53
    [50, 51, 52, 100]
    <type 'list'>
```

リストの最初の対象が 1 ではなく 0 の位置であり、これによってリストの最後の対象の位置はそのリストの長さよりも一つ小さくなる事に注意しましょう。また、変数の右側に位置番号を大括弧で括れば、リストの中の対象を参照することになります。

次の例では異なった型の対象がリストの中に置ける事を示します：

```
x = [1, 1/2, .75, 'Hello', [50,51,52,53]]
print x
|
   [1, 1/2, 0.750000000000000, 'Hello', [50, 51, 52, 53]]
```

タプル（Tuples)もまた**列**(sequences）で、不可変であることを除いてリストに似ています。タプルは大括弧ではなく**小括弧**を使うことで生成され、不可変(immutable)であると言う事は、一旦、タプル対象を生成すると、大きく、小さくしたり、それが包含する対象を変えられない事を意味します。次のプログラムは最初のリストプログラムに似ていますが、リストの代わりにタプルを用いており、位置 4 の対象を変更しようとせず、print 文の代わりにセミコロンを使う手法で複数の結果を表示させています：

```
print type(())
x = (50,51,52,53)
x;x[0];x[3];x;type(x)
|
   <type 'tuple'>
   (50, 51, 52, 53)
   50
   53
   (50, 51, 52, 53)
   <type 'tuple'>
```

3.15.3.1 多変数の取りまとめと取り崩し

コンマで分離された多変数が一つの変数として割り当てられると、その変数は自動的にタプルとして置かれ、これを**タプル纏め**(tuple packing)と呼びます：

```
t = 1,2
t
|
   (1, 2)
```

タプルにコンマで区切った複数の変数を割り当てる時、これを**タプル展開(tuple unpacking)**と呼びます：

```
a,b,c = (1,2,3)
a;b;c
|
    1
    2
    3
```

タプル展開をする為にはタプル内部の対象の数は等号左側にある変数の総数と一致しなけれなりません。

3.16 リストとタプルを使った while 反復処理の利用

反復処理を行う文はリストやタプルの各成分を次々と選択して、処理がそれらの対象に対して処理されるようにすることが出来ます。次のプログラムはリストの各成分を表示する為に while 文による反復処理を使っています：

```
#リストの各成分を印字
x = [50,51,52,53,54,55,56,57,58,59]
y = 0
while y <= 9:
    print x[y]
    y = y + 1
|
   50
   51
   52
   53
   54
   55
   56
   57
   58
   59
```

反復処理はまたリストでの検索にも使えます。次のプログラムでは**while** による反復処理と if 文を使って、数 53 を含むかどうかリストの検索で用います。もし、53 をリストの中で見

つけると、メッセージを表示します。

```
# 53 がリストにあるかどうかを判断。
x = [50,51,52,53,54,55,56,57,58,59]
y = 0
while y <= 9:
    if x[y] == 53:
        print "53 was found in the list at position", y
    y = y + 1
|
    53 was found in the list at position 3
```

3.17 in 演算子

反復処理は非常に便利な機能であり、内部的な反復を行う in と名付けられた演算子さえも SAGE は持っています。 in 演算子はリストに与えられた対象が含まれているかどうかを判断する為に自動的に検索を行うことが出来ます。もし、その対象を見つけると、True を返却しますが、対象がみつからなければ False を返します。次のプログラムは両方の場合を示しています:

```
print 53 in [50,51,52,53,54,55,56,57,58,59]
print 75 in [50,51,52,53,54,55,56,57,58,59]
|
    True
    False
```

not 演算子はその結果を変更する為に in 演算子と併用する事も出来ます:

```
print 53 not in [50,51,52,53,54,55,56,57,58,59]
print 75 not in [50,51,52,53,54,55,56,57,58,59]
|
    False
    True
```

3.18 for 文を使った反復処理

For 文は while 文でやっている様に、リストやタプルからの添字を用いた反復を使いますが、もっと柔軟で自律的です。ここでは for 文の簡略化した構文を示しておきます：

```
for <target> in <対象>:
```

```
        <文>
        <文>
        <文>
        .
        .
        .
```

この構文で、<target> は通常変数であり、<対象>は通常、複数の対象を包含する一つの対象です。この節の残りでは、<対象>はリストであると仮定しておきます。**for** 文は交互にそのリストの対象を選出し、<target>に割り当て、インデントされた箇所の内部の文を実行します。次のプログラムでは for 文をリスト内部の項目のすべてを表示する為に用いています：

```
for x in [50,51,52,53,54,55,56,57,58,59]:
    print x

   50
   51
   52
   53
   54
   55
   56
   57
   58
   59
```

3.19 函数

プログラムした **Function** は、コードの名付けられたブロックで構成されていて、一つ、あるいは何回もそのプログラムの他の部分から呼び出されることによって実行され得る文です。函数は呼び出したコードから引き渡される対象を持ち、函数は呼び出したコードに対象を返します。函数の例として、対象の型を決定する事に使える type()命令を挙げておきます。

函数は SAGE がコードを再利用できる一つの手法です。大半のプログラム言語では、この方法でコードが用いられませんが、再利用される文のコードの型はサブルーチン、あるいは手続と呼ばれています。函数名はすべての小文字を使います。もし、函数名が一つの語(例えば、calculatesum の様に)を含んでいれば、下線を語と語の間に置いて(calculate_sum のように)読み易くする事が出来ます。

3.20 def 文を用いた函数の定義

函数定義で用いられる文は **def** と呼ばれ、その構文は次の通りです：

```
def <函数名>(arg1, arg2, ... argN):
    <文>
    <文>
    <文>
    .
    .
    .
```

def 文は引数を伴った函数名を包含するヘッダを含みます。ここで引数は省略が可能です。函数は 0、又はそれ以上の引数を持つことが可能で、それらの引数は括弧の中に置かれます。函数が呼び出されたときに実行される文はインデント付けられたプログラムブロックを使った函数の内部に置かれます。

次のプログラムは addnums と呼ばれる函数を定義し、この函数は二つの数を引数として取り、それらを足し合わせ、それから return 文を使って輪を返します：

```
def addnums(num1, num2):
    """
    num1 と num2 の和を返す
    """
    answer = num1 + num2
    return answer

#函数を呼び出し、2 に 3 を加えさせる C
a = addnums(2, 3)
print a

#函数を呼び出して 4 に 5 を加えさせる
b = addnums(4, 5)
print b
|
  5
  9
```

最初にこの函数を呼び出すと、数字の 2 と 3 が引き渡されて、これらの数は変数 **num1** と **num2** にそれぞれ割り当てられます。函数の呼び出しの間に対象が引き渡された引数変数は必要に応じて函数内部で利用することが出来ます。

956 函数がそれを呼び出したものに結果を返却する時に、return 文の右側に置かれた対象は呼
957 び出したプログラムの内部で利用出来ることに注意してください。函数自体が、その返却す
958 る対象で置き換えられるかのようです。返却された対象について別の見方は、呼び出した
959 コードの中で函数名の左側に送り出され、しかしながら、符号は一致し、その変数に割当て
960 られるというものです。最初の函数呼出では、函数が返す対象は変数'a'に割り当てられたも
961 ので、この対象が印字されています。第２の函数呼出は最初の函数呼出に似ていますが、異
962 なった数(4, 5)を函数に与えている点が異なります。

963 3.21 SAGE 内部の函数の一部

964 SAGE は幅広く多様な目的に対応出来る様に莫大な数の組込み函数があります。
965 表 3 にはこれらの函数の一部を示しておきますが、SAGE の文書にはもっと長い函数の一覧
966 があります。より完全な函数の一覧は *SAGE Reference Manual* にあります。

Function Name	Description
abs	引数の絶対値を返却
acos	逆余弦函数.
add	数列(文字列ではない)と媒介変数'start'の値の総和を返します。列が空であれば start の値を返却します。
additive_order	x の additive order を返す。
asin	逆正弦函数。
atan	逆正接函数。
binomial	二項係数を返す函数。
ceil	足切り函数
combinations	複合集合(同時に同じ対象を含んでいるかもしれない対象のリスト)の組合せの mset は反復なしで順序付けられていない選出であり、mset のソートされた副リストによって表現されています。k 成分の複合集合 mset の全ての組合せの集合を返却します。
complex	実部とオプションの虚部から複素数を生成します。つまり(実部 + 虚部*1j)と同値で、実部の附置は０です。
cos	正弦函数
cosh	双曲余弦函数
coth	双曲余接函数
csch	双曲余割函数
denominator	x の分母
derivative	f の微分
det	x の行列式
diff	f の微分
dir	与えられた対象の(幾つかの)属性込みの名前や対象から到達可能な属性の一覧をアルファベット順で返します。
divisors	全ての整数因子を返す
dumps	文字列 s に対象 obj を吐き出す。対象 obj の復元には load(s)を使います。
e	自然底
eratosthenes	素数<= n のリストを返す。
exists	S が P(x)が True となる成分 x を含む場合、この函数は True と成分 x を返し、それ以外は False と None を返します
exp	指数函数, exp(x) = e^x.
expand	多項式を展開した書式で返す

factor	整数の因数分解をタプルのリストとして返します
factorial	nの階乗を計算します。階乗は積　1 * 2 * 3 ... (n-1) nです。
fibonacci	n番目のFibonacci数を返却.
fibonacci_sequence	n=startからn=stop迄の全てのfibonacci数f_nに対して、Fibonacci列の反復を返却します。
fibonacci_xrange	与えた範囲でFibonacci数のリストを返しますが、f_n=startを含みますが、f_n=stopは含みません。
find_root	閉区間[a, b](または[b, a])に於ける根を可能なら探す。ここでfは1変数の函数である。
floor	floor函数
forall	P(x)がSの全てのxで真であればTrueとNoneを返します。もし、Sのある成分xでPがTrueにならなければ、Falseとxを返します
forget	仮定を忘却する、あるいは無引数で全ての仮定を忘却する。ここで仮定は記号拘束の一種です
function	記号函数を生成します。
gaussian_binomial	ガウス分布を返します。
gcd	aとbの最大公約因子を返却
generic_power	mが非負であれば、aのm乗を返します。
get_memory_usage	記憶の状態を返します。
hex	整数、あるいは長整数の16進表現を返す
imag	xの虚部を返す
imaginary	複素数の虚部を返す
integer_ceil	xの四捨五入を返す.
integer_floor	整数で <= xを満たす最大の整数を返す
integral	対象xの不定積分を返す
integrate	積分
interval	aとbの間にある整数(aとbは整数)
is_AlgebraElement	xの型がAlgebraElementであればTrueを返す
is_commutative	
is_ComplexNumber	
is_even	偶数、つまり、2で割り切れるかどうかを返します。
is_Functor	
is_Infinite	

is_Integer	
is_odd	奇数であるかどうかを返す。定義によるis_evenの補完
is_power_of_two	2の冪乗のときだけTtueを返す。
is_prime	素数であればTrueを返し、そうでなければFalseを返す。
is_prime_power	素数の冪乗であればTrue、それ以外はFalse。
is_pseudoprime	擬素数であればTrue、そうでなければFalse。
is_RealNumber	RealNumber型であればTrueを返し、 meaning that it is an element of the MPFR real field with some precision.
is_Set	SAGE Setであれば Trueを返す
is_square	平方数であるかどうかを返します。平方数であれば平方根を返し、そうでなければNoneを返します。
is_SymbolicExpression	
isqrt	整数の平方根を返します。つまり、平方根を越えない整数を返します。
laplace	Laplace変換を計算して返します。
latex	SAGEの対象を latex(...) でlatexの書式に変換します。
lcm	aとbの最小公倍数、The least common multiple of a and b, or if a is a list and b is omitted the least common multiple of all elements of v.
len	列や写像の成分の総数を返します。R
lim	与えられた方向からの変数vのaへの極限を返す
limit	与えられた方向からの変数vのaへの極限を返す
list	list() -> new list, list(sequence) -> new listでsequenceの成分で初期化されます。
list_plot	list_plotは単与件リストを取り、これはタプル(i,di)のリストとなります。ここでiは0からlen(data)-1迄で、diは与件のi番目の値です。そして、点はこれらのタプル上に配置されます。list_plotはまたタプル(dxi,dyi)のリストを取ります。ここでdxiはX座標を表現するi番目の与件、dyiはi番目の値で、plotjoined=Trueなら全ての点が繋がれて描かれます。
load	名前がfilenameのファイルからSAGEの対象を読み込みます。ファイル名には.sobjという修飾子がもしもなければ追加されます。註:loadと呼ばれる特殊なSGAE命令(Pythonでは使えません)もあり、load filename.sageと入力して用います。
loads	s=dumps(x)を使うことで文字列sに吐き出された対象xを復元します。
log	底を2とする自然対数。
matrix	行列を生成

max	単列の変数で、その最も大きな元を返します。二つ、あるいはそれ以上の引数に対しては、最大の引数を返します。
min	単列の変数で、その最も小さな元を返します。二つ、あるいはそれ以上の引数に対しては、最小の引数を返します。
minimal_polynomial	x で最小多項式を返す
mod	
mrange	与えられた大きさと型の複合リストを返します。
mul	リストの成分の積を返します。
next_prime	整数 n の次に大きな整数
next_prime_power	n が素数の時に、n よりも大きく、n に最も近い素数を返します
norm	ノルムを返す
normalvariate	正規分布
nth_prime	
number_of_arrangements	arrangements(mset,k)の大きさを返します。
number_of_combinations	combinations(mset,k)の大きさを返します。
number_of_derangements	derangements(mset)の大きさを返します。
number_of_divisors	因子の数を返します。
number_of_permutations	permutations(mset)の大きさを返します。
numerator	分子を返す
numerical_integral	区間 xmin から xmax で函数の数値積分を返します。R
numerical_sqrt	x の平方根を返します
oct	整数や多倍長整数の 8 進数表現を返します。
order	x の次数を返す。x が環、あるいは可換環の元であれば、これは x の aadditive order となる。
parametric_plot	parametric_plot はリストやタプルとしての二つの函数を取り、最初の函数が x 座標を与え、二番目の函数が y 座標を与えるグラフを描きます。
parent	定義されていれば x.parent() 、そうでなければ type(x)を返します。
permutations	mset として丁度同じ成分を含み、順序が異なるリストによる置換の表現。
pg	置換群。SAGE では置換は分離した円環表示を使って置換を表現する文字列か、分離した円環で表現されたタプルのリストの何れかで表現されています。
pi	円周率
plot	
pow	二つの引数を取り、x^y と同値になります。三引数では、(x^y) % z と同値

	になりますが、もっと効率的かもしれません(e.g. longに対し)
power_mod	The m-th power of a modulo the integer n.
prange	startとstop-1の間の素数のリスト。
previous_prime	nよりも小さな素数で最大の物を返します。
previous_prime_power	nよりも小さな素数の冪乗で最大の物を返します。
prime_divisors	The prime divisors of the integer n, sorted in increasing order.
prime_factors	The prime divisors of the integer n, sorted in increasing order.
prime_powers	List of all positive primes powers between start and stop-1, inclusive.
primes	startからstop-1の間の全ての素数の列を返します。.
primes_first_n	最初のn個の素数を返します。
prod	リストxの元の積を返します。
quo	Return the quotient object x/y, e.g., a quotient of numbers or of a polynomial ring x by the ideal generated by y, etc.
quotient	Return the quotient object x/y, e.g., a quotient of numbers or of a polynomial ring x by the ideal generated by y, etc.
random	区間[0,1]での乱数を返します。
random_prime	2とnの間の素数pをランダムに返します(つまり、 2 <= p <= n)。
randrange	範囲(start, stop[, step])からの乱数を選出します
range	Returns a list containing an arithmetic progression of integers.
rational_reconstruction	この函数はx/yを計算しようとします, ここでx/y は有理数です。
real	xの実部を返します。
reduce	Apply a function of two arguments cumulatively to the items of a sequence, from left to right, so as to reduce the sequence to a single value.
repr	Return the canonical string representation of the object.
reset	Delete all user defined variables, reset all globals variables back to their default state, and reset all interfaces to other computer algebra systems. If vars is specified, just restore the value of vars and leave all other variables alone (i.e., call restore).
restore	大域変数を再定義することで、それらの初期値に戻します。
round	Round a number to a given precision in decimal digits (default 0 digits). This always returns a real double field element.
sample	Chooses k unique random elements from a population sequence.
save	Save obj to the file with name filename, which will have an .sobj

	extension added if it doesn't have one. This will *replace* the contents of filename.
save_session	Save all variables that can be saved wto the given filename.
search	Return (True,i) where i is such that v[i] == x if there is such an i, or (False,j) otherwise, where j is the position that a should be inserted so that v remains sorted.
search_doc	Full text search of the SAGE HTML documentation for lines containing s.
search_src	Search sage source code for lines containing s.
sec	正割函数
sech	双曲正割函数
seed	
seq	A mutable list of elements with a common guaranteed universe, which can be set immutable.
set	順序のない単一元で構成される集まりを生成する。
show	グラフィックス対象を表示する
show_default	Set the default for showing plots using the following commands: plot, parametric_plot, polar_plot, and list_plot.
shuffle	
sigma	Return the sum of the k-th powers of the divisors of n.
simplify	式を簡易化する
sin	正弦函数
sinh	双曲正弦函数
sleep	
slice	Create a slice object. This is used for extended slicing (e.g. a[0:10:2]).
slide	Use latex(...) to typeset a SAGE object. Use %slide instead to typeset slides.
solve	単一の方程式や方程式系を与えられた変数に対して代数的に解く。
sorted	
sqrt	平方根函数。 これは記号平方根です。
square_free_part	Return the square free part of x, i.e., a divisor z such that x = z y^2, for a perfect square y^2.
srange	Return list of numbers ¥code{a, a+step, ..., a+k*step}, where a+k*step < b and a+(k+1)*step > b. The type of the entries in the

	list are the type of the starting value.
str	対象の文字列表現を返します。
subfactorial	Subfactorial or rencontres numbers, or derangements: number of permutations of n elements with no fixed points.
sum	Returns the sum of a sequence of numbers (NOT strings) plus the value of parameter 'start'
super	Typically used to call a cooperative superclass method.
symbolic_expression	
sys	This module provides access to some objects used or maintained by the interpreter and to functions that interact strongly with the interpreter.
tan	正接函数
tanh	双曲正接函数
taylor	Expands self in a truncated Taylor or Laurent series in the variable v around the point a, containing terms through (x - a)^n.
transpose	
trial_division	Return the smallest prime divisor <= bound of the positive integer n, or n if there is no such prime.
two_squares	Write the integer n as a sum of two integer squares if possible; otherwise raise a ValueError.
type	対象の型を返します
union	xとyの結合をリストとして返します
uniq	Return the sublist of all elements in the list x that is sorted and is such that the entries in the sublist are unique.
valuation	p>0を割り切る整数mによるpの冪を返します
var	記号変数を生成します。
vars	引数なしでlocals()と同値です。引数があればobject._dict_と同値です。
vector	与えられた成分を持つR上のベクトルを返します。
version	SAGEの版を返す。
view	対象のlatex表現を計算します。註:ノートブックモードではこの函数は単純に出力にpng画像を埋め込みます。
walltime	wall timeを返却します
xgcd	g = s*a+t*b = gcd(a,b)を満たす三個の整数(g,s,t)を返します。
xinterval	Iterator over the integers between a and b, inclusive.
xrange	range()に似ていますが、リストを返す代わりに要求された範囲の数を生成

	する対象を返します。
zip	Return a list of tuples, where each tuple contains the i-th element from each of the argument sequences.

表 3: SSAGE 函数の一部

3.22 SAGE 函数の情報を得る

表 3 には各函数がどの様な事をするか短い解説の一覧があります。しかしながら、これらの函数をどのように使うかを示す情報としては不十分です。任意の函数の追加の情報を得る一つの方法は疑問符'?'をワークシートのセルの中で、函数名のうしろに続けて入力してから <tab>キーを押すことです：

```
is_even?<tab>
|
File: /opt/sage-2.7.1-debian-32bit-i686-
Linux/local/lib/python2.5/site-packages/sage/misc/functional.py
Type:           <type 'function'>
Definition:     is_even(x)
Docstring:

   Return whether or not an integer x is even, e.g., divisible by 2.

   EXAMPLES:
       sage: is_even(-1)
       False
       sage: is_even(4)
       True
       sage: is_even(-2)
       True
```

表示される灰色のウィンドウには函数についての次の情報が含まれています：

File: 函数を実装するソースコードを含むファイル名を与えます。函数がどのように実装されているかを見る為、あるいは編集する為にファイルを探そうとしていれば便利です。

Type: 参照している情報サービスに引き渡された名前の対象の型を示します。

Definition: 函数の呼び出され方を示します。

Docstring: 函数のソースコードの中に置かれた文書文字列を表示します。

表 3 で挙げられた任意の函数のヘルプ、あるいは SAGE のマニュアルがこの方法を使って得られます。また、二つの疑問符'??'を函数名のうしろに置いて、<tab>キーを押せば、函数のソースコードが表示されます。

3.23 利用者が入力した函数についても情報が使える

情報サービスは利用者入力の函数についての情報を得ることにも使え、情報サービスがどのように動作するかをより良く理解すれば、一度、これを入力することでご利益があります。もしも、貴方が既に現時点でのワークシート上でそれを実行していたのであれば、再び addnums 函数を入力して、それを実行してください:

```
def addnums(num1, num2):
    """
    num1 と num2 の和を返す。
    """
    answer = num1 + num2
    return answer

#函数を呼出し、３に２を加える。
a = addnums(2, 3)
print a
|
  5
```

そして、ここで新しく再入力した函数についての情報が前節の手法を使って得られます：

```
addnums?<tab>
|
File: /home/sage/sage_notebook/worksheets/root/9/code/8.py
Type: <type 'function'>
Definition: addnums(num1, num2)
Docstring:

    Returns the sum of num1 and num2.
```

これは函数に関して表示された情報が函数のソースコードから得られた事を示しています。

3.24 SAGE に含まれる函数を用いた例

次の短いプログラムは 表 3 に挙げられていた幾つかの函数がどのように用いられるかを示すものです：

```
#1 から１０までの数の総和。
add([1,2,3,4,5,6,7,8,9,10])
|
   55

#1 ラジアンの余弦函数の値
cos(1.0)
|
   0.540302305868140

#15/64 の分母
denominator(15/64)
|
   64

#20 の全ての約数のリストを得る
divisors(20)
|
   [1, 2, 4, 5, 10, 20]

#４０と 132 の最大公約数を求める。
gcd(40,132)
|
   4

#2, 3, と 4 の積。
mul([2,3,4])
|
   24

#リストの長さ。
a = [1,2,3,4,5,6,7]
len(a)
|
   7

#０から１０までの整数リストを生成。
a = srange(11)
a
|
   [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

#0.0 と 10.5 の間の実数を 0.5 刻みでリストで出力。
a = srange(11,step=.5)
```

```
a
|
   [0.0000000, 0.5000000, 1.000000, 1.500000, 2.000000, 2.500000,
   3.000000, 3.500000, 4.000000, 4.500000, 5.000000, 5.500000,
   6.000000, 6.500000, 7.000000, 7.500000, 8.000000, 8.500000,
   9.000000, 9.500000, 10.00000, 10.50000]
```

```
#-5 から 5 までの整数を含むリストを生成。
a = srange(-5,6)
a
|
   [-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5]
```

```
#The zip() function takes multiple sequences and groups
#parallel members inside tuples in an output list.  One
#application this is useful for is creating points from
#table data so they can be plotted.
a = [1,2,3,4,5]
b = [6,7,8,9,10]
c = zip(a,b)
c
|
   [(1, 6), (2, 7), (3, 8), (4, 9), (5, 10)]
```

3.25 for 文で srange()と zip()を利用

for 文用に手動で列を構成する代わりに、srange()が自律的に列を生成することに使えます：

```
for t in srange(6):
   print t,
|
   0 1 2 3 4 5
```

for 文はまた zip()函数の利用から複数の列を並べて使うことが出来ます：

```
t1 = (0,1,2,3,4)
t2 = (5,6,7,8,9)
for (a,b) in zip(t1,t2):
```

```
    print a,b
|
   0 5
   1 6
   2 7
   3 8
   4 9
```

3.26 リスト内包(list comprehension)

ここまでで、我々は **if 文、for 文、list と函数**が、個別に、そして互いに用いられた時に、それぞれ非常に強力なことを見ています。より強力で、リスト内包(list comprehension)と呼ばれる特殊な文では、最小の構文でこれらを合わせて使うことができます。

ここでリスト内包の簡略化した構文は：

[式 for *変数* in 列 [if *条件*]]

リスト内包が行うことは、列を通じて、各列の成分をかわるがわる指定した*変数*で置いた反復処理を行うことです。その式はまた*変数*を含み、*変数*の中に数を置くのにしたがって*式*が評価され、その結果は新しいリストに置かれます。列の中の全ての数が処理されると、新しいリストが返されます。

次の例では、 t が変数で、 2*t が式です。 そして [1,2,3,4,5] が列です:

```
a = [2*t for t in [0,1,2,3,4,5]]
a
|
   [0, 2, 4, 6, 8, 10]
```

列を人手で生成する代わりに、列の自律的生成で srange()函数がしばしば利用さ:

```
a = [2*t for t in srange(6)]
a
|
   [0, 2, 4, 6, 8, 10]
```

おまけの if 文は、リスト内包で新しいリストで置き換えられる結果を篩にかける為に用いることが出来ます：

```
a = [b^2 for b in range(20) if b % 2 == 0]
a
|
   [0, 4, 16, 36, 64, 100, 144, 196, 256, 324]
```

この場合、２で割り切れるものに対する結果のみが出力リストに置かれます。

4 オブジェクト指向プログラミング

この章の目的は、オブジェクト指向の SAGE プログラムがどのように動作し、どの様に問題を解く為の用いられるかといった背後にある主要な概念を紹介する事です。貴方が僅か、あるいは全くオブジェクト指向プログラミング（OOP）の経験がないと仮定しており、貴方が OOP を十分理解することで問題を解く為に SAGE の対象を効果的に使えるようになる事を目的にしています。もしも、この OOP の詰込が完全に飲み込めなかったとしても心配する必要はありません。何故なら、貴方自身、何もない状態から対象をプログラムすることに必要とされる技能を持っていなくても、問題を解く為に SAGE の対象が使えるからです。そういいながらも、この章では、どの様に真っ新な状態から対象をプログラムするかを示すことで、貴方が SAGE の組込の対象がどの様に動作するかがより良く理解出来るでしょう。

4.1 オブジェクト指向な心の書換

私の意見では、この類のプログラミングで貴方が遂行する物事で最も難しい物の一つが手続き型プログラミングの世界からオブジェクト指向プログラミングの世界へと心の切り替えを行うことです。問題はオブジェクト指向プログラミングが手続き型プログラミングよりも必然的に難しいことではありません。問題はプログラミングの問題を解くその手法に違いが大きくて、貴方が本当に”それを会得する”前に幾つかの心的な書換が生じなければならないことです。この心理的な書換は貴方がオブジェクト指向プログラムを書いたり、一体全体何が OOP であるかを本当に理解する為に苦労してオブジェクト指向の書籍に耽溺するに従って、非常にゆっくりと生じる過程です。

まず初めに、貴方が何か特別で強力な何かが進行していると思うでしょうが、それに固く結びついた貴方の労力を逃してしまうでしょう。貴方がやっと”理解する”事は、普通、明るい光が差し込むように一挙に生じる事ではありません。それは薄明にとても似たもので、貴方の意識の背後で、その耀きをゆっくりと感じられるものです。貴方が関わる新しいプログラミングの問題の各々に対し、貴方の心の正面側では依然、それを解く為に手続的な計画を作成しようとするでしょう。しかし、貴方の心の背後で、この薄明がオブジェクト指向的な戦略(最初はぼんやりとしていますが、やがて明瞭さを増してゆきます)を表現し、それでまた問題が解けることに気付くでしょう。そして、これらのオブジェクト指向的な戦略がやがて興味深い物となって、やがて、それらにより多くの時間を費やしている自分に気付く事になるでしょう多くのプログラミングの問題が、それらを解く為に豊かなオブジェクト指向的な戦略と云う生産の引き金を引くことになったその時に、輝くオブジェクト指向的な部分が貴方の心に訪れるでしょう。

4.2 属性と振舞

オブジェクト指向プログラミングはソフトウエア設計の哲学であり、そこでソフトウエアは物理的な世界で対象が作用する方法に似た動作をさせます。全ての物理的な対象は**属性(attribute)**と**振舞(behavior)**があります。一つの例として、典型的な事務所の椅子には、**色**、**コロの数**、そして**物質的な型**を属性とし、**スピン**、**ロール**や**高さの設定**が振舞です。

ソフトウエア対象は物理的対象の様に働かせられ、それらはまた属性と振舞を持っています。ソフトウエア対象の**属性**は**インスタンス変数**と呼ばれる特殊な変数で保持され、その**振舞**は**メソッド**(これは**構成員函数(menber function)**とも呼ばれます)で保持されるプログラムで決定されます。**メソッド**は標準の函数と似ていますが、ただ、それは"自由に漂っている"代わりに対象に関連付けられています。SAGEでは、インスタンス変数とメソッドはしばしば**属性**として参照されます。

対象が生成されると、メソッドの呼出し、あるいは起動を意味するメッセージの送付に使われます。

椅子の場合は、char.spin(3)メッセージでその椅子を3回スピンさせる事を伝え、さらには、chair.setHeight(32)メッセージでその椅子を32センチメートルに設定する事を伝えることを想像してもよいでしょう。

4.3 類（対象を生成する為の設計図）

類(class)は対象を構築する為に用いられる**設計図**として考えられ、概念的に家の設計図に似たものです。建築家は、与えられた家がどのように構築されるべきか、どの様な材料が使われ、様々な寸法がどうあるべきか等が正確に定義された設計図を使います。設計図が終わると、ひとつ、あるいはそれ以上の家の建設でそれが使えますが、何故なら、設計図には家をどの様に建築するかと言ったことが記述された情報を含んでいるからですが、家それ自体にはありません。プログラマーが**類**を生成することは建築家が**家の設計図**を作成するのと非常に似ていますが、建築家が図面台やCADシステムを使って設計図を作るのに対し、プログラマーはテキストエディタやIDE(総合開発環境:Integrated Development Environment)を使って類を開発する点が異なります。

4.4 オブジェクト指向プログラムは必要に応じて対象の生成と破壊を行う

次の類比はオブジェクト指向プログラム内部でソフトウエア対象が必要に応じてどのように生成され、破壊されるかを解説するものです。対象の生成はまたインスタンスによる表現と

も呼ばれますが、それは対象を定義する類(設計図)が対象インスタンスを生成するために用いられているからです。対象を破壊し、記憶と他の資源を再要求してそれを用いる行為のことを**ゴミ収集(garbage collection)**と呼びます。

オブジェクト指向プログラムに似せた手法で与えられたジェット旅客機を操作し、それから、そのジェット機が大西洋を渡ってニューヨークからロンドンに飛ぶ準備をしているところを想像してください。離陸の前に、飛行機の全ての部品の設計図が滑走路に運び込まれ、飛行機の構築に必要とされる部品の全てを非常に素早く製作する為に一団の作業員が送り込まれます。各部品が出来上がると、飛行機の正しい位置に据え付けられ、短い時間の間に飛行機が完成して利用できるようになります。乗客はジェット機に乗り込み、そしていよいよ離陸です。

飛行機が地面を離れると滑走車輪を解体してしまいます(ゴミ収集されます)。なぜなら、それらは飛行中は不要で、大西洋を渡っている間に車輪を出して風切り音をたてていることは全く燃料の無駄だからです。だからといって心配する必要はありません。と言うのもロンドンに着陸する前に滑走車輪を適正な設計図(類)を使って再構成してしまえばよいからです。

離陸後の数分後に、パイロットは飛行機のジェットエンジンを作った会社が丁度、新モデルを出し、今使っているものよりも15%以上燃料効率がよいので飛行中でも航空会社が飛行機のエンジンの取替えを行うとの連絡を受け取ります。航空会社はニューヨークから新しいエンジンの設計図を飛行機に送り込み、これらは新しい三個のエンジンの組立て(インスタンス化)で用いられます。新しいエンジンが組み立てられると、古い三個のエンジンは同時に停止させられて、新しいエンジンで置き換えられて解体されてしまいます。エンジンの取替えは素早く行われ、旅客はこの取替えが行われたことに気付きさえしません。この飛行には世界的重要人物が乗っており、途上で遭遇した敵対機が我らがパイロットに航路を外れるように要求します。この要求に応じる代わりに、パイロットは設計図のライブラリから一群の50㎜機銃砲塔の設計図を取り出し、4個の砲塔を構築し、それから飛行機の頭、腹、鼻と尻尾にそれらを取り付けます。機銃からの多少の砲火は敵機を驚かせるに十分で、さっさと逃げてしまって、やがてレーダーからも消えてしまいました。残りの飛行は何も問題はありません。飛行機がロンドンに近づくに従って機銃の砲塔は解体され、新しい着陸車輪が設計図を使って構成されると飛行機は無事に着陸します。旅客がターミナルに入ると、飛行機全体は解体されてしまいます。

4.5 オブジェクト指向プログラム例

次の二節では、Hellosという名前の単純なオブジェクト指向プログラムの話をします。最初の節ではプログラムを含んでいますが、そのプログラム自体を見やすいように何らの注釈も入れていません。第二の節では、注釈をしっかり入れたプログラムがあり、そのプログラ

ムがどのようにして動作するのかが詳細に記述されています。

4.5.1 Hellos オブジェクト指向プログラム例（無注釈）

```
class Hellos:

  def __init__(self, mess):
     self.message = mess

  def print_message(self):
     print"The message is: ", self.message

  def say_goodbye(self):
     print "Goodbye!"

  def print_hellos(self, total):
     count = 1
     while count <= total:
        print"Hello ", count
        count = count + 1

     print " "


obj1 = Hellos("Are you having fun yet?")
obj2 = Hellos("Yes I am!")

obj1.print_message()
obj2.print_message()
print " "

obj1.print_hellos(3)
obj2.print_hellos(5)

obj1.say_goodbye()
obj2.say_goodbye()

   The message is:  Are you having fun yet?
   The message is:  Yes I am!

   Hello  1
   Hello  2
   Hello  3

   Hello  1
   Hello  2
```

```
   Hello  3
   Hello  4
   Hello  5

   Goodbye!
   Goodbye!
```

4.5.2 Hellos オブジェクト指向プログラム例（注釈付き）

では、Hellos プログラムの詳細を見ることにしましょう。このプログラムの版は注釈を付けたものです。プログラム左側の行数とコロンはプログラムそれ自身ではなく、それらはプログラムの異なった部分を参照し易くする為に追加したものです。

```
 1:class Hellos:
 2:    """
 3:    Hellos は'類'であり、類は対象生成の青写真になります。
 4:    類はインスタンス変数(属性,Attribute) とメソッド(振舞,Behavior)
 5:    で構成されます.
 6:    """
 7:
 8:    def __init__(self, mess):
 9:        """
10:        __init__ はコンストラクタが呼び出した組込の特殊な
11:        メソッドです。コンストラクタ　メソッドは、対象が生成
12:        された時に一度だけ呼び出され、その作業は対象の構築を
13:        完遂する事です。その対象が生成されたのちには、
14:        そのコンストラクタはもはや用いられません。
15:        コンストラクタの目的は'伝言'と呼ばれるインスタンス
16:        変数を生成することで、それから、文字列を使って、
17:        それを初期化します。
18:        """
19:
20:        """
21:        このコードはインスタンス変数を生成します。　この'設計図'
22:        類から生成された対象インスタンスは任意のインスタンス変数
23:        のそれ自身のユニークな複製を持っています。インスタンス
24:        変数は対象の属性(あるいは状態)を保持します。
25:        ここでの self 変数は現行の対象への参照を保ちます。
26:
```

```
        """
        self.message = mess;


    def print_message(self):
        """
        print_message はインスタンスメソッドであり、この類を使って、
        生成された対象に'伝言を印字する'振舞を与えます。
        """
        print"The message is: ", self.message


    def say_goodbye(self):
        """
        say_goodbye はインスタンスメソッドであり、この類を使って、
        生成された対象に'goodby と言う'振舞を与えます。
        """
        print "Goodbye!"


    def print_hellos(self, total):
        """
        print_hellos はインスタンスメソッドで、Hello の数を
       印字の引数として取り、画面にこの Hello を沢山印字し
      ます。
        """
        count = 1
        while count <= total:
            print"Hello ", count
            count = count + 1

        print " "


"""
次のプログラムは二つの分離した Hellos 対象(インスタンス)
を生成し、これらは変数 obj1 と obj2 としてそれぞれが表現
されます。インスタント化された時に、ユニークな文字列
助変数が各対象に、引き渡され、この文字列は対象の状態を
初期化する為に用いられます。
```

```
70:
71:  対象が生成されると、それらに振舞を実行させる為に、
72:  メソッドを呼び出すことで対象に対して伝言が送られます。
73:  これは'対象の取り上げ'による参照(obj1 とします)で実行
74:  されますが、その参照ではうしろに点を置いて、呼び出したい。
75:  対象の名前を記入します。
76:"""
77:
78:obj1 = Hellos("Are you having fun yet?")
79:obj2 = Hellos("Yes I am!")
80:
81:obj1.print_message()
82:obj2.print_message()
83:print " "
84:
85:obj1.print_hellos(3)
86:obj2.print_hellos(5)
87:
88:obj1.say_goodbye()
89:obj2.say_goodbye()
```

1 行での Hellos 類は **class** 文を使って定義され、類名は習慣から大文字で始めます。類名が複数の語で構成されていれば、各語の最初の文字を大文字にして、残りの全ての文字を小文字で記述します(例えば、HelloWorld)。その類は行 1 で始まり行 61 で終わります。この行 61 はインデントをつけたコードの最終行です。類の一部としての全ての**メソッド**と**インスタンス変数**は類のインデント付けられた区画の中にある必要があります。

Hellos 類は一つの**コンストラクタ**メソッドを 8 行目に、28 行目で生成された一つの**インスタンス変数**と 32 行目、41 行目と 50 行目にそれぞれ 3 個の**インスタンス**メソッドを含んでいます。**インスタンス変数**の目的は対象に、与えられた類から生成された他の対象と異なるユニークな**属性**を与えることです。**インスタンスメソッド**の目的は各対象にその**振舞**を与えることです。対象の全てのメソッドは、対象のインスタンス変数に接続し、これらのインスタンス変数はこれらのメソッドの中のプログラムによって接続が可能です。インスタンス変数名は函数名と同様の習慣に従います。

8 行目のメソッドは**コンストラクタ**と呼ばれる特殊なメソッドです。 **コンストラクタ**メソッドは対象が生成される時に一度だけ呼び出され、その目的は対象の構築を完遂することです。 対象が生成されたあとには、そのコンストラクタは最早使われません。8 行目のコンストラクタの目的は、Hellos 対象の **message** インスタンス変数を Hellow 型の新しい対象が生成されたとき(78 行と 79 行を参照)にそれに引き渡される文字列で初期化する事です。

全てのインスタンスメソッドは呼び出されたメソッドからの特定の対象に対する参照を含む一つの引数を持ちます。この引数は常に左端の位置に置かれ、通常、この位置に置かれる変数を**self**と呼びます。この**self**変数は特定の対象のインスタンス変数を生成したり、利用する為に使われます。

28行目の**self.message=mess**でコンストラクタ**mess**変数に引き渡された対象を取り、**message**と呼ばれるインスタンス変数に引き渡します。インスタンス変数は通常の変数であるかの様に、割り当てによって生成されます。この点演算子'.'は対象のインスタンス変数に対象を参照する変数とインスタンス変数名の間に(self.messageやobj1.messageの様に)置かれることで対象のインスタンス変数を参照する為に用いられます。

32行目，41行目と50行目のメソッドはHellos類を用いて生成された対象に、それらの振舞を与えます。print_message()はその対象の**message**インスタンス変数にある文字列を印字するという振舞を実行し、sat_goodbye()メソッドは、文字列"Goodbye!"を印字するという振舞を実行します。print_hellos()メソッドは引数として一つの整数を取り、その回数程、'Hello'という言葉を印字します。メソッドの名付け方は函数名で用いたものと同様です。

Hellos類の下のコードは二つの分離した対象(インスタンス)を生成し、これらは変数**obj1**と変数**obj2**にそれぞれ割り当てられます。対象はその類名を括弧に続けて入力することで生成されます。括弧の中に置かれた引数はコンストラクタメソッドに引き渡されます。Hellos類が呼出されたとき、文字列がそれのコンストラクタメソッドに引き渡され、この文字列は対象の**状態**の初期化で用いられます。対象の状態はそのインスタンス変数の内容で決定されます。もし、対象のインスタンス変数が変更されると、対象状態も変更されます。Hellos対象は**message**と呼ばれるインスタンス変数のみを持っているので、それらの状態はこの変数によって決定されます。

対象が生成されると、それらの振舞はメソッドの呼出によって要求されます。
これは参照する変数(obj1と呼びましょう)による"対象の摘み上げ"によって実行され、
点のうしろにこの変数を置き、呼び出したい対象のメソッドの内の一つの名前を入力し、
その引数のうしろに括弧を続けます。

4.6 SAGE類(Class)と対象(Object)

SAGEの函数は沢山の機能を持っているので、SAGEの機能の殆どは**類**と類から実体化された**対象**に含まれいます。SAGEの類と対象は説明するには莫大な情報を表現します。とはいえ、貴方が可能な限り早くSAGEの対象を使って作業を始めることが出来るように、より簡単な素

材を最初に示すことにしましょう。

4.7 SAGE の対象について情報を得ること

セルに次のコードを入力し、それを実行てみましょう:

```
x = 5
print type(x)
|
   <type 'sage.rings.integer.Integer'>
```

我々は既に type()函数を整数の型を定める為に用いていますが、ここで我々は型がどのようなものであるか詳細を説明する事が出来ます。Sage.rings.integer.Integer と疑問符'?'に続けて入力して<tab>キーを押してみましょう：

```
sage.rings.integer.Integer?<tab>
|
File:/opt/sage-2.7.1-debian-32bit-i686-
Linux/local/lib/python2.5/site-packages/sage/rings/integer.so
Type:   <type 'sage.rings.integer.Integer'>
Definition: sage.rings.integer.Integer([noargspec])

Docstring:

      The class{Integer} class represents arbitrary precision
      integers.  It derives from the class{Element} class, so
      integers can be used as ring elements anywhere in SAGE.

      begin{notice}
      The class class{Integer} is implemented in Pyrex,
      as a wrapper of the GMP mpz_t integer type.
      end{notice}
```

この情報は、sage.rings.integer.Integer が本当に類であり、整数類を生成できる事を示しています。また、貴方が二つの疑問符'??'を類名のうしろに置いて、<tab>キーを押せば、類のソースコードが表示されます。

では、別のセルで x.と入力し、 それから<tab>キーを押しましょう:

```
x.<tab>
|
```

```
x.additive_order          x.gcd                    x.numerator
x.base_base_extend        x.inverse_mod            x.ord
x.inverse_of_unit         x.order                  x.parent
x.base_extend             x.is_nilpotent           x.plot
x.base_extend_canonical   x.is_one                 x.powermodm_ui
x.is_perfect_power        x.powermod               x.quo_rem
x.base_extend_recursive   x.is_power               x.rename
x.base_ring               x.is_power_of            x.reset_name
x.binary                  x.is_prime               x.save
x.category                x.is_prime_power         x.set_si
x.ceil                    x.is_pseudoprime         x.set_str
x.coprime_integers        x.is_square              x.sqrt
x.crt                     x.is_squarefree          x.sqrt_approx
x.db                      x.is_unit                x.square_free_part
x.degree                  x.is_zero                x.str
x.denominator             x.isqrt                  x.substitute
x.digits                  x.jacobi                 x.test_bit
x.div                     x.kronecker              x.val_unit
x.lcm                     x.subs                   x.valuation
x.divides                 x.leading_coefficient    x.version
x.dump                    x.list                   x.xgcd
x.dumps                   x.mod                    x.parent
x.exact_log               x.multiplicative_order   x.plot
x.factor                  x.next_prime             x.rename
x.factorial               x.next_probable_prime    x.reset_name
x.floor                   x.nth_root               x.powermodm_ui
```

表示させる灰色のウィンドウには対象が包含する全てのメソッドが含まれています。これらの任意のメソッドがマウスで選択されると、その名前がドット演算子のうしろのセルに置かれます。ここでは、`is_prime` メソッドを選択しましょう。その名前がセルに置かれた時点で、疑問符'?'をそのうしろに入力して、<tab>キーを押すと、このメソッドの情報が得られます:

```
x.is_prime?
```

```
File:       /opt/sage-2.7.1-debian-32bit-i686-
Linux/local/lib/python/site-packages/sage/rings/integer/pyx
Type:       <type 'builtin_function_or_method '>
Definition: x.is_prime()

Docstring:

          Retuns True if self is prime

          EXAMPLES:
```

```
            sage: z = 2^31 - 1
            sage: z.is_prime()
            True
            sage: z = 2^31
            sage: z.is_prime()
            False
```

定義の節から is_prime()メソッドが引数を引き渡されるに呼び出さ事が分かり、Docstring の節から、このメソッドが対象が素数であれば True を返すものであることが分かります。次のコードは変数 x(これは 5 を包含しています)が is_prime()メソッドを呼び出す為に用いられています:

```
x.is_prime()
|
   True
```

4.8 対象のメソッドのリスト

リストは対象であり、それ故、リストには便利な機能を持ったメソッドを含んでいます:

```
a = []
a.<tab>
|
a.append   a.extend   a.insert   a.remove   a.sort
a.count    a.index    a.pop      a.reverse
```

次のプログラムは対象としてのリストの幾つかのメソッドのデモです:

```
# リストの末端に対象を追加.
a = [1,2,3,4,5,6]
print a
a.append(7)
print a
|
[1, 2, 3, 4, 5, 6]
[1, 2, 3, 4, 5, 6, 7]

# リストに対象を挿入.
a = [1,2,4,5]
print a
a.insert(2,3)
print a
|
```

```
[1, 2, 4, 5]
[1, 2, 3, 4, 5]
```

```
# リストの成分を並び替え.
a = [8,2,7,1,6,4]
print a
a.sort()
print a
```

```
[8, 2, 7, 1, 6, 4]
[1, 2, 4, 6, 7, 8]
```

4.9 継承による類の拡張

オブジェクト技術は微妙ですが強力です。これは複雑なものを扱う為の幾つかの機構をもっており、**類継承(class inheritance)**はそのうちの一つの物です。**類継承**は、最小の処理で、他の類(親の類、スパークラス、あるいはベースクラスと呼ばれます)のインスタンス変数やメソッドの全てを得たり、あるいは継承する為の類の能力です。親の類から継承された類のことを**子供の類**、あるいは**サブクラス**と呼びます。この意味は、子供の類はその親で実行出来るものが利用可能だからです。

次のプログラムでは、Person 類が組込みの object 類から継承し、ArmyPrivate 類が Person 類から継承するという類継承を示します：

```
class Person(object):
     def __init__(self):
          self.rank = "I am just a Person, I have no rank."

     def __str__(self):
          return "str: " + self.rank

     def __repr__(self):
          return "repr: " + self.rank


class ArmyPrivate(Person):
     def __init__(self):
          self.rank = "ArmyPrivate."


a = object()
print type(a)
```

```
b = Person()
print type(b)

c = ArmyPrivate()
print type(c)
|
   <type 'object'>
   <class '__main__.Person'>
   <class '__main__.ArmyPrivate'>
```

類が生成されたあとに、このプログラムは object 型の対象を変数'a'に割当て、Person 型の対象を変数'b'に割当て、ArmyPrivate 型の対象を変数'c'に割当ててインスタンス化します。

次の処理は任意の対象の継承階層を表示させることに使えます。上のプログラムを実行したあとで別のセルでそれを実行すれば、ArmyPrivate 類の継承階層が表示されます.
(この処理がどの様にして動作するか考え込まないように。使ってみましょう。):

```
#対象の継承階層を表示します。註： このプログラムが度のように動作するのか
#思い悩む必要はありません。
def class_hierarchy(cls, indent):
   print '.'*indent, cls
   for supercls in cls.__bases__:
      class_hierarchy(supercls, indent+1)

def instance_hierarchy(inst):
   print 'Inheritance hierarchy of', inst
   class_hierarchy(inst.__class__, 3)

z = ArmyPrivate()

instance_hierarchy(z)
|
   Inheritance hierarchy of str: ArmyPrivate
   ... <class '__main__.ArmyPrivate'>
   .... <class '__main__.Person'>
   ..... <type 'object'>
```

instance_hierarchy 函数がそれに引き渡された任意の対象の継承階層を表示します。この場合、ArmyPrivate 類がインスタンス化されて instance_hierarchy 函数に引き渡され、それから対象の継承階層が表示されるのです。その階層で最上の類、つまり、object 類ですが、これが最後に表示され、Person は object から継承し、ArmyPrivate は Person から継承します。

4.10 対象類、dir() 函数、組込メソッド

対象類は SAGE に組込まれており、いくつかの便利なメソッドを持っています。これらのメソッドは非常に便利なので、多くの SAGE の類がその **object** 類から 1)直接、あるいは 2)間接的に継承しますが、それは object 類から継承している類からの継承によります。
さて、**object** 類に含まれるメソッドを調べることで、継承プログラムについて議論を始めることにしましょう。**dir()**函数は全ての属性(インスタンス変数とメソッドの双方を意味します)を一覧表示するので、**object** 型の対象がどのメソッドを含むかを見るためにそれが使えます：

```
dir(a)
|
   ['__class__', '__delattr__', '__doc__',
   '__getattribute__','__hash__','__init__', '__new__', '__reduce__',
   '__reduce_ex__', '__repr__','__setattr__', '__str__']
```

二重の'__'で開始し、それで終わる名前が SAGE の一部であり、下線によって、これらの名前がプログラマーが定義した名前と衝突する事がない様にしています。Person 類は object 類からのこれらの属性の全てを継承しますが、それらの内の幾つかだけしか使いません。メソッドが親の類から継承されたとき、子供の類はメソッドの親の実装を使ったり、親のものとは異なった振舞をするように再定義することが出来ます。

先に議論した様に、__init__メソッドはコンストラクタであり、それは類を使って生成された新しい類の完全な構成を助けるものです。Person 類は__init__メソッドを再定義して、**rank** と呼ばれるインスタンス変数を生成して文字列"I am just a Person, I have no rank"をそれに割り当てます。 **__ repr__**と **__str__** メソッドもまた Person 類で再定義されています。__repr__メソッドは、対象の一部である対象の文字列表現を返します：

```
|
   repr: I am just a Person, I have no rank.

```

__str__函数はまた対象の一部でもある対象の文字列表現を返しますが、print の様な文に引き渡された時だけです：

```
print b
|
   str: I am just a Person, I have no rank.
```

__str__メソッドは通常、__repr__メソッドよりもより利用者にやさしい文字列を返す為に用

いられますが、この例では、とても似た文字列が返されます。

4.11 Sage.rings.integer.Integer 類の継承階層

次のプログラムは sage.rigns.integer.Integer 類の継承階層を表示します：

```
# 類の継承階層を表示します。　註：このプログラムがどの様にして動作するのかを
# 理解しなくても構いません。使ってみましょう。
def class_hierarchy(cls, indent):
    print '.'*indent, cls
    for supercls in cls.__bases__:
        class_hierarchy(supercls, indent+1)

def instance_hierarchy(inst):
    print 'Inheritance hierarchy of', inst
    class_hierarchy(inst.__class__, 3)

instance_hierarchy(1)
|
   Inheritance hierarchy of 1
    ... <type 'sage.rings.integer.Integer'>
    .... <type 'sage.structure.element.EuclideanDomainElement'>
    ..... <type 'sage.structure.element.PrincipalIdealDomainElement'>
    ...... <type 'sage.structure.element.DedekindDomainElement'>
    ....... <type 'sage.structure.element.IntegralDomainElement'>
    ........ <type 'sage.structure.element.CommutativeRingElement'>
    ......... <type 'sage.structure.element.RingElement'>
    .......... <type 'sage.structure.element.ModuleElement'>
    ........... <type 'sage.structure.element.Element'>
    ............ <type 'sage.structure.sage_object.SAGEObject'>
    ............. <type 'object'>
```

次の解説では、私は節約の為に類名の”sage.xxx.xxx...”の頭の部分を外す事にします。instance_hierarchy 函数からの出力は、数字 1 が類型 Integer であることを示しています。その出力は Integer は EuclideanDomainElement から継承し、EuclideanDomainElement は PrincipalIdealDomainElement から継承し..等であることを示しています。階層の頂点(この一覧の底になります)には SAGEObject が object から継承しています。

ここで二つの他の広く使われる SAGE 類の継承階層は：

```
instancehierarchy(1/2)
|
   Inheritance hierarchy of 1/2
```

```
   ... <type 'sage.rings.rational.Rational'>
   .... <type 'sage.structure.element.FieldElement'>
   ..... <type 'sage.structure.element.CommutativeRingElement'>
   ...... <type 'sage.structure.element.RingElement'>
   ....... <type 'sage.structure.element.ModuleElement'>
   ........ <type 'sage.structure.element.Element'>
   ......... <type 'sage.structure.sage_object.SAGEObject'>
   .......... <type 'object'>
```

```
instancehierarchy(1.2)
|
   Inheritance hierarchy of 1.20000000000000
   ... <type 'sage.rings.real_mpfr.RealNumber'>
   .... <type 'sage.structure.element.RingElement'>
   ..... <type 'sage.structure.element.ModuleElement'>
   ...... <type 'sage.structure.element.Element'>
   ....... <type 'sage.structure.sage_object.SAGEObject'>
   ........ <type 'object'>
```

4.12 ”は一つの”関係

継承の概念の別の側面とは、親が出来ることなら何でも子供の類でも出来るので、親の類が使えるどの様な場所でも子供の類が使えます。整数類(Integer class)の継承階層に注目してみましょう。この階層は、整数は EuclideanDomainElement であり、EuclideanDomainElement は PrincipalIdealDomainElement であり、PrincipalIdealDomain は DedekindDomainElement である等となっています。最後に辿り着く SAGEObject は一つの対象です(丁度、SAGE での他の殆ど全ての類の様に object 類が根本の類で、全てはそこからの子孫になるからです。)。一般的には、その先祖が使える任意の場所で子孫の類が使えるということです。

4.13 こんがらがったかな?

この章は多分貴方を困惑させるものでしょうが、そのことに悩むことはありません。この本の残りは SAGE で対象がどのように使われているかを示す例を含んでおり、たくさんの用いられている対象を見て、それらのお陰でより快適になることでしょう。

5 いろいろなこと

5.1 前回の処理結果の参照

一つのワークシートに複数のセルを広げて問題を解くとき、以前の処理結果を参照する事が望ましい事がしばしばあります。下線記号'_'は次の例で示すようにこの目的に使えます：

```
2 + 3
|
   5
_
|
   5

_ + 6
|
   11

a = _ * 2
a
|
   22
```

5.2 例外処理

SAGE のプログラムが動作している間に生じるかもしれない例外条件の処理を行う為に単一の方法を持っている事を保証する為、例外表示と処理の仕組は SAGE に組み込まれています。この節では単に表示された例外についてのみ解説します。何故なら、例外処理はこの文書の領域を越える進んだ話題だからです。

次のプログラムは例外が生じる原因となり、それから、その例外の情報が表示されます：

```
1/0
|
   Exception (click to the left for traceback):
   ...
   ZeroDivisionError: Rational division by zero
```

何故なら、1/0 は未定義の数学的演算なので、SAGE はその計算の実行が出来ません。そのプログラムの実行を停止し、この問題に関してプログラムの別の領域、或いは、利用者に報せ

る例外を生成します。その例外を扱うプログラムの別部分がなければ、その例外のテキストによる説明が表示されます。この場合、その例外が利用者に ZeroDivisionError が生じたことと、これが”rational division by zero(零による有理数の割算)”を実行しようとした為に生じたことを利用者に報せています。

とにかく、これは利用者にとってソースコードにある問題を突き止めて修正するのに十分な情報です。時には、利用者はその問題を突き止める為により多くの情報を必要とすることもあるので、例外はマウスを表示された例外のテキストの左側をクリックすれば、追加の情報が表示される様になっています。

```
   Traceback (most recent call last):
     File "", line 1, in
     File "/home/sage/sage_notebook/worksheets/tkosan/2/code/2.py",
        line 4, in
       Integer(1)/Integer(0)
     File "/opt/sage-2.8.3-linux-32bit-debian-4.0-i686-
        Linux/data/extcode/sage/", line 1, in

     File "element.pyx", line 1471, in element.RingElement.__div__
     File "element.pyx", line 1485, in element.RingElement._div_c
     File "integer.pyx", line 735, in integer.Integer._div_c_impl
     File "integer_ring.pyx", line 185, in
   integer_ring.IntegerRing_class._div
   ZeroDivisionError: Rational division by zero
```

この追加情報で、例外が生じた時に利用されている SAGE ライブラリの全てのプログラムの追跡結果とそのプログラムが含まれているファイル名が見られます。そのお陰で、例外が SAGE の虫か入力されたプログラムの虫を原因とするものかどうかを決定する為に、SAGE の猛者はソースコードを見ることが出来ます。

5.3 数値結果を得る

対象の数値近似が必要になることがありますが、SAGE はこれを遂行するいくつかの方法があります。一つの方法は n()函数で、別のもう一つの方法は n()メソッドを使うことです。次の例では双方を利用した物を示します：

```
a = 3/4
print a
print n(a)
print a.n()
|
   3/4
```

```
    0.750000000000000
    0.750000000000000
```

返却される対象の桁数は digits 助変数を使って調整で来ます：

```
a = 3/4
print a.n(digits=30)
|
    0.750000000000000000000000000000
```

そして、精度の桁数は prec 助変数を使って調整で来ます：

```
a = 4/3
print a.n(prec=2)
print a.n(prec=3)
print a.n(prec=4)
print a.n(prec=10)
print a.n(prec=20)
|
    1.5
    1.2
    1.4
    1.3
    1.3333
```

5.4 式の表記指南

常に、次の二項演算子には両方に空行一つを置きます:

割当 '=', 引数付き割当 (+=, -=, 等), 比較 (==, <, >, !=, <>, <=, >=, in, not in, is, is not), 論理値 (and, or, not).

算術演算子+ と - のまわりには空行を置きますが、算術演算子 *, /, %, とˆには空行を置いてはいけません：

```
    x = x + 1
    x = x*3 - 5%2
    c = (a + b)/(a - b)
```

添字引数や引数の附置として指す場合には等号'='の回りに空行を置かないように：

```
a.n(digits=5)
```

5.5 組込定数

SAGE は組込の幾つかの数学定数を有しており、次は最も一般に使われるもののリストです:

Pi, pi: 円周率

E, e: 自然底

I, i: 純虚数

log2: 実数２を底とする対数

Infinity, infinity: 正や負の無限大を指す為に + や - を前に置けます。

次の例では定数の利用が示されています:

```
a = pi.n()
b = e.n()
c = i.n()
a,b,c
|
    (3.14159265358979, 2.71828182845905, 1.00000000000000*I)
```

```
r = 4
a = 2*pi*r
a,a.n()
|
(8*pi, 25.1327412287183)
```

SAGE での定数は大域変数として定義され、この**大域変数**は大半の SAGE のプログラムから、函数の内部やメソッドも含めて参照出来ます。定数は単純に、それらに一定の対象が割り当てられた変数なので、必要であっても定数対象が失われているのであれば、その変数に最割り当てが可能です。もし、変数にそれが通常持っていんる定数を再度割り当てなければならない場合は **restore()**函数が用いられます。 次のプログラムではどの様に変数 pi がそれに

割り当てられた対象 7 を持つことができ、それから、restore()函数にその名前を単引用符で括って引き渡すことで再びそのデフォルトの定数値を持つようになるかを示しています：

```
print pi.n()

pi = 7
print pi

restore('pi')
print pi.n()
|
   3.14159265358979
   7
   3.14159265358979
```

restore()函数が引数なしで呼び出されると、一度別の値を割り当てられた全ての定数は元の値に戻されます。

5.6 根

sqtrt()函数は値の平方根を得る為に使えるだけではなく、より一般的な手法が値の他の根を得る為に用いられています。例えば、8 の三乗根を得たければ:

$$\sqrt[3]{8}$$

```
8 would be raised to the 1/3 power:
8^(1/3)
|
   2
```

演算子の順序により、有理数 1/3 は指数として評価されるために括弧で括る必要があります。

5.7 記号変数

ここまでで、我々が利用している全ての変数は割当のときに生成されています。たとえば、次の処理では変数 w が生成されて、数 8 がそれに割り当てられます：

```
w = 7
w
```

```
|
   7
```

しかし、指定した値を何も割り当てていない変数を処理する必要があればどうなるでしょうか？次の処理では変数 z の値を表示させようとします。しかし、z にはまだ値が割り当てられていないので、例外処理が返されてしまいます：

```
print z
|
   Exception (click to the left for traceback):
   ...
   NameError: name 'z' is not defined
```

数学で、”割り当てられていない変数”はいつでも使われます。SAGE は数学指向のソフトウエアなので、割り当てられていない変数を使って処理する能力があります。SAGE では、割り当てられていない変数は**記号変数**と呼ばれ、それらは **var()**函数を用いて定義されます。ワークシートが最初に開かれた時に、変数 **x** は自律的に記号変数として定義され、貴方のコードの中で別の値を割り当てるまではそのままです。

次のコードが新しく開いたワークシート上で実行されました：

```
print x
type(x)
|
   x
   <class 'sage.calculus.calculus.SymbolicVariable'>
```

変数 **x** には **SymbolicVariable** 型の対象が自律的に SAGE 環境によって割り当てられる事に注意してください。

もし、記号変数として **y** と **z** も使いたければ、**var()**函数をこの為に使わなければなりません。**var('x,y')**、または **var('x y')**の何れかが入力出来ます。var()函数は一つ、またはそれ以上の変数名を文字列の中に受け入れられる様に設計されており、その名前はコンマか空行の何れかで分離することが出来ます。

次のプログラムでは **var()**が **y** と **z** を記号変数として初期化することに用いている事を示します：

```
var('y,z')
```

```
y,z
|
   (y, z)
```

ひとつ、あるいはそれ以上の記号変数が定義された後で、reset()函数はそれらを未定義にすることが出来ます：

```
reset('y,z')
y,z
|
   Exception (click to the left for traceback):
   ...
   NameError: name 'y' is not defined
```

5.8 記号変数

記号変数を含む式の事を**記号式(symbolic expressions)**と呼びます。次の例で、b は記号変数として定義されており、それから、記号式 2*b の生成で用いられています。

```
var('b')
type(2*b)
|
   <class 'sage.calculus.calculus.SymbolicArithmetic'>
```

この例で見られる様に。記号式 2*b は SymbolicArithmetics 型の対象で置き換えられます。この式はまた変数に割り当てる事が可能です：

```
m = 2*b
type(m)
|
   <class 'sage.calculus.calculus.SymbolicArithmetic'>
```

次のプログラムは二つの記号式を生成し、それらを変数に割り当てて、それから、それらに関する演算を行います：

```
m = 2*b
n = 3*b
m+n, m-n, m*n, m/n
|
   (5*b, -b, 6*b^2, 2/3)
```

二つの記号式同士の積のもう一つの別例です：

```
m = 5 + b
n = 8 + b
y = m*n
y
```

```
    (b + 5)*(b + 8)
```

5.9 展開と因子分解

前節の式の展開式が必要であれば、**expand()**メソッドを呼び出すことで簡単に得られます（この例では前節のセルが動作していると仮定しています）：

```
z = y.expand()
z
```

```
    b^2 + 13*b + 40
```

式の**展開**式は変数 z に割り当てられ、**因子分解**式は z から **factor()**メソッドから得られます：

```
z.factor()
```

```
    (b + 5)*(b + 8)
```

兎に角、数は変数に割当てをしなくても、回りを括弧で括って factor()メソッドを呼び出せば因子分解が行えます：

```
(90).factor()
```

```
    2 * 3^2 * 5
```

5.10 いろいろな記号式の例

```
var('a,b,c')
```

```
(5*a + b + 4*c) + (2*a + 3*b + c)
|
   5*c + 4*b + 7*a

(a + b) - (x + 2*b)
|
   -x - b + a

3*a^2 - a*(a -5)
|
   3*a^2 - (a - 5)*a

_.factor()
|
   a*(2*a + 5)
```

5.11 記号式への値の引き渡し

もし、値が記号式に引き渡されると、それは評価され、その結果が返却されます。もし、式が一つの変数を持っていれば、その値を次の様に単純に引き渡すことが出来ます：

```
a = x^2
a(5)
|
   25
```

しかしながら、式が二つ以上の変数を持っていれば、各変数には名前を使って値を割り当てる必要があります：

```
var('y')
a = x^2 + y
a(x=2, y=3)
|
   7
```

5.12 記号方程式と solve()函数

記号式の処理に加え、SAGE はまた記号方程式が扱えます：

```
var('a')
type(x^2 == 16*a^2)
|
   <class 'sage.calculus.equations.SymbolicEquation'>
```

この例で示すことが出来るように、記号方程式 x^2 == 16*a^2 は二重等号'=='を使う必要があるので、単等号'='を次のように使って変数に割り当てることが出来ます：

```
m = x^2 == 16*a^2
m, type(m)
|
  (x^2 == 16*a^2, <class 'sage.calculus.equations.SymbolicEquation'>)
```

たくさんの方程式が solve()函数を用いて代数的に解けます：

```
solve(m, a)
|
   [a == -x/4, a == x/4]
```

solve()函数の第 1 引数は記号方定式で、第 2 の引数は解くべき記号変数です。

solve() 函数もまた同時に方程式を解けます：

```
var('i1,i2,i3,v0')

a = (i1 - i3)*2 + (i1 - i2)*5 + 10 - 25 == 0
b = (i2 - i3)*3 + i2*1 - 10 + (i2 - i1)*5 == 0
c = i3*14 + (i3 - i2)*3 + (i3 - i1)*2 - (-3*v0) == 0
d = v0 == (i2 - i3)*3

solve([a,b,c,d], i1,i2,i3,v0)
|
   [[i1 == 4, i2 == 3, i3 == -1, v0 == 12]]
```

一つ以上の方程式を solve()に与える場合、リストの形式で与えなければならないことに注意してください・

5.13 記号数学函数

SAGE には数学の構文を使った函数を定義する能力があります。次の例では x を変数の函数として函数 f を定義する様子を示しておきます：

```
f(x) = x^2
```

```
f, type(f)
|
   (x |--> x^2,
   <class'sage.calculus.calculus.CallableSymbolicExpression'>)
```

この方法で生成された対象は CallableSymbolicExpression 型で、これらは次の例題で示される様に呼出可能な事を意味します：

```
f(4), f(50), f(.2)
|
   (16, 2500, 0.040000000000000010)
```

ここでは複製の代わりに上の CallableSymbolicExpression を使います：

```
a = 0
while a <= 9:
    f(a)
    a = a + 1
|
   0
   1
   4
   9
   16
   25
   36
   49
   64
   81
```

次の例では前の例題で行った同じ作業を完遂しますが、より進んだ言語機能を用いる点で異なります：

```
a = srange(10)
a
|
   [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
for num in a:
    f(num)
|
   0
   1
   4
   9
```

```
    16
    25
    36
    49
    64
    81
```

5.14 グラフを使った根の検出と find_root()メソッドを利用した数値的な根の検出

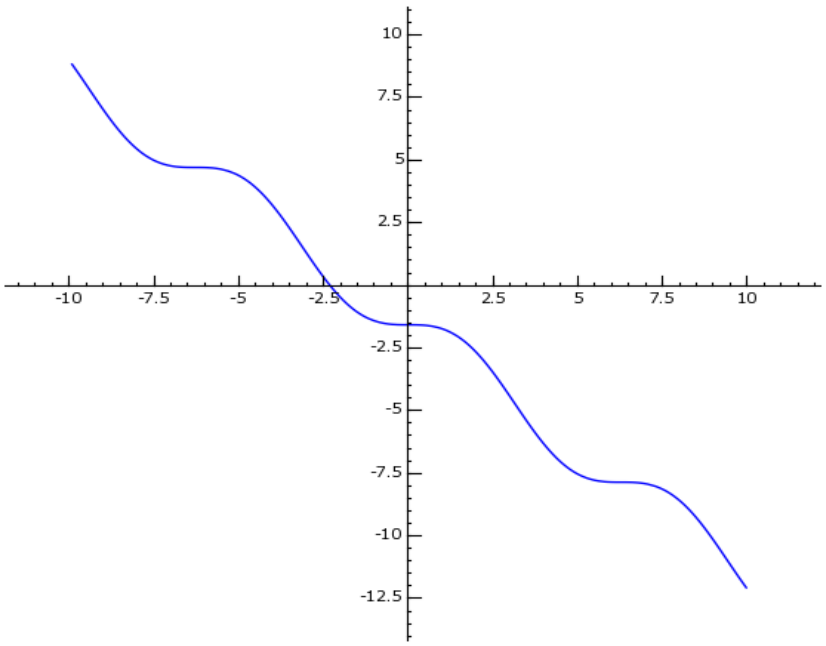
方程式が代数的に解けない場合には、solve()函数は引き渡された入力の複製を返す事でこれを示唆します。これは次の例で見られます：

```
f(x) = sin(x) - x - pi/2
eqn = (f == 0)
solve(eqn, x)
```

```
   [x == (2*sin(x) - pi)/2]
```

ところで、代数的に解けない方程式はグラフを使ったり数値的な方法の両方で解く事が出来ます。次の例では上の方程式をグラフを使って解いています：

```
show(plot(f,-10,10))
```



このグラフは方程式の根が-2.5 よりも僅かに小さいことを示しています。

次の例で、この方程式をより正確に find_root()メソッドを使って解けることを示しています：

```
f.find_root(-10,10)
|
   -2.309881460010057
```

-10 と 10 を find_root()メソッドに引き渡す事で、根を探す区間を教えています

5.15 伝統的な書式で数学対象を表示

早くから、SAGE は数学的対象を**テキスト書式**か**伝統的書式**の何れかで表示出来る様にしていました。この点について、我々は通常はテキスト書式を用います。数学対象を伝統的な書式で表示したければ、show()函数が使えます。次の例題では数学式を生成し、それからテキスト書式と伝統的書式の両方で表示します：

```
var('y,b,c')
z = (3*y^(2*b))/(4*x^c)^2

#テキスト書式で式を表示します。
z
|
   3*y^(2*b)/(16*x^(2*c))

#伝統的な書式で式を表示します。
show(z)
|
```

$$\frac{3 \cdot y^{2 \cdot b}}{16 \cdot x^{2 \cdot c}}$$

5.15.1 伝統的な数式で対象を表示する為に LaTeX を利用

LaTex（ラテフと発音、 http://en.wikipedia.org/wiki/LaTeX)は文書作成言語で、莫大な数学記号を扱う事が出来ます。SAGE の対象は latex()メソッドが呼び出されると、対応する LaTeX 表記を出力します。対象の LaTeX 表記は latex()函数からも得られます：

```
a = (2*x^2)/7
```

```
latex(a)
|
   \frac{{2 \cdot {x}^{2} }}{7}
```

この結果を LaTeX に与えると、次に示す様な伝統的な数式出力を生成します：

$$\frac{2x^2}{7}$$

図 2.5 で参照されている jsMath パッケージは SAGE のノートブックが LaTeX 入力を伝統的な数式に変換する為に用いているソフトウエアです。

5.16 集合

次の例題では SAGE が集合上で行える操作を示しています：

```
a = Set([0,1,2,3,4])
b = Set([5,6,7,8,9,0])
a,b
|
   ({0, 1, 2, 3, 4}, {0, 5, 6, 7, 8, 9})

a.cardinality()
|
   5

3 in a
|
   True

3 in b
|
   False

a.union(b)
|
   {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

a.intersection(b)
|
   {0}
```

6 2D 描画

6.1 plot()と show()函数

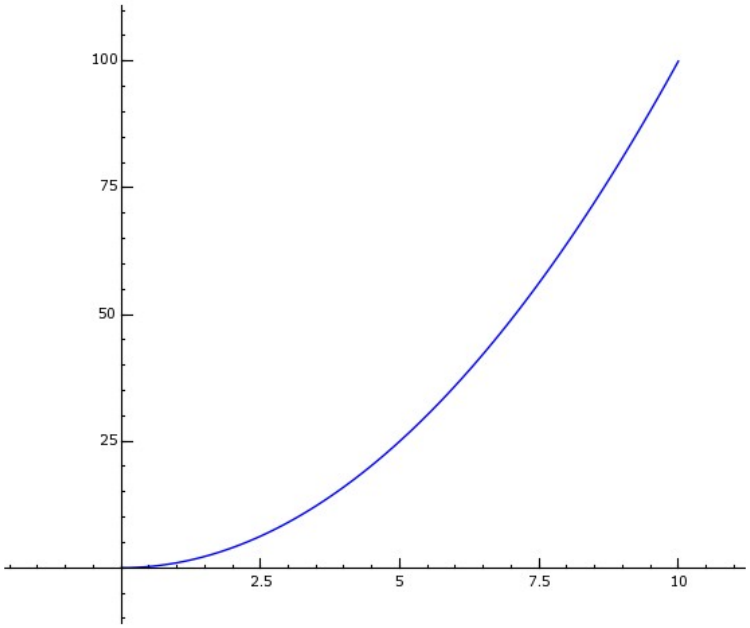
SAGE は数学函数の 2D 描画を生成する為の沢山の方法を提供し、そのうちの一つが**plot()**函数を **show()**函数と絡めて用いる事です。次の例題では記号式を plot()函数の最初の引数として引き渡されることを示しています。第 2 の引数は X 軸上での描画の始点を指し、第 3 の引数が描画の終点を指します：

```
a = x^2
b = plot(a, 0, 10)
type(b)
```

```
<class 'sage.plot.plot.Graphics'>
```

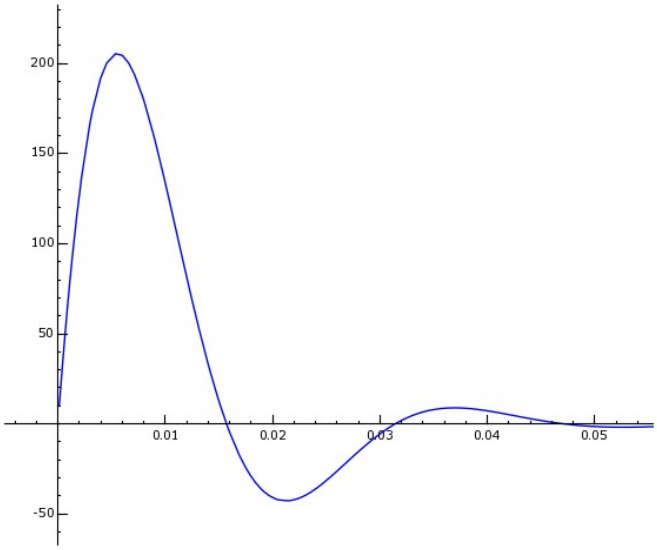
plot()函数は描画を表示する函数ではないことに注意してください。代わりに、それは sage.plot.plot.Graphics 型の対象を生成しますが、この対象は描画の与件を含んでいます。**show()**函数がこの描画の表示で使えるのです:

```
show(b)
```



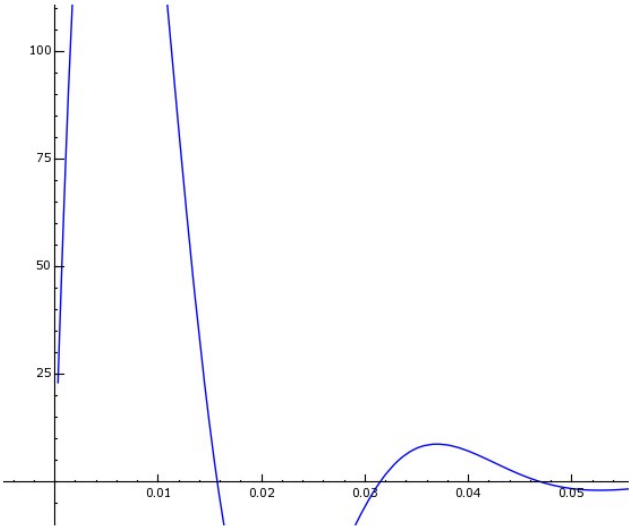
show()函数は**xmin**, **xmax**, **ymin** と **ymax** と呼ばれる 4 個の引数を持ち、これらは表示する描画を調整するために用いられます。**figsize** 助変数もあり、これは画像の大きさを決定します。次の例では、**xmin** と **xmax** を X 軸上の 0 と.05 の間の描画を表示する様に用いています。plot()函数は入力の労を減らす為に show()函数への第 1 引数として用いる事が可能です(註：x 以外の任意の記号変数が用いられていれば、最初に var()函数を使って宣言を行っていなければなりません):

```
v = 400*e^(-100*x)*sin(200*x)
show(plot(v,0,.1),xmin=0, xmax=.05, figsize=[3,3])
```



引数**ymin** と **ymax** は上の描画で表示される y 軸の大きさを調整する為に使えます：

```
show(plot(v,0,.1),xmin=0, xmax=.05, ymin=0, ymax=100, figsize=[3,3])
```



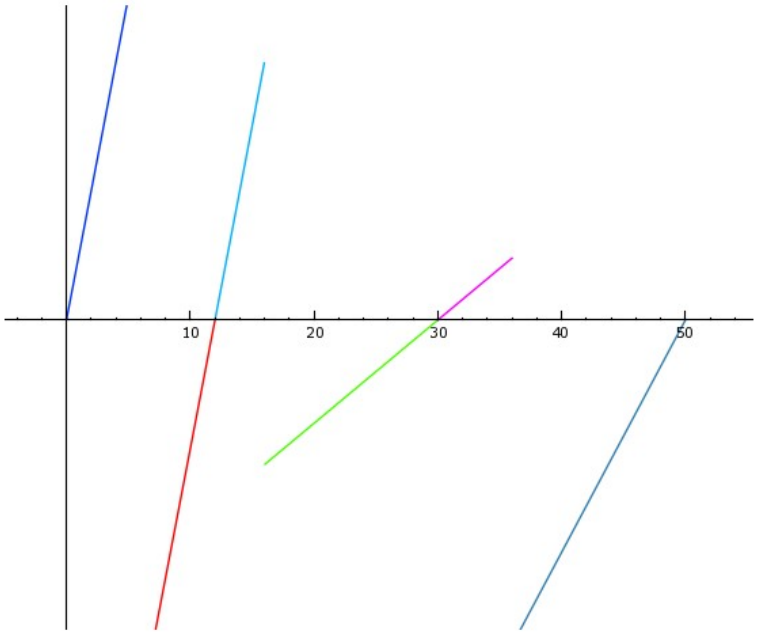
6.1.1 描画の結合と描画の色の変更

一つ、あるいはそれ以上の描画を纏めて一つの描画にする必要があるかもしれません。次の例では、show()函数を用いて 6 個の描画を結合します：

```
var('t')
p1 = t/4E5
p2 = (5*(t - 8)/2 - 10)/1000000
p3 = (t - 12)/400000
p4 = 0.0000004*(t - 30)
p5 = 0.0000004*(t - 30)
p6 = -0.0000006*(6 - 3*(t - 46)/2)

g1 = plot(p1,0,6,rgbcolor=(0,.2,1))
g2 = plot(p2,6,12,rgbcolor=(1,0,0))
g3 = plot(p3,12,16,rgbcolor=(0,.7,1))
g4 = plot(p4,16,30,rgbcolor=(.3,1,0))
g5 = plot(p5,30,36,rgbcolor=(1,0,1))
g6 = plot(p6,36,50,rgbcolor=(.2,.5,.7))

show(g1+g2+g3+g4+g5+g6,xmin=0, xmax=50, ymin=-.00001, ymax=.00001)
```



各描画の色は **rgbcolor** 助変数を用いて変更する事が出来ることに注意してください。RGB は、赤、緑、青のタプルで成り立ち、**rgbcolor** 助変数には 0 から 1 の間の三つの値を割り当てます。最初の値は最初の値は、描画の**赤**をどの程度(0 から 100%の間)似するかを指定し、二番目の値は描画の**緑**をどの程度にするかを指定し、それから三番目の値は描画の**青**をどの程度にするかを指定します。

6.1.2 グラフィックス対象とグラフィックスの結合

一つの画像に様々な種類のグラフィックスを纏めることはしばしば便利な事です。次の例では、6 点が描画されて、各点にはラベルがあります：

```
"""
次の点をグラフに描画:

A (0,0)
B (9,23)
C (-15,20)
D (22,-12)
E (-5,-12)
F (-22,-4)
"""

# 複数のグラフィックス対象を持足せるためにグラフィックス対象を生成します。
```

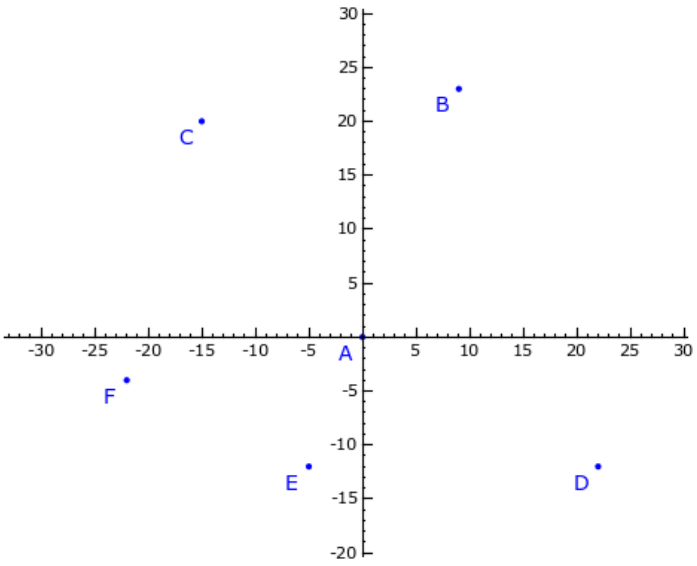
```
# これらのグラフィックス対象は同じ画像上に表示されます。
g = Graphics()

# 点のリストを生成し、それらをグラフィックス対象に追加します。
points=[(0,0), (9,23), (-15,20), (22,-12), (-5,-12), (-22,-4)]
g += point(points)

# 点のラベルをグラフィックス対象に追加します。
for (pnt,letter) in zip(points,['A','B','C','D','E','F']):
    g += text(letter,(pnt[0]-1.5, pnt[1]-1.5))

#結合したグラフィクス対象を表示します。
show(g,figsize=[5,4])

```

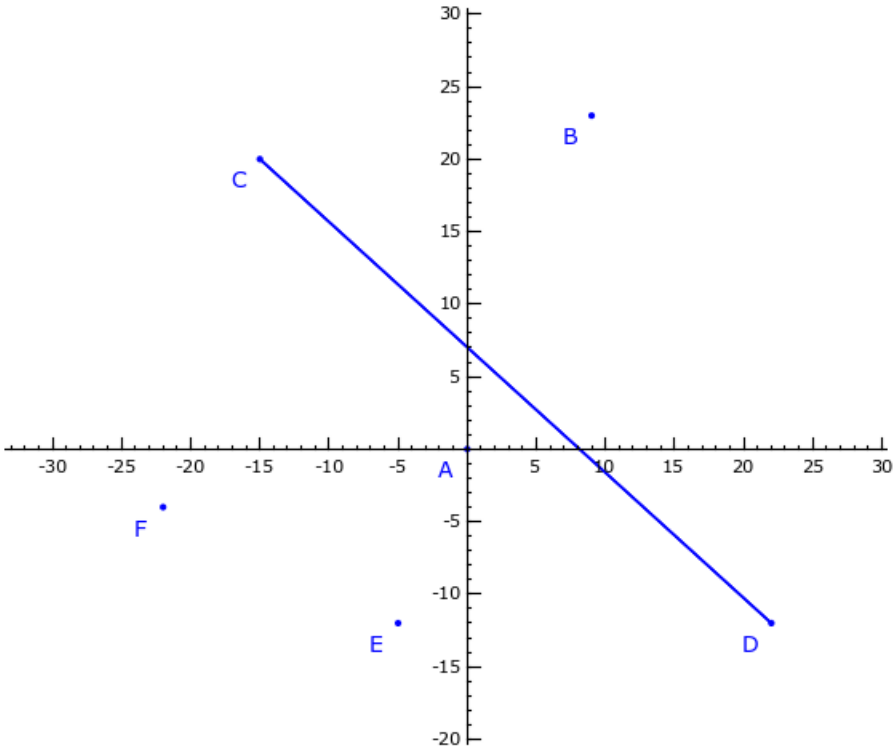


最初に、空のグラフィックス対象を構築し、描画点のリストを point()函数を使って生成します。それから、これらの描画される点は+=演算子を使ってグラフィックス対象に追加されます。次に、各点のラベルはグラフィックス対象に for 文を使って追加されます。最後に、グラフィックス対象は show()函数を使ってワークシート上に表示されます。

表示し終わっても、グラフィックス対象はそれの中に置かれた全てのグラフィックスを含んでおり、その上、必要であればグラフィックスをさらに追加することが可能です。たとえば、点 C と D の間の線分を描く必要があれば、これを完遂する為に別のセルで次のコードを実行することが出来ます：

```
g += line([(-15,20), (22,-12)])
```

```
show(g)
```



6.2 matplotlibによる進んだ描画

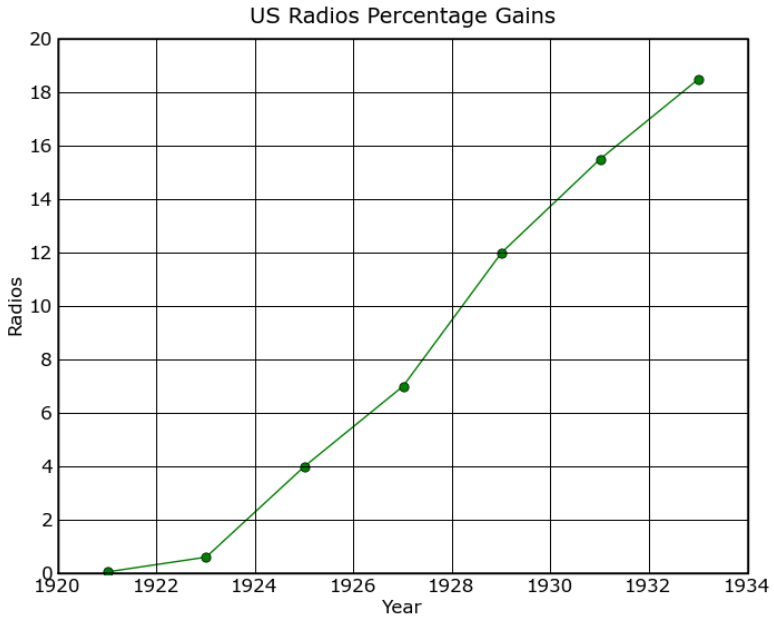
SAGEは**matplotlib**（http://matplotlib.sourceforge.net）ライブラリを必要であれば描画で用い、もし、plot()が提供する機能以上の描画制御が必要になれば、matplotの機能を直接使うことが出来ます。matplotlibがどのように動作するかの完全な解説はこの本の程度を越えてしまうので、この節では貴方の手助けになりそうな例を示しておきます。

6.2.1 網目と軸のラベルを持ったリストデータの描画

```
x = [1921, 1923, 1925, 1927, 1929, 1931, 1933]
y = [ .05, .6, 4.0, 7.0, 12.0, 15.5, 18.5]

from matplotlib.backends.backend_agg import FigureCanvasAgg as \
FigureCanvas
from matplotlib.figure import Figure
from matplotlib.ticker import *
fig = Figure()
```

```
2175  canvas = FigureCanvas(fig)
2176  ax = fig.add_subplot(111)
2177  ax.xaxis.set_major_formatter( FormatStrFormatter( '%d' ))
2178  ax.yaxis.set_major_locator( MaxNLocator(10) )
2179  ax.yaxis.set_major_formatter( FormatStrFormatter( '%d' ))
2180  ax.yaxis.grid(True, linestyle='-', which='minor')
2181  ax.grid(True, linestyle='-', linewidth=.5)
2182  ax.set_title('US Radios Percentage Gains')
2183  ax.set_xlabel('Year')
2184  ax.set_ylabel('Radios')
2185  ax.plot(x,y, 'go-', linewidth=1.0 )
2186  canvas.print_figure('ex1_linear.png')
2187
```

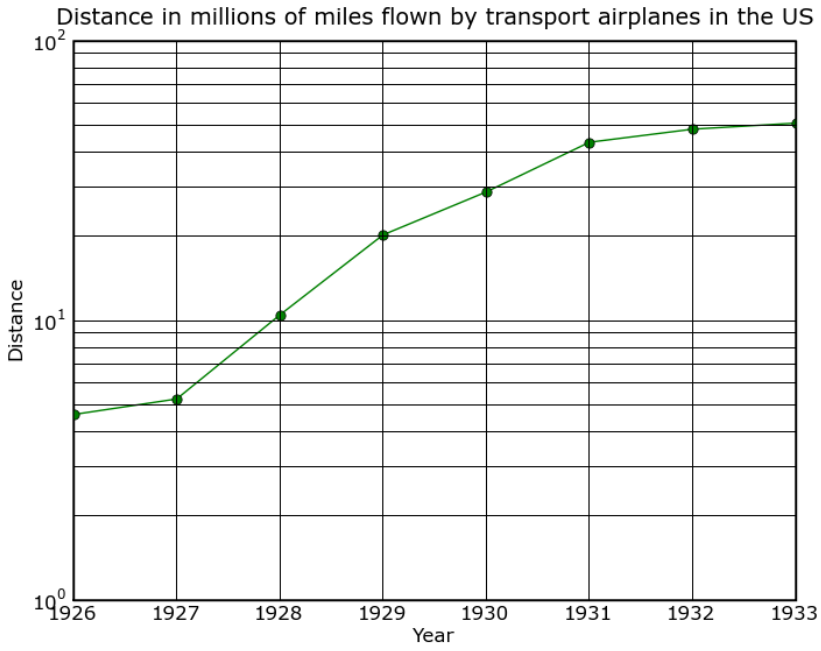


6.2.2 対数目盛のY軸を持った描画

```
2188  x = [1926, 1927, 1928, 1929, 1930, 1931, 1932, 1933]
2189  y = [ 4.61,5.24, 10.47, 20.24, 28.83, 43.40, 48.34, 50.80]

2190  from matplotlib.backends.backend_agg import FigureCanvasAgg as \
2191  FigureCanvas
2192  from matplotlib.figure import Figure
2193  from matplotlib.ticker import *
2194  fig = Figure()
2195  canvas = FigureCanvas(fig)
2196  ax = fig.add_subplot(111)
```

```
2197  ax.xaxis.set_major_formatter( FormatStrFormatter( '%d' ))
2198  ax.yaxis.set_major_locator( MaxNLocator(10) )
2199  ax.yaxis.set_major_formatter( FormatStrFormatter( '%d' ))
2200  ax.yaxis.grid(True, linestyle='-', which='minor')
2201  ax.grid(True, linestyle='-', linewidth=.5)
2202  ax.set_title('Distance in millions of miles flown by transport
2203  airplanes in the US')
2204  ax.set_xlabel('Year')
2205  ax.set_ylabel('Distance')
2206  ax.semilogy(x,y, 'go-', linewidth=1.0 )
2207  canvas.print_figure('ex2_log.png')
2208
```

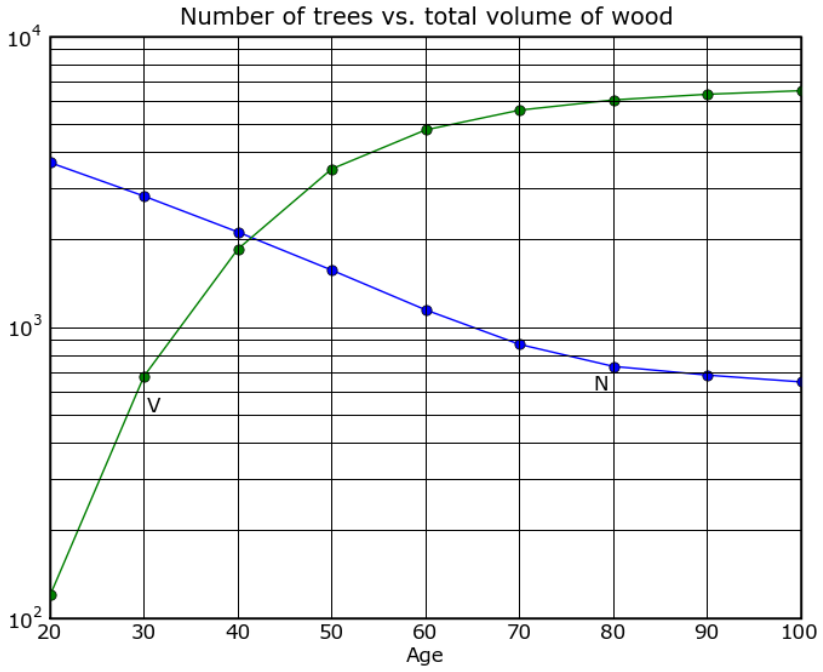


6.2.3 描画の中にラベル付きの二つのグラフ

```
2209  x = [20,30,40,50,60,70,80,90,100]
2210  y = [3690,2830,2130,1575,1150,875,735,686,650]
2211  z = [120,680,1860,3510,4780,5590,6060,6340,6520]

2212  from matplotlib.backends.backend_agg import FigureCanvasAgg as \
2213  FigureCanvas
2214  from matplotlib.figure import Figure
2215  from matplotlib.ticker import *
2216  from matplotlib.dates import *
2217  fig = Figure()
```

```
2218  canvas = FigureCanvas(fig)
2219  ax = fig.add_subplot(111)
2220  ax.xaxis.set_major_formatter( FormatStrFormatter( '%d' ))
2221  ax.yaxis.set_major_locator( MaxNLocator(10) )
2222  ax.yaxis.set_major_formatter( FormatStrFormatter( '%d' ))
2223  ax.yaxis.grid(True, linestyle='-', which='minor')
2224  ax.grid(True, linestyle='-', linewidth=.5)
2225  ax.set_title('Number of trees vs. total volume of wood')
2226  ax.set_xlabel('Age')
2227  ax.set_ylabel('')
2228  ax.semilogy(x,y, 'bo-', linewidth=1.0 )
2229  ax.semilogy(x,z, 'go-', linewidth=1.0 )
2230  ax.annotate('N', xy=(550, 248),  xycoords='figure pixels')
2231  ax.annotate('V', xy=(180, 230),  xycoords='figure pixels')
2232  canvas.print_figure('ex5_log.png')
2233
```



7 SAGE の書式

SAGE はとても柔軟な環境なので、それ故に様々な利用方法があります。この章では二つの SAGE の構文書式について議論しますが、それらは**スピード書式(Speed Usage Style)**と **OpenOffice プレゼン書式(OpenOffice Presentation Style)**と呼びます。

スピード書式は問題を可能な限り素早く解くことを目的に設計されており、結果の見栄えを良くする為に割く労力を最低にしています。この書式は通常の数学の教科書にある章末問題を解く事にとりわけ適しています。

OpenOffice プレゼン書式は数学文書の作成能力がない人でも最小の労力で数学文書か書ける様に設計されています。このプレゼン書式はお家での宿題、報告書、記事や本等の作成に便利で、この本はこの書式を使って構築したものです。

7.1 スピード書式

（まだ...）

7.2 オープンオフィスプレゼン書式

（まだ...）

8 高校数学の問題（大半がまだ著作中）

8.1 Pre-Algebra

Wikipedia entry.	http://en.wikipedia.org/wiki/Pre-algebra

(In development...)

8.1.1 方程式

Wikipedia entry.	http://en.wikipedia.org/wiki/Equation

(In development...)

8.1.2 式

Wikipedia entry.	http://en.wikipedia.org/wiki/Mathematical_expression

(In development...)

8.1.3 幾何学

Wikipedia entry.	http://en.wikipedia.org/wiki/Geometry

(In development...)

8.1.4 不等式

Wikipedia entry.	http://en.wikipedia.org/wiki/Inequality

(In development...)

8.1.5 線形函数

Wikipedia entry.	http://en.wikipedia.org/wiki/Linear_functions

(In development...)

8.1.6 Measurement

Wikipedia entry.	http://en.wikipedia.org/wiki/Measurement

(In development...)

8.1.7 非線形方程式

Wikipedia entry.	http://en.wikipedia.org/wiki/Nonlinear_system

(In development...)

8.1.8 Number Sense And Operations

Wikipedia entry.	http://en.wikipedia.org/wiki/Number_sense
Wikipedia entry.	http://en.wikipedia.org/wiki/Operation_(mathematics)

(In development...)

8.1.8.1 分数の約分

```
"""
問題:
90/105 を約分しなさい。

解:
この問題を解く一つの方法は、分子と分母の双方を素数因子に素因数分解して共通の因子を
見つけて、それから分子と分母の双方をこれらの因子で割ってしまいます。
"""
n = 90
d = 105
print n,n.factor()
print d,d.factor()

    Numerator: 2 * 3^2 * 5
    Denominator: 3 * 5 * 7


"""
分母と分子の双方に因子 3 と 5 が現れる事が分かり、そこで、分子と分母の双方を 3*5 で割
ります:
"""
n2 = n/(3*5)
```

```
d2 = d/(3*5)
print "Numerator2:",n2
print "Denominator2:",d2
```

```
    Numerator2: 6
    Denominator2: 7
```

```
"""
```

それ故に, 6/7 が 90/105 の約分になります。

この問題はまた、単に 90/105 をセルの中に入れるだけでも解けます。何故なら、有理数対象は自律的に約分されるからです：

```
"""
90/105
```

```
    6/7
```

8.1.9 多項式函数

Wikipedia entry.	http://en.wikipedia.org/wiki/Polynomial_function

(まだ...)

8.2 代数

Wikipedia entry.	http://en.wikipedia.org/wiki/Algebra_1

(まだ...)

8.2.1 絶対値函数

Wikipedia entry.	http://en.wikipedia.org/wiki/Absolute_value

(まだ...)

8.2.2 複素数

Wikipedia entry.	http://en.wikipedia.org/wiki/Complex_numbers

(まだ...)

8.2.3 合成函数

Wikipedia entry.	http://en.wikipedia.org/wiki/Composite_function

(まだ...)

8.2.4 Conics

Wikipedia entry.	http://en.wikipedia.org/wiki/Conics

(まだ...)

8.2.5 データ解析

Wikipedia entry.	http://en.wikipedia.org/wiki/Data_analysis

(まだ...)

9 離散数学：初等的数とグラフ理論

Wikipedia entry.	http://en.wikipedia.org/wiki/Discrete_mathematics

(まだ...)

9.1.1 方程式

Wikipedia entry.	http://en.wikipedia.org/wiki/Equation

(まだ...)

9.1.1.1 記号分数の約分

"""

問題:

式 (6*x^2 - b) / (b - 6*a*b) を約分しなさい。ここで a と b は正整数を表現します。

解:

"""

```
var('a,b')
n = 6*a^2 - a
d = b - 6 * a * b
```

```
print n
print "                                    ---------"
print d
```

```
                                 2
                              6 a  - a
                              ---------
                              b - 6 a b
```

"""
分子と分母の双方の因子分解を始め、それから共通因子を探します：
"""

```
n2 = n.factor()
d2 = d.factor()
print "Factored numerator:",n2.__repr__()
print "Factored denominator:",d2.__repr__()
```

```
    Factored numerator: a*(6*a - 1)
    Factored denominator: -(6*a - 1)*b
```

"""
最初に、分子と分母の双方に何も共通因子が現れません。ところで、分子をもっと調べる
と、-1 を掛けることで(1 - 6 a)が見つけられるので、この(6 a - 1)が答で、この因子も分
子に含まれています。だから、次の段階では分子と分母の双方に-1 を掛けます：
"""

```
n3 = n2 * -1
d3 = d2 * -1
print "Numerator * -1:",n3.__repr__()
print "Denominator * -1:",d3.__repr__()
```

```
    Numerator * -1: -a*(6*a - 1)
    Denominator * -1: (6*a - 1)*b
```

"""
そこで、約分の為に分子と分母の双方が(6*a - 1)で割り切ることが出来ます：

```
"""
common_factor = 6*a - 1
n4 = n3 / common_factor
d4 = d3 / common_factor
print n4
print "                                          ---"
print d4
```

```
                                  - a
                                  ---
                                   b
```

```
"""
この問題は SymbolicArithmetic 対象を使うことで、より直接的に解けます：
"""
z = n/d
z.simplify_rational()
```

```
    -a/b
```

9.1.1.2 二つの記号分数の積を計算

次の計算を実行:　$(\frac{x}{2y})^2 . (\frac{4y^2}{3x})^3$

```
"""
記号式は通常自律的に簡易化されるので、この問題で実行されるべき全ての事は式を入力し
てある変数に割り当てる事です：
"""

var('y')
a = (x/(2*y))^2 * ((4*y^2)/(3*x))^3

#テキスト形式で式を表示:
a
```

```
    16*y^4/(27*x)
```

```
#伝統的な書式で式を表示：
show(a)
```

|

$$\frac{16 \cdot y^4}{27 \cdot x}$$

9.1.1.3 x について線型方程式を解く

$3x+2x-8=5x-3x+7$ を解け

```
"""
この方程式を SymbolicEquation 対象として置くと、項の様に自律的に結合されます：
"""
a = 5*x + 2*x - 8 == 5*x - 3*x + 7
a
```

|

```
    7*x - 8 == 2*x + 7
```

```
"""
最初に、方程式の左側に x の項を移動させる為に 2x を両側から引きます（註：下線記
号'_'は直前に実行されたセルの結果を保っています:
"""
_ - 2*x
```

|

```
    5*x - 8 == 7
```

```
"""
両側に 8 を加えます:
"""
_+8
```

|

```
    5*x == 15
```

```
"""
```

最後に、解を決める為に両側を 5 で割ります：

```
"""
_/5
```

```
    x == 3
```

```
"""
```

この問題は solve()を使うと自律的に解かれます：

```
"""
solve(a,x)
```

```
    [x == 3]
```

9.1.1.4 *分数を持つ線型方程式の解法*

$\frac{16x-13}{6}=\frac{3x+5}{2}-\frac{4-x}{3}$ を解け

```
"""
```

最初の段階は方程式を SymbolicEquation 対象に置き換える事です。方程式を表示させる事は、方程式が正しく入力されたかを検証する事が出来るので良い思いつきです：

```
"""
a = (16*x - 13)/6 == (3*x + 5)/2 - (4 - x)/3
a
```

```
    (16*x - 13)/6 == (3*x + 5)/2 - (4 - x)/3
```

```
"""
```

テキストとして方程式がされると、方程式が正しく入力されたかどうか確認する事は困難です。だから、伝統的な書式でも表示させましょう：

```
"""
show(a)
```

$$\frac{16\cdot x-13}{6} = \frac{3\cdot x+5}{2} - \frac{4-x}{3}$$

```
"""
```

次の段階で、方程式の分母を消せるように最小公倍数(LCD)を計算しましょう：

```
"""
```

```
lcm([6,2,3])
|
    6

"""
方程式の LCD は 6 なので、 6 を掛けて分数を消してしまいます：
"""
b = a*6
b
|
    16*x - 13 == 6*((3*x + 5)/2 - (4 - x)/3)

"""
方程式の右辺にはまだ分数があるので展開してみましょう：
"""
c = b.expand()
c
|
    16*x - 13 == 11*x + 7

"""
11x を方程式の左辺に移す為に、SymbolEquation から 11x を引きます:
"""
d = c - 11*x
d
|
    5*x - 13 == 7

"""
-13 を方程式の右辺に移動させるために SymbolEquation に対して 13 を加えましょう:
"""
e = d + 13
e
|
    5*x == 20

"""
```

最後に、SymbolEquation を 5 で割って、方程式の左側を x だけにすると解が得られます：
"""
```
f = e / 5
f
```
```
    x == 4
```

"""
この問題は solve()を使うと自律的に解かれます：
"""
```
solve(a,x)
```
```
    [x == 4]
```

9.1.2 指数函数

Wikipedia の見出	http://en.wikipedia.org/wiki/Exponential_function

(まだ...)

9.1.3 冪

Wikipedia の見出.	http://en.wikipedia.org/wiki/Exponent

(まだ...)

9.1.4 式

Wikipedia の見出.	http://en.wikipedia.org/wiki/Expression (mathematics)

(まだ...)

9.1.5 不等式

Wikipedia の見出.	http://en.wikipedia.org/wiki/Inequality

(まだ...)

9.1.6 逆函数

Wikipedia の見出.	http://en.wikipedia.org/wiki/Inverse_function

(まだ...)

9.1.7 線型方程式と函数

Wikipedia の見出.	http://en.wikipedia.org/wiki/Linear_functions

(まだ...)

9.1.8 線型プログラミング

Wikipedia の見出.	http://en.wikipedia.org/wiki/Linear_programming

(まだ...)

9.1.9 対数函数

Wikipedia の見出.	http://en.wikipedia.org/wiki/Logarithmic_function

(まだ...)

9.1.10 兵站函数

Wikipedia の見出.	http://en.wikipedia.org/wiki/Logistic_function

(まだ...)

9.1.11 行列

Wikipedia の見出.	http://en.wikipedia.org/wiki/Matrix_(mathematics)

(まだ...)

9.1.12 Parametric Equations

Wikipedia 見出.	http://en.wikipedia.org/wiki/Parametric_equation

(In development...)

9.1.13 区分函数

Wikipedia 見出.	http://en.wikipedia.org/wiki/Piecewise_function

(In development...)

9.1.14 多項式函数

Wikipedia 見出.	http://en.wikipedia.org/wiki/Polynomial_function

(In development...)

9.1.15 冪級数函数

Wikipedia の見出.	http://en.wikipedia.org/wiki/Power_function

(In development...)

9.1.16 Quadratic Functions

Wikipedia entry.	http://en.wikipedia.org/wiki/Quadratic_function

(In development...)

9.1.17 Radical Functions

Wikipedia の見出.	http://en.wikipedia.org/wiki/Nth_root

(まだ...)

9.1.18 有理函数

Wikipedia の見出.	http://en.wikipedia.org/wiki/Rational_function

(In development...)

9.1.19 列

Wikipedia の見出.	http://en.wikipedia.org/wiki/Sequence

(In development...)

9.1.20 級数

Wikipedia の見出.	http://en.wikipedia.org/wiki/Series_mathematics

(In development...)

9.1.21 方程式系

Wikipedia の見出.	http://en.wikipedia.org/wiki/System_of_equations

(In development...)

9.1.22 変換

Wikipedia の見出.	http://en.wikipedia.org/wiki/Transformation_(geometry)

(In development...)

9.1.23 三角函数

Wikipedia の見出.	http://en.wikipedia.org/wiki/Trigonometric_function

(In development...)

9.2 Precalculus And Trigonometry

Wikipedia entry.	http://en.wikipedia.org/wiki/Precalculus
	http://en.wikipedia.org/wiki/Trigonometry

(In development...)

9.2.1 二項定理

Wikipedia entry.	http://en.wikipedia.org/wiki/Binomial_theorem

(In development...)

9.2.2 複素数

Wikipedia entry.	http://en.wikipedia.org/wiki/Complex_numbers

(In development...)

9.2.3 合成函数

Wikipedia entry.	http://en.wikipedia.org/wiki/Composite_function

(In development...)

9.2.4 Conics

Wikipedia entry.	http://en.wikipedia.org/wiki/Conics

(In development...)

9.2.5 データ解析

Wikipedia entry.	http://en.wikipedia.org/wiki/Data_analysis

(In development...)

10 離散数学: Elementary Number とグラフ理論

Wikipedia entry.	http://en.wikipedia.org/wiki/Discrete_mathematics

(In development...)

10.1.1 方程式

Wikipedia entry.	http://en.wikipedia.org/wiki/Equation

(In development...)

10.1.2 指数函数

Wikipedia entry.	http://en.wikipedia.org/wiki/Equation

(In development...)

10.1.3 逆函数

Wikipedia entry.	http://en.wikipedia.org/wiki/Inverse_function

(In development...)

10.1.4 対数函数

Wikipedia entry.	http://en.wikipedia.org/wiki/Logarithmic_function

(In development...)

10.1.5 兵站函数

Wikipedia entry.	http://en.wikipedia.org/wiki/Logistic_function

(In development...)

10.1.6 行列と行列代数

Wikipedia entry.	http://en.wikipedia.org/wiki/Matrix_(mathematics)

(In development...)

10.1.7 数学的解析

Wikipedia entry.	http://en.wikipedia.org/wiki/Mathematical_analysis

(In development...)

10.1.8 Parametric Equations

Wikipedia entry.	http://en.wikipedia.org/wiki/Parametric_equation

(In development...)

10.1.9 区分函数

Wikipedia entry.	http://en.wikipedia.org/wiki/Piecewise_function

(In development...)

10.1.10 Polar Equations

Wikipedia entry.	http://en.wikipedia.org/wiki/Polar_equation

(In development...)

10.1.11 多項式函数

Wikipedia entry.	http://en.wikipedia.org/wiki/Polynomial_function

(In development...)

10.1.12 冪関数

Wikipedia entry.	http://en.wikipedia.org/wiki/Power_function

(In development...)

10.1.13 多項式函数

Wikipedia entry.	http://en.wikipedia.org/wiki/Quadratic_function

(In development...)

10.1.14 Radical Functions

Wikipedia entry.	http://en.wikipedia.org/wiki/Nth_root

(In development...)

10.1.15 有理函数

Wikipedia entry.	http://en.wikipedia.org/wiki/Rational_function

(In development...)

10.1.16 実数

Wikipedia entry.	http://en.wikipedia.org/wiki/Real_number

(In development...)

10.1.17 列

Wikipedia entry.	http://en.wikipedia.org/wiki/Sequence

(In development...)

10.1.18 級数

Wikipedia entry.	http://en.wikipedia.org/wiki/Series_(mathematics)

(In development...)

10.1.19 集合

Wikipedia entry.	http://en.wikipedia.org/wiki/Set

(In development...)

10.1.20 方程式系

Wikipedia entry.	http://en.wikipedia.org/wiki/System_of_equations

(In development...)

10.1.21 変換

Wikipedia entry.	http://en.wikipedia.org/wiki/Transformation_(geometry)

(In development...)

10.1.22 三角函数

Wikipedia entry.	http://en.wikipedia.org/wiki/Trigonometric_function

(In development...)

10.1.23 ベクトル

Wikipedia entry.	http://en.wikipedia.org/wiki/Vector

(In development...)

10.2 解析

Wikipedia entry.	http://en.wikipedia.org/wiki/Calculus

(In development...)

10.2.1 微分

Wikipedia entry.	http://en.wikipedia.org/wiki/Derivative

(In development...)

10.2.2 積分

Wikipedia entry.	http://en.wikipedia.org/wiki/Integral

(In development...)

10.2.3 極限

Wikipedia entry.	http://en.wikipedia.org/wiki/Limit_(mathematics)

(In development...)

10.2.4 多項式近似と級数

Wikipedia entry.	http://en.wikipedia.org/wiki/Convergent_series

(In development...)

10.3 統計

Wikipedia entry.	http://en.wikipedia.org/wiki/Statistics

(In development...)

10.3.1 データ解析

Wikipedia entry.	http://en.wikipedia.org/wiki/Data_analysis

(In development...)

10.3.2 Inferential Statistics

Wikipedia entry.	http://en.wikipedia.org/wiki/Inferential_statistics

(In development...)

10.3.3 標準分布

Wikipedia entry.	http://en.wikipedia.org/wiki/Normal_distribution

(In development...)

10.3.4 1変数解析

Wikipedia entry.	http://en.wikipedia.org/wiki/Univariate

(In development...)

10.3.5 確率と試行

Wikipedia entry.	http://en.wikipedia.org/wiki/Probability

(In development...)

10.3.6 ２変数解析

Wikipedia entry.	http://en.wikipedia.org/wiki/Multivariate

(In development...)

11 高校生の科学の問題

(In development...)

11.1 物理学

Wikipedia entry.	http://en.wikipedia.org/wiki/Physics

(In development...)

11.1.1 原子物理

Wikipedia entry.	http://en.wikipedia.org/wiki/Atomic_physics

(In development...)

11.1.2 円運動

Wikipedia entry.	http://en.wikipedia.org/wiki/Circular_motion

(In development...)

11.1.3 力学

Wikipedia entry.	http://en.wikipedia.org/wiki/Dynamics_(physics)

(In development...)

11.1.4 電磁界

Wikipedia entry.	http://en.wikipedia.org/wiki/Electricity
	http://en.wikipedia.org/wiki/Magnetism

(In development...)

11.1.5 流体

Wikipedia entry.	http://en.wikipedia.org/wiki/Fluids

(In development...)

11.1.6 運動学

Wikipedia entry.	http://en.wikipedia.org/wiki/Kinematics

(In development...)

11.1.7 光

Wikipedia entry.	http://en.wikipedia.org/wiki/Light

(In development...)

11.1.8 光学

Wikipedia entry.	http://en.wikipedia.org/wiki/Optics

(In development...)

11.1.9 相対性理論

Wikipedia entry.	http://en.wikipedia.org/wiki/Relativity

(In development...)

11.1.10 回転運動

Wikipedia entry.	http://en.wikipedia.org/wiki/Rotational_motion

(In development...)

11.1.11 音響

Wikipedia entry.	http://en.wikipedia.org/wiki/Sound

(In development...)

11.1.12 波

Wikipedia entry.	http://en.wikipedia.org/wiki/Waves

(In development...)

11.1.13 熱力学

Wikipedia entry.	http://en.wikipedia.org/wiki/Thermodynamics

(In development...)

11.1.14 仕事

Wikipedia entry.	http://en.wikipedia.org/wiki/Mechanical_work

(In development...)

11.1.15 エネルギー

Wikipedia entry.	http://en.wikipedia.org/wiki/Energy

(In development...)

11.1.16 モーメント

Wikipedia entry.	http://en.wikipedia.org/wiki/Momentum

(In development...)

11.1.17 Boiling

Wikipedia entry.	http://en.wikipedia.org/wiki/Boiling

(In development...)

11.1.18 浮力

Wikipedia entry.	http://en.wikipedia.org/wiki/Bouyancy

(In development...)

11.1.19 Convection

Wikipedia entry.	http://en.wikipedia.org/wiki/Convection

(In development...)

11.1.20 密度

Wikipedia entry.	http://en.wikipedia.org/wiki/Density

(In development...)

11.1.21 Diffusion

Wikipedia entry.	http://en.wikipedia.org/wiki/Diffusion

(In development...)

11.1.22 Freezing

Wikipedia entry.	http://en.wikipedia.org/wiki/Freezing

(In development...)

11.1.23 摩擦

Wikipedia entry.	http://en.wikipedia.org/wiki/Friction

(In development...)

11.1.24 熱伝導

Wikipedia entry.	http://en.wikipedia.org/wiki/Heat_transfer

(In development...)

11.1.25 Insulation

Wikipedia entry.	http://en.wikipedia.org/wiki/Insulation

(In development...)

11.1.26 Newtonの法則

Wikipedia entry.	http://en.wikipedia.org/wiki/Newtons_laws

(In development...)

11.1.27 圧力

Wikipedia entry.	http://en.wikipedia.org/wiki/Pressure

(In development...)

11.1.28 プーリー

Wikipedia entry.	http://en.wikipedia.org/wiki/Pulley

(In development...)

12 計算の基礎

12.1 計算機ってなあに

多くの人々は計算機が複雑である為に理解し難いと思っています。計算機は実際に複雑ですが、これが理解し難くしている理由ではありません。計算機が理解し難いのは、実体のある世界には計算機の小さな部分しか存在しないからです。実体としての計算機は人間が見ることの出来る部分でしかなく、計算機の残りは不可視の実体の無い世界に存在するからです。この不可視の世界は概念（イデア）の世界であり、計算機の大半は概念としてこの実体の無い世界にあるのです。

計算機を理解する為の鍵は、概念に基づく機械の目的が全ての型の概念の自動操作である事を理解する事にあります。'計算機'と言う名前は計算機が現実に何であるかを描くには余り助けになるものではなく、寧ろ、概念操作機器(Idea Manipulation Device)、即ち、IMDと言う名前の方が良いでしょう。

さて、概念は実体の無い対象なので、それらは実世界に持ち込めませんし、実世界の対象を概念の世界に持ち込む事も勿論出来ません。これらの二つの世界は互いに分離しているので、実世界の対象は記号による遠隔制御で概念の世界の対象を操作する事になります。

12.2 記号(symbol)って何?

記号(symbol) は別の対象を表現する為に用いられる対象です。　図 12.1 は電話の記号の例で、この記号は実際の電話を表現する為に用いています。

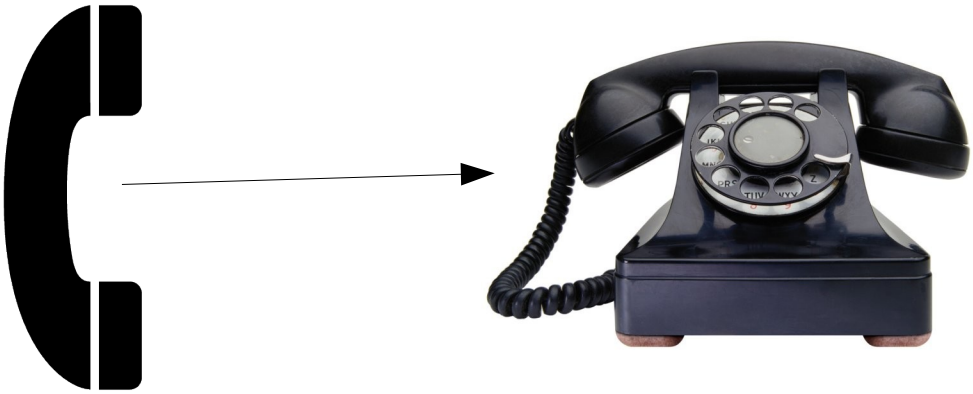


図 *12.1:* 実体を持つ対象と関連付けられた記号

図 12.1 で示した電話の記号は通常、（紙の様な）平面上でインクで印刷される事で生成されています。一般的に、ある種の並びとして調整された実体を持つ物の任意の型（或いは、実体を持つ物の属性）は記号として用いる事が出来ます。

12.3 記号としてビットの並びを使う計算機

実体のある物から出来上がった記号は全ての実体のある対象の型を表現出来ますが、それらはまた、概念の世界の実体を持たない対象を表現する事にも使えます(図 12.2 参照)

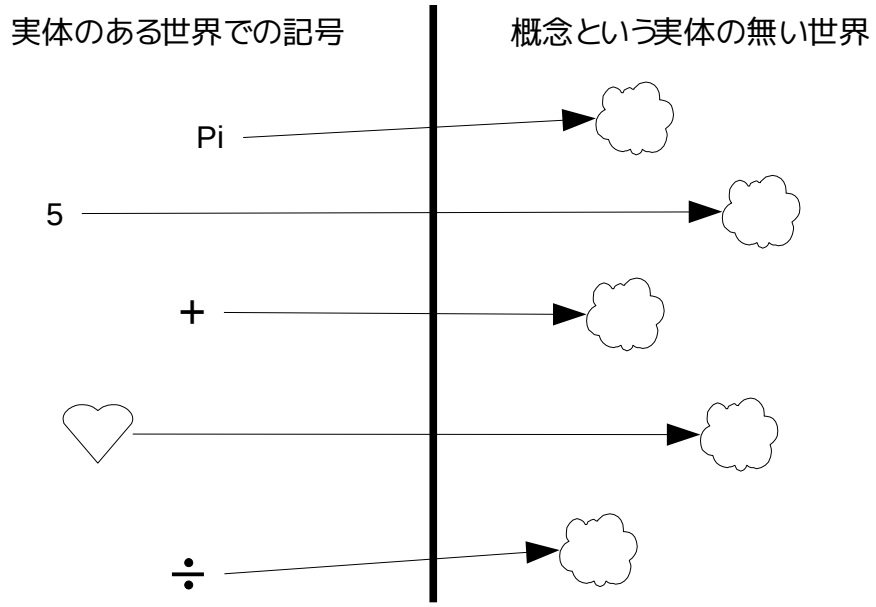


図 *12.2:* 実体のある記号は実体を持たない概念を表現出来る。

実体を持ったものより構成された記号で最も単純なものの中はビット(0/1)とビット(0/1)の羅列です。ビット単体は on 状態と off 状態という二つの状態に単純に置き換えられます。on の状態を書いたり、印字したり、表示すると、数値 1 で表現され、off の状態は数値 0 で表現されます。ビットの並びを書いたり、印字したり、表示すると次の様なものになります: 101, 100101101, 0101001100101, 10010.

図 12.3 はビットの並びが、実体を持たない概念を表現する為、実体を持つ任意のから構成された記号として簡単に用いられる事を示すものです。

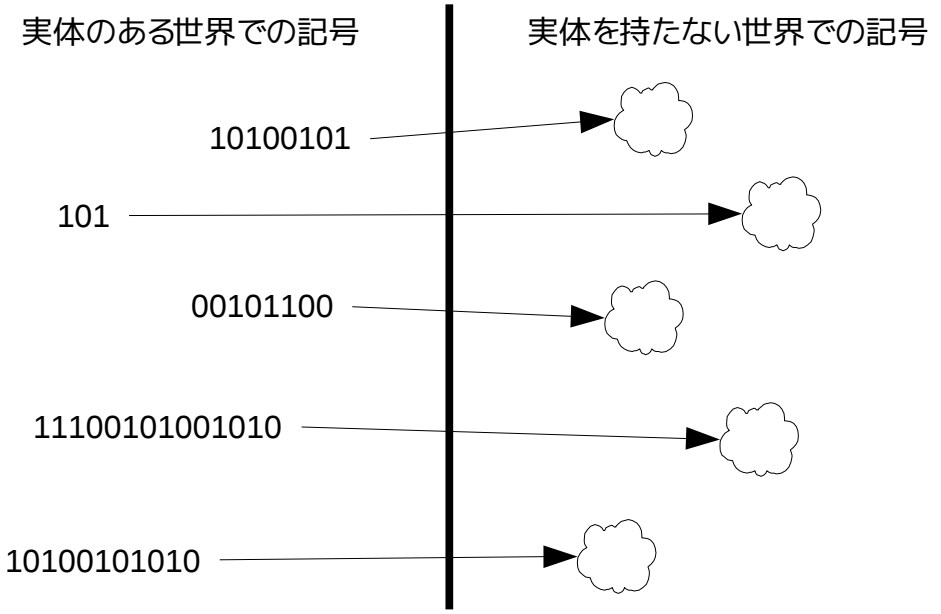


図 *12.3:* ビットは実体を持たない概念も表現出来る。

実体を持ったものをビットやビットの並びとして構成する他の手法は:二つの周波数の間の音声信号のトーンを変えたり、灯りを点けたり消したり、ある物体の表面上に磁場を置いたり取り除いたり、電子ディバイスで二つの水準の間で電圧を変えるといった方法があります。大半の計算機が概念を表現するビットの並びを保つ為にさいごのしゅほうをもちいています。

計算機の内部記憶は様々な”箱”、つまり、**記憶番地(memory location)と各記憶番地は概念を表現する為に用いることが可能なビットの並びで構成さています**。大半の計算機は数百万の番地をもっており、同時に数百万の概念を参照する事が容易に行えます。より大きな計算機は数十億の記憶番地を持っています。例えば、２００７年版の典型的な個人向けの計算機は十億を越える記憶番地を持っています。

図 12.4 に小さな計算機の内部記憶の断片にビットの並びが含まれている様子を示します。

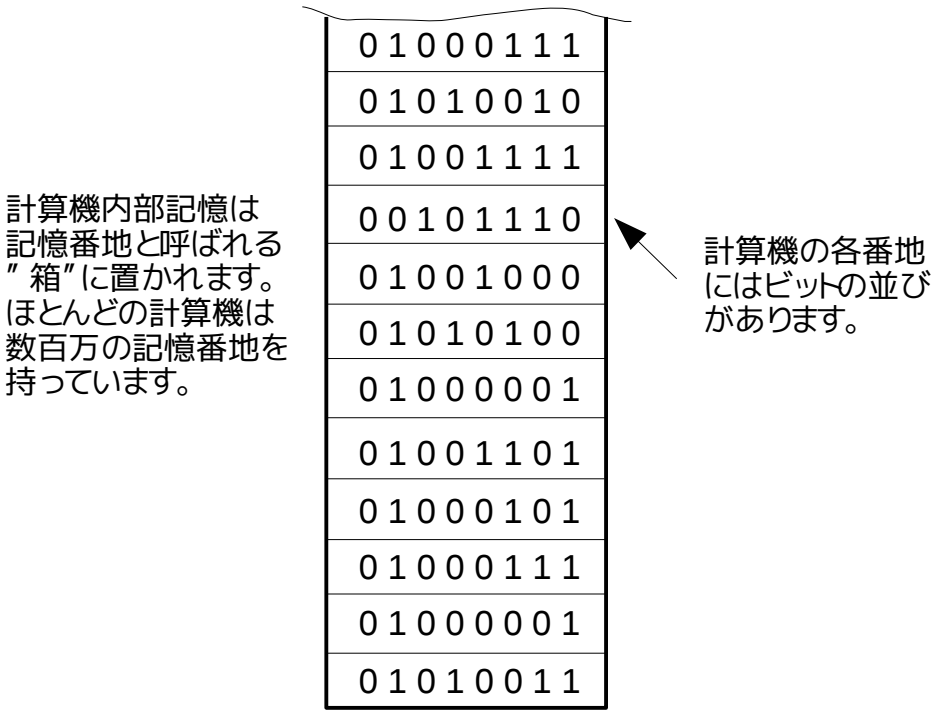


図 *12.4:* 計算機の記憶番地にはビットの並びがある。

計算機内部記憶での何百ものビットの並びのそれぞれが、人間が思考し得る任意の概念を表現する能力を持っています。ほとんどの計算機が保有する莫大なビットの並びは、ある種の組織化された体系を使わず配置を保つことは困難です。

計算機が保持する沢山のビットの並びを保つ為に用いている体系は、図 12.5 に示す様な一意な番地を各記憶番地に与えるものです。

図 *12.5:*各記憶番地には一意に番地が与えられます。

12.4 文脈的意味

この点で、貴方は”記憶番地のビットの並び、すなわち、記憶番地の集まり、意味をどのように決定しするの？”と不思議に思うかもしれません。この答えはビットの並びに意味を与える文脈的意味付けと呼ばれる概念になります。

文脈は状況の中で生じた事象、あるいは何かが置かれた環境のことです。**文脈的意味付**は、すなわち、文脈中に置かれた事象や物事に対して与える意味付のことです。

大半の人々が日々文脈的意味付を用いていますが、それに皆が気付いていません。文脈的意味付は非常に強力な概念であり、人間が思い浮かべられる任意の概念と計算機の記憶位置を結び付けるものです。 もし、一つの記憶で与えられた並びを保つ為により多くのビットが必要であれば、その並びは一つの場所以上に広げられるのです。

12.5 変項

計算機は数を覚えておくことがとても得意なので、簡単に膨大な番地を覚えられます。人間は、残念なことに計算機の様に数を覚える事は然程得意ではないので、この問題を解決する為に変項と呼ばれる概念が発明されました。.

変項は**数**の代わりに**名前**を使って記憶の中のビットの並び記号を人間が参照できる記憶番地に結びつけられる**名前**です。 図 12.6 に計算機内部での４つの記憶番地に結びつけられた四個の変項を示しておきます。

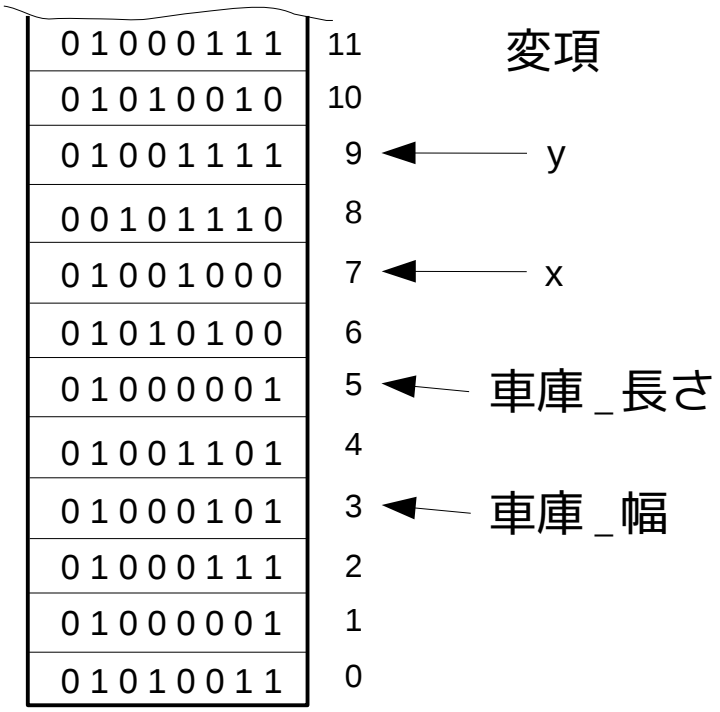


図 *12.6:* 記憶番地の代わりに変項を使うこと

車庫_幅と**車庫_長さ**と言う名前の変項は車庫の大きさを表現する並びが保存されている記憶番地を参照し、変項 **x** と **y** は方程式に於ける数を表現するような記憶番地を参照しています。上の変項の記述が正確であるものの、これはさすがに使うには冗長で、人々が”変数「車庫_長さ」は車庫の長さである”の様な発言や記述には時間がかかります。

変数は記号的に対象の属性を表現する為に用いられます。典型的な個人向けの計算機でさえも数百万もの変数を擁する能力があるとはいえ、大半の対象が、ほとんどの計算機が擁することのできる能力以上の莫大な属性を所有しています。例えば、１キログラムの岩はおおよそ、 10,000,000,000,000,000,000,000,000 個の原子を含んでいます。この岩の原子の位置を表現することでさえも、最も進んだ計算機の能力を凌駕しているのです。それ故に、計算機は通常、対象の完全な表現の代わりにモデルを用いて作業するのです。

12.6 モデル

モデルは対象の簡易化した表現で、対象のある種の属性のみに注目したものです。典型的な対象の属性としては、重さ、長さや色彩が含まれます。 モデルの為に選択される属性は所定

の目的の為に選ばれます。 モデルで表現される属性を増やせば増やす程、モデルの構築は高くつきます。 それ故に、所定の目的を達成する上で絶対的に必要とされる属性のみがモデルで通常表現されるのです。対象の属性の幾つかだけをモデルの構成時に選択する行為を**抽象化**と呼びます。

次の例はモデルを用いて問題を解く手順を描いたものです。2 台の車が入る車庫を作ることを考えましょう。この車庫には車に沿って工作台、対の収納庫に芝刈り機があります。この倉庫は適切な天井の高さがあり、最初に述べた目的に必要な大きさ以上の車庫を作る気が無いと仮定すると、車庫の適切な長さ幅をどの様に決めれば良いでしょうか?

車庫の大きさを決定するための一つの戦略は広い野原で大体 10 個のいろいろな大きさの車庫を作ることです。まず、車庫を作り終えると、二台の車を工作台、対の倉庫と芝刈り機と一緒に原っぱに置きます。それから、これらの物を車庫に交互に入れて、引っかからずに適合するもっとも小さな車庫を探します。

原っぱのテスト車庫は破棄しても良く、選択した物と同じ大きさの車庫を希望する場所に建てられます。不幸にして、一つではなく 11 個の車庫がこの方式では必要で、これは非常に高価で非効率でしょう。

この問題を費用をかけずに解く方法は**車庫のモデル**と**その中に置く物のモデル**を用いることです。車庫の床の大きさを決めたいだけなので、紙を使って床の縮小モデルを作ることが出来ます。

車庫に据える物も紙で縮小したもので表現する事が出来ます。すると、これらの物を表現する紙製のものは床を表現する大きな紙の上に置く事ができ、紙の小物はどうすれば適合するかを見るために動かす事が出来ます。もし、これらの小物があまりにも窮屈であれば、床を表現する為に、より大きな紙を切れば良く、スカスカであれば、床の為により小さな紙を切れば良いのです。

丁度良い具合になったら、床を表現する紙の長さと幅を計って、それらを実際の車庫向けに使える様に拡大します。この手法では、あとで捨ててしまう 10 個の実物の車庫の代わりに、幾らかの紙だけが問題を解くために必要とされます。

紙の小物に複製された実物大の対象の属性は対象の長さと幅でした。この例で示す様に、紙のモデルはそれらが表現する対象よりも明らかに作業し易いものです。ところで、**計算機変数は紙、あるいは他の大半のモデル手法よりもモデル化で扱うにはより簡単なものです。**

この点で、紙モデル化の技術は我々が注目している計算機変数について一つの重要な長所があります。紙モデルは小物のモデルを動かしたり、紙の車庫の床の大きさを変更することによって変更可能です。我々が議論する変数は対象属性を表現する為に与えられた能力がありますが、変数を変更する為の与えられた機構はありません。その変数の内容を変更する能力を持たない計算機は現実的に利用価値がありません。

12.7 機械語

計算機の記憶番地のビットの並びが人間が思考出来る任意の概念を表現する為に使えると以前述べました。記憶番地が任意の概念を表現していれば、それらが概念を参照できる事を意味し、この参照は計算機が自律的に記憶の中の変数をどの様に操作するか計算機に指図する命令(instruction)になります。

記憶の中の命令に追随する計算機の側は中央演算装置(CPU)、すなわち、マイクロプロセッサと呼ばれます。マイクロプロセッサが記憶の中の命令を追随しているときに、それらを走らせているとか**実行している**とも呼びます。

マイクロプロセッサはファミリーに分類され、各マイクロプロセッサのファミリーはそれ自身の命令の集合(**命令集合:instruction set**と呼ばれます)を持っており、他のマイクロプロセッサのファミリーが用いている命令とは異なっています。マイクロプロセッサの命令集合はマイクロプロセッサに何をさせるかを伝えることが出来る組み上げた積木の様な言語を表現します。この言語は記憶に送り込まれる命令集合からの命令列で構成されており、マイクロプロセッサが理解できる唯一の言語です。これはマイクロプロセッサが理解できる唯一の言語なので、**機械語**と呼ばれます。計算機にやらせることを伝える為の機械語の命令の列は**計算機プログラム**と呼ばれ、計算機にやらせることを伝えるための機械語の命令の列を作成する人は**プログラマー**と呼ばれます。

そこで、単純なマイクロプロセッサの命令集合がどのようなもので、この命令集合を使って構築された単純なプログラムを見てみましょう。

これがマイクロプロセッサ6500ファミリーの命令集合です：

ADC　キャリー付きで記憶を加算器に追加(ADd memory to accumulator with Carry)。
AND　記憶と加算器のANDをとる(AND memory with accumulator)。
ASL　1ビット算術的にずらす(Arithmetic Shift Left one bit)。
BCC　キャリークリアなら分岐(Branch on Carry Clear)。
BCS　キャリーセットなら分岐(Branch on Carry Set)。

BEQ　結果が零に等しければ分岐(Branch on result EQual to zero)。
BIT　加算器のビットをメモリと比較(test BITs in accumulator with memory)。
BMI　結果が負になれば分岐(Branch on result MInus)。
BNE　結果が零に等しくなければ分岐(Branch on result Not Equal to zero)。
BPL　結果が正なら分岐(Branch on result PLus)。
BRK　force Break.
BVC　オーバーフローフラグがクリアであれば分岐(Branch on oVerflow flag Clear)。
BVS　オーバーフローフラグがセットされれば分(Branch on oVerflow flag Set)。
CLC　キャリーフラグのクリア(CLear Carry flag)。
CLD　十進モードをクリア(CLear Decimal mode)。
CLI　インタラプトディセイブルフラグをクリア(CLear Interrupt disable flag)。
CLV　オーバーフローフラグをクリア(CLear oVerflow flag)。
CMP　記憶と加算器を比較(CoMPare memory and accumulator)。
CPX　記憶とXレジスタを比較(ComPare memory and index X)。
CPY　記憶とYレジスタを比較(ComPare memory and index Y)。
DEC　記憶を1減算(DECrement memory by one)。
DEX　レジスタSを1減算(DEcrement register S by one)。
DEY　レジスタYを1減算(DEcrement register Y by one)。
EOR　記憶と加算器のXORを取る(Exclusive OR memory with accumulator)。
INC　記憶に1加算(INCrement memory by one)。
INX　レジスタXに1加算(INcrement register X by one)。
INY　レジスタYに1加算(INcrement register Y by one)。
JMP　新しい記憶位置に跳ぶ(JuMP to new memory location)。
JSR　新しいサブルーチンに跳ぶ(Jump to SubRoutine)。
LDA　記憶から加算器に読込み(LoaD Accumulator from memory)。
LDX　記憶からXレジスタに読込み(LoaD X register from memory)。
LDY　記憶からYレジスタに読込み(LoaD Y register from memory)。
LSR　論理的1ビット右にずらし(Logical Shift Right one bit)。
NOP　無操作(No Operation)。
ORA　記憶と加算器のORを取る(OR memory with Accumulator)。
PHA　スタックに加算器のものを押し出す(PusH Accumulator on stack)。
PHP　スタックにプロセッサ状態を押し出す(PusH Processor status on stack)。
PLA　スタックから加算器に取り込み(PuLl Accumulator from stack)。
PLP　スタックからプロセッサ状態に取り込み(PuLl Processor status from stack)。
ROL　1ビット左にずらしてビット列を回します(ROtate Left one bit)。
ROR　1ビット右にずらしてビット列を回します(ROtate Right one bit)。
RTI　インタラプトから復帰(ReTurn from Interrupt)。

RTS　サブルーチンから復帰(ReTurn from Subroutine)。
SBC　キャリーで減算(SuBtract with Carry)。
SEC　キャリーフラグのセット(SEt Carry flag)。
SED　十進モードのセット(SEt Decimal mode)。
SEI　インタラプトディセイブルフラグのセット(SEt Interrupt disable flag)。
STA　加算機の内容を記憶に保全(STore Accumulator in memory)。
STX　X レジスタの内容を記憶に保全(STore Register X in memory)。
STY　Y レジスタの内容を記憶に保全(STore Register Y in memory)。
TAX　加算機の内容をレジスタ X に転送(Transfer Accumulator to register X)。
TAY　加算機の内容をレジスタ Y に転送(Transfer Accumulator to register Y)。
TSX　スタックポインタの内容を X レジスタに転送(Transfer Stack pointer to register X)。
TXA　X レジスタを加算器に転送(Transfer register X to Accumulator)。
TXS　X レジスタをスタックポインタに転送(Transfer register X to Stack pointer)。
TYA　Y レジスタを加算器に転送(Transfer register Y to Accumulator)。

次は 6500 ファミリーの命令集合を使って書かれた小さなプログラムです。プログラムの目的は 16 進数の番地 0200 から始まる記憶に置かれた 10 個の数の和を計算する事です。

ここで、記憶の中の 10 個の数(これらは青で表示しています)が記憶位置にあり、そこに総和が保管されます(これは赤字で表示されています)。ここで 0200 は記憶の中の最初の数の番地です。

```
0200  01 02 03 04 05 06 07 08 - 09 0A 00 00 00 00 00 00  ................
```

これらの 10 個の数の和を計算するプログラムです：

```
0250  A2 00     LDX #00h
0252  A9 00     LDA #00h
0254  18        CLC
0255  7D 00 02  ADC 0200h,X
0258  E8        INX
0259  E0 0A     CPX #0Ah
025B  D0 F8     BNE 0255h
025D  8D 0A 02  STA 020Ah
0260  00        BRK
...
```

このプログラムが実行されたあとに、それが計算した和は記憶に蓄えられます。和は 16 進数で 37(十進数なら 55)になり、ここでは赤で示しておきます：

```
0200  01 02 03 04 05 06 07 08 - 09 0A 37 00 00 00 00 00  ..........7.....
```

もちろん、貴方にこのアセンブリ言語のプログラムがどの様に働くかを理解することを期待していません。それを貴方に示した目的は、この様なマイクロプロセッサーの命令集合を使ったプログラムがどの様なものであるかを貴方に見て頂く為です。

低水準言語と高水準言語

プログラマはその命令集合の命令を使って計算機のプログラムが出来ると入っても、これはうんざりする作業です。初期の計算機プログラマはマイクロプロセッサが理解できる機械語よりも自然言語、例えば英語にもっと似た言語でプログラムを開発することを望んでいました。機械語は**低水準言語**として考えられています。何故なら、それはマイクロプロセッサの回路で簡単に実行できる様に単純に設計されているからです。

プログラマはそこでプログラムで彼らが使いたい**高水準言語**を低水準言語を使って作り上げる事を思いました。こうして、FORTRAN（1957年)、ALGOL（1958年)、LISP（1959年)、COBOL（1960年)、BASIC（1964年）とC（1972年)か作られました。不幸なことに、マイクロプロセッサは機械語しか理解できないので、高水準言語で記述された全てのプログラムはマイクロプロセッサで実行出来る様に機械語に変換しなければなりません。

与えられたプログラミング言語向けにプログラムでどの様に入力するかを指図する規則のことを**構文規則**と呼びます。もしも、プログラマがプログラムの記述時に言語の構文規則に従わなければ、ソースコードを機械語に変換するソフトウエアが混乱し、所謂、構文エラーを返すことになります。

構文エラーがどの様な物であるかを示す例として、'印刷する'と云う言葉を考えてください。もし、'印刷する'という言葉が所与のプログラム言語の命令であったとしましょう。そこでプログラマーが'印刷する'の代わりに'印刷すろ'と打ち込んでしまうと、これが構文エラーになるわけです。

12.8 コンパイラとインタプリタ

高水準言語から機械語に変換する為に用いられるプログラムに通常二種類あります。最初の種類が**コンパイラ**と呼ばれ、高水準言語の**ソースコード**(通常は入力された書式です)をその入力として機械語に変換します。ソースコードと同値な機械語が生成されたのちに、それが計算機の記憶に読込まれて実行されます。コンパイルされたプログラムはまた記憶装置にも保存可能で、そうすることで必要な時なら何時でも計算機の記憶に読込む事が出来ます。

第二の機械語へ高水準言語を変換することに通常用いられる第二のプログラムの種類は**インタプリタ**と呼ばれます。コンパイラがする様なソースコードを機械語に変換する代わりに、インタプリタはソースコードを読み取り(通常は一度に一行)、ソースコードの一行で遂行すべきとされる処理を決定し、それからこれらの処理を実行します。そして、その下にあるソースコードの次の行を見つけると、プログラムのこの次の行で望まれる事の処理を決定すると、これらの処理を実行し、以下同様となります。

幾千もの計算言語が１９４０年代から生成されていますが、２から3百程度の歴史的に重要な言語が現在あります。ここで歴史的に重要な計算機言語の一覧を掲げるウエブサイトのリンクがあります：http://en.wikipedia.org/wiki/Timeline_of_programming_languages

12.9 アルゴリズム

計算機プログラマは少なくとも一つのプログラム言語は知っている必要がありますが、プログラム間が問題を解く時には、より抽象的な計算機言語よりも抽象性ではより高度な水準でそれを遂行します。

問題が解かれたのちには、その解がプログラム言語として展開されているのです。それはプログラマがあたかも二人の人であるかのようです。最初の人間は**問題を解く人**であり、第二の**人はプログラム作成者**です。

単純な問題では、多くのプログラマが心の中でアルゴリズムを生成し、それらのアルゴリズムを直接プログラム言語に展開します。この過程での問題を解く人とプログラム作成者との間を切り替えています。

複雑なプログラムになると、流石に、この問題を解く局面とプログラム作成の局面がより分離します。与えられた問題を解くアルゴリズムはプログラム言語よりも意味を用いて開発され、それは文書の中に記録されます。この文書は問題を解く人からプログラム言語への展開を行うプログラム作成者への橋渡しとなっています。

ある問題に対して問題を解く人が最初に行うことは**分析**です。これは殊に重要な段階です。というのも、問題を分析しなければ、適切に解く事が出来ないからです。あるものを**分析**するということは、構成部品にバラバラに分解して、これらの部分についてそれ等がどのようにして動くかを定める為に研究する事を意味します。有名な文句は'**分解して征服せよ**'で、だから、難しい問題を分析する時には問題全体を解くよりも、それぞれをより単純にして小さな問題にバラバラにしてしまいます。**問題を解く人**は、より単純な問題をそれぞれ解くアルゴリズムを開発し、それから、これらの**アルゴリズム**を結合して、問題全体に対する回を構成するのです。

アルゴリズム(「ある-ご-りずむ」と発音します)は命令の列であり、この命令は与えられた作業をどのように達成するかを記述したものです。これらの命令は(英語の様な)自然言語による記述、それらの図式の描画、プログラム言語の記述を含めた様々な方法で 表現されています。

アルゴリズムの概念は数学者が数学の問題、例えば、二つの数の和やそれ等の積と云った様なものを解く為に開発した様々な手続に由来します。

アルゴリズムはより一般的な問題にも使う事が可能です。例えば、紙モデルを使った車庫の大きさの問題を解きたいと考えている人にとって、次のアルゴリズムに従うことが出来るものです：

1) 車庫に据え付ける小物の長さと幅を計って、これらの計測量を記録します。

2) 第一からの計測量を 100 で割って、それから、元の小物のモデルとして使えるような大きさに合致する紙片を切り出す。

3) 紙片を切り出します。ここで紙片は長さを一番大きな車のモデルの長さの 1.5 倍、幅を 3 倍とし、車庫の床モデルとします。

4)車庫床のモデルに車庫の扉を配置し、鉛筆を使ってその位置に印を付け、その紙の区画と鉛筆の印の間にちょうど入るように車庫の床モデルの上に両方の車を置きます。

5)　紙の床もデルの上の車が占有していない空いている場所に小物を置きます。

6)小物のモデルを、それら全てが、この大きさの車庫の中で適合する様に、この空いた場所でいろいろな位置に動かしてみます。

7)受け入れられるものであれば段階 10 に行く。

8)車庫が窮屈だったら、車庫モデルの床の長さ、幅(あるいはその両方)を 10%増やして新しい床モデルを作って段階 4 に行く。

9)もし、車庫に余裕が有り過ぎれば、車庫の長さ、幅(あるいはその双方)を 10%減らして、新しい車庫の床モデルを生成し、段階 4 に行きます。

10)車庫の床モデルの長さと幅を計測し、これらを本来の大きさに拡大して戻し、これらの大きさの車庫を建てます。

この例で見る事が出来る様に、アルゴリズムは往々にして、かなりの数の段階を含みます。何故なら、望ましい解を導出するに十分な程、詳細でなければならないからです。これらの段階を発展させて文書に記録したあとでやっと、その与えられた問題を解かなければならない人々が、幾度も、その手順に従うことになるのです。

12.1 計算

人間がどのようにしてアルゴリズムの各段階に従うかを理解することはわりと容易な事ですが、計算機のマイクロプロセッサが単純な機械言語の命令だけしか処理できない時に、これらの段階をどのように計算機が処理をするのかを理解することはより難しいことです。

マイクロプロセッサがアルゴリズムの各段階をどのようにして処理が出来ているかを理解する為には、**計算**(これは**演算**としても知られています)とは何であるかを最初に理解しなければなりません。インターネット上でこれらの言葉のよい定義を探し、どう言っているかを読んでみましょう。

計算(computation)と言う言葉には二つの定義があります：

1) 固定された規則に従った数や記号の操作。通常、算術的電子計算機の操作に対して使われるが、心や脳で処理されたある種の過程の実行にも使われる。
(www.informatics.susx.ac.uk/books/computers-and-thought/gloss/nodel.html)

2) 計算は計算機と呼ばれる閉じた物理系内部で生じる純粋な物理的現象として観察できる。このような物理系の例としては、デジタル計算機、量子計算機、DNA 計算機、分子計算機、アナログ計算機や、wetware 計算機がある(www.informatics.susx.ac.uk/books/computers-and-thought/gloss/nodel.html)

これらの二つの定義は**”計算が決まった規則に基づいた数や記号の操作”**であり、**”計算機と呼ばれる閉じた物理的系内部での物理的現象として観察される”**ということを指しています。双方の定義は、我々が通常計算機と呼んでいる機械が**計算機という一つの種族**であり、閉じた物理系の他の種族もまた計算機として動作することが出来ることを意味しています。その他の計算機というものにはDNA 計算機、分子計算機、アナログ計算機やwetware 計算機(すなわち、脳)が含まれます。

演算(calculation)の次の二つの定義は、通常の計算機、脳や他の種類の計算機を統べるものに光を当てるものです：

1)演算は一つ、あるいはそれ以上の入力を一つそれ以上の結果に変換する内的な手続である。
(en.wikipedia.org/wiki/Calculation)

2)演算：計算する手続き；数学的、あるいは論理的手法によって何かを決定すること
(wordnet.princeton.edu/perl/webwn)

演算に関するこれらの定義によると、”**一つ、あるいはそれ以上の入力を一つ、あるいはそれ以上の結果に変換する内的な手続**”であり、これは”**数学的、あるいは論理的**”に実行されるということを指しています。我々は脳が計算を実行する為に、どの様な数学的、論理的なものを用いるかをまだ完全に理解していませんが、この領域は急激に展開されている状態です。

演算の第二の定義では**論理的**という言葉を使っており、話を進める前に、この言葉の定義が必要です：

論理系は規則の全体を構成するもので、その体系の中での任意の意味付に用いられなければなりません。数学の大半が良く理解された規則の構造上を根底とし、高度に論理的なものとして考えられています。任意の論理が適用される以前に、どの規則が用いられるかを述べたり、理解しておく必要が常にあります(ddi.cs.uni-potsdam.de/Lehre/TuringLectures/MathNotions.htm)

意味付(Reasoning)はその体系での一つの点から別の点へ動かす為の予め定義された規則を用いる手続の事です。例えば、人が紙片の二つの数を足し合わせる時に、それらは正しい和を得る為の和のアルゴリズムの規則に従わなければなりません。和のアルゴリズム規則がその**論理**であり、誰かが演算の間に、これらの規則を適用する時、これらは規則によって**意味付**けられています。 ここで、計算機がアルゴリズムの段階を、そのマイクロプロセッサが単純な機械語命令が実行出来るだけの時に、どの様にして実行するかという疑問に対して、これらの概念を適応してみましょう。

人がアルゴリズムを開発するとき、アルゴリズムの段階は通常、各作業を実行する為に必要なより小さな段階の全てを含まない、高水準の作業として述べられています。

例えば、ある人が”ニューヨークからサンフランシスコ迄の運転”という手引を書こうとします。この大きな手引は”その交差点を左に曲がり、西に10キロ進め等”の手順を含むより小さな手引に落とせます。大きな手引の全ての小さな手引が完成されれば、大きな手引も完成されます。

この大きな運転の手引を使わなければならない人は通常、それを完遂する為に遂行しなければならない小さな手引を思い描く事が出来なければなりません。計算機は非常におバカなので、任意のアルゴリズムは計算機上で実行出来る以前に、アルゴリズムの段階がより小さな段階に分けられて、それらの小さな段階がまたさらに小さな段階に分けられて、それらの段階がマイクロプロセッサの命令集合で実行されるのに十分に小さなものになっていなければなりません。

ある場合は、いくつかの小さな段階が大きな段階を実装する上で必要なだけかもしれませんが、一方では数百、数千のより小さな段階が必要とされます。数百、数千の小さな段階は、アルゴリズムが機械語に変換される時に数百、数膳の機械語命令に変換されます。

機械語は計算機でプログラム出来る唯一の言語であれば、人間によって計算機に送り込まれるアルゴリズムの殆どが大き過ぎるものになるでしょう。高水準言語で記述されたアルゴリズムはしかしながら、機械語で必要とされるより小さくて沢山の段階に分割する必要がありません。

高水準言語で実装されるアルゴリズムを分解するという大変な作業は自律的にコンパイラやインタプリタで実行されます。これが時間の大半をプログラマが機械語の代わりに高水準言語を使って開発する理由です。

12.2 アルゴリズムの記録に図式が使える

前に、英語の様な自然言語でアルゴリズムは記録できるだけではないと述べましたが、図式を使って記録する事も出来ます。貴方は、殆ど図式に基づく言語が創られていて、プログラムが用いているアルゴリズムを含めて、'問題を解く人'がプログラムを設計出来ることを学べば驚くかもしれません。この言語は**UML**と呼ばれ、**Unified Modeling Language(統合モデル化言語)**の事です。UML図式の一つは**活動図式(activity diagram)**と呼ばれ、幾つかの論理の組の段階(即ち、活動(activity))の列を示すことが出来ます。次の例は、アルゴリズムが活動図式でどの様に表現されるかを示すものです。

12.3 1から１０までの数の総和を計算

分析や求解の前に問題で実行されなければならない第一のことは、明確に、そして明瞭に問題を記述することです。ここではアルゴリズムを使って求解する問題の短い陳述を挙げておきます：

陳述： この問題では、1から10の間に含まれる数の総和を決定することが必要とされる。

2952　これはかなり単純な問題なので、分析にたっぷり時間を使う必要はありません。 図 12.7 に
2953　活動図式に置き換えた、この問題を解くアルゴリズムを示しています。

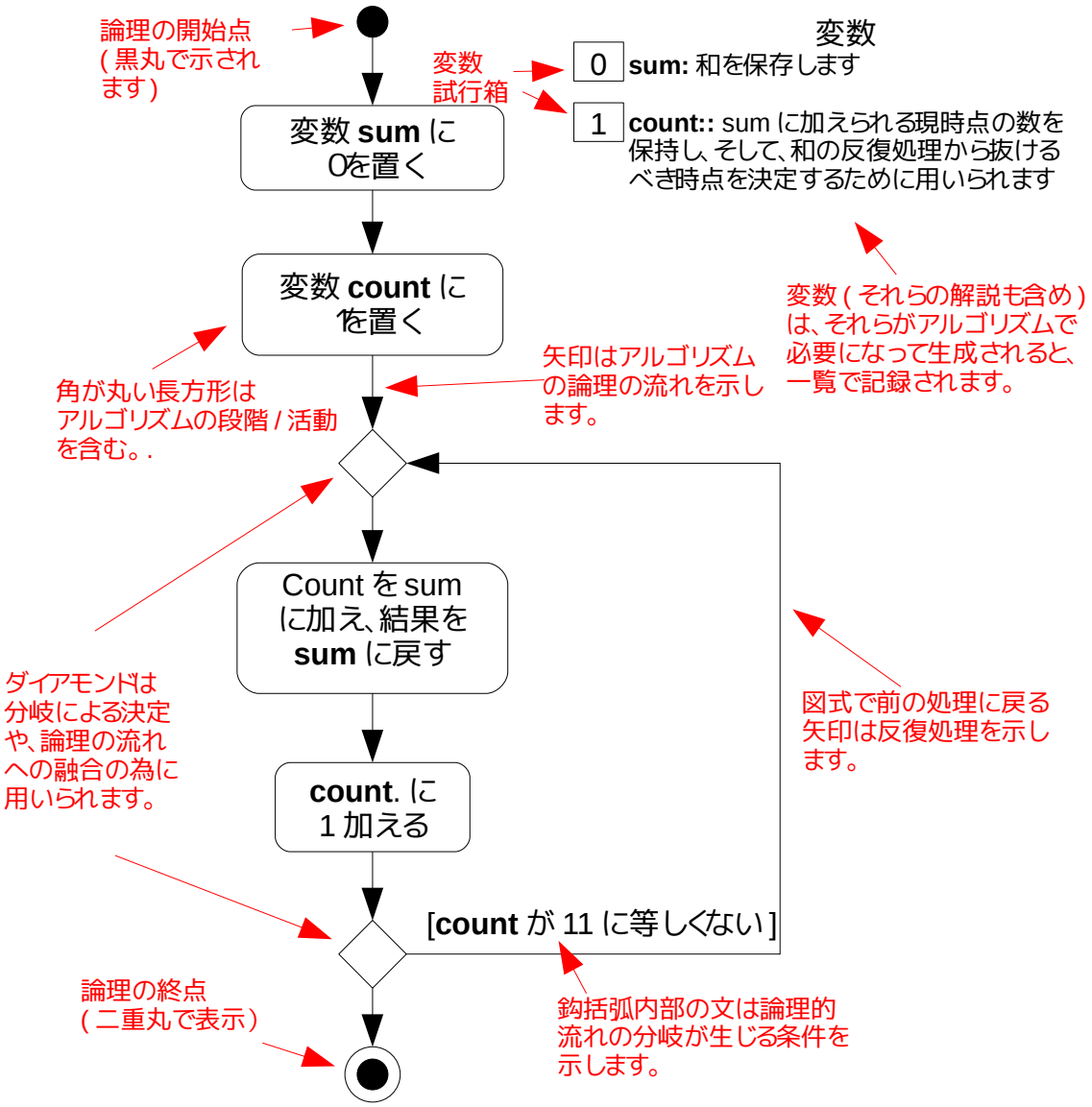


図 *12.7:* あるアルゴリズムの活動図式

2954　アルゴリズムと活動図式は同時に開発されます。開発途上、変数は必要に応じて生成され、
2955　それらの名前は通常、それらの陳述に沿ったリストに記録されます。開発者は定期的に出発
2956　点から開始して、それが正しい事を確認する為に論理的に一通り試してみます。
2957　試行箱は各変数の隣に置かれて、論理が変数をどのように変更するかを記録と更新ができる
2958　ようになっています。一通り試している間に、誤りが見つかると、これらは流れの矢印を動
2959　かして、活動長方形の内部の文言を調整して修正しなければなりません。

論理で間違いがないことを見つけると、開発者は**問題を解く人**でありつづける事を止めて、アルゴリズムをプログラミング言語でプログラムするプログラマに引き渡す事が可能になります。

12.4 数学計算系の数学の部分

数学は”並びの科学”とし記述されます。ここで並びの定義ですが：

1）体系的整理...
（http://www.answers.com/topic/pattern）

そして、系の定義は：

1）相互作用し、相互に関連し、あるいは互いに依存しあい、複雑な全体を構成する要素の一群。

2）相互の関連する概念や原理の体系化された集合。
（http://www.answers.com/topic/system）

それ故に、数学は物理的と非物理的対象の組織的な属性を扱う科学として考えられます。数学は非常に強力なので、全ての物理的、及び非物理的対象は組織的な属性を有し、それゆえに数学は、理解されて操作され得るこれらの対象によって、ひとつの平均値となります。数学を知れば知るほど物理的世界に対して制御できるようになります。このことが数学をもっとも有用なものの一つで人間が得ることの出来る知識の中で興味深い領域にしているのです。

伝統的に、数学は、人手の計算による数学に必要な、ワクワクすることでもなければ複雑なアルゴリズムを散々学ぶことを要求しました。典型的な数学の教科書の50%以上の内容が手書に基づくアルゴリズムの教示に割かれ、その上、教科書を通じた勉強では、より高い割合でこれらのアルゴリズムを人手で操作することに費やされています。

ほとんどの人にとって、退屈な学習や実習、複雑な手計算アルゴリズムはあまりにも複雑で、心を踊らす事のない退屈なことで、決して数学のとてつもなく面白く、強力で、美しい側面を見る機会を持つことがないのです。

悪いことに、手書中心の計算アルゴリズムは常にワクワクするものでもなく、複雑で退屈なものです。良いことは数学計算環境の発見によって人々が手書中心のアルゴリズムを使う必要性が目立って減少した事です。

13 SAGE サーバーの立ち上げ

前節で示した様に、大半の人々が最初に SAGE をウエブサービスとして使い、この本の冒頭で読者は既に SAGE サーバーに接続したと思っても良いでしょう。

この節は自分自身の SAGE サーバが欲しい人向けのもので、Windows や Linux 上での、入手、インストール、設定と保守を範囲とします。SAGE ノートブックサーバはインターネット技術に基づくので、この節はこれらの技術の幾つかを包含することで始めましょう。SAGE の構成の高水準な俯瞰は SAGE 配布ファイルに含まれている議論でつづけられるでしょう。最後に、Linux と Windows を基盤とする SAGE サーバー双方の立ち上げも包含します。

13.1 インターネットに基づく技術への入門

インターネットは現在、我々の文明の最重要の技術の一つであり、その重要性は今後増大する一方でしょう。実際、インターネットは非常に迅速に拡大しており、次の資料では、やがて殆ど全ての計算機器がそれに繋がる事を示しています (https://embeddedjava.dev.java.net/resources/waves_of_the_internet_telemetry.pdf) 。

だから、インターネットに関連する技術がどの様に働くのかを理解する事は計算機を使って仕事をすることに興味を持つ人々にとって価値のある事です。

インターネットがどの様に創造されたかという歴史を理解する事も価値がありますが、我々はここで、この歴史を議論をするつもりはありません。何故なら、よそできちんと文書としてまとめられているからです。私は貴方がインターネットの歴史についてインターネットの検索を使って、貴方が見つけた記事を読む事を高く勧めます。そうする事が貴方の時間の優れた投資となる事を私が保証します。

13.1.1 複数の計算機はどの様にして互いに通信するの?

二つの計算機だけが互いに通信する必要がある時、状況は単純です。何故なら、必要とされる全ての事は通信媒体（電線、光ファイバーケーブルや無線信号といった代物）で互いをつなげる事です。一方の計算機から出発した情報はもう一つの計算機に送り込まれる云々です。しかし、複数の計算機が互いに通信しなければならない状況であればどうでしょうか？この問題を解く方法は沢山ありますが、最も一般的な方法が図１１で示す方法です:

図 11 には、複数の計算機が所謂、**局所ネットワーク**、即ち、**LAN** と呼ばれるもので繋がっている様子を示しています。LAN は互いに物理的に閉じた複数の計算機（通常は同じ部屋や同じ建物の中）で構成され、通信媒介として用いられる物で互いに結合されています。図

13.1 では計算機は銅線のイサーケーブルを使って switch と呼ばれる機器に繋げられています。

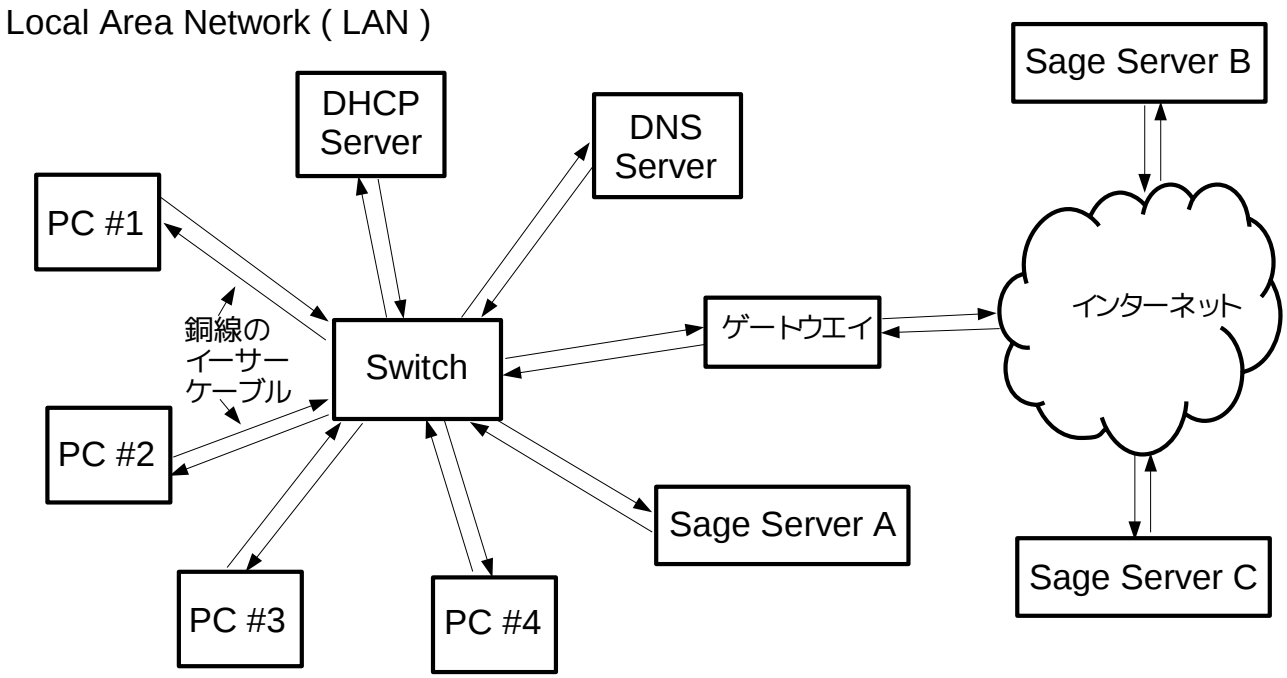


図 *13.1:* 局所ネットワーク*(LAN)*

ネットワーク上の計算機は**伝言**を用いて互いに通信しますが、伝言を送る事は文通の手紙に似ています。**switch** の目的はそれに送り込まれる各伝言を監視し、その伝言が送り込まれるべき計算機を決定し、それからその伝言を当の計算機に送り込みます。

図 11 のモデルには問題があります。というのは、このネットワーク上の各計算機に結びつけられた名前がそれらをユニークに同定することに、計算機の数が数百、数千に増加してしまえば不向きになるからです。これはさておき、絵の右側の雲はインターネットとそれに接続されている数百万の計算機(これらはホストとも呼ばれます)を表現しています。インターネット上の各計算機がある方法でユニークに同定されれば、伝言はまたこれらの計算機に送られたり、これらから受け取られたりする事が出来ます。これはさておき、伝言がどの様にして交換されるべきかという規則も存在していなければなりません。

13.1.2 TCP/IP プロトコルについて

インターネットが出現する以前、二つの解決すべき問題は 1)各計算機をユニークに同定する必要があったことと 2)どのようにメッセージが交換されるかを取り決める通信規則(**プロトコル**とも呼ばれます)を発達させる必要でした。インターネットのお陰で、**プロトコル**は

「系の間の通信の為にキチンとした書式を定義する規則の集合」(www.unitedyellowpages.com/internet/terminology.html)として定義することができます。プロトコルの数は互いに用いられるときに、それらは**プロトコル集**(protocol suite)と呼ばれます

インターネット向けに発達したプロトコル集は TCP/IP と呼ばれ、その名前は、この集まりの中で最も煩雑に用いられる二つの名前の組み合わせです(TCP は**転送制御プロトコル** (Transmission Control Protocol)で、IP は Internet Protocol によります)。**インターネットプロトコル**は番地系を用いてインターネット上の計算機をユニークに同定する方法を定義しています。 IP version 4 (**Ipv4**)、これは現在もっとも広く用いられている IP プロトコルの版で、**ドットで区切った 0 から 255 の間の 4 個の数字**で構成されています。IP アドレスの例には 207.21.94.50、 54.3.59.2 と 204.74.99.100 があります。0.0.0.0 から 255.255.255.255 までも全ての IPv4 アドレスが番地空間を生成し、 ここには 4,294,967,296 番地あります。

IP version 6 (**IPv6**) は IP プロトコルの最新版で、その番地空間が保有する番地は 340,282,366,920,938,463,463,374,607,431,768,211,456 番地です！ IPv4 から IPv5 へのいこうは始まっていますが、ゆっくりとしています。 大半のインターネット上のホストは長い間 IPv4 を使い続けており、それ故に、IPv4 はこの文書でも使われています。

図 13.2 には 図 13.1 で示されたネットワークと同じモデルを含んでいますが、各計算機は

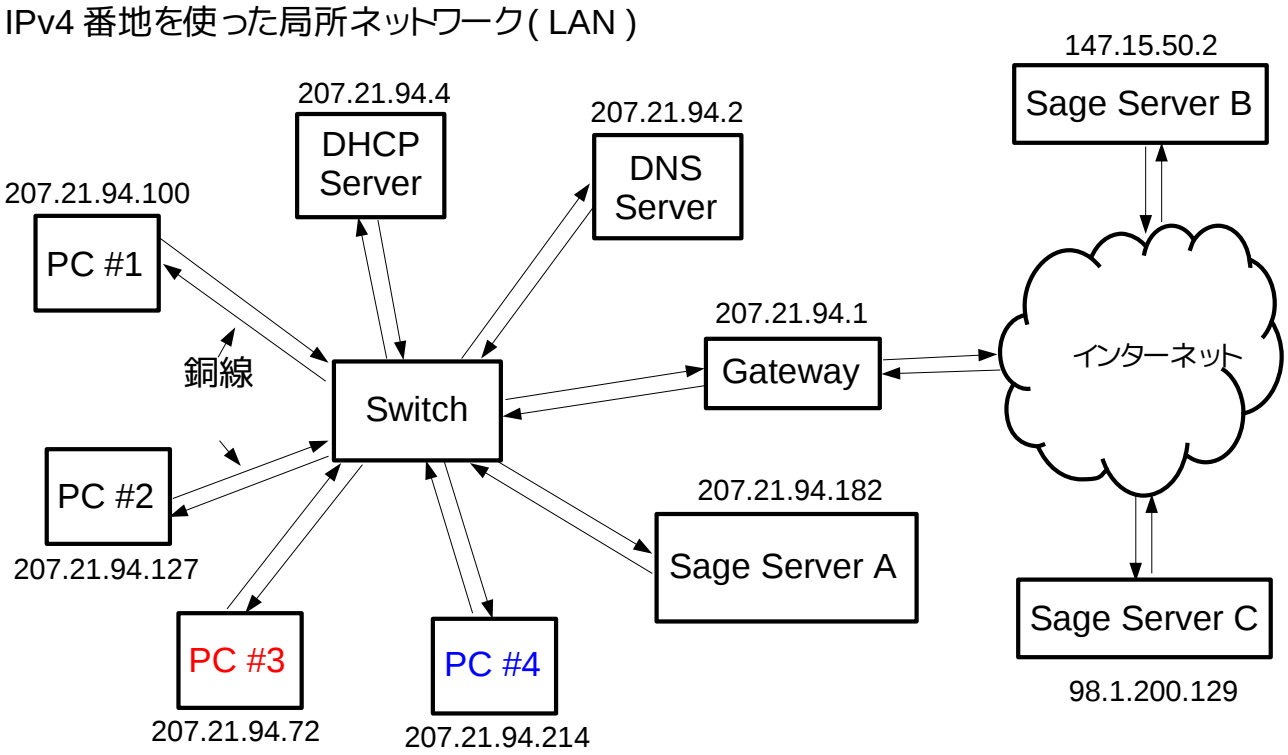


図 *13.2: IP* アドレス

IPv4 番地が割り当てられています：

もし、PC #3 が PC #4 に伝言を送る必要があれば、PC #4 の IP アドレス(これは 207.21.94.214 です)に宛てて伝言を送り込みます。送り手の IP アドレス(207.21.94.72)もまた、PC #4 が返事を送る必要があった場合に備えて、伝言の中に置かれます(これは返送先の番地を手紙に添えておくことと同様です)。PC #3 はそれから switch に伝言を送り、switch は伝言の送り先の番地を探し、それから PC #4 に伝言を引き渡します。

もし、この局所ネットワーク上の一つの計算機が、その LAN 上にない計算機に伝言を送る必要があれば、その伝言は**ゲートウエイ**計算機に送られて、そのゲートウエイはそれからその伝言をインターネットに流します。

13.1.3 クライアントとサーバー

LAN 上やインターネット上で、組織化された計算機同士が通信する幾つかの方法があり、そして、それらの組織の事を構成(architecture)と呼びます。 一つの構成は**Peer-to-Peer**(P2P)とよばれ、ネットワーク上の計算機を互いに情報を交換する等価なものとして扱います。P2P アプリケーションの例はインスタントメッセージングです。

ネットワーク上の計算機を使った別の構成は**クライアント-サーバ**と呼ばれます。クライアント・サーバー構成では、サーバーはネットワーク上の他の計算機の要求を受け入れる計算機で、要求された仕事を果たし、請求者に向けて作業結果を返します。クライアントは計算機で、サーバーに要求を送って、応答を受け取って、それから応答に含まれている情報を利用します。

図 11 に示す LAN では、三個のサーバ(DHCP サーバー、DNS サーバと SAGE サーバー)と四個のクライアントがあります。DHCP と DNS サーバーは次の二つの節で議論しましょう。

13.1.4 DHCP

DHCP は**Dynamic Host Configuration Protocol**(動的ホスト設定プロトコル)の事で、その目的は LAN 上の計算機を立ち上げるときにネットワークの接続に必要な情報を使って自律的に設定させることです。 この情報には**IP アドレス**、**ゲートウエイのアドレス**に**DNS サーバーのアドレス**が含まれます。 IP アドレスやゲートウエイが何であるかを我々は既に議論しています。DBNS サーバーは次の節で解説することにしましょう。

IP アドレスを持ってもいない(何故なら立ち上げだから)計算機がどの様にして IP アドレスを取得する為に DHCP サーバーと接続する為にネットワークが使えるかという点に疑問を感じ

るかもしれません。この問題は立ち上げ持の計算機がLANに対して**broadcast**メッセージを流すことで解決されます。broadcastメッセージはLAN上の特定の計算機に送られるものではありません。その代わりに、broadcastメッセージはLANに対して送られて、LAN上の全ての計算機がその伝言を受け取ります。

DHCP要求伝言がLAN上に送り込まれると、DHCPサーバーは他の計算機と同時に要求を受け取ります。その他の計算機は伝言の内容を読み、そのDHCP要求を含む伝言を見て、それを無視します。DHCPサーバーは、伝言の内容を読んで、その伝言がそれに対して意味することを理解すると、DHCP設定情報を送り主に送り返すのです。

13.1.5 DNS

インターネット上の数百万もの計算機の各々がそれらのIPアドレスを使ってアクセスされます。例えば、sagemath.orgウエブサイトを包含するサーバーのIPアドレスは**12.208.160.192**です。ウエブブラウザを立ち上げて、**http://128.208.160.192/sage**と入力すれば、貴方は直接このウエブサイトに接続できます。

莫大な数を覚えることは人間にとっては困難な事ですが、しかしながら、**IPアドレス番号を使って関連付けた名前で構成された系**がインターネット向けに生成されています。その系の名前が**DNS**で、**Domain Name System**のことです。一つ以上のIPアドレスに対応付けられた名前は**ドメイン名(domain name)**と呼ばれ、頭に与えられた計算機の**ホスト名**(とお尻にピリオド)があるドメイン名のことを **fully qualified ドメイン名**と呼びます。ドメイン名の例としては：

```
gentoo.org
yahoo.com
sourceforge.net
google.com
sagemath.org
wikipedia.com
```

fully qualifiedドメイン名の例は:

```
kiwi.gentoo.org.
loon.gentoo.org.
wren.gentoo.org.
```

DNSはインターネット全体に渡って分散した膨大なデータベースとして実装されています。

ドメイン名は、それらを DNS 系に入れる前に、**ドメイン名登録**機関で登録しておく必要があります。ドメイン名登録会社の例には godaddy.com、 networksolutions.com と register.com があります。

図 12 の LAN 上の DNS サーバーは三つの機能を持っています。最初の機能はドメイン名を含むクライアントからのメッセージを受理し、それらの名前に対応する IP アドレスを返すことです。利用者が sagemath.org の様なドメイン名をブラウザの URL バーに入力したときに、ブラウザはまだ SAGE ウエブサイトと接触出来ません。というのも、その IP アドレスを知らないからです。ブラウザが動作しているオペレーティングシステムは、そのために、ドメイン名を(DHCP を通じて得た DNS サーバーの IP アドレスを使って)DNS サーバーに送りつけます。すると、DNS サーバーは sage.org ドメイン名に関連する一つ以上の IP アドレスを返します。そこで、システムは SAGE サーバーが乗っているサーバーと背食するために、これらの IP アドレスを使います。

局所 DNS サーバが持つ第二の機能は局所ネットワーク上の計算機に対応する **IP アドレス**に対して**ドメイン名**を定義することです。もし、インターネット上で遠くにある計算機がその局所ネットワーク上のある計算機の IP アドレスを知りたがっているものの、DNS サーバがその対応を知らないとします。遠隔 DNS サーバが局所**権威(authotative)**DNS サーバに接触して、その対応がどうなっているかを尋ねます。そして、遠隔 DBS サーバは一定期間、今後、その遠隔ネットワーク上の計算機が対応関係を知る必要がある場合に備えて、この対応関係を覚えておきます。

第三の DNS サーバが持つ機能は **IP アドレス**を含む伝言を取得し、これらのアドレスに対応する**ドメイン名**に返却することです。

13.1.6 プロセスとポート

ここで、インターネットに関連するより重要な幾つかの技術について議論してきましたが、IP メッセージ(今後、メッセージとして参照します)が計算機に届いたときに何が生じ、ある計算機から送られたメッセージの前に何がメッセージを生成するかを話す時です。

殆ど全ての個人向けの計算機では同時に計算喜寿尾で複数のプログラムを動かせます。ここでは典型的な利用者の計算機上で同時に動作しているプログラムのリストを挙げておきましょう：

- ウエブブラウザ
- インスタントメッセージクライアント
- ワープロ

- ファイルダウンロードユーティリティ
- 音楽ファイルプレイヤ
- ゲーム

殆どの計算機のオペレーティングシステムで、動作しているプログラムは**プロセス**と呼ばれます。Windows では現時点で動作している全てのプロセスの一覧は<ctrl><alt>と<delete>キーを同時に押すことで起動するタスクマネージャーで見られます。 Linux の様な UNIX 系のシステムでは、動作しているプロセスの一覧は ps -e 命令を実行することで得られます。ここで、私が使っている Linux システム上で動作しているプロセスの一覧を示しておきましょう：

```
manage@sage:~$ ps -e
  PID TTY          TIME CMD
    1 ?        00:00:00 init
    2 ?        00:00:00 ksoftirqd/0
    3 ?        00:00:00 watchdog/0
    4 ?        00:00:00 events/0
    5 ?        00:00:00 khelper
    6 ?        00:00:00 kthread
    8 ?        00:00:00 kblockd/0
    9 ?        00:00:00 kacpid
   10 ?        00:00:00 kacpi_notify
   67 ?        00:00:00 kseriod
  100 ?        00:00:00 pdflush
  101 ?        00:00:00 pdflush
  102 ?        00:00:00 kswapd0
  103 ?        00:00:00 aio/0
 1545 ?        00:00:00 scsi_eh_0
 1547 ?        00:00:00 scsi_eh_1
 1728 ?        00:00:02 kjournald
 1796 ?        00:00:00 logd
 1914 ?        00:00:01 udevd
 2611 ?        00:00:00 shpchpd
 2620 ?        00:00:00 kpsmoused
 3208 tty2     00:00:00 getty
 3209 tty3     00:00:00 getty
 3210 tty4     00:00:00 getty
 3211 tty5     00:00:00 getty
 3212 tty6     00:00:00 getty
 3263 ?        00:00:00 dd
 3265 ?        00:00:00 klogd
 3345 ?        00:00:14 vmware-guestd
 3381 ?        00:00:00 sshd
 3404 ?        00:00:00 atd
```

```
 3414 ?        00:00:00 cron
 3959 ?        00:00:00 dhclient3
 4140 tty1     00:00:00 login
 4141 tty1     00:00:00 bash
 4429 ?        00:00:00 syslogd
 4507 ?        00:00:00 sshd
 4508 pts/1    00:00:00 bash
 4538 tty1     00:00:00 sage
 4541 tty1     00:00:00 sage-sage
 4554 tty1     00:00:00 python
 4555 tty1     00:00:05 sage-ipython
 4573 tty1     00:00:00 sh
 4574 tty1     00:00:00 sage
 4580 tty1     00:00:00 sage-sage
 4591 tty1     00:00:02 python
 4592 pts/2    00:00:00 sage
 4600 pts/2    00:00:00 sage-sage
 4611 pts/2    00:00:06 python
 4611 pts/1    00:00:00 ps
```

もし、貴方がこのリストの下側を見れば, SAGE ノートブックサーバと一書に SAGE が動作している事が分かるでしょう。ps 命令はそれ自身を一覧に含めていますが、その一覧が生成された時点で、それが動作していたからだということに注意してください。

この一覧には四列あります。各プロセスには、プロセスが生成された時点で単一の**プロセス ID(PID)**番号が与えられ、これらの番号は PID 列に挙げられています。TTY 列はプロセスが端末に結びつけられているかどうかと、もしも繋がっていれば、どの端末に結び付いているかを示しています。TIME 列は CPU 時間をそのプロセスが時間、分と秒でどれだけ使ったかを示しています。

ネットワークから計算機にメッセージが届いたときに、計算機はどのプロセスがメッセージを受けとるかを判断しなければなりません。TCP/IP プロトコルはこの問題を解く手法で、ソフトウエアに基づく通信**ポート**を使います。

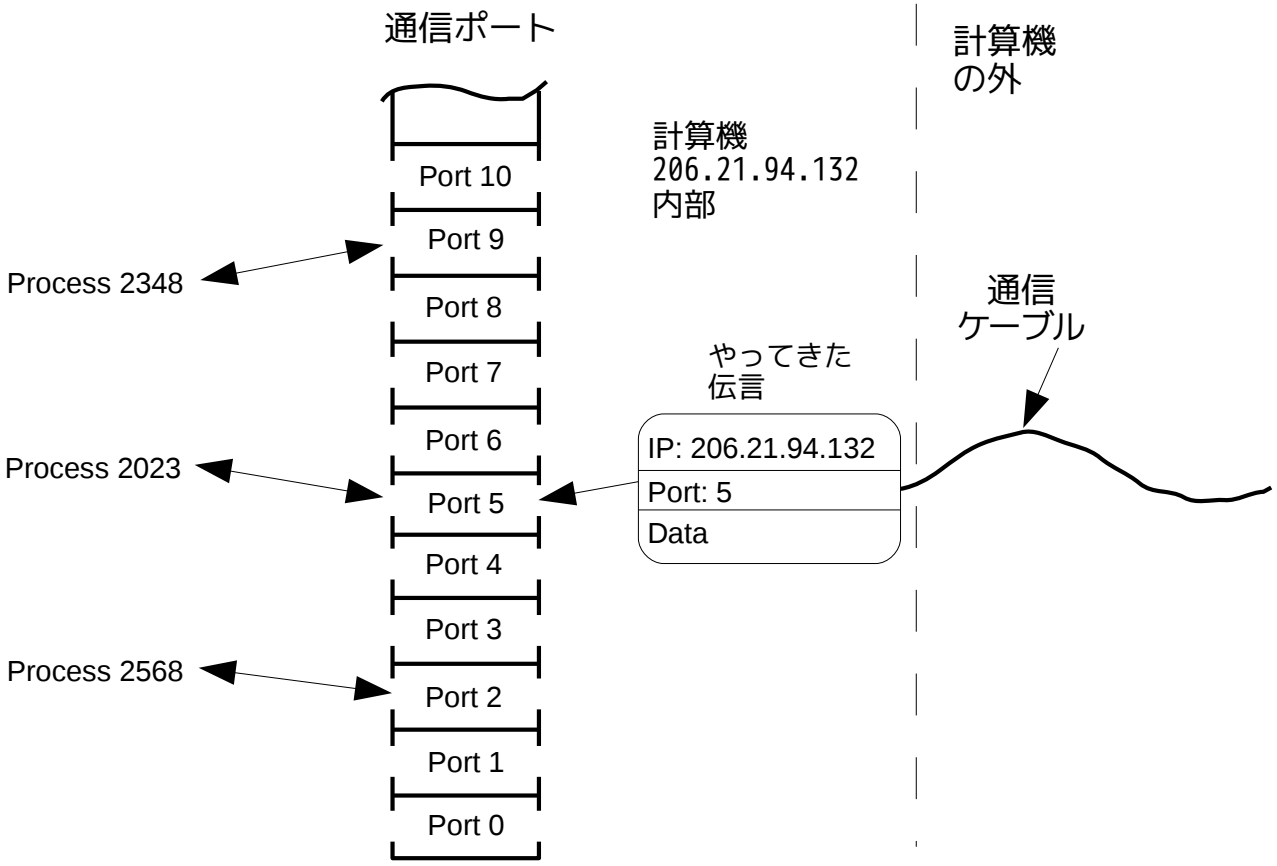


図 *13.3:* 通信ポート

図 13.3 では、ネットワークに繋がった IP アドレスが 206.21.94.132 の計算機の中と外を示しています。通信ポートはプロセス間に置かれ、動いているプロセスは左側、右側にはネットワーク接続があります。各ポートは最低のポート番号が 0 から最も高いポート番号が 65535 で単一の番号が与えられています。ネットワークから届く各伝言は、その中にポート番号を持っており、その番号には系がどのポートが伝言が送ったたかが分かる様になっています。

図 13.3 で、port 5 を行き先に持つ伝言がネットワークから到着しているので、系は port 5 にその伝言を送り込みます。process 2023 は系が port 5 に伝言を送り込んだ時点でそのポートに束縛され、process 2023 は伝言を受け取り、それからそれに含まれる情報を使って何かを実行します。

図 13.4 に IP アドレスが 65.2.8.3 を持つネットワーク上の別の計算機に送られた process 2023 からの伝言を示します。この伝言が目的の計算機に到達したときに、その伝言をその

ポート 8 に置けば、port 8 に束縛されたその計算機のプロセスがある筈です。

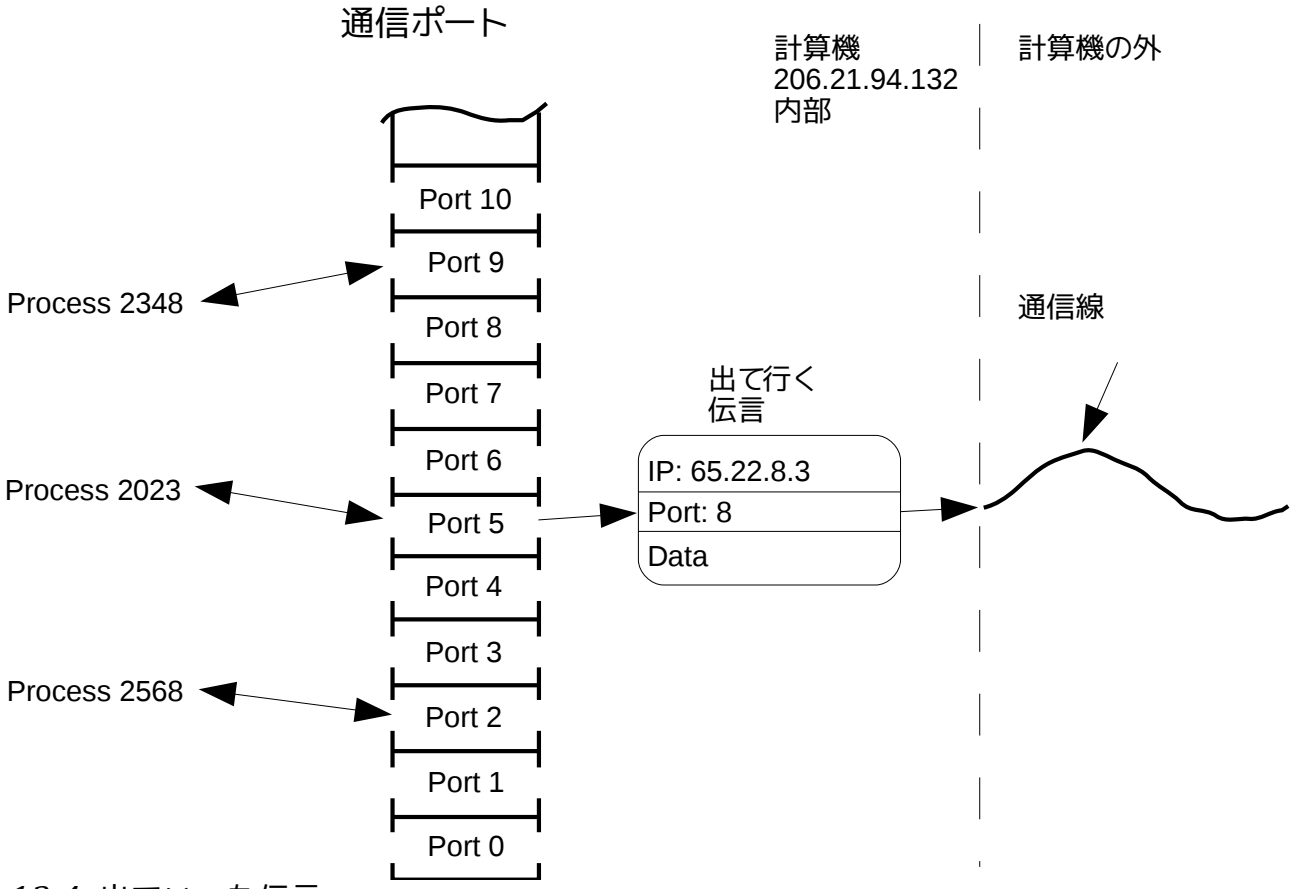


図 *13.4:* 出ていった伝言

13.1.7 良く知られたポート，登録されたポートと動的ポート，

ここで、貴方はポートが何であり、それらに拘束されたプロセスがどうするかが分かっていますが、人がどのプロセスがどのポートに束縛されるべきかをどうやって決定するか不思議に思っているかもしれません。IANA(Internet Assigned Numbers Authority)と呼ばれる組織が、インターネットに関連する様々な手法、その一つが TCP/IP ポート手法ですが、に関して責任を持っています。IANA は 0 - 635535 ポート範囲を次の三つの区画に分けています：

0 - 1023 -> 著名なポート

1024 - 49151 -> 登録されたポート

49152 - 65535 -> 動的、或いは個人的ポート

13.1.7.1 著名なポート (0 - 1023)

IANA が管理している一覧には、どの種類のプログラムが、この範囲内の特定のポート番号に通常、束縛されるかが示されています。例えば、**ウエブサーバ**は port 80 に束縛され、**SSH(secure shell)サーバ**は port 22 に束縛され、**FTP(ファイル転送プロトコル:File Transfer Protocol)サーバ**は**port 20** に束縛されて、**DNS サーバー**は**port 53** に束縛されています。ここで、最初の 25 の著名なポートを示しますが、一覧全体は http://www.iana.org/assignments/port-numbers から入手できます:

キーワード	十進数	概要	参照文献
	0/tcp	Reserved	
	0/udp	Reserved	
#		Jon Postel <postel@isi.edu>	
tcpmux	1/tcp	TCP Port Service Multiplexer	
tcpmux	1/udp	TCP Port Service Multiplexer	
#		Mark Lottor <MKL@nisc.sri.com>	
compressnet	2/tcp	Management Utility	
compressnet	2/udp	Management Utility	
compressnet	3/tcp	Compression Process	
compressnet	3/udp	Compression Process	
#		Bernie Volz <volz@cisco.com>	
#	4/tcp	Unassigned	
#	4/udp	Unassigned	
rje	5/tcp	Remote Job Entry	
rje	5/udp	Remote Job Entry	
#		Jon Postel <postel@isi.edu>	
#	6/tcp	Unassigned	
#	6/udp	Unassigned	
echo	7/tcp	Echo	
echo	7/udp	Echo	
#		Jon Postel <postel@isi.edu>	
#	8/tcp	Unassigned	
#	8/udp	Unassigned	
discard	9/tcp	Discard	

```
3260  discard           9/udp    Discard
3261  #                          Jon Postel <postel@isi.edu>
3262  discard           9/dccp   Discard SC:DISC
3263  #                          IETF dccp WG, Eddie Kohler <kohler@cs.ucla.edu>,
3264  [RFC4340]
3265  #                 10/tcp   Unassigned
3266  #                 10/udp   Unassigned
3267  systat            11/tcp   Active Users
3268  systat            11/udp   Active Users
3269  #                          Jon Postel <postel@isi.edu>
3270  #                 12/tcp   Unassigned
3271  #                 12/udp   Unassigned
3272  daytime           13/tcp   Daytime (RFC 867)
3273  daytime           13/udp   Daytime (RFC 867)
3274  #                          Jon Postel <postel@isi.edu>
3275  #                 14/tcp   Unassigned
3276  #                 14/udp   Unassigned
3277  #                 15/tcp   Unassigned [was netstat]
3278  #                 15/udp   Unassigned
3279  #                 16/tcp   Unassigned
3280  #                 16/udp   Unassigned
3281  qotd              17/tcp   Quote of the Day
3282  qotd              17/udp   Quote of the Day
3283  #                          Jon Postel <postel@isi.edu>
3284  msp               18/tcp   Message Send Protocol
3285  msp               18/udp   Message Send Protocol
3286  #                          Rina Nethaniel <---none--->
3287  chargen           19/tcp   Character Generator
3288  chargen           19/udp   Character Generator
3289  ftp-data          20/tcp   File Transfer [Default Data]
3290  ftp-data          20/udp   File Transfer [Default Data]
3291  ftp               21/tcp   File Transfer [Control]
3292  ftp               21/udp   File Transfer [Control]
3293  #                          Jon Postel <postel@isi.edu>
3294  ssh               22/tcp   SSH Remote Login Protocol
3295  ssh               22/udp   SSH Remote Login Protocol
3296  #                          Tatu Ylonen <ylo@cs.hut.fi>
```

```
telnet          23/tcp    Telnet
telnet          23/udp    Telnet
#                         Jon Postel <postel@isi.edu>
              24/tcp    any private mail system
              24/udp    any private mail system
#                         Rick Adams <rick@UUNET.UU.NET>
smtp            25/tcp    Simple Mail Transfer
smtp            25/udp    Simple Mail Transfer
```

ネットワーク上の一つの計算機がネットワーク上の別の計算機上の特定のサービスを利用したいと希望したとき、最初の計算機は伝言を作成し、その伝言の中に希望するサービスのポート番号に置いて、それから伝言を行き先の計算機に送ります。もし、そのポート向けの著名なサービスを行うプロセスがそのポートに束縛されていれば、このプロセスがその伝言を受け取り、要求された作業を実行します。

著名なポート範囲内のポートに束縛されているプロセスについての主要な制約は、それらが**特権利用者権限**(super user priviledge)で動作していなければ**ならないこと**です。

13.1.7.2 登録されたポート (1024 - 49151)

登録されたポートは**著名なポート**と同様に動作していますが、関連付けられた**プロセス**は**特権利用者権限**で**動作している必要がありません**。**登録されたポート**の一覧は、**著名なポート**の一覧を含んでいるのと同じ **IANA 文書**に含まれています。

13.1.7.3 動的/個人ポート (49152 - 65535)

これらのポートは必要時応じて使われ、それらに関連する特定のプロセス型はありません。この領域のポートの典型的な利用は、ウエブブラウザ向けにウエブサーバを用いた外部接続です。

13.1.8 SSH (Secure SHell) サービス

著名なポートを経由して利用されるサービスの例として、**SSH**(Secure Shell)サービスがあり、それは通常 **port 22** に束縛されています。**SSH サービス**はネットワーク上の別の計算機から利用者がログインすることを許可します。遠隔計算機にログインする前に、利用者は**利用者名**と**パスワード**を知っていなければなりません。そして、遠隔計算機は(プロセスの形式で)動作し、port 22 に束縛された SSH サービスを持っていなければなりません。SSH は計算機間を、その間を行き来するデータの暗号化によって機密を守った接続が行え

ます。

UNIX 系のシステムでは、SSH クライアントプログラムは単に SSH と呼ばれ、Windows システムでは、貴方は ssh サービスが動作している計算機にリモートでログインが行える様に putty.exe と呼ばれるプログラムをダウンロードとインストールを行わなければなりません。この putty.exe プログラムは(http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html)から入手できます。

ssh クライアントプログラムが最初にリモートマシンにログインしようとする時、利用者にこのホストに対する暗号情報を現時点で持っていない事を告げて、続けるかどうかを尋ねます。"yes"と入力することで答えます。すると、このプログラムはこのホストに関する情報を既知のホストの一覧に追加し、その後は二度と質問をしません。

13.1.9 ネットワーク上の計算機間のファイルを複写する為の scp の利用について

SSH サービスは利用者が計算機にリモートログインを許容するだけではなく、ネットワーク上の計算機の間のファイルの複写を行う為にも使えます。ファイル複写の為の Linux のクライアントプログラムは scp (Secure Copy) と呼ばれ、そして、人気のある Windows scp クライアントの pscp.exe は putty.exe と同じ url から入手出来ます。

13.2 SAGE の構成 (まだ)

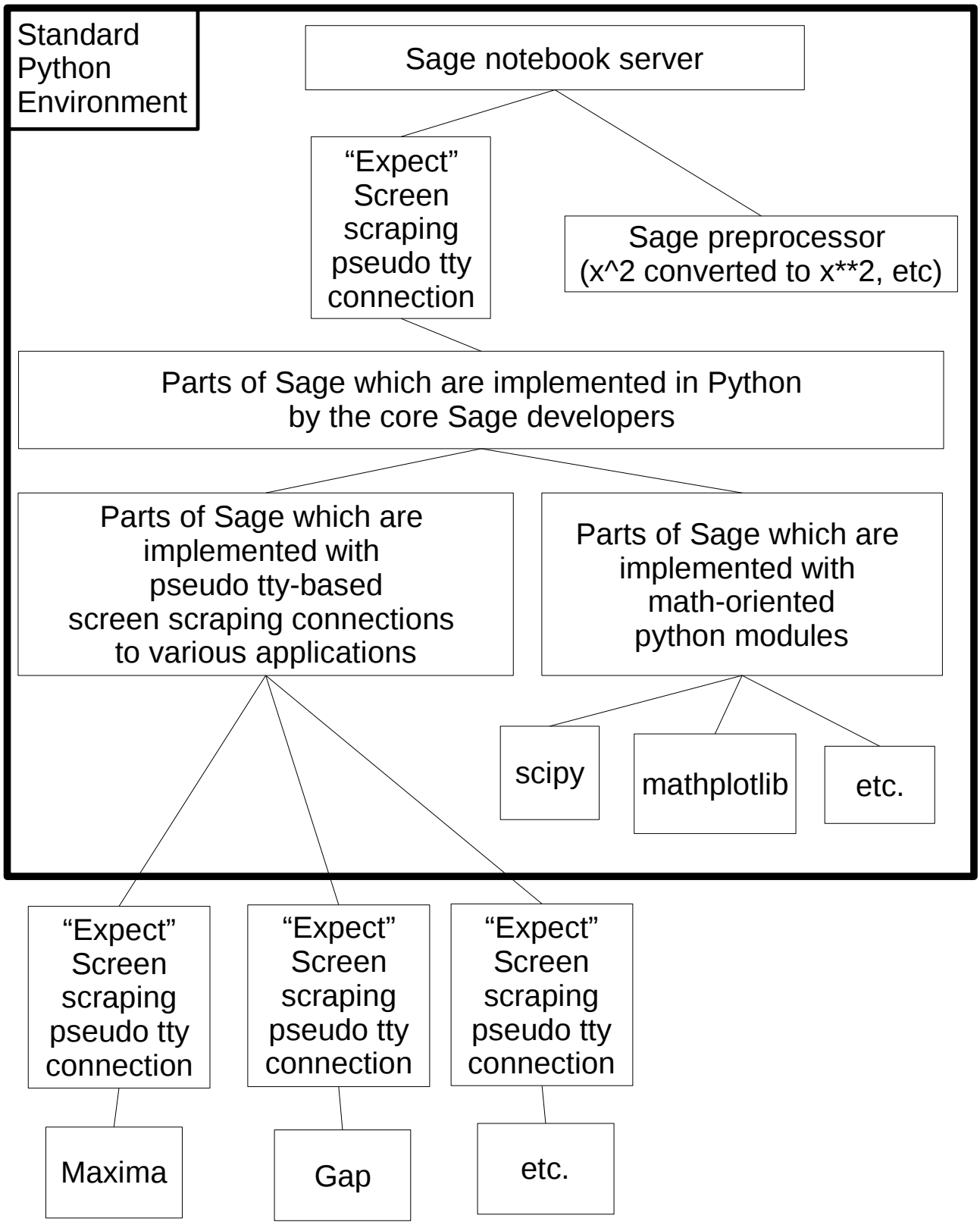


図 *13.5: SAGE*の構成

13.3 Linuxに基づくSAGE版

（まだ...）

13.4 SAGEのVMware仮想計算機版(大半のWindows利用者向け)

（まだ...）