

数式処理システム SageMathへの招待

- Python で記述された統合数学環境 -

USO794 版

横田博史

平成 30 年 02 月 14 日 (水)

数式処理システム SageMath への招待 ©(2018) 横田 博史著
この著作の誤り, 誤植等で生じた損害に対して MathLibre
のメンバー, 著者は一切の責任を負いません.

まえがき

SageMath は非常にユニークなシステムです。開発手法のユニークさに加え、SageMath が土台にしている Python 言語の柔軟度の高さによって他の数式処理システムとの違いを際立たせています。この点は SageMath という数式処理が既存のさまざまなアプリケーションを組み込んでゆくことで仮想環境やクラウド環境へと柔軟に対応しながら機能を拡充していくことに顕著に現われています。このように SageMath は鶴やキメラのような存在ですが、システムとして大きな統一感を Python が与えていますが、このことは非常に重大なことです。巨大なシステムを構築する際にフルスクラッチで全てを構成するよりも既存のものを上手く使う方がコスト的にもスケジュール的にも、さらには実現可能な機能の見積の上でも有利であり、これだけ計算機を使った解析が進んだ現在では、そのような事例やデータの積み重ねを利用しないということは考えられないことです。だから、この SageMath の「**車輪の再発明をしない**」という開発手法は非常に興味深い手段であり、その方針で構築したシステムはそれだけでも注目に値するシステムです。

この文書は SageMath の解説書と銘打っているものの、読んで頂ければ判りますが、実質的に Python の話が主になります。そして、私自身の興味も数学上の概念を Python でどのようにして表現するかということに集中しているために、そのことに必要と思われる哲学的なことや数学的なことがら、それと Python 自体のことを鬼火がフラフラと漂うように記述しています。その意味でこの文書は SageMath を直ちに習得することには向かないでしょう。逆にその浮遊具合を色々と楽しんで頂ければと思う次第です。

なお、この文書はまだ下書き以前の段階で、多少の間違いどころか致命的な間違いや嘘も大量に含んでいます！だから USO 版です。だから現時点ではこの文書の内容の保証は十分できないこと、そのためこの文書の二次配布はご遠慮願います。とはいって絶対秘密の文書ではありません。GitHub で公開しているためにリンク先の紹介等は構いませんし、間違いや問題点の指摘は歓迎します。どこに転がるか判らない代物ですが、どうぞ（生？）暖く見守ってやって下さい。

目次

第1章 SageMathについて	1
1.1 背景	2
1.2 SageMathの簡単な使い方	7
第2章 オブジェクト指向について	35
2.1 SageMathの中核としてのPython	36
2.2 オブジェクト指向プログラミングの哲学的側面	37
2.3 判断と推論	53
2.4 集合論について	65
2.5 関数と関係	73
2.6 代数的構造について	76
2.7 番 (Category)	82
2.8 トポス (Topos)	121
2.9 トポスの基本定理	127
2.10 高階論理 λ -h.o.l. とトポス	130
2.11 ニューラルネットワークについて	138
第3章 Pythonについて	143
3.1 Pythonの概要	144
3.2 有理数を構築してみよう	154
3.3 バッカス・ナウア記法 (BNF)について	160
3.4 Pythonの字句解析について	164
3.5 Pythonの式と文	173
3.6 複合文	194
3.7 オブジェクトについて	197
3.8 特殊メソッド	223
3.9 記述子 (descriptor)	230
3.10 クラス属性の参照について	232

3.11	名前空間とスコープ	236
3.12	例外	238
第 4 章	数値行列処理について	245
4.1	数式処理と数値行列処理	245
4.2	IEEE 754 による実数の表現	246
4.3	数値行列ライブラリについて	254
4.4	NumPy による数値計算	268
4.5	Numba について	289
4.6	SageMath と NumPy の計算比較	293
第 5 章	数学的対象の表現	297
5.1	はじめに	298
5.2	Python の数の構成	300
5.3	SageMath の数の構成	305
5.4	SageMath のオブジェクト	310
5.5	SageObject の各階層	312
第 6 章	結び目理論への適用	317
6.1	概要	317
6.2	結び目/絡み目とは	317
6.3	正則射影図	319
6.4	結び目/絡み目の同値性	321
6.5	群について	323
6.6	結び目/絡み目を表現する群	327
6.7	組紐群と置換群	334
6.8	ガウス・コード	340
6.9	LinkClass クラス	350
6.10	多項式不变量	373
6.11	カウフマンのブラケット多項式を計算するプログラム	376
参考文献		391

第1章

SageMathについて

πάντες ἀνθρώποι τοῦ εἰδέναι ὅρέγνονται φύσει.

凡ての人は自然に知ることを欲する。

アリストテレス, 形而上学

1.1 背景

1.1.1 車輪の再発明をしない

SageMath は非常にユニークなオープンソース ソフトウェア (Open Source Software, OSS と略記) のシステムです。SageMath の開発者 (William Stein) の「**車輪の再発明をしない**」との言葉からも判るように最初から自前のソフトウェアを構築するのではなく、既存の優れた OSS のソフトウェアを取込むことで必要な機能を実現するという手法です。このような開発手法が可能になった背景に高機能の OSS のアプリケーションが多数存在していることに加え、それらを組合せて動作させる計算機環境に余裕があるという二つの状況があります。これらの事実の背景について簡単に説明しておきましょう。

OSS のアプリケーションには研究機関の研究成果として一般に公開したものが多くあり、それらは専門分野で非常に優れた性能を持っていますが、用途と利用者が限定されるために処理言語やデータ構造が汎用性を持たない独特な仕様になり易く、その結果、専門家でも使い難いものや狭い世界での利用のためにマニュアル等の文書化に難があったりと誰にでも簡単に使えるものでないことがあります。そこで、アプリケーションの入出力を抽象化して汎用性を持たせ、処理言語やデータ構造も Python で統一するとどうなるでしょうか？このときに利用者に見えるものは処理言語の Python と、その Python を使って定義したデータしかありません。さらに新たに定義したデータも専門家の観点のみで定義したものから、より広範囲な観点を加えた普遍的なものとして再定義された結果、どのように対象が表現されているかということさえ理解していれば門外漢でも高度な処理が行え、Python を共通基盤にしたことでの統一的な操作環境が得られ、以前は利用できなかった別分野で有用な専門アプリケーションとの連携を視野に含めた基盤までもが整備されます。

このように既存のアプリケーションを連携して使うというやり方が十分に実用的になった背景に、近年の計算機環境が非常に贅沢な環境、実際、携帯電話の CPU でさえも 1GHz 以上の動作周波数で複数のコアが動作し、2GB 以上のメモリ、最低でも 16GB 程度の記憶媒体を持ち、高速ネットワークに当然のように接続可能、といったことが挙げられます。このような「**贅沢な環境**」では既存のアプリケーションを Python 言語のような比較的低速な対話処理言語で繋ぎ合せたシステムでも常識的なプログラミングで実用的な処理速度で動作します。その上、職人技で最適化したシステムよりもリリースの期間や作業量等を含めて全体的なコストが安く上がるという長所まであります^{*1}。

^{*1} だからといって高速処理への要求がなくなることではありません。あくまでもコストか所要時間のどちら

1.1.2 SageMath の中核としての Python

SageMath は Python を基盤とし、その処理言語も Python です。Python はオブジェクト指向プログラミングの考えが取り入れられた言語で、さらに「**多重模範言語 (multi-paradim language)**」と呼ばれる多様なプログラミング様式に適応できる言語です。さらに Python は「**継承**」と呼ばれる機能を享受できます。この機能は新しいクラスを構築する際に既存のクラスを雛形し、その既存のクラスに付随する属性やメソッドを新しいクラスのメソッドとして使い、その書き換えもできるというものです。たとえば貴方が開発した言語には実数があっても複素数がなかったとします。その言語で複素数を扱う必要が生じたとき、複素数が実数の対として表現できるためにオブジェクト指向プログラミング言語であれば実数のクラスを雛型に複素数のクラスが構築できます。このときに実数に付随する四則演算に対応するメソッドが複素数クラスにそのまま継承され、複素数上の四則演算を頭から構築する必要がなく、実数の四則演算を基に足りない純虚数に対する処理を追加することで複素数の四則演算が定められます。このことはライブラリを構築したときに、それを基礎とするさまざまなライブラリが容易に構築できることを意味します。このように継承を活用してソフトウェア資産の効率的な利用ができます。だから、オブジェクト指向プログラミングである言語 Python を基底に用いている SageMath であれば、数学上のさまざまな概念が Python のクラスとして表現されているために処理すべき問題に適合したクラスを選択してメソッドの修正や追加を行い、そうでなければ類似のクラスの概念を継承する新たなクラスを自らが定義しさえすればよいのです。このように数学の諸問題への日々の対処が将来への資産として容易に生かせることが SageMath の強みです。

1.1.3 電池込みだよ

SageMath に類似したものに Anaconda, Inc. の Anaconda があります。この Anaconda はデータ・サイエンティスト向けの Python 環境の位置付けで、ノートブック形式のユーザ・インターフェイスとして Jupyter が利用できる点に加えて SageMath とも重なる OSS のアプリケーションやライブラリを纏めたパッケージであり、共にクラウド環境 (CoCalc と AnacondaCloud) があるといった類似点がありますが、システムとしての統合度は大きく異なります。たとえば、Python で記述された数式処理パッケージ SymPy が SageMath と Anaconda の双方に含まれていますが、Anaconda で SymPy の扱いは、NumPy や Matplotlib 同様に Python の一つのパッケージで、利用するためには通常のモジュールと同様に import 文で読み込みます。それが SageMath では SymPy は数式の表現を受け持つ不可分の部品の一つです。ところで、SymPy にも多項式の展開・因子分解、初

かを優先するかという問に対する一つの現実的な解答です。

等函数の微分・積分といった数式処理の機能がありますが、SageMathではCommon Lispで記述された汎用の数式処理 **Maxima**を数式処理の中心に据えています。このMaximaはMITで人工知能の研究と並行して開発が行われた最古参の数式処理 Macsyma の OSS版で、文脈(Context)等の興味深い機能を持っていますが、近年、発展した計算機代数の結果が十分に取り入れられておらず、単体として、古風な数式処理であることは否定できません。しかし、SageMathでは可換環の処理は **Singular**、数論は **PARI/GP**、有限群論は **GAP**といったOSSの専門の数式処理に、数値行列の処理とプログラミング機能はPythonに、そして、フロントエンドは IPython や Jupyter に任せることで弱点を克服しています。しかし、利用者には Python をその処理言語として用いる一つの大きなアプリケーションにしか見えません。このように SageMath は多様なアプリケーションを一纏めにした便利なパッケージではなく、一つの有機的に纏まったシステムを利用者に提供している点に大きな特徴があります。

多くの数式処理システムの注意事項として挙げられる点が数値行列処理の遅さです。これは任意精度演算を採用していることも一因ですが、それ以上に数値行列計算ライブラリの最適化が不十分であることに加え、数値行列を処理するための構文や函数が貧弱であることが拍車をかけています。SageMathでは任意精度数値計算ではGMP等のCライブラリを利用し、数値行列処理ではパッケージ NumPy を利用します。このNumPyはPythonに多次元配列と関連する処理を導入する基本的なパッケージです。SageMathには効率的な数値行列の処理が行えるように数値行列処理向けのライブラリ「**BLAS(Basic Linear Algebra Subprograms)**」^{*2}としてOSSの「**OpenBLAS**」が組込まれ、さらにPython上でMATLAB本体と同等の機能を実現するためのパッケージ Matplotlibも組込まれているために商用の数式処理システム *Mathematica*に匹敵する数式処理能力と業界標準の数値行列処理システム MATLAB本体に匹敵する数値行列処理能力の双方を SageMath は兼ね備えています^{*3}。

このように SageMath は OSS のさまざまな分野の成果を統合したソフトウェアですが、これは計算処理に限った話ではなく、ユーザ・インターフェイスについても同様です。SageMathは仮想端末向けのシェルとウェブ・ブラウザが利用可能な環境向けのノートブック形式のユーザ・インターフェイスを標準で持ち、仮想端末向けのシェルがIPython、ノートブック形式がJupyterです。ここでIPythonはPythonを仮想端末やウェブブラウザ上で利用するためのユーザ・インターフェイスとして開発され、SageMathはウェブ・ブ

^{*2} 公式標準実装は [netlib\(<http://www.netlib.org/blas/>\)](http://www.netlib.org/blas/) で公開されています。

^{*3} これらの商用のアプリケーションに取って代るという意味ではありません。商用のアプリケーションはアプリケーションやライブラリを含めたファミリー展開を行っており、さらに過去の蓄積を越えることは容易なことではありません。

ラウザ上のノートブック形式のフロント・エンドとして IPython の Notebook(SageNB)も利用していました。Jupyter は IPython から Python 以外のアプリケーションに対してもウェブ・ブラウザ上でノートブック形式のユーザインターフェイスが使えるように IPython から分岐したもので、入出力はセル単位で行われ、利用可能なアプリケーションに対応する kernel をセル単位で切り替えることで一つのノートブックに複数のアプリケーションを混在させることができます。現在、SageMath に用意された kernel は SageMath と Python 2 だけですが商用サービスの CoCalc^{*4}では無課金であっても SageMath 本体、Python や Haskell、GNU Octave や GNU R といった言語やアプリケーションが利用可能であり、さらには組版指示言語の LaTeX や reStructuredText(reST)をプリビュー付きで編集できます。

また、SageMath で描画したグラフィックスもノートブックに表示させることも可能で、組版指示言語の Markdown の文書であれば評価させるだけでレンダリングが可能なために、ちょっとした文書の作成も容易にできます。

このように SageMath では多くのアプリケーションとライブラリが有機的に取り込まれて一つのアプリケーションとしての外観を持たせることに成功したシステムです。SageMath を数式処理システムとして使うも良ければ、SageMath に組込まれたさまざまなアプリケーションやライブラリを使ってシステムの構築を行うのも良し、CoCalc を使ってスマートフォンからちょっとした計算処理や協調作業をさせたりと、この融通無碍さ加減は Python のいわゆる「電池込みだよ(Battery Included)^{*5}」を彷彿させるものです。

1.1.4 SageMath が使える環境は？

SageMath は UNIX 環境で動作します。ここで UNIX 環境は Solaris, FreeBSD や Apple の macOS(OSX) と各種 LINUX ディストリビューションが対応します。とくに macOS 版は Linux 版のように仮想端末上で動作するものと macOS のアプリケーションとして動作するものの二種類があり、アプリケーション版 SageMath では図 1.2 に示すよ

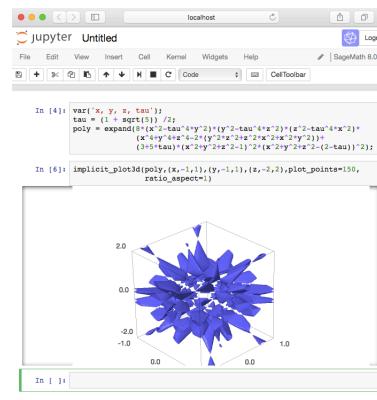


図 1.1 ウェブ・ブラウザを用いた SageMath のノートブック

^{*4} 2017/05 より SageMathCloud から CoCalc(Colaborative Calc Cloud に改名しました。

^{*5} 子供の玩具を買って帰宅したときにパッケージに「電池別売」と貼ってあるシールを見て、ある種の落胆を感じたことがあるのは私だけではないでしょう。

うに Finder 上に SageMath のアイコン現れ、そのメニューに SageMath のノートブック形式や仮想端末上の CUI 形式の SageMath、さらには Maxima 等のアプリケーションを個別に仮想端末上で動かせます。さらに Mac と比べて異質な環境であった MS-Windows でも SageMath が動作するようになりました。

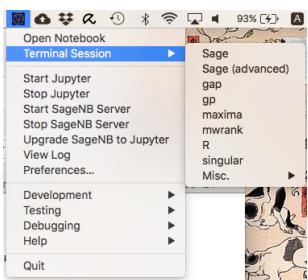


図 1.2 Sage.app のメニュー

ここで SageMath のような複合的なシステムではライブラリの整合性の問題で障害が出易く、さらに苦労して構築した環境を各種ソフトウェアの更新を含めて保守・運営しなければならないという厄介な課題が生じます。そこで SageMath の開発者が採用した手法は、SageMath が必要な必要とするアプリケーションやライブラリの一切合財を収録した巨大なパッケージとして配布することです。こうすることで利用者が微妙なバージョンの違いで悩ま

される可能性を大きく下げられます^{*6}。

なお、SageMath はソースファイルで 280MB 程度、バイナリ版で 700MB、MS-Windows 版で 1GB 程度、仮想計算機版になると 3.2GB、後述の贅沢な SageMath 環境である CoCalc の Docker イメージになると 8GB のディスク容量を必要とする大きなシステムです。とはいっても近年の計算機の能力の向上、記憶容量の増大、高速ネットワーク環境といった御利益によって入手がさほどの負担にならなくなっています。しかし、SageMath のインストールや仮想計算機が利用できなければ、最後の手段として CoCalc^{*7}を使ってみましょう！CoCalc はクラウドベースの SageMath でちょっとした計算であれば無課金で利用できます。こちらは LATEX 等の揃った Ubuntu のシステムとしても使えるために非常に便利です。図 1.3 に iPhone6 Plus から CoCalc に接続して（ハート状の）代数曲面の表示を行った様子を示していますが、最近の大画面化したスマートフォンであれば、ソフトウェアキーボードが邪魔であるとは言え、ちょっとした計算や可視化さえも可能です。また、Bluetooth に対応したキーボードとマウスがあれば十分に快適な操作環境さえも得られます。また、Chromebook であればネットワークさえ確保できれば快適な利用環境が得られるでしょう。このように SageMath を導入することは貴方の計算機に強力な数学環境を構築するということだけではなく、CoCalc も併用すれば、スマートフォンさえあれば何処でも数学の問題に対処できることを意味します。

^{*6} それでもシステム側のコンパイラの問題で異常が生じることがあります。

^{*7} <https://cocalc.com>

また、計算機のディスクや処理能力に余裕があれば MathLibre^{*8}を導入されでは如何でしょうか? MathLibre は ISO イメージで 3G 程度で、SageMath だけではなくさまざまな数学アプリケーションや TeX 環境、さらには数学アプリケーションに関する日本語文書を包含しています。SageMath で遊ぶも良し、他のアプリケーションで遊ぶのも楽しいでしょう。そして、これらのアプリケーションの中から自分に本当に必要なものを見つけることもできるでしょう! MathLibre はそのようなソフトウェアのカタログとしても使えます^{*9}。また、SageMath のためにディスクの領域を割けなければ USB メモリに MathLibre をインストールして、そこから起動するといった手段もあります。こちらを利用する場合は仮想計算機版の SageMath よりもキーボードの設定が容易であったりと、使い勝手もとても良いものです。

1.2 SageMath の簡単な使い方

1.2.1 SageMath のユーザ・インターフェイス

SageMath は Python 2 を基盤にしています。この Python 2 の言語的な側面は §3 にて詳細を述べることとし、ここでの解説は数学に関連する話に限定して幾つかの例題を示すことにします。SageMath があらかじめ貴方の計算機に導入されていると仮定して解説しますが、そうでなければ CoCalc を試してみると良いでしょう。

まず、SageMath の起動は UNIX 系の OS であれば仮想端末上で `sage` と入力します。これで IPython をフロントエンドにして SageMath が立ち上がります。LibreMath や macOS であればラウンチャから立ち上げるとノートブック形式のフロントエンドが立ち上がります。そして、MS-Windows 版であれば SageMath のアイコンをクリックします。仮想計算機版を利用するのであれば最初に VirtualBox を立ち上げてから SageMath の仮想計算機を起動させます。この場合は仮想計算機のウィンドウいっぱいにノートブック形式のフロントエンドが立ち上がるるために SageMath というアプリケーションが立ち上がったように見えます。

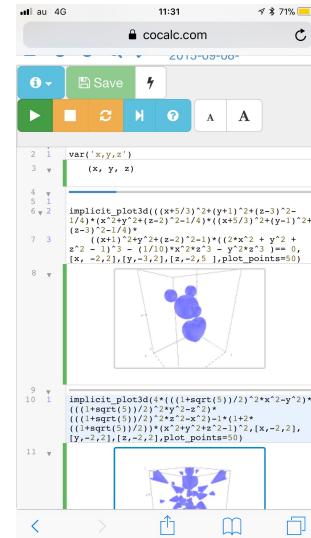


図 1.3 スマートフォンで利用

^{*8} 以前は KNOPPIX ベースの「KNOPPIX/Math」として開発・配布されていましたが、現在は Debian Live ベースです。

^{*9} とは言え、SageMath の容量が TeXLive 等の他のアプリケーションをこのところ顕著に圧迫しているのが現状です。

SageMath のフロントエンドには仮想端末上で動作する IPython とノートブック形式の Jupyter, SageNB と CoCalc のフロントエンドがあります。IPython は標準の Python のシェルよりも履歴機能, ログ出力や GUI 等の機能が強化されています。ノートブック形式の Jupyter はセル単位で kernel として登録されたアプリケーションを利用することができます。また, SageNB は Jupyter が標準になる前のフロントエンドで, IPython のノートブックを利用しています。なお, CoCalc で用いられているフロントエンドは Jupyter ノートブックを基盤としていますが, CoCalc 専用で SageMath には提供されていません。

この本では対話的な処理では IPython をフロントエンドにした様子を示し, プログラムの記述や複数行に亘る処理の表示では Jupyter ノートブックの様子を示します。まず, 仮想端末上で対話処理を行ったときの様子を示します:

```
sage: p1 = 1 + 1
sage: p1
2
```

ここで ‘sage:’ は仮想端末で利用するときの SageMath のプロンプトで, 式の入力は Python と同様にプロンプトに続いて入力します。Python のシェルと違って入力式の先頭に空白文字が入っていてもインデント関連のエラーが出ませんが, 条件分岐, 反復文等で複数行に及ぶ文については Python のシェルと同様にブロック単位でインデントを行わなければなりません。入力式の評価は仮想端末であれば Enter キー, ノートブック形式であれば式を入力したセル内で Mathematica と同様に Shift+Enter で入力セルの評価を行います。ここでの例では最初の行の処理が変数 p1 への $1 + 1$ の計算結果の割当を行っていますが, Python/SageMath では割当て評価のエコーバックが行なわれず, 名前 p1 を入力すると変数 p1 に割り当てられた値が表示されます。また, Python/SageMath では入力行末尾に記号 “;” を置くと評価のエコーバックを行ないません。

1.2.2 ノートブック形式のフロントエンド

SageMath のノートブック環境はウェブ・ブラウザを利用し、出力式を美しくレンダリングしたり、グラフやアニメーションのノートブック上での表示が行えます。以前の SageMath では IPython notebook を基にした SageNB をノートブック環境として利用していましたが、現在は Jupyter Notebook が標準です。なお、仮想端末上で ‘sage –notebook=sagenb’ で起動すると SageNB, ‘sage –notebook=jupyter’ で Jupyter Notebook で立ち上がり, ‘sage’ とオプションなしであれば、SageNB を既定値のノートブックに指定していない限り Jupyter で立ち上がります。また、macOS 上の Sage.app を利用していれば、普通の macOS アプリケーションと同様に Launchpad から呼び出せばノートブック形式の SageMath が立ち上がります。このときに Finder 上に SageMath のアイコンメニューが現われ、そこから Jupyter や SageNB をフロントエンドで SageMath を利用したり、仮想端末から SageMath の起動ができます。

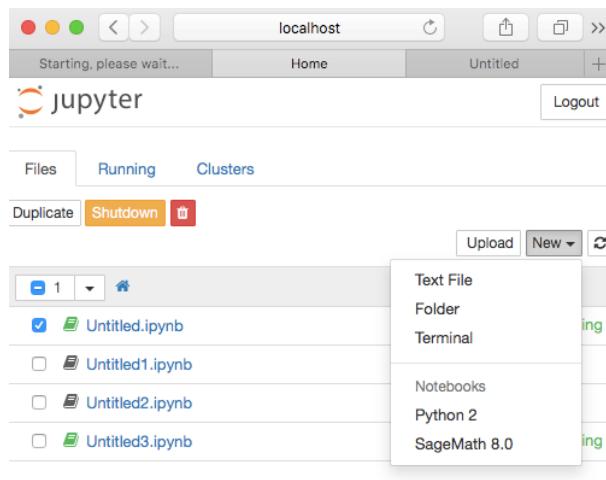


図 1.4 jupyter の Home

最初に SageMath をノートブック形式で立ち上げると「Starting, Please wait...」というタブがブラウザに現われ、それから「Home」というノートの一覧が表示されたタブが現われます。ここに表示されたノートを選択してクリックするか右肩の [New] を押して「SageMath 8.1」を選択して SageMath 用の新規ノート生成します。ノートブック形式ではセルが入力・評価の一つの単位になっています。そのためには「Home」で「Python 2」等の SageMath 以外のノートブックを選択していても kernel 切替で SageMath が利用可能で、逆に SageMath のノートブックから「Python 2」を kernel の切替で利用できます。セルには複数行の式やプログラムを入力しても構いません。このときに [Enter] キーがセル内部の改行で、セルへの入力の評価は Mathematica と同様に [SHIFT+Enter] で行い、評価結果は入力セルの直下に出力されます。ここで出力式は昔風のキャラクタを用いた数式表示が既定値で、Jupyter ノートブックであれば ‘pretty_print_default(False)’ の評価後

は出力がキャラクタ式で, ‘pretty_print_default(True)’ の評価後は出力が MathJax^{*10} を用いてレンダリングされた数式表示になります. 図 1.5 にこの数式表示の違いを示します.

```
In [14]: pretty_print_default(False)
integrate(1/(x^3-1),x)

Out[14]: -1/3*sqrt(3)*arctan(1/3*sqrt(3)*(2*x + 1)) - 1/6*log(x^2 + x + 1)
+ 1/3*log(x - 1)

In [15]: pretty_print_default(True)
integrate(1/(x^3-1),x)

Out[15]: -1/3 √3 arctan(1/3 √3(2x + 1)) - 1/6 log(x² + x + 1) + 1/3 log(x - 1)
```

図 1.5 ノートブックでの数式の表示

この Jupyter は簡易的なワープロとして使えます. そのためにセルを Markdown 向けに切替えて Markdown でセルに書き込みます. この Markdown は組版言語と呼ばれる HTML ように文書の組版を指定できる言語で, 似たものに reStructuredText(reST) があり, Markdown と reST は共にプレーンテキストで記述し, 組版の指示方法もアスキーアート風に指示するだけですが, 対応するアプリケーションで指示した組版で文書が表示されます. ちなみに Markdown はウェブ・ブラウザで表示するための比較的簡易な文書の作成, reST は Sphinx と組合せて, より一般的な文書作成が行えます.

ここで簡単に Markdown の解説をしておきます. まず, 見出は行の先頭に記号 “#” を置き, “#” の数が見出のレベルに対応します. なお, 行直下に文字数だけ記号 “=”, 行直下に文字数だけ記号 “-” を配置しても, “#” や “##” を行の先頭に配置するのと同じ効果があり, こちらの表記は reST と同じです. そして本文は記載し, 改行は行末に空白文字を二文字入れます. 箇条書は先頭に記号 “-”, “+” か “*” を入れ, これらの入れ子構造はインデントで行います. そして, 箇条書の先頭を数字にするときは数字のうしろに記号 “.” を入れます. この番号は自動で振ることが可能なため, 単に “1.” と入力するだけで実は十分です. 文字の強調は “**”, イタリックは記号 “*” で強調すべき箇所を括ります. それからリンク先は見出を “[]” で括り, その後に URL を記載します. 表もアスキーアート風に記号 “|” を縦線, 記号 “-” を横線として構築できます. この Markdown 文書はレンダリングを行っていないプレーンテキストのままでも昔ながらのアスキー文字による WYSWYG

^{*10} 数式をウェブ・ブラウザ上で表示するための JavaScript ライブリ.

な文書であり、対応するアプリケーションでレンダリングすれば意図したとおりに読める点で、段組の制御命令文で構成され、ブラウザで見るかコンパイルしなければ出来上がりを実際に確認できない HTML や TeX と大きく異なります。

ここで Markdown の実例を挙げておきましょう。まず、入力を行うセルに対してファイルメニューの「Cell」にて「Cell Type」を既定値の「Code」から「Markdown」に切替えます。それから以下の内容をそのまま入力します：

```
# Markdownの例

## はじめに

**Markdown**はなかなか便利ですよ。こんな風に*イタリック*も書けます。
- こんな風に簡単に箇条書きできます。
- 別の書式の箇条書きと分離させるためには二行以上開けます。

1. 多項式：
   1. 一変数: $x^2+1$ 
1. 積分：
   1. $\displaystyle \int_0^1 x^2 = \frac{1}{3}$
1. 微分：
   1. $\displaystyle \frac{d \sin x}{dx} = \cos x$ 
   1. $\Large \frac{d^2 f(x)}{dx^2}$

引用もこう書けます。改行したければ改行する行の行末に空白文字を
二文字入れます;
>var('x, y, z, tau');
tau = (1 + sqrt(5)) /2;
poly = expand(8*(x^2-tau^4*y^2)*(y^2-tau^4*z^2)*(z^2-tau^4*x^2)*
              (x^4+y^4+z^4-2*(y^2*z^2+z^2*x^2+x^2*y^2))+*
              (3+5*tau)*(x^2+y^2+z^2-1)^2*(x^2+y^2+z^2-(2-tau))^2);

表
--

表はアスキーアート風に記載します(この表の「**サイト**」にはリンクが
張られています):

| 数式処理 | サイト |
|:-----|:-----|
| SageMath | [SageMathのサイト](http://www.sagemath.org/) |
| Maxima | [Maximaのサイト](http://maxima.sourceforge.net/) |
```

数式と同様に Shift+Enter で評価すると図 1.6 に示すレンダリング結果が得られます：

The screenshot shows a Jupyter Notebook interface with the title 'jupyter Markdown_Sample (autosaved)'. The notebook contains the following content:

Markdownの例

はじめに

Markdownはなかなか便利ですよ。こんな風にイタリックも書けます。

- こんな風に簡単に箇条書きができます。
- 別の書式の箇条書きと分離させるためには二行以上開けます。

1. 多項式 :

- 一変数: Sx^2+1

2. 積分:

- $\int x^2 dx = \frac{1}{3}x^3$

3. 微分 :

- $\frac{d}{dx}(\sin x) = \cos x$
- $\frac{d}{dx}(x^2 \cos x) = 2x \cos x + x^2 \sin x$

引用もこう書けます。改行したければ改行する行の末に空白文字を2文字入れます;

```
var('x, y, z, tau');
tau = (1 + sqrt(5))/2;
poly = expand((x^2 - tau^4)^2)(y^2 - tau^4 z^2)^2(x^4 - y^4 - z^4 - 2(y^2 x^2 + z^2 x^2 + x^2 y^2))^(3 - 5tau)(x^2 + y^2 + z^2 - 1)^2(x^2 + y^2 + z^2 - tau)^2;
```

表

表はアスキーアート風に記載します(この表の「サイト」にはリンクが張られています):

数式処理	サイト
SageMath	SageMathのサイト
Maxima	Maximaのサイト

図 1.6 Markdown の例

なお、レンダリングされた文書が表示されているセルをクリックすると編集モードに戻つて編集ができます。このように Markdown は HTML や LaTeX のようなタグや命令を使わずに組版を持った文書を容易に作成できます。

SageMath のフロントエンドには命令入力の補完機能があります。これは入力中に TAB キーを押すと、入力中の文節に合致する函数、メソッドや属性の候補を挙げ、それらの選択が行えます。仮想端末上ではその機能に依存する面もありますが、複数の候補があれば候補を出力し、補完可能であれば入力の補完を行います。ノートブック形式のユーザ・インターフェイスで TAB キーによる名前の補完は複数の候補が表示されたセレクタ・ウィンドウが現われて、その中からマウスや TAB キーを押すことで候補を選択できます。macOS 版の SageMath を仮想端末で動作させても同様です。この方法は Mathematica と同様で、この点からも商用のものと比べてさほど劣るものではありません。

SageMath のもうひとつの利用者支援の機能にヘルプ機能があります。これには幾つかの函数や演算子があります。まず、最も代表的な方法が Python と同様に函数 help() を使う方法です。たとえば、函数 expand() を調べたければ ‘help(expand)’ と入力するとヘルプ文書を読むことができます。このヘルプの内容は「文書文字列 (docstring)」と呼ばれるプログラム内に記述された文字列です。ここで函数 help() は文書文字列の表示を行う

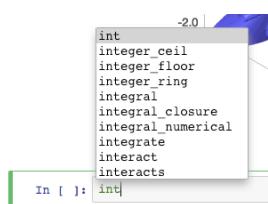


図 1.7 補完の様子

Python 組込関数ですが、Jupyter では記号 “?” もオンラインヘルプとして使えます。たとえば、関数 expand() を調べるときは [expand?] のように調べる事項のうしろに記号 “?” を追記し、さらに “??” でソースファイルを見るすることができます。たとえば [expand??] と入力すると関数 expand() のソースファイルを眺めることができます。

1.2.3 SageMath での数の表現

つぎに SageMath での数の表現について説明しましょう。高校までに扱う数に自然数、整数、有理数、実数と複素数があり、これらの数は本質的に無限個ですが、計算機はメモリ容量の上限があるために数の表現もメモリ容量が許す範囲内になります。ところで自然数、整数と有理数は計算機内部で 2 進数として表現できますが、実数と複素数の表現には工夫が必要です。ここで複素数は実数対として考えられるために実数の表現が重要です。ところで、実数の表現方法には二つの方法があります。たとえば、 π という数は有理数で表現できない無理数ですが、実際の生活では 3.14 と近似して利用しています。計算機でも同様で、この数をある桁数に制限した数として扱うか、そうではなく、 $2 \cos^{-1} 0$ と等しい数と考えるかで処理が異なります。数値計算を主に行うアプリケーションでは前者の方法である桁数で近似した数、つまり、浮動小数点数と呼ばれる数として扱い、数式処理であれば、浮動小数点数による表現に加えて、ある方程式の解や関数の値として扱います。

SageMath では円周率 π 、ネイピア数、黄金比のように重要な数は定数としてあらかじめ登録されています。これらの重要な数には名前があり、実際の処理では記号的な数として SageMath で扱われます。たとえば円周率は pi、ネイピア数は e、そして黄金比は golden_ratio、log2 は log2 といった名前を持ち、これらの数はその名前を SageMath に評価させても名前がそのまま返却されるだけですが、メソッド n()、あるいは関数 N() でその具体的な数値の表示が行われます：

```
sage: pi
pi
sage: N(pi, digits=3)
3.14
sage: pi.n(digits=6)
3.14159
sage: log2
log2
sage: log2.n(16)
0.6931
sage: log2.n(prec=16)
0.6931
sage: log2.n(digits=16)
```

0.6931471805599453

函数 `N()` は *Mathematica* に同名の函数があり、それと同様に指定した精度で数を表示し、メソッド `n()` がこの函数 `N()` に対応しますが、これらの函数とメソッドは型(クラス)を変換せずに指定された範囲の数を表示し、表示された数は同じでも等しくないという一見して意味不明なことが生じます。このことをオブジェクトの所属するクラス(型)を返却する函数 `type()` を使って確認してみましょう：

```
sage: N(pi, digits=3)
3.14
sage: N(pi, digits=3)==3.14
False
sage: type(N(pi, digits=3))
<type 'sage.rings.real_mpfr.RealNumber'>
sage: type(3.14)
<type 'sage.rings.real_mpfr.RealLiteral'>
sage: RR(N(pi, digits=3))
3.14160156250000
sage: a = 3.14
sage: b = a.n(digits=10)
sage: b
3.140000000
sage: type(b)
<type 'sage.rings.real_mpfr.RealNumber'>
sage: type(a + a)
<type 'sage.rings.real_mpfr.RealNumber'>
```

ここでの例では ‘`N(pi, digits=3)`’ で円周率 π を 3 柱表示させたものと ‘3.14’ という(リテラルと呼ばれる)キーボードから入力した数字を比較していますが同値ではありません。このことを調べるために函数 `type()` でオブジェクトが何であるかを調べています。まず、‘`N(pi,digits=3)`’ が所属するクラスである `sage.rings.real_mpfr.RealNumber` は Sage-Math の任意精度の数のクラスです。‘3.14’ が所属する `sage.rings.real_mpfr.RealLiteral` は入力された数値が小数点 ‘.’ を持つときに暫定的に所属するクラスで、このクラスに所属のオブジェクトは任意精度の数 (`real_mpfr.RealNumber`) に変換できます。なお、双方のクラス名に現れる ‘`real_mpfr`’ は実数の実装に関わる MPFR と呼ばれる多倍長浮動小数点数ライブラリに由来します。そこで、クラスが違うためのように思えますが、実は函数 `N()` とメソッド `n()` で表示させた数は指定した精度の部分だけで他の部分もあるために違っても当然です。ただ、ここで示したように、実数は単に小数点付きの数字列として入力した時点ではリテラルであり、実際に計算処理されることで特に指定がなければ多倍長浮動小数点数の数として表現されます。なお、多倍長浮動小数点数では処理時間がかかるために CPU に実装されている倍精度浮動小数点数で計算しなければならないことがあります。

ます。SageMath では倍精度の浮動小数点数のクラスも用意されおり、構築子 `RDF()` で初期化して型の変換を行います：

```
sage: a = 1.3
sage: type(a)
<type 'sage.rings.real_mpfr.RealLiteral'>
sage: b = RDF(a)
sage: type(b)
<type 'sage.rings.real_double.RealDoubleElement'>
```

つぎに SageMath の定数、函数式や整数と有理数のクラスを示しておきましょう：

```
sage: type(pi)
<type 'sage.symbolic.expression.Expression'>
sage: type(I)
<type 'sage.symbolic.expression.Expression'>
sage: type(e)
<type 'sage.symbolic.constants-c.E'>
sage: type(1)
<type 'sage.symbolic.expression.Integer'>
sage: type(1/4)
<type 'sage.symbolic.expression.Rational'>
sage: type(sqrt(5))
<type 'sage.symbolic.expression.Expression'>
```

ここでネイピア数 e のクラス名が他と異なっていますが、これは函数 `exp()` を使って `exp(1)` を包含するダミークラスでネイピア数を定義しているためです。

SageMath の自然数 \mathbb{N} 、整数 \mathbb{Z} 、有理数 \mathbb{Q} 、実数 \mathbb{R} と複素数 \mathbb{C} の数を初期化する構築子（コンストラクタ）を以下にまとめておきます：

数の構築子

数	表記	事例
自然数 \mathbb{N} :	<code>NN()</code>	$NN(1.0) \mapsto 1$
整数 \mathbb{Z} :	<code>ZZ()</code>	$ZZ(1.0) \mapsto 1$
有理数 \mathbb{Q} :	<code>QQ()</code>	$QQ(1.4) \mapsto 7/5$
実数 \mathbb{R} (任意精度):	<code>RR()</code>	$RR(7/5) \mapsto 1.400000000000000$
実数 \mathbb{R} (倍精度):	<code>RDF()</code>	$RDF(7/5) \mapsto 1.4$
複素数 \mathbb{C} (任意精度):	<code>CC()</code>	$CC(7/5+I) \mapsto 1.400000000000000 + 1.000000000000000*I$
複素数 \mathbb{C} (倍精度):	<code>CDF()</code>	$D(RDF(7/5+I)) \mapsto 1.4+1.0*I$

数の初期化では、包含関係 $\mathbb{N} \subset \mathbb{Z} \subset \mathbb{Q} \subset \mathbb{R}$ で包含される数が初期化されますが、包含されない数の初期化はできません。この数の初期化で挙げた構築子の名前が環 (ring) と呼ばれる数学的構造と密接に関連しています^{*11}。ところで、整数への変換では ZZ() よりも明示的な Integer() が使われ、実数と複素数はその計算機上の表現で任意精度の浮動小数点数と倍精度浮動小数点数に分けられます。ここで軽く述べた数学的構造と SageMath での実装と表現については §5 でその詳細を述べます。

1.2.4 算術演算

複素数は実数と純虚数から構成される式でもあるため、ここでは整数、有理数と実数の算術演算式について述べます：

SageMath の算術演算子

演算	演算子	式の書式	式の意味と変数の領域
和	+	$x + y$	$x + y$
差	-	$x - y$	$x - y$
積	*	$x * y$	$x \times y$
商	/	x / y	$\frac{x}{y}$
商	//	$x // y$	$(x - (x \bmod y)) \div y \quad x, y \in \mathbb{Z}$
冪	^	$x ^ y$	x^y
剰余	%	$x \% y$	$x \bmod y \quad x, y \in \mathbb{Z}$

基本的に SageMath の算術演算式は他の計算機言語とほぼ同様の演算子と書式を持ちます。ただし、商の演算子 “/” では被演算子に浮動小数点数が含まれると浮動小数点数を返す計算処理になります。なお、被演算子が π のような SageMath の数学的定数、 $\sqrt{2}$ のような代数的数のときは有理式 (= 分数式) の形です。それと被演算子が整数のときの演算子 “//” は Python 2 の商演算子 “/” と同様に整数の商を返します。演算子 “^” は SageMath の数学的対象に対しては冪演算子として作用します。ちなみに Python で演算子 “^” は XOR 演算子として用いられていますが、この演算子は TeX で数式の冪演算子として扱われており、このこともあって数式処理システムの多くや MATLAB 系言語で冪演算子として扱われています。

SageMath にはさまざまな型 (クラス) の数があります。SageMath での算術演算では、算術式に複数の型のオブジェクトが混在していても、型の変換が必要に応じて自動的に行われます。この型変換では数の包含関係 $\mathbb{N} \subset \mathbb{Z} \subset \mathbb{Q} \subset \mathbb{R} \subset \mathbb{C}$ から被演算子全体を包含す

^{*11} 数式処理 Singular の環の命名に対応します。

る数のクラスのオブジェクトになるように変換されます。なお、自然数、整数、有理数のインスタンスのみが被演算子で、その演算結果が整数になるときは整数として返却されます：

```
sage: 10 + 1/2
21/2
sage: 10 + 0.5
10.5000000000000
sage: 10 * 0.5
5.00000000000000
sage: 10 * 1/2
5
sage: (10 + I) * (10 - I)
101
```

これらの例で示すように浮動小数点数の書式を持った被演算子を含む算術演算の結果は浮動小数点数を含む書式になります。

整数、有理数や実数では大小関係があります。Python では一般的に同じクラスのオブジェクトの同値性は演算子 “`==`” で調べられ、その値の大小関係は “`>=`”, “`>`”, “`<=`”, “`<`” といった比較の演算子で調べることができます。また、これらの演算子は和演算子 “`+`” と同様に被演算子の型が異っていても、数学的対象として大小関係で比較可能であれば、これら比較の演算子が使えます。なお、実数や複素数で浮動小数点数を用いるときは、実際は異なる数でも丸めによる誤差で同じ浮動小数点数を使って表現されるために同じ値になることがあります。

なお、算術演算子は C と同様ですが、他の数式処理システム、たとえば Maxima や *Mathematica* にある階乗の演算子 “`!`” は SageMath になく、函数 `factorial()` を使います。基本的に SageMath の演算子は Maxima と比較して多くはありません。そもそも、Maxima では演算子の定義は非常に簡単です。つまり、函数を定義して、その函数がラプラスian Δf のように被演算子の前に演算子記号を配置する「前置演算子」であるか、和 $1 + 2$ のような被演算子の間に演算子記号を配置する「中置演算子」であるか、階乗 $n!$ のように被演算子の後に演算子記号を配置する「後置演算子」であるか、あるいは絶対値 “`| |`” のような被演算子を内包する演算子であるかを宣言すると目的の演算子が定義できてしまいます。しかし、SageMath は Python 言語の制約上、そこまで自由な演算子の定義が行えず、記述子 `infix_operator` を使って中置演算子の定義が行える程度です：

```
sage: def dao(x, y): return x * y + 1
sage: dao = infix_operator('multiply')(dao)
sage: 3 *dao* 4
```

1.2.5 名前と変数

SageMath の基盤である Python が扱うデータは全てオブジェクトです。これらのオブジェクトの参照で「名前 (name)」が用いられます。この名前はオブジェクトの名札に対応します。そして、名前とオブジェクトの対応関係を「名前空間 (name space)」と呼びます。この名前とオブジェクトとの対応付けは「名前への束縛」と呼ばれる操作で行われ、演算子 “=” を使って行います。たとえば、「 $a = 1$ 」という Python の式は、文字列 a が 1 というオブジェクトの名前であるという対応関係を与える式になります。そして、名前空間に含まれている名前の一覧は函数 `dir()` で確認できます。このように名前には必ず何かのオブジェクトが紐付けられており、名前は数式の定数項に対応しているとも言えます。

SageMath では自由変数としてあらかじめ “ x ” が登録されています。この変数 x を使って ‘ $2*x$ ’ のような多項式、‘ $\cos(x)$ ’ のような函数式を SageMath に入力することができます。ここで函数 `type()` で x に対応しているオブジェクトが何であるか確認しておきましょう：

```
sage: type(x)
<type 'sage.symbolic.expression.Expression'>
```

名前 x には ‘`sage.symbolic.expression.Expression`’ という型のオブジェクトが束縛されています。つまり、名前 x はそれ単体で多項式 x としての意味を持っていることを示しています。なお、 x 以外に自由変数が登録されていないために、‘ $y + x$ ’ のような x 以外の自由変数を持つ式を入力したければ、その変数を函数 `var()` を使って登録しておく必要があります：

```
sage: var('x, y, z')
(x, y, z)
sage: a = x + 2*y + 3*z
sage: a
x + 2*y + 3*z
```

このように函数 `var()` の引数として自由変数とする名前を一つの文字列として与えたり、文字列のリストやタプルとして与えることで、新たに名前が自由変数として登録されます¹²。

¹² 函数 `var()` は SymPy 由来の函数で、SymPy でも自由変数 x,y の宣言は `var('x,y')` とします。

1.2.6 多項式の扱い

SageMath では変数 x があかじめ登録されているために, x の多項式は C や FORTRAN の書式の数式や, Maxima や LATEX の書式の数式が直接, 記述することができます. また, x 以外の自由変数が必要になると函数 var() で宣言しておきます. なお, SageMath では数学的対象に対して幕演算子として演算子 “ \wedge ” が使われており, この本でも数式中の幕演算子として演算子 “ \wedge ” を主に用います.

また, SageMath では入力した数式の項を一定の順序で並び替えを行い, 和や差, それと項に対する自明な積や商の結果を整理した式を出力します:

```
sage: 1 + x
x + 1
sage: 3*x - 10 * x + 3 - 1 + 3 * x^2/x^4 * x^(2 + 1)
-4*x + 2
```

この項の入替や項の変数の並び替えは「**項順序**」に従って行われます. 特に指定を行っていない場合は「**辞書式順序**」と呼ばれる項順序が用いられます. この順序は辞書と同様の並びで変数や項を並べる順序で, 項順序を導入することで多項式は指定の順序に対して一意に定まります. なお, 多項式同士の積や商, 式の展開は自動ではありません. そのため式の展開は函数 expand(), あるいはメソッド expand(), 式の因子分解は函数 factor(), あるいはメソッド factor() を用います. ただし, 式の幕乗の次数が整数以外のときは式の展開ができません:

```
sage: p1 = (x + 1)^3
sage: p1
(x + 1)^3
sage: type(p1)
<type 'sage.symbolic.expression.Expression'>
sage: p2 = p1.expand()
sage: p2
x^3 + 3*x^2 + 3*x + 1
sage: expand(p1)
x^3 + 3*x^2 + 3*x + 1
sage: expand((x + 1)^3.0)
(x + 1)^3.00000000000000
sage: type((x + 1)^3.0)
<type 'sage.symbolic.expression.Expression'>
```

ここで $(x+1)^3$ の展開ができるいても $(x+1)^{3.0}$ の展開ができていませんが, 函数 type() の結果から双方の多項式は ‘symbolic.expression.Expression’ というクラスのオブジェ

クトです。式で違っているのは次数の‘3’と‘3.0’だけです。ところで，‘3.0’に対応するSageMathのオブジェクトは浮動小数点数で、この浮動小数点数は本質的に近似値です。多項式の展開のような代数的な処理を行うためには対象となる式を構成する各成分も代数的な対象でなければなりません。

1.2.7 リストや集合の扱い

SageMath/Pythonのリストは演算子“[]”を使ってオブジェクトや名前の列を‘[‘x’, ‘y’]’のように括った書式です。なお、SageMathのリストの生成はPythonのものよりも拡張されています。たとえば、1から10までの自然数のリストの生成は‘[1..10]’でできます：

```
sage: L = [1..10]
sage: L[0]
1
sage: L[1]
2
sage: L[0:5]
[1, 2, 3, 4, 5]
sage: L[-1]
10
sage: L[-4:-1]
[7, 8, 9]
sage: L[-1:-4:-1]
[10, 9, 8]
```

この例では自然数のリストを生成し、それからリストの成分の取出を行っています。この演算子“..”は数式処理システムMapleでも見られる演算子ですが、SageMathで使えてPythonでは使えません。また、Pythonでは配列、リスト等の添字はMaximaやMATLABのように1からではなく、Cと同様に0から開始します。PythonではMATLAB風のリスト処理として**スライス処理**と呼ばれる処理ができますが、添字0から開始するためにMATLABと微妙な違いが生じるために注意が必要です。たとえばPythonでは‘L[0:5]’で添字が0から4までの名前Lに束縛された5成分のリストを返却しますが、MATLABで‘L[1:5]’とすると添字が1から5までの5成分のリストを返します。また、添字を負の整数とすることでリストの末尾、つまり、右端からの成分を返却できます。‘L[-4:-1]’で添字が-4, -3, -2のLの成分のリストを返却し、‘L[-1:-4:-1]’で初期値が-1、増分-1で-1から-4までの添字、すなわち、-1, -2, -3の添字のLの成分リストを返却します。このようにSageMathはMATLABに類似の方法で多次元の配列操作が可能であり、数値行列処理システムのMATLABと大差がない機能を持っています。そして、数値ベクトルと行列はNumPyの数値配列に変換することにより効率的に処理できます。このNumPyを使った数値計算に関しては§4.4を参照して下さい。

SageMath の集合は Python の集合と同じで, SageMath の対象の列を記号 “{ }” で括って生成します. SageMath の集合は Python の集合と同様に, 内部の順序ないために配列のように添字を使って取り出せません.

1.2.8 線形代数

SageMath は NumPy を擁するために MATLAB 風の数値行列処理が可能で, 倍精度浮動小数点数の多次元配列を NumPy を使って処理することで迅速な処理が可能です. ここでは多項式等を含む一般的な SageMath の対象から構成された行列の処理について述べ, NumPy の数値配列の処理については §4.4 で解説します. 最初にベクトルと行列の定義について述べましょう. ベクトルの定義は関数 `vector()`, 行列の定義は関数 `matrix()` を使います. ベクトルや行列の定義では成分の型を指定することもできます. この指定は数と同様ですが, 多項式や函数式を含むときは “SR” を指定します. この指定は省略できます:

```
sage: vector([1,2,3])
(1, 2, 3)
sage: vector(ZZ, [1, 2, 3])
(1, 2, 3)
sage: vector(RR, [1, 2, 3])
(1.00000000000000, 2.00000000000000, 3.00000000000000)
sage: vector(SR, [1,x,x^2-1])
(1, x, x^2 - 1)
sage: matrix(ZZ, 3, 3, [1, 2, 3, 4, 5, 6, 7, 8, 9])

[1 2 3]
[4 5 6]
[7 8 9]
sage: matrix(SR, 2, 5 , [1, 2, 3, 4, 5, 6, 7, 8, x^2-1, x-1])

[      1      2      3      4      5]
[      6      7      8 x^2 - 1  x - 1]
sage: matrix(2, [1, 2, 3, 4, 5, 6, 7, 8, x^2-1, x-1])

[      1      2      3      4      5]
[      6      7      8 x^2 - 1  x - 1]
sage: matrix([[1,2,3,4,5],[6,7,8,x^2-1,x-1]])

[      1      2      3      4      5]
[      6      7      8 x^2 - 1  x - 1]
sage: A = matrix(QQ, 3, 3, [1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
sage: C = A.numpy()
sage: C
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
sage: type(C)
<type 'numpy.ndarray'>
```

ベクトルの定義では明示的に成分が所属するクラスを指定したり、明示的に指定せずに定義することもできます。また、行列の定義ではサイズを明示的に指定する方法、行数のみを指定する方法と “[]” で行を直接指定する方法があります。また、行列の成分の型は成分単位で異なるものを許容し、成分を揃える必要がなければ型の指定は不要です。また、SageMath の数値行列はメソッド `numpy()` で NumPy の配列に変換できます。もちろん、NumPy の `ndarray` 型の配列の構築子 `array()` や NumPy の `matrix` 型の構築子 `matrix()` を使って SageMath の数値行列を NumPy の `ndarray` 型や `matrix` 型の配列として構築することもできます。

行列の転置はメソッド `transpose()`、あるいは函数 `transpose()` を使います。転置は行列のビューの変換で、行列データそのものの変更ではありません。そのため `a.transpose()` や `transpose(a)` の処理後行列 `a` が以前と異なる点は何もありません。なお、`vector()` で初期化したベクトルは縦ベクトルでも横ベクトルでもなく、単なるベクトルとして扱われるため、行列との積で積演算子 “`**`” の右側に配置されると縦ベクトル、左側であれば横ベクトルとして扱われます。逆行列計算はメソッド `inverse()` や函数 `inverse()` も使えますが、数と同様に演算子 “`~`” を利用できます。行列式の計算はメソッド `det()`、`determinant()`、あるいはこれらのメソッドの函数型からも行え、固有多項式はメソッド `charpoly()` で計算できます：

```
sage: A = matrix([[1, 2, 3], [0, -2, 1], [0, 0, 1]])
sage: A
[ 1, 2, 3]
[ 0, -2, 1]
[ 0, 0, 1]
sage: A.transpose()
[ 1, 0, 0]
[ 2, -2, 0]
[ 3, 1, 1]
sage: A.inverse()
```

```
[ 1,      1,      -4 ]
[ 0, -1/2,  1/2 ]
[ 0,      0,      1 ]
sage: A^(-1)

[ 1,      1,      -4 ]
[ 0, -1/2,  1/2 ]
[ 0,      0,      1 ]
sage: A.charpoly()
x^3 - 3*x + 2
sage: Ax=matrix([[1,2,3],[4,5,6],[7,8,x^2-1]])
sage: Ax

[ 1      2      3]
[ 4      5      6]
[ 7      8 x^2 - 1]
sage: Ax.det()
-3*x^2 + 30
sage: det(Ax)
-3*x^2 + 30
sage: Ax.determinant()
-3*x^2 + 30
```

そして、和、差は演算子“+”と“-”，スカラー積や行列同士、行列とベクトルの積は演算子“*”を用います。ベクトルと行列の積で、構築子 vector() で生成したベクトルは縦ベクトルでも横ベクトルでもなく、ベクトルとして扱われ、行列との積では行列の右側であれば縦ベクトルとして、左にあれば横ベクトルとして処理されます。ベクトルと行列の加法で 0 は零ベクトル、あるいは零行列として作用し、正方形行列の加法で 1 は単位行列として作用します。そして、SageMath では行列の指數函数も計算できます：

```
sage: v = vector([1,0,3])
sage: v
(1, 0, 3)
sage: w = vector([0,10,-4])
sage: 2*v - w/3
sage: 2*v - w/3
(2, -10/3, 22/3)
sage: Ax

[ 1      2      3]
[ 4      5      6]
[ 7      8 x^2 - 1]
sage: v * Ax
```

```
(22, 26, 3*x^2)
sage: Ax * v
(10, 22, 3*x^2 + 4)
sage: B = matrix([[1,2,3], [0, -2,1],[0,0,-3]])
sage: B
[ 1  2  3]
[ 0 -2  1]
[ 0  0 -3]
sage: 1 + B + 0

[ 2  2  3]
[ 0 -1  1]
[ 0  0 -2]
sage: 2*B + B^2

[ 3  2  2]
[ 0  0 -3]
[ 0  0  3]
sage: C = matrix([[1,2,3], [0, -2,1],[0,0,1]])
sage: exp(C)

[          e   2/3*(e^3 - 1)*e^(-2) 1/9*(31*e^3 + 2)*e^(-2)]
[          0           e^(-2)      1/3*(e^3 - 1)*e^(-2)]
[          0                 0                  e]
```

さらにベクトルの内積と外積はそれぞれメソッド `dot_product()` と `cross_product()` を使い、ベクトルのノルムは `norm()` で計算可能です。このメソッド `norm()` で引数がなければ通常のユークリッドノルムを計算しますが、1を指定したときは成分の絶対値の和である `1-norm` を返却します：

```
sage: w = vector([5,-3,-1])
sage: v = vector([1,2,3])
sage: v.inner_product(w)
-4
sage: w.cross_product(v)
(-7, -16, 13)
sage: w.inner_product(v)
-4
sage: w.dot_product(v)
-4
sage: v.norm()
sqrt(14)
sage: v.norm(2)
```

```
sqrt(14)
sage: v.norm(1)
6
```

なお、外積が計算できるのは 3 成分か 7 成分のベクトルの場合に限られ、ベクトルのノルムに関しては、引数のないメソッド norm() と引数が 2 のときに 2-norm(通常の長さ) を計算し、引数が 1 のときに 1-norm(成分の絶対値の和) を計算していることが判ります。

1.2.9 解析

SageMath で函数の定義は大きく分けて 3 種類あります。一つは簡易的に演算子 “=” を使って定義する方法と def 文を使って定義する方法と形式的な函数を定義する方法です：

```
sage: f(x, a, b) = a*x + b
sage: f
(x, a, b) |--> a*x + b
sage: def h(x, a, b): a*x + b
sage: h
<function h at 0x1be9729b0
sage: var('t')
sage: T = function('T')(t)
```

ここで SageMath で数式として処理が可能な函数は最初の演算子 “=” を用いた函数と、最後の形式的な函数で、def 文を用いた函数は Python の函数です。

つぎに微分と積分は古くから数式処理の標的になってきた処理です。SageMath がその数式処理の中核に置いた Maxima は従来から積分処理の機能の高さが評価されています。SageMath では数式の微分は函数 diff()、積分は函数 integrate() を用います：

```
sage: diff(x^2+2*x-1,x)
2*x + 2
sage: diff(x^2+2*x-1,x,2)
2
sage: var('x,y,z')
(x, y, z)
sage: diff(x^2*y*z^4,x,y,z)
8*x*z^3
sage: diff(x^2*y*z^4,x,2,y,z,3)
48*z
sage: integrate(x^2+2*x-2,x)
1/3*x^3 + x^2 - 2*x
sage: integrate(x^2+2*x-2,x,0,1)
-2/3
```

微分を行う函数 `diff()` には第1引数に函数, 第2引数に変数を記載します。ここで微分の階数が2以上であれば, 第3引数に階数を記載します。多変数の函数の微分では第1変数の微分の記載を終えると次に第2変数以降の変数と部分の階数を記載しますが, 階数が1のときは省略可能です。函数の積分では第1引数に積分すべき函数, 第2引数に積分変数を指示します。なお, 函数 `integrate()` の不定積分では積分定数の既定値が0になっています。

SageMathには無限大 `inf` があり, その無限大の表記としては‘infinity’と‘oo’の二種類があります。この無限大を用いた実例を幾つか示しておきましょう:

```
sage: [1/(i*(i-1)) for i in range(2,11)]
[1/2, 1/6, 1/12, 1/20, 1/30, 1/42, 1/56, 1/72, 1/90]
sage: sum([1/(i*(i-1)) for i in range(2,11)])
9/10
sage: sum(1/(x*(x-1)),x,2,10)
9/10
sage: var('n')
n
sage: sum(1/(x*(x-1)),x,2,n)
(n - 1)/n
sage: sum(1/(x*(x-1)),x,2,oo)
1
sage: limit(sum(1/(x*(x-1)),x,2,n),n=infinity)
1
```

ここでは最初に Python の内包表現を用いてリストを生成しています。それからリストの総和を計算していますが, これは $\sum_{i=2}^{10} \frac{1}{i(i-1)}$ と同じで, SageMathの函数 `sum()` を使って‘`sum(1/(x*(x-1)), x ,2, 10)`’と表現できます。つぎに変数 `n` を宣言しておき, $\sum_{i=2}^n \frac{1}{i(i-1)}$ を求めています。それから `n` を‘`oo`’で置換えると $\sum_{i=2}^{\infty} \frac{1}{i(i-1)}$ の計算ができます。もちろん, 函数 `limit()` を使って極限を計算する方法もあります。

1.2.10 方程式

ここではいろいろな方程式を解いてみましょう。まず, 代数方程式は函数 `solve()` で解くことができます:

```
sage: solve(x-123,x)
[x == 123]
sage: solve(x^2-3*x+1 == 0,x)
[x == -1/2*sqrt(5) + 3/2, x == 1/2*sqrt(5) + 3/2]
sage: var(['y','z'])
(y, z)
```

```
sage: solve([2*x -y + z == 0, x^2-y^2+z^2-1 == 0,x^3-z^2+2*y-1 == 0],[x,y,z])
[[x == (0.0255946656987 - 1.63139339042*I),
y == (0.0295897155531 - 2.19244726264*I),
z == (-0.0215996158442 + 1.0703395182*I)],
[x == (0.0255946656987 + 1.63139339042*I),
y == (0.0295897155531 + 2.19244726264*I),
z == (-0.0215996158442 - 1.0703395182*I)],
[x == 0.788032678295, y == 0.667795080117,
z == -0.908270133622],
[x == (-0.138360986224 - 0.103194243068*I),
y == (0.988075254834 - 0.994925122194*I),
z == (1.26479722728 - 0.788536636057*I)],
[x == (-0.138360986224 + 0.103194243068*I),
y == (0.988075254834 + 0.994925122194*I),
z == (1.26479722728 + 0.788536636057*I)]]
```

函数 `solve()` は引数として方程式と変数の二つを少なくとも引数として取ります。ここで方程式の表記は演算子 “`==`” を持つ式ですが、0 に等しいときは演算子 “`==`” と 0 を除いた式でも構いません。たとえば、方程式 $x - 123 = 0$ を解くときは ‘`solve(x - 123 == 0,x)`’ でも ‘`solve(x-123,x)`’ でも構いません。そして、函数 `solve()` は常にリストで解を返却します。また、函数 `solve()` で連立方程式を解くこともできます。この場合、連立方程式は式のリストとして表現し、求めるべき変数も変数リストとして与えます。函数 `solve` は可能であれば代数的数^{*13}を用いた厳密解を返却しますが、代数的に解けないときは浮動小数点数を用いた近似解を返却します。

常微分方程式も解くことができます。この常微分方程式の解の計算では Maxima が用いられており、このことが常微分方程式を解く函数のパラメータの指定に影響が出ています。ここでは簡単な常微分方程式を解いてみましょう。

```
sage: var('t');
(t)
sage: T = function('T')(t)
sage: f=desolve(diff(T,t,2) + 2*diff(T,t) == 1, T).expand()
sage: f
_K2*e^(-2*t) + _K1 + 1/2*t - 1/4
sage: g=desolve(diff(T,t,2) + 2*diff(T,t) == 1, T, [0,0,0]).expand()
sage: g
1/2*t + 1/4*e^(-2*t) - 1/4
```

^{*13} 整数係数の多項式に代入すると 0 になる数です。代表的数には整数、有理数、純虚数や n 乗根が挙げられます。

この例では変数 t を函数 `var()` で宣言し、それから変数 t の函数として変数 T を定義しています。こうすることで実態は何か判らなくても変数 t の函数 T が定義できます。SageMath で常微分方程式は `desolve` が使えます。最初は一般解を求めていますが、次の例では最後の引数のリスト $[0, 0, 0]$ で $T(0) = 0, \frac{dT}{dt}(0) = 0, \frac{d^2T}{dt^2}(0) = 0$ と函数 $T(t)$ の各微分の初期値を与えることで、その特殊解を求めていきます。

1.2.11 代入

自由変数や部分式への代入はメソッド `subs()` やメソッド `substitute()` を使います。ちなみに SymPy のメソッド `subs()` では置換えるべき式とその値の対のタプルを引数にしますが、SageMath のこれらのメソッドでは ‘`x=1`’ や ‘`x==1`’ のように演算子 “=” や “`==`” の式として与えます。

```
sage: (x^2 - 1).subs(x=2)
3
sage: (x^2 - 1).subs(x==3)
8
sage: var('x, y');
sage: ans = solve([x^2 + y^2 - 1, x - 2*x + 1/4], [x,y])
sage: ans
[[x == (1/4), y == -1/4*sqrt(15)], [x == (1/4), y == 1/4*sqrt(15)]]
sage: (x + 4*y - 3).subs(ans[0])
-sqrt(15) - 11/4
sage: f = cos(x)^2 - sin(x)^2
sage: f.subs(sin(x)^2==1-cos(x)^2)
2*cos(x)^2 - 1
```

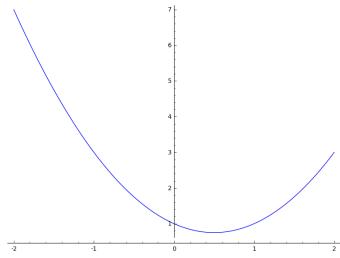
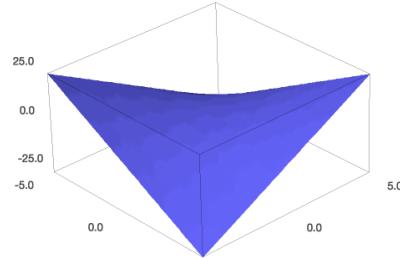
1.2.12 グラフ表示

SageMath のグラフ表示は函数 `plot()` で 2 次元グラフ、函数 `plot3d()` で 3 次元グラフが描画できます。そして、 $y^2 - x^3 + 4xy + 1 = 0$ のような零点集合を描くための函数に `implicit_plot()` と `implicit_plot3d()` があります。より高度な描画を行いたければ Matplotlib の函数を利用することもできます。

■函数 `plot()` と `plot3d()`: 最も良く使われる描画函数です。これらの函数は、表示すべき函数や式を第 1 引数に、以降、変数の領域を指示するタプルやリストが並びます:

```
sage: var('x, y');
(x, y)
sage: plot(x^2 - x + 1, (x, -2, 2))
```

```
sage: plot3d(x * y, (x, -2, 2), (y, -2, 2))
```

図 1.8 $x^2 - x + 1$ のグラフ図 1.9 xy のグラフ

このように数式と表示領域の指定を行うだけで簡単に 2 次元と 3 次元グラフを描けます。ここで 3 次元グラフはマウスを使って回転や拡大も行えます。なお、仮想端末から 3 次元グラフの表示を行うと Jmol そのものが立ち上がりります。本来、分子モデルの可視化で用いられているアプリケーションのためにメニューがグラフ表示向けに転用されていることが非常に判り易くなっています。なお、ノートブック上でグラフ表示を行うときはノートブック上に結果が表示され、メニューの内容も通常のグラフ操作の内容になっています。

■`implicit_plot()` と `implicit_plot3d()`: 与えられた式の零点集合を表示する函数です。これらの函数は零点集合を求める必要があるために描画に時間がかかります:

```
sage: implicit_plot(y^2-x^3 + 4*x*y+1,(x,-10,10),(y,-10,10))
sage: var('x, y, z, tau');
sage: implicit_plot(y^2-x^3 + 4*x*y+1,(x,-10,10),(y,-10,10))
sage: tau = (1 + sqrt(5)) / 2;
sage: poly = expand(8*(x^2-tau^4*y^2)*(y^2-tau^4*z^2)*(z^2-tau^4*x^2)*
....: (x^4+y^4+z^4-2*(y^2*z^2+z^2*x^2+x^2*y^2))+
....: (3+5*tau)*(x^2+y^2+z^2-1)^2*(x^2+y^2+z^2-(2-tau))^2);
sage: implicit_plot3d(poly ,(x,-2,2),(y,-2,2),(z,-2,2),plot_points=150)
```

オプションの `plot_points` の値は描画する曲面の点数になりますが、最初に 100 程度の値を設定して、出来栄えを確認してからより大きな値に設定すると良いでしょう。

また、CSV データを取り込んで表示することもできます。ここでは NumPy の函数 `loadtxt()` を使って CSV 形式のファイルを読み込み、函数 `list_plot()` で表示します:

```
sage: import numpy as np
sage: L = list(np.loadtxt("/users/yokotahiroshi/testfunc.csv"))
sage: list_plot(L)
```

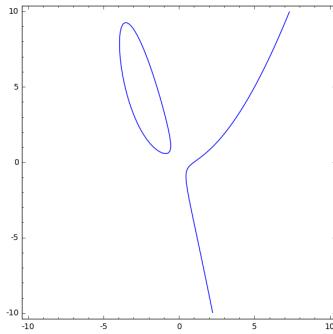


図 1.10 implicit_plot の例

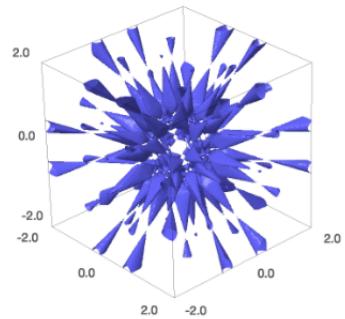


図 1.11 implicit_plot3d の例

この例では testfunc.csv に X-Y の CSV データを格納しています。これを NumPy の函数 loadtxt() で読み込んでいます。ここでは構築子 list() で list 型に変換していますが、そのまま ndarray 型の配列のままでも函数 list_plot() で表示できます。

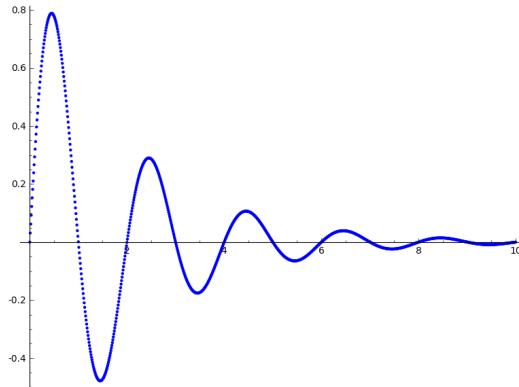


図 1.12 list_plot の例

グラフィックス・オブジェクトに関しては和演算が可能です。この場合は複数のグラフを同一グラフに表示することができます。

```
sage: P=list_plot(L)
sage: Q=plot(exp(-x/2)*sin(pi*x),(x,0,10),color="red",legend_label="1/2")
sage: R=plot(exp(-x/4)*sin(pi*x),(x,0,10),color="green",legend_label="1/4")
sage: (P + Q +R).show()
```

この例はさきほど読み込んだ CSV データのリスト L を再度、函数 list_plot() で描画しますが、その描画オブジェクトを P に保存します。同様に函数 plot() で生成した描画オブ

ジェクトを Q と R に保存し、これらのオブジェクトの和 $P+Q+R$ をメソッド `show()` で表示したものが次のグラフです：

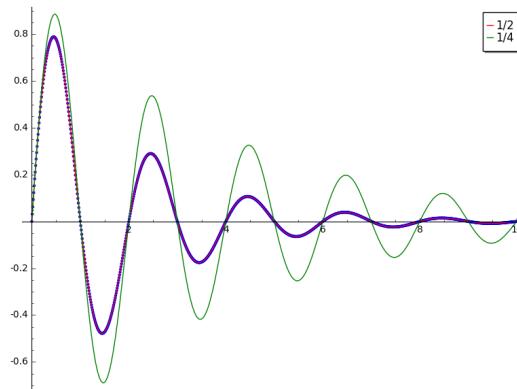


図 1.13 グラフィックス・オブジェクトの和の例

このように容易にグラフの重ね合わせができます。

1.2.13 画像の簡単な処理

ここでは SageMath で画像の簡単な処理を行ってみましょう。そのために Matplotlib を使ってみましょう。この Matplotlib は商用の数値行列処理システム MATLAB を強く意識した NumPy 上で構築された Python のパッケージで、数値配列の処理やグラフ表示で MATLAB に類似した処理が行えることが大きな特徴です。ところで画像の処理のために必要な函数が最初から読み込まれていないために幾つかのモジュールを読み込んでおく必要がありますが、SageMath の基本的な操作やプログラミングの骨子は Python そのもので、Python と同様にパッケージやモジュールの読み込みを `import` 文で行います：

```
sage: import matplotlib.pyplot as plt
sage: from matplotlib import image as mi
sage: imat=mi.imread('/Users/yokotahiroshi/Documents/ScuolaDiAtene.png')
sage: type(imat)
<type 'numpy.ndarray'>
sage: imat.shape
(509, 800, 3)
sage: matrix_plot(imat).show()
```

この例では配列の描画用にパッケージ Matplotlib のモジュール `pyplot` を ‘`plt`’、画像読み込み用のモジュール `image` を ‘`mi`’ と読み替えて SageMath に読み込みます。`pyplot` と `image` の読み込み方に違いがありますが、これらはパッケージを構成するモジュールの呼び出し方

の例です。これらは Python のパッケージ/モジュールの階層構造に関係します。Python ではパッケージ、モジュール等の階層構造を表現するために区切文字として記号 “.” を用い、左側に上層、下側に下層を記載します。pyplot の場合は上位が matplotlib であるためにパッケージ内部の階層を含めた表記は matplotlib.pyplot になります。これで pyplot の所在が分かるために import 文でそのまま表記すれば良いことになります。もう一つは Matplotlib の配下からモジュールを取り出す方法で、どこから取り出すかを明示するために from 節を使って ‘from matolotlib’ とし、その下にあるモジュール image を読み込みます。ここでの例では共に as 節を用いてモジュールの読み替えを行っていますが、これは読み込んだモジュールの函数や属性の呼出を簡易に済ますための方法です。実際、画像の読み込みで ‘mi.imread()’ としていますが、これはモジュール image に含まれた函数 imread() を用いるという意味で、本来なら ‘matplotlib.image.imread()’ としなければならないところを ‘matplotlib.image’ の箇所を ‘mi’ で読み替えて ‘mi.imread()’ としています。さて、画像は指定のディレクトリ内にある ‘ScuolaDiAtene.png’ ファイルです。この画像を函数 imread() で読み込んでいます。この函数 imread() で読み込まれた画像データが束縛された変数 imat を函数 type() を使って調べると ‘numpy.ndarray’、つまり、NumPy の多次元配列として読み込まれていることが判ります。この数値配列の大きさはメソッド shape() で調べられますが、ここでは ‘(509, 800, 3)’ と返却されています。この結果の最初の二つの整数値が画像の縦と横の画素数で、最後の ‘3’ が Red, Green, Blue の RGB に対応します。この数値行列の表示は SageMath の函数 matrix_plot() で行えます。この表示で SageMath を仮想端末で利用していれば画像表示のためのウィンドウが開かれ、Jupyter を利用していれば図 1.14 のようにノートブック側に表示されます：

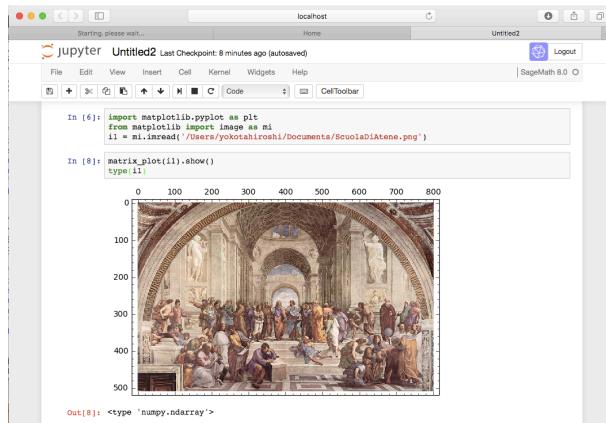


図 1.14 読込画像の表示例

次に RGB 画像の輝度のヒストグラムを赤、緑と青で出力してみましょう。ヒストグラ

ムのグラフ出力は SageMath の関数 `histogram()` でできます。この関数 `histogram()` は 1 次元数値配列のヒストグラムを描きますが、画像は 2 次元です。そこで画像データを NumPy の多次元配列のメソッド `reshape()` を使って 1 次元化し、ヒストグラムの棒を 16 本にして描きます：

```
sage: hR = histogram((i1[:, :, 0]).reshape(509*800), bins=16, color="red")
sage: hG = histogram((i1[:, :, 1]).reshape(509*800), bins=16, color="green")
sage: hB = histogram((i1[:, :, 2]).reshape(509*800), bins=16, color="blue")
sage: hR.show()
sage: hG.show()
sage: hB.show()
sage: (hR + hG + hB).show()
```

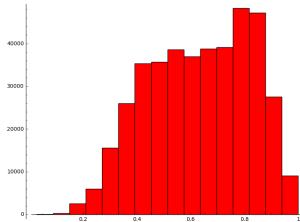


図 1.15 赤のヒストグラム

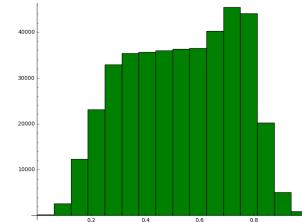


図 1.16 緑のヒストグラム

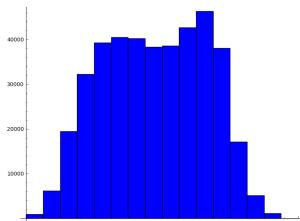


図 1.17 青のヒストグラム

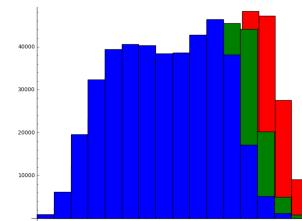


図 1.18 RGB のヒストグラム

ここでは関数 `histogram()` が output するグラフィックス・オブジェクトを `hR`, `hG`, `hB` に割り当て、それらをメソッド `show()` を使って表示させています。グラフィックス・オブジェクトであれば演算子 “+” を使って重ね描きを生成することもできます。このヒストグラムを Matplotlib の `plot` パッケージの関数 `hist()` で描くこともできますが、仮想端末であれば `plot` パッケージの関数 `savefig()` を使って画像出力したものを別のアプリケーションで表示するか、Jupyter を使っているのであれば、あらかじめ ‘%matplotlib inline’ を評価させておくとノートブックに画像が貼り付けられます。

このヒストグラムの例で `imat[:, :, 0]`, `imat[:, :, 1]`, `imat[:, :, 2]` という表記がありました。ここでの 0, 1, 2 は赤 (R), 緑 (G) と青 (B) の輝度に対応する配列であることを指示するも

のですが、他の添字記号 ‘:’ は、その添字記号が置かれた場所で添字が取り得る値の全てを意味しする MATLAB 系言語を特徴づける構文で、Python でスライス・オブジェクトと呼ばれるオブジェクトです。この構文は配列の i から j までの $j - i$ 個の添字で指示される成分を配列として取り出すための構文です。MATLAB では添字が 1 から開始するために、Python の ‘`a[i:j]`’ と同値な MATLAB の表記は ‘`a(i+1:j)`’ になります。ここで添字の増分が 1 に固定されていますが、増分を 1 以外に設定するときは Python では ‘`10 : 0 : -1`’ のように ⟨始点⟩ : ⟨終点 + 1⟩ : ⟨増分⟩ と表記し、MATLAB 系の言語では ⟨始点⟩ : ⟨増分⟩ : ⟨終点⟩ と始点と終点の間に増分を挟みます。

ここで SageMath に取り込んだ画像データ `imat` は 509×800 の 3 個の数値配列データですが、では ‘`imat[200:300,300:500,0]`’ は何になるでしょうか？ここで画像の座標系は左隅を原点とし、縦下方向が Y 軸の正方向、横軸右方向が X 軸の正方向になるために ‘`imat[200:300,300:500,0]`’ の意味は Y 軸座標が 200 から 299、X 軸座標が 300 から 499 で表現される短冊で、‘0’ ということは RGB の赤を指示することから図 1.19 に示す赤の輝度を示す画像になります：

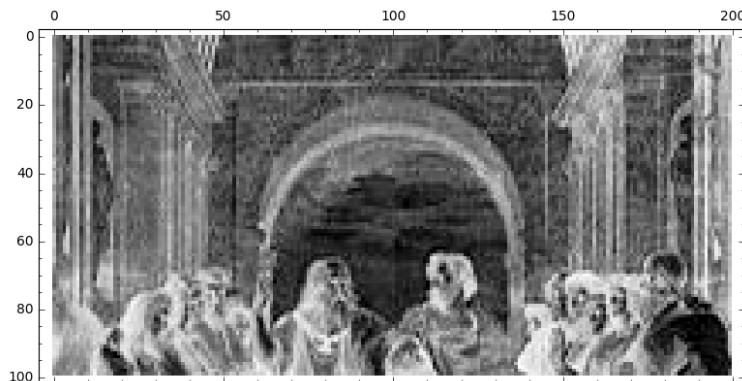


図 1.19 画像の一部切り取り

この切り取りでは座標と配列の添字との対応関係を利用していますが、必要な領域の切り抜きも、この手法の応用で容易に行えます。

いかがでしょうか？このように SageMath はノートブック環境を持った Python 環境として非常に気楽に使えます。そして、まっさらな環境にさまざまなパッケージを取捨選択しながら導入する必要もなしに、ただ、SageMath だけを入れてしまえば本格的な数式処理や統計処理も行えるノートブック形式フロントエンドを持つ Python 環境が導入できます。

第2章

オブジェクト指向について

Heil dir, Sonne!
Heil dir, Licht!
Heil dir, leuchtender Tag!
Lang war mein Schlaf;
ich bin erwacht.
Wer ist der Held, der mich erweckt'?

2.1 SageMath の中核としての Python

SageMath は既存の数学アプリケーションを Python で繋ぎ合せて創り上げた数学のための統合環境です。したがって、数学に関するアプリケーションを使う必要があれば是非とも使い倒すべきシステムですが、そのためには Python がどのような言語であるかを熟知する必要があります。ところで「Python はどのような言語なのか?」という問い合わせに対しては Google 等の検索エンジンや Wikipedia で調べればおおよそが判ります。実際、Wikipedia には作者がオランダ人のグイド・ヴァンロッサム (Guido van Rossum)^{*1} が作った OSS (Open Source Software) であり、オブジェクト指向プログラミングに対応した言語であること、加えて BBC 制作のコメディ番組「空飛ぶモンティ・パイソン」への言及もあります。さらに記事を読み進めてゆくとプログラマの生産性とコードの信頼性を重視した設計、核となる構文や文法を必要最小限に抑えていること、そして大規模な標準ライブラリがあることが書いてあります。この他の Python には文化的な側面もあります。たとえば Python それ自体の開発にはコミュニティの存在が前提にあり、コミュニティでの議論を反映した「PEP」と呼ばれる文書を基に開発が進められている点が挙げられます^{*2}。

このように Python という言語がオブジェクト指向プログラミング言語と呼ばれる言語であることが判りましたが、では、「オブジェクト指向プログラミング (Object Oriented Programming)」はどのようなものでしょうか? そこで再度、Wikipedia で調べてみると「相互にメッセージを送りあうオブジェクトの集まりとしてプログラムを構成する技法」と最初にまとめていますが、この一節だけでも「オブジェクトって何?」、「メッセージとは何?」といった疑問が出てきます。そこで「オブジェクト」を「計算機上で扱うべき対象を抽象化したもの」、「メッセージ」を「個々のオブジェクトを結びつける関係」と暫定的に定義し、「抽象化」と「関係」はあとで考察しましょう。すると疑問の残りは「どのような技法?」になりますが、ここで述べている「技法」には「クラスに基くもの (class based)」と「プロトタイプに基くもの (prototype based)」の二種類があります。前者のクラスに基くものは扱うべき対象を「クラス」と呼ばれる対象(オブジェクト)として表現し、実際の処理では、その処理すべきデータを、それを表現するクラスが具現化した「インスタンス」と呼ばれるオブジェクトとして処理を行います。このクラスには属性値と呼ばれる固有の値とメソッドと呼ばれるインスタンスに作用できる手続を備

^{*1} カレル・チャベックの戯曲「ロボット (R.U.R.)」は Rossom's Universal Robot(ロッサム汎用ロボット)です。

^{*2} ちなみに開発者のヴァンロッサムは「慈悲部深き終身独裁者 (Benevolent Dictator for Life, BDFL)」として Python の開発を総覧しています。

えており、これらでオブジェクト間の関係が与えられます。さらにクラスには親子関係に類似した階層構造が入り、下位のクラスで上位の属性やメソッドを継承と呼ばれる機能を利用ができます。このような言語に Python の他に Java や Ruby があります。後者のプロトタイプに基く言語ではクラスを構築しませんが、既存のオブジェクトをプログラムで実際に処理するインスタンスの雛型として用います。このプロトタイプに基づく言語には JavaScript や Lua があります。これらの技法をたとえるなら、ある証明書を作成するときに、その証明書が何であり、どのような書式であるかを決定し、その雛形を作成しておく方法がクラスに基く方法、似たものを探し出して複製を生成し、その複製を適宜改変して再利用する方法がプロトタイプに基く方法にたとえられるでしょう。では、残りの「抽象化」と「関係」はどのようなものでしょうか？また、その根底に潜んでいる動機、意図や概念はどのようなものでしょうか？そこで、この章では Python の言語仕様ではなく、Python の基礎にあるクラスに基づくオブジェクト指向プログラミングがどのようなものであるかを語ろうと思います。

2.2 オブジェクト指向プログラミングの哲学的側面

2.2.1 プラトンのイデア論

クラスに基づくオブジェクト指向プログラミングの説明で、何かとプラトン (Πλάτων, Plato) *3 の「イデア論 (Theory of Forms)」が引っ張り出されます。このイデア論では我々が考察の対象になる現実のモノ、つまり、「個体 (individual)」には「思惟によってのみ知られる世界」、すなわちイデア界に「イデア (ἰδέα, idea)」が存在して個体はそのイデアの像であると主張しています。だから、あなたのそばにいる三毛猫の「みけ」には対応する「三毛猫のイデア」が「イデア界」に存在し、そのイデアの現世の像が「みけ」となります。そして、イデアは思惟によってのみ知覚可能で、さらには「永遠不滅」という超越的な性質を持っています。つまり、イデアは現実にある対象を「理想化したもの」で、ちょうど「鋳型」に対応しますが、プラトンはイデア界こそが真実の世界で、現世はイデアが投影された「影の世界」、「模倣物 (εἰκόνη) の世界」と見なしています (c.f. 「洞窟の比喩」 [25]) *4。これをオブジェクト指向プログラミングに当て嵌めると、まず、クラスがイデアに対応し、計算機で扱うデータは対応するクラスが計算機内部で「実体化したもの」と説明できます。ちなみにクラスが計算機上のデータとして「実体化」することを「インスタンス化 (instantiation)」、そして、「実体化したクラス」を「インスタンス (instance)」と呼びます。ちなみに「イデアの現世における実体化」も英語では同

*3 体格が良くて肩幅が広かった (πλάτυς) ことに由来する渾名です。

*4 「こんなにまずい家の普請を誰がした！」と言いたいところですが、現世の否定的側面をことさら強調するトグノーシス主義になります。

じ「instantiation」で、このようにクラスとインスタンスの関係はイデアと個体の関係に類似しています。ところでイデアを実体化したのが誰で、その理由はプラトンによると「デーミウールゴス (*δημιουργός*, demiurge)」がイデアを実体化した張本人で、イデアを模倣して世界を創世した理由は貧欲な神「エロース (*"Ερως*, Eros)」がイデアの美に憧れたためと主張していますが納得できるものではないでしょう。また、イデアは美や善に関わるものであって、醜いものや悪にイデアは存在しないと述べていますが、そうであれば「何が美なのかをヒキガエルに聞いてみろ!」とヴォルテール (Voltaire) ならずとも言いたくなるでしょう。

このような「機械仕掛けの神 (Deus ex machina)」^{*5}を持ち出されても信じるしかないところは哲学であるよりもむしろ宗教であり、実際、イデア論はヘレニズム世界のさまざまな宗教に大きな影響を及ぼします。まず、プラトンのイデア論を基に超越的な「一者 (*το εν, to hen*)」からの流出による世界の創造というプロティノスの「流出説」を取り入れた「新プラトン主義」^{*6} からデミウールゴスによる惡しき世界の創造、肉体という牢獄に囚われ星辰の支配を受け、死後に神への魂の帰一を柱とする「グノーシス主義 (*Γνωσις*)」に繋がります。このグノーシス主義に関してはヘルメス・トリスマギストス (三重に偉大なヘルメス, Hermes Trismegistus, *Ἑρμῆς ὁ Τρισμέγιστος*) が記したとされる「ヘルメス文書」と呼ばれる一群の文書があります。ここでのヘルメスはギリシャ神話の神ヘルメスとエジプト神話の神トート (*Θώρ*)^{*7}がヘレニズム時代に、おそらくエジプトで融合したと考えられ、鍊金術では「賢者の石」^{*8}を実際に手にした人物^{*9}とされています。そのヘルメス文書の一つの「ポイマンドレース (Poimandres)」[14]によると人間は



図 2.1 ヘルメス・トリスマギストス

^{*5} 古代ギリシャ・ローマ悲劇で收拾がつかなくなってしまった話を解決するためにいきなり神を登場させることです。たとえば、ソフォクレース (*Σοφοκλής*) の「ピロクテーテース (*Φίλοκτήτης*)」では終盤にヘーラクレース (*Ηρακλῆς*) が現れて入った話を一刀両断で解決してしまいます。とはいえ、この物語の結末は当時の観客にとって既知のことと、水戸黄門の「葵の印籠」を待つような心情だったのかもしれません。

^{*6} これは後世の呼び名で、創始者のプロティノスと信奉者達はプラトンの思想そのものと思っていました。

^{*7} 頭がトキ (ibis) の神様です。

^{*8} 鍊金術師が探し求めた究極の薬草で、鉄などの非貴金属を貴金属の金に変え、人間を不老不死にします。

^{*9} 図 2.1 の恰好の人物をどこかで見たことがありませんか? MIT の SICP (Structure and Interpretation of Computer Programs) の扉絵の人物に似てます。つまり、 λ -函数概念は計算機科学の「賢者の石」で「A ニシテ Ω 」です!

元来、美しい神の似姿として創られた神の子で、あるとき彼は高次で純粋な天上界^{*10}から地上へと下降します。その際に通過した恒星、土星、木星等の星辰の支配を受けることになり、たどり着いた地上にてフュシス ($\varphiύσις$, 物質) 内に写った自分の姿に恋してフュシスとの愛欲に陥り、「**フュシスは愛する者を捕へ、全身で抱きしめて互に交わった**」結果、人間はフュシスに捕えられて肉体を牢獄とする存在になったと主張しています。この伝説^{*11}は人間の本質が神の似姿のために不死であるものの、星辰に支配されて消滅する肉体に囚われた存在であるという二面性を説明すると同時にオリエント諸国からの占星術の影響とイデア論を中心とした哲学が秘儀化して宗教へと変じてゆくありさまが刻印されています。実際、プラトンの「饗宴」等に見られる対話で人々を正しさへと導こうとするソクラテスから、「ポイマンドレース」の自説への反駁を一切許さない高圧的なヘルメスとの違いにも、その雰囲気が伺えます。ところで、この世はデーミウールゴスが誤って創造したものだという厭世的な観点は新プラトン主義はもちろんのこと、キリスト教主流派からも反駁されます^{*12}。それどころか本質的に默示的な宗教であったキリスト教は、徐々に合理的な宗教へと変貌します^{*13}。この変貌は教父と呼ばれるキリスト教神学者によるもので、特に青年時代にマニ教徒^{*14}であった教父アウグスティヌス (Augustinus Hippoensis, Augustine of Hippo) が新プラトン主義をキリスト教神学の理論付けに用いたことが大きく影響しています。このようにキリスト教と古代の間には大きな断絶がある一方で、地球中心の同心円で階層的な宇宙観、星辰信仰やイシス信仰をマリア崇敬として引継ぐ等、直接、あるいはイスラム文化を介した間接的な方法等でヘレニズム文明の遺産を引き継いでいます。

さて、本筋に話を戻すとプラトンのイデア論はオブジェクト指向プログラミングのクラスとインスタンスの双方の関係に類似がみられる程度です。実際、真っ新たなシステムで「**三毛猫!**」と唱えれば完全無欠な三毛猫のクラスが我等のシステム上に降臨する訳でもなく、クラス自体も「**神聖ニシテ侵スヘキアラス**」な超越的な代物ではなく、現実の対象から抽出されるべきものです。そして、我々が扱う対象やクラスは「**どのようなものであるかを語ることができるもの**」でなければなりませんが、このようなものに「**概念**」があります。そこで概念が何であるかを語りましょう。

^{*10} この宇宙観は同心円状階層構造を有する天動説です。

^{*11} おおよそ宗教や宗教的な代物は「**伝説**」を続々と生成するものです。現在でもカトリックでは列聖で、共産主義は英雄という形でその宗教の聖者とそれにまつわる伝説を生産するという有様です。そして新たに伝説や聖人達を量産することを止めて博物館や図書館で安心して閲覧できるようになった時点が**宗教の死**です。

^{*12} 全知全能の神が半端なことをする筈がないという反論です。

^{*13} その際にグノーシスの影響にあった教義の排除が行われています。たとえば「ユダの福音書」等を含むナグ・ハマディ文書は瓶に入れて洞窟に埋められています。とは言え、表面からは消えてもその痕跡は伏流として残り、10世紀のカタリ派のように再び表舞台に現れることがあります。

^{*14} マニ教 (摩尼教, Manichaeism) はマニ ($Μάνης$, Mani) が開祖のグノーシス主義の世界的宗教です。

2.2.2 概念について

プラトンのイデアは前述のように「**思惟にのみによって知覚されるもの**」で「**永遠不滅の存在**」です。こういった代物が個体と別に実在し、その上、「**思惟で知覚できる**」ように努力しなければならないという状況は人間の手に余る状況です。このイデアのように思惟によって知覚されるものに「**概念 (concept)**」があり、「**それが何であるか?**」や「**それがどのようなものであるか?**」という問に対する回答から特徴付ける形や色や機能等をまとめたものです。つまり、概念は対象を特徴付ける「**微表**」から「**属性**」を抽出し、これらの属性を共通性で纏めることから得られます。このように概念はイデアのように超越的ではなく、人間が認知し得る対象の形や色といった具体的な特徴、つまり、「**形相 (εἶδος)**」から出発し、我々が対象をどのように語るかということ、つまり、「**説明規定 (λόγος, 口ゴス, account)**」です。

概念は「**名辞 (term)**」としても現れます。名辞は概念が載る器であって概念そのものではありません。我々と無関係にどこかに実在して永遠不滅であるという超越的存在のイデアと異なって、概念は人間が認知し得る具体的な事物から出発し、その対象を我々がどのように語るかという説明規定であるために、対象への理解が深まれば語られるこの内容も深まるという性格を持ちます^{*15}。この「**それが何であるか**」と「**それがどのようなものであるか**」といった問い合わせにより深く考察した人物がアリストテレス (*Αριστοτέλης*, Aristotle) です。



図 2.2 アテネの学堂より: プラトンとアリストテレス

トテレスは地上 (形相=具象) を示すという形で両者の思索の方向性の違いが表現されて

このアリストテレスとプラトンの思索の方向性の違いを判り易く図示したものにラファエロ・サンティ (Raffaello Santi) の有名な絵画「**アテナイの学堂**」[53]^{*16}が挙げられます。図 2.2 に示すようにプラトンは天上 (イデア=抽象) を指し、アリストテレスは地上 (形相=具象) を示すという形で両者の思索の方向性の違いが表現されて

^{*15} 創世記で混沌の中で漂っていた「**神の靈**」が λόγος とギリシャ語に翻訳されていたことは興味深いことです。

^{*16} イタリア・ルネサンスにて前述のヘルメス文書の幾つかとプラトンの全集がフィチーノ (Fichino) によってラテン語に翻訳され、これらは「**古代神学**」と呼ばれます。これを契機に新プラトン主義は時代を越えて大きな影響をルネサンスに与え、「アテナイの学堂」もその影響を受けた作品の一つです。

います。すなわち、イデアは天界に存在する超越的な存在であるのに対し、概念は地上の個体の徵表から取り出されることです^{*17}。

とは言え、イデアが天界に安住する存在で、概念が地べたを這い回る代物という意味ではありません。まず、概念は複数の主語の述語になり得るという性質を持ちます。ここで複数の主語の述語になりうる性質を「普遍」と呼びます^{*18}。具体的には「猫」という「概念」は、その辺にいる「みけ」や「たま」、その他の貴方の周りで見掛ける野良猫 x についても「 x は猫である」という命題が作られるために「猫」は「普遍」ですが、「みけ」や「たま」は個体に強く結びついているために「これがたまです」のように個体を特定するだけで複数の主語を取り得るという意味の普遍ではありません。この「みけ」のように個体に結び付けられた概念を「個体概念」と呼びます。ちなみに、いろいろなものを取り替えて使える道具の名前で「ユニバーサル」を冠する理由は、このように主語を取り替えられる性質に擬したもののです。

ところで「猫」には「三毛猫」、「黒猫」、「白猫」、「虎猫」といった「猫」がいます。これらは「猫」の毛並について述べたもので、より詳細に「猫」を説明しようとする意図があります。このように概念には「類似する個体とまとめてより包括的に説明しようとする概念」、すなわち、「個体から離れた側の概念」、それから逆に「個体をより詳しく説明しようとする概念」、すなわち、「個体に近い側の概念」の二種類があることが判ります。そして、「対象を類似する対象も含めて包括的に語ろうとする概念」は「個体をより詳しく説明する概念」を包含します。このように一つの対象を語る二つの概念があり、一方の概念が他方の概念を包含するときに包含する側の概念を「上位概念」と呼び、もう一方の個体をより詳しく語ろうとする概念を「下位概念」と呼びます。これらの二つの概念をその普遍性で比較すると上位概念がより普遍です。たとえば、「三毛猫は猫である」という命題で、「猫」が上位概念、「三毛猫」が下位概念になります。「みけは猫である」「みけは三毛猫である」という命題に対して、「猫」と「三毛猫」の二つの概念を比較すると、より詳細に個体の「みけ」を説明している概念が下位概念の「三毛猫」です。実際、「三毛猫」は「猫である」ことに加えて「毛の色が黒・茶・白の三色である」といった三毛猫の特徴が述べられているためです。さらに上位概念のことを「類概念」、あるいは「類 (genus)」、下位概念を「種概念」、あるいは「種 (species)」と呼びます^{*19}。先程の「猫」で解説するならば「三毛猫の類概念」が「猫」、「三毛猫」が「猫の種概念」になります。そして種の違い

*17 ここから神的なものは天界にあるという同心円状の階層を有する天動説に基づく宇宙観が伺えます。

*18

*19 類と種の関係を上位概念と下位概念として述べていますが、「種類」という言葉があるように類 (genus) と種 (species) は分類学では属 (genus) と種 (species) に対応し、種は類の直下の概念としての性格があります。

を示す徵表(特徴)を「種差」と呼びます。たとえば先程の「三毛猫」、「虎猫」、…の例では「毛並」の違いが種差です。それから「上位」と「下位」の意味はどちらがより普遍的であるかに対応し、より普遍的な概念である上位概念は下位概念を包含します。そして、概念にはその上限と下限があり、上限になる最上位の概念を「範疇(カテゴリー, Category)」、下限になる最下位の概念を「単独概念」、あるいは「個体概念」と呼びます。この個体概念は個体を直接指示する個体に最も近い概念で、それに対して範疇は個体を含む概念の中で最も普遍的な概念です^{*20}。このように概念は対象を説明する一方で普遍性を目指す性質を持ちます。この概念の階層構造をアリストテレスは「範疇(カテゴリー)論」等の著作で述べおり、イデア論に批判的なアリストテレスも、プラトンと方向性が逆でも、同じことを主張しているように思え、イデア論が秘儀と化した新プラトン主義の哲学者達もアリストテレスを「師の思想を秘匿するために批判していた」と捉えています。そして、新プラトン主義の創始者であるプロティノスの弟子であるポルフュリオス(Πορφύριος, Porphyry of Tyre)は「手引(エイサゴーゲー, Eἰσαγωγή, Isagoge[37])」^{*21}でプラトンとアリストテレスの思想を矛盾なく結び付けることに成功しますが、この「手引」は(新プラトン主義)哲学を学ぶにあたって最初に読むべき本とされます。さらに「手引」はギリシャ語が判る「最後のローマ人」と呼ばれたボエティウスによってラテン語に翻訳され、それが西ローマ帝国崩壊後に西ヨーロッパに残された数少ない哲学書のひとつになって西ヨーロッパの哲学に大きな影響を及ぼすことになります。ちなみにアリストテレスの哲学の西ヨーロッパでの本格的な受容は12世紀末から13世紀初頭にかけてイスラム文化圏を経由して行われますが、イスラム文化圏でのアリストテレスの哲学の受容はプロティノスの著書「エンネアデス」^{*22}の影響で新プラトン主義化したものでした。この解釈で代表的な哲学者がスコラ哲学でアビケンナ(Avicenna)として知られるイブン・スィーナ(ibn Sīnā)で、新プラトン主義的な夾雜物を排した「純正アリストテレス」を標榜したアベロエス(Averroes, 註釈者)として知られるイブン・ルシュド(ibn rušd)がそのうちに現れます、西ヨーロッパへの伝播では逆にアベロエスがやや早く、アビケンナがそれに続き、これらの哲学者の影響でスコラ哲学は新しい局面を迎えることになります[32]。

さて、この「手引」によると「ものごとを語る」ことには「類」、「種」と「種差」に加えて「特有性」と「偶有性」があると述べています。まず、「類(genus)」と「種(species)」をアリストテレスは「γένος」と「εἶδος」と呼んでおり、これらの語源には「形」という意味があります。のことから概念は対象の形に依存するものであったことが伺え、類が種

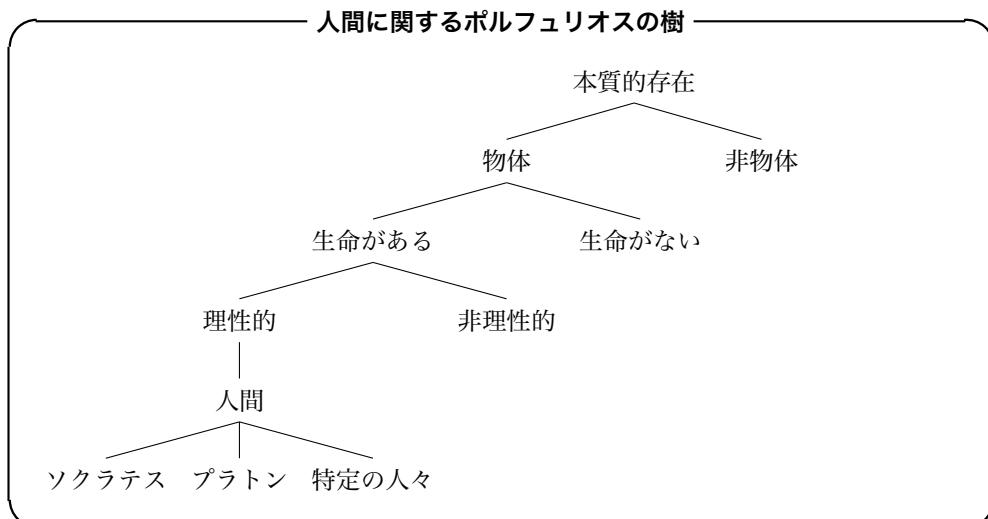
^{*20} アプリケーションのメニューで最上段が「カテゴリー」という名称で分類されている理由です。

^{*21} 英訳はボエティウス(Boethius)のラテン語訳(イサゴーゲー, Isagoge)をオーエン(Owen)が翻訳したもの[47]とバーンズ(Barnes)がギリシャ語文献から翻訳したもの[37]があり、前者はエイサゴーゲーが範疇論の入門書という古来からの立場、後者がアリストテレスの論理学全体の入門書としての立場で、さらに詳細な解説があります。

^{*22} 「手引」の著者であるポルフュリオスがプロティノスの遺稿を編集したものです。

の上位概念、種は類の下位概念に対応し、共に「それが何であるか?」という問への回答になります。つぎに「種差」、「特有性」と「偶有性」は「それがどのようなものであるか?」という問への回答になります。具体的には「種差」は「種を特徴付ける属性」で、「動物」を類とする「人間」という種が他の「猫」等の動物との違いは「理性を持つこと」で、これが人間の種差です。また、「特有性」は「それが何であるかを語るものではないがそれを指示できるもの」で、たとえば、「下町のナポレオン」や「貴志駅の猫駅長」といった個体が持つ属性です。それに対して「偶有性」は「その程度を語ることができるもの」、たとえば、日焼けした子供を「薄く日焼けしている」、「よく日焼けしている」と日焼けの程度が表現可能であり、さらには「日焼けしていない」と日焼けしているという属性を持たないという状況も考えられる属性で、他の属性と違って形や模様、あるいは量の把握ができるもの、つまり、「感覚的に把握できる属性」です。さらに感覚され得るものは変化するものであるため、偶有性は永遠不滅な属性ではなく、その都度、変化し得る属性です。

このポルフュリオスによる述語の「類」、「種」、「種差」、「特有性」と「偶有性」による分類はさまざまな分野に大きな影響を与えています。その一つに「ポルフュリオスの樹 (Arbor Porphyrianae)」があります：



ここでは人間にに関するポリフュリオスの樹の一例を示していますが、この樹形図の根元側の概念が上位概念、枝側の概念が下位概念です。この図はポリフュリオス本人ではなく「手引」の註釈者達が種で類を分類することを視覚化したもので、さまざまな分野で階層構造を示す「樹形図」のもとになっています。また、リンネ (Carl von Linné) による学名の命名方法は「二名法」と呼ばれる方法で、動物/植物が属する種とその種を包含する属に対して最初に属 (genus) のラテン語名、それから種 (species) のラテン語名を列記する方法です。たとえば、人類の学名は ‘Homo sapiens’ ですが、ここで属が Homo、種が sapiens

で、この方法は種による類の分類を線的に記述したものです。この二名法はオブジェクト指向プログラミングでクラス属性やメソッド、あるいはクラスとその直下のサブクラスの表記で用いられています。そして、全体を俯瞰するときに樹形図が用いられます。

2.2.3 定義すること

さて、我々は事物を抽象することで概念に辿りつきましたが、名辞 X について「Y を充足ものが X である」とも言える筈です。この操作を「(X を) 定義付ける」と言います。具体的には「定義付ける」ということには「タマは猫である」のように類や種で定義付ける「実体的定義」、あるいは「分析的定義」と呼ばれる方法、「点は平面上の平行でない二直線の交わりとして構成される」という点の定義のように対象がどのような条件で発生、あるいは成立するかを記述する「発生的定義」、または「総合的定義」と呼ばれる内包的な定義と外延を用いる「実例、または代表・典型を用いた定義」があります。ちなみにアリストテレスが創始者である逍遙学派の「定義」は類と種や種差を用いてその「説明規定」(*λόγος*, logos, account) を与えることです。

ところで、キュニコス(犬儒)派のアンティステネス(Αντισθένες, Antisthenes)は定義(*λόγος*)について「それが何であるかを説明するもの」と述べています。その一方で「一つの主語は一つの述語あるのみ」[2]と主張し、「馬は認めて馬性を認めない」と類や種といった概念を認めないと立場です。これと彼の弟子のディオデゲネス(Διογένης, Diogenes)がプラトンの「人間とは二本足で羽根のない動物である」という定義に対して羽を巣り取った鶏を持ち込んで「これがプラトンの人間だ!」と言った逸話を思い起こせば、説明規定を並べたところで個体そのものにならないとの主張と言えるでしょう。この立場に立脚するならクラスを定義することは妥当なことではなく、むしろ、クラスを持たずに既存のものを複製するプロトタイプに基くオブジェクト指向プログラミングが相応しいでしょう。

2.2.4 形相(Εἶδος)

イデア論への有名な反論が「第三の人間」です。これはイデアの存在を認める「人間自体」という人間の類としてのイデアと「ソクラテス」や「プラトン」といった個体のイデアが存在しなければなりません。すると、人間としての類似を示す尺度としての「人間のイデア」が必要で、これを「第三の人間」と呼びます。この第三の人間を認めると今度は「第三の人間」とその他のイデアに対しても類似の尺度になるイデアが存在しなければならず、以降、第四、第五、第六…の人間が存在しなければなりません。このように議論が收拾できないこと自体にイデア論に無理があるのではないかという反論です。

また理想的な人間として例えられるソクラテスにしても、乳児、少年、青年、壮年…といった過程を辿りますが、それぞれの瞬間にイデアがあり、それらのイデア同士の関係を含めると話が簡単になるどころか逆に複雑になります。また、種から芽が出てやがて木になり、それが老木になって倒れて腐るといった個体の生成、変化や運動、最後に消滅する理由がイデア論からは説明できません。結局、機械仕掛けの神を引っ張り出して創世神話を語ったり、生物の生殖の理由を説明しても、何気ない現象の説明には無理があります。この有様にアリストテレスも「形而上学」にて「物を数えようとする場合に、数が少なくては数えられないと思って、その数を増やして数えようとする者のごときである」とイデア論を批判しています^{*23}。

アリストテレスは師匠のプラトンと異なり、観察に立脚したより現実に則した考え方をしています。まず、アルストテレスの「形相 ($\epsilon\hat{\imath}\delta\omega\varsigma$, eidos)」はプラトンの「イデア ($\iota\delta\varepsilon\alpha$)」のような「個体から離れた存在 ($\chi\omega\rho\iota\sigma\tau\acute{a}$)」ではなく、現実の個体を「形相」とこれといった特性を持たない「質料 ($\beta\lambda\eta$)」との「結合体 ($\sigma\acute{u}\nu\omega\lambda\omega\nu$)」として捉え、形相こそが個体を個体たらしめる原因、つまり「形相因」という設計図とプログラム双方の働きをする要因として捉えています。これを木の種の話に戻すと、木としての形相が種(たね)の内部に存在し、その形相が結合体としての質料に働きかけることで木として育ち、成熟し、やがて形相が木から消えることで木としての特性を失って朽ちて質料に戻るという説明になります。このアリストテレスの考察を現在の科学と比べてどうかと言えば細かな点では怪しいかもしれません、現代の科学でも対象である個体が何であるか、どのような理由でその個体がそれ自体であるかを説明しようとするもので、この流儀はアリストテレスの考察に源流があることが判ります。それゆえにアリストテレスが「万学の祖」と呼ばれる所以です。また、この形相があるからこそ、金属の金が金であり、鉄が鉄であるとするなら、それらの形相に直接働きかけて鉄を金にすることができるという考えが鍊金術に繋り、鍊金術究極の目的がそれを可能にする「賢者の石」です。鍊金術が非常に長い時代にわたって行われていたこと、それにニュートン等の著名な科学者や哲学者が関係したものこのような自然観があつてのことです。

さて、この形相と質料を計算機上で考えるとそれなりに面白いことが判ります。まず、質料それ自体は何らの特性を持ちませんが、これをビットの列に、それから形相をデータ構造等の意味付けに対応付けることができるでしょう。すると計算機内部のデータは形相と質料の結合として表現されます。この形相因は時計をモデルにした機械論では何とも不明瞭なもので、それこそ「機械仕掛けの神」でも持ち出さなければ收拾がつきませんが、現代

^{*23} 形而上学 [2] 第一卷九章

のようにソフトウェアも含めて考慮すれば形相因は非常に説得力を持ちます。

2.2.5 普遍の実在

概念/イデアの存在は物理学の原理や数学の定理の方が先に存在し、それらを学者が発見すると考えるか、到達した概念から原理や定理が導出されると考えるかといった議論にも繋がります。ここで事物に先行して概念があると考える立場を「**プラトニズム (Platonism)**」、あるいはプラトンの「**実在論 (Realism)**」と呼びます。

では概念やイデアは実在するものでしょうか？最初に概念は実在の「みけ」から「猫」や「三毛猫」といった概念に到達するために「**事物のあと普遍**」と呼ばれる普遍です。一方のイデアは事物がイデアの像であることから事物に先立つことになるために「**事物の前の普遍**」と呼ばれる普遍になります。ここでプラトンのイデア論を認めてしまえば、イデアは個体から独立して存在しますが、「第三の人間」のような厄介な問題が生じます。また、アリストテレスは範疇論で類や種を第二の本質的な存在と呼んでいますが、この第二の本質的な存在について、それが存在するかどうかをアリストテレスは明確に述べていません。さらにポルフュリオスの「手引」の一節には

…類と種については - それらが存在するものかどうか、それらが実際にそのままの思考にだけ依存するものなのかどうか、もし、それらが存在するのであれば、それらは物体 (*σώμα, body*) を持つものなのか、それとも非物体のもの (*ἀσώματος, incorporeal*) なのか、そして、それらは離在可能 (*χωριστός, separable*) なものなのか、あるいは明瞭に知覚できるものの中にあって、それらに関わって存在するもののか - こういったことの議論を私は避けようと思います…

とあります。この「手引」をラテン語に翻訳したボエティウス (Boethius) は「手引」の註釈を二つ記しており、「**第二注釈**」が西ヨーロッパ中世のスコラ哲学で「**普遍論争**」を引き起すことになります。この普遍の実在が問題となった背景に、アリストテレスが創始し、そのうちに発展した伝統的形式論理学で扱う命題に「**存在含意 (external import)**」と呼ばれる条件が付随していることが関係します。この存在含意は命題の主語が実際に存在しているという暗黙の条件で、このことはアリストテレスが用いた古代ギリシア語が属する印欧語族では ‘A = B’ という命題にて、その主語 A と述語 B の関係として表現する「**繋辞 (copula)**」として主語 A が存在する意味が付随する「**存在動詞**」と呼ばれる動詞が用いられることが多い関係しているでしょう。たとえば日本語の「**A は B である**」^{*24}を印欧語族の一つである英語で「A is B」と置換したときに日本語の「は」は A と B が

^{*24} 「A は B である」という命題に「ある」が何気に含まれていることに、このような用語を作り定着させた人々の何気ない凄さを私は感じます。

一致すること意味する以上の意味を持ちませんが, be 動詞は主語の A が存在するという意味が付随する「**存在動詞**」と呼ばれる動詞であるために「A = B」の意味だけではなく, むしろ, 「A が存在し, かつ, A = B である」の意味を持つ命題になります。たとえば, 前述のトマス・アクィナスの「形而上学叙説」[22] には

もし、其れに就いて肯定的命題が形成せられ得るならば、かかるものすべては有と呼ばれる。

と, ある命題が真であれば, その命題に存在含意が含まれているために命題の主語も存在すると主張しています。また, 後述の三段論法は本質的に命題の外延の包含関係を念頭に置いた推論であるため, 命題の主語が存在するかどうか不確かなものに対しては推論が行えません。

この存在含意は現代の論理学の創始者のフレーゲ (Frege) の概念記法で除外され, 現在の論理学にはありません。このことは「**神は全能である**」という命題が真であれば自動的に神の存在が保障された伝統的形式論理学と異なり, 命題の正しさと主語の存在性を別問題とする現代の形式論理学はある意味, 千年にも及ぶ神の実在から神の存在の不確実さを招来し, それはある意味, 「**神を殺した**」とも言えなくもないでしょう。

ところで中世のスコラ哲学の「**普遍論争**」と呼ばれる論争で争点になった「**普遍**」は注意が必要です。まず, イデアは「**事物の前の普遍**」, 実在の個体から抽出された類, 種差等の普遍は「**事後の普遍**」と呼ばれており, 前者についてはアリストテレスの批判もあって, その存在は否定的で, 後者は実物が存在することもあって, その実在については肯定的でした。そして, これら二つの普遍は普遍論争の争点にはならず, 問題になったのは「**存在における普遍**」, あるいは「**形而上学上の普遍**」と呼ばれる普遍です。たとえば, 「馬」を考えてみましょう。この「馬」の実体は牧場や動物園に個々の馬として存在します。では「馬」という概念はどうでしょうか? アリストテレスなら馬を種差を使って説明するでしょう。ところで, この馬の概念は個々の馬に共通する何かです。この何かを「**馬性**」と呼ぶことにしましょう。すると, この「馬性」は「馬」という概念とは異なった振る舞いを示します。実際, 「A は馬である」という主張があつても「A は馬性である」と主張しないために論理学上の普遍ではありません, ところが「馬」であるためには個々の馬に共通する特徴である「馬性」がなければなりません。したがって, この「馬性」と論理学上の「普遍性」が加わって初めて「馬」という概念が生じていると考えられ, このことから「馬性」と「馬という概念」は別物で, さらに「馬性」は類, 種差といった論理学上の普遍でもなく, 「馬という概念」の属性であつて, 偶有等に類似したものです。そのためには「**馬性**」

「馬性に他ならない」と主張するしかなくなります。この議論は前述のアビケンナが主張^{*25}したもので、この議論を計算機言語の「型 (type)」を使って「馬性 = 馬という型」とすると、この「馬という型」は「馬という概念」そのものとは異なっているものの、個々の馬に付属する「型」としか言いようがなく、「馬という型は馬という型に他ならない」という論点も何となくその雰囲気が分かることではないでしょうか。また、普遍論争で唯名論の代表者として挙げられるオッカム (Ockham)[15] は

概念把握された項辞およびそれらから構成される命題は、聖アウグスティヌスが言うところの“心のことば (verba mentalia)”に相当する。

と概念について述べており、アリストテレスや手引でほとんど触れられなかった概念を認識する心の働きを加えた考察になっています。このようにスコラ哲学は心の働きを含めた考察を行っており、このこともあって心理的な表記があり、この「**心理的な側面**」は19世紀のデーデキントの著書「数は何であり、何であるべきか」[20] で、数概念を説明する上で各個人の心理的な動機を含めた説明を行うことに繋がります。この数概念で現れる「**心理主義**」、それと数を単なる記号とみなす素朴（粗雑）な「**形式主義**」をフレーゲは非難し、彼自身が「**概念記法**」と名付けた言語を定め、その言語を用いて著書「概念記法」[26] で純粹に論理学上の対象として数を定義し、さらには「算術の基本法則」[27] で論理学から数学を構築するという「**論理主義**」を実際に遂行しています。

2.2.6 範疇 (Category)

個体が何であり、どのようなものであるかを語ること、すなわち、どのように述語付けられるかをアリストテレスは「**範疇 (カテゴリー)**」[1] で分類します。ここで範疇は最上位の概念であり、最も普遍的な概念であると述べましたが、この「**範疇**」に対応するギリシャ語のカテゴリアー ($\kappaατηγορία$) は法律用語の「**責を負わせる**」という意味のカテゴレイスタイル ($\kappaατηγορίεσται$) に由来し、実際に、それが何であり、どのようなものであるかを語るように責を負わされています。そして、「**A は B である**」という命題の述語 B を次の10種類の範疇に分類しています：

^{*25} ラテン語訳: Equinitas est Equinitas tantum.

アリストテレスによる範疇

1. まさにそれであるもの (本質的存在): 「人間」, 「猫」
2. どれだけか (量): 「128cm」
3. どのように (性質, 質): 「面白い」, 「文法的」
4. 何に対する (関係): 「二倍」, 「半分」, 「より大きい」, 「より小さい」
5. どこか (場所): 「千代田公園」, 「ペットショップ」
6. 何時か (時間): 「昨日」, 「去年」
7. 置かれている (態勢): 「寝転んでいる」, 「立っている」
8. 持っている (所有): 「靴を履いている」, 「首輪を付けている」
9. 作用する (能動): 「齧る」
10. 作用を受ける (受動): 「齧られる」

ここで「**本質的存在 (実体, οὐσία)**」は「**第二の本質的存在 (第二実体)**」と呼ばれ、「人間」, 「猫」, 「哲学者」等の主語にも述語になり得るもの, すなわち類や種になる性質を持ちます。ちなみに「**第一の本質的存在**」は「私」, 「みけ」, 「ソクラテス」等の個体により近くて普遍性を持たないもので, これらの本質的存在はギリシア語で「**ウーシア (οὐσία)**」と呼ばれ, 「**存在**」を意味する動詞 εἶναι を名詞化したのに由来し, 「**実体**」が訛語として当てられています。

このアリストテレスの述語の分類に対応するように, カント (Kant) は命題 (判断) を量, 質, 関係と様相の 4 纏目に分け, さらに各自を 3 項目に分けて 12 の範疇に分類しています:

カントによる判断の分類

量	单一性
	数多性
	全体性
質	实在性
	否定性
	制限性
関係	属性と実体性
	因果性 (原因と結果)
	交互性
様相	可能性 (不可能性)
	現実性 (非現実性)
	必然性 (偶然性)

述語の範疇で重要なことは、「**それが何であるか?**」や「**それがどのようなものであるか?**」という問に対する答は、ここで述べた範疇に分類されます。このように概念には類種(種差)による階層が入り、語られる内容も範疇で分類されることになります。そして、これらは我々がこれから考察しようとするオブジェクト指向プログラミングのクラスとその構造に深く関わります。また、判断に関する範疇については、三段論法で適用することができます。

2.2.7 内包と外延

「概念」を語る場合はまず「**それが何であるか?**」という問に対して我々はそれがどのようなものであるかを特徴を列挙するか、それに該当する個体を列挙するかどちらの方法になります。このように説明には二通りの方法があり、前者が「**内包**」、後者が「**外延**」と呼ばれる方法です。最初の「**内包**」は概念が持つ微表/属性から構成され、「**外延**」は概念が適用される個体等の対象の列記で構成されます。たとえば、「**猫**」という概念であれば、その内包は「**動物である**」、「**4本足で歩く**」、「**柔らかい肉球を持つ**」、「**ニヤオと鳴く**」等の属性(性質)から構成され、外延なら「**ペルシャ猫**」、「**シャム猫**」といった猫の種、「**黒猫**」、「**白猫**」、「**虎猫**」、「**三毛猫**」といった毛並で分類する方法、あるいは「**栗根さんのペットのタマ**」のように個体を列記する方法になるでしょう。このように内包は概念を説明する述語から、外延は概念に対応する具体的な個体や下位概念の列記から構成されます。そして、内包と外延には「**内包外延反比例増減の法則**」と呼ばれる関係があります。これは内包が増大すると外延が減少し、逆に外延が増加すれば内包が減少するという反比例の関係です。たとえば「**猫**」という概念に対して「**茶、黒、白の三色の毛並である**」という内包を追加すると「**三毛猫**」以外の「**白猫**」、「**黒猫**」等の猫が「**猫**」と「**茶、黒、白の三色の毛並**」の外延から消えてしまいますが、逆に「**三毛猫**」という外延に「**白猫**」という外延を追加すると「**茶、黒、白の三色の毛並である**」という内包が消えてしまいます。つまり、内包が増えるということは、それだけ述語付けられることで個体に近付く結果、外延を構成する個体が絞られ、逆に外延を構成する個体が増えると個体から離れて普遍的な事柄を抽出するために内包が減少するという関係です。つまり、上位概念とその下位概念とを外延で比較すると、より大きな外延を上位概念が持ち、下位概念では外延がより小さくなります。内包については下位概念が上位概念より詳細な記述になります。

外延で表現された概念は内包で説明規定することができますが、逆に内包で説明規定された概念は外延で表現できるとは限りません。さらに任意の命題が外延を持つとは限りません。たとえば ' $x \neq x$ ' という命題の外延は存在しません。これは発見者のイギリスの哲学者ラッセル(Russell)の名前から「**ラッセルの逆理**」と呼ばれる有名な逆理に対応する論理式です。一般にはラッセルが言い換えた「**床屋の逆理**」の名前で知られています：

——床屋の逆理——

とある村には床屋が一軒だけあります。その床屋の主人は自分で髪を剃らない人の髪だけを剃ると言っています。では、その床屋の主人の髪を誰が剃ればよいのでしょうか？

床屋の主人が女であったという与太話は除いて、この手の逆理は古来より「**クレタ人の逆理**」として知られていました：

——クレタ人の逆理——

クレタ人はうそつきである。

この命題をエジプト人やギリシャ人が主張したのであれば問題がありませんが、エピメニデス (*Ἐπιμενίδης*, Epimenides) というクレタ人^{*26}が主張したためにややこしくなっています。これらの逆理の本質は前述の論理式 ' $x \notin x$ ' で「**自分自身を元として持たないもの**」と自分を定義するために自己を引用する循環的な定義であることです。このようにラッセルの逆理は非常に単純な式ですがその効果は絶大で、ラッセルが書き上げたばかりの著作「Principles of Mathematics」[49] やフレーゲが独自の論理式のために嫌がる出版社を説得して二部に分けて出版した「算術の基本法則」[27] といった著作の成果を葬り去るに十分でした^{*27}。

ラッセルやフレーゲの論理主義^{*28}とカントール (Cantor) の(素朴)集合論^{*29}に批判的であったポアンカレ (Poincaré) は彼のエッセイ「科学と方法」[28] で幾つかの逆理を分析しています。たとえば「偶数の集合」や「身長 170cm 以下の人の集合」といった集合の定義では「自然数の集合」や「人間の集合」といった集合の概念に触れずに集合がきちんと定義ができます。このような定義方法を「**可述的**」と呼びますが、床屋の逆理のような循環論法に訴えなければ自分自身を定義できない定義を「**非可述的**」と呼び、ポアンカレは非可述的な定義に問題があると述べています ([28], p.204)。ただし、この非可述的な定義を全部を排してしまえば良いものではありません。たとえば、実数の連続性で「実数 \mathbb{R} の有限部分集合はその最小上界を持つ」はある対象を含む上界全体に言及しているながら、その対象を定義しているために非可述的です。そこでラッセルは「**型理論**」と「**悪循環原理**」を導入することで病的な非可述的な命題の排除に成功したものの、今度は数学的帰納法が使えないという重大な副作用が生じます。そこで、今度は「**還元可能性公理**」^{*30}

^{*26} 紀元前 6 世紀頃のクレタのクノッソスの学者とのことです。

^{*27} フレーゲは「算術の基本法則」のあとがきにこの逆理に対する悲痛なコメントを残しています。

^{*28} 論理学から数学を導出しようとする数学上の哲学です。この立場は最終的にはラッセルとホワイトヘッドの「Principia Mathematica」[50] で完成しています。この立場の成果はドイツの数学者ヒルベルト (Hilbert) の形式主義に引き継がれ、現在の数学の基礎の一つになっています。

^{*29} 素朴集合論とは命題の外延を集合とみなす立場の集合論です。

^{*30} 「任意の階の命題函数には、それと同値な可述的函数が存在する」, Principia Mathematica の表記を用

を導入すれば今度はその天下り的な性格が問題になるといったありさまでラッセルの試みが成功したとは言えません。なお、現在の集合論ではその公理系で「**集合**」を定め、それ以外の命題の外延を「**類**」、あるいは「**クラス**」と呼んで集合と区分し、「ラッセルの逆理」を集合論の体系から排除しています。

2.2.8 オブジェクト指向プログラミングにおけるクラスの表現

これらの考察を基にオブジェクト指向プログラミングを吟味してみましょう。まず、扱うべきデータが個体と考えるなら、データを抽象することで得られる概念に対応するクラスがあり、データはそのクラスが実体化したものとして捉えられます。ここでクラスは、「**それがどのようなものなのか**」という間にに対する属性で語られ、属性が何らかの値で表現されるのであればその値、機能であれば、それをメソッドとして表現することになります。たとえば、「猫」であれば「足の本数」、「尻尾の有無」、「体重」、「体長」や「月齢」といった特徴、それに加えて「柔らかい肉球を持つ」、「猫パンチで殴る」、「雨の前に顔を洗うような仕草をする」等の機能があるでしょう。すると、「猫」というクラスはこれらの猫の特徴(足の本数、尻尾の有無等々)を列記し、猫が持つ機能(「猫パンチ」、「忍び足」、「雨の前に顔を洗うような仕草」、「ネズミを掴まえる」等々)をメソッドとして列記します。そして、「みけ」は「猫」というクラスが実体化したもの、すなわち、インスタンスになります。このときにクラス間の関係はどのようになるでしょうか？概念では類と種といった階層があります。これに似たものとして次に述べる「**継承**」という機能があります。

2.2.9 継承

概念には階層があり、より普遍的な上位概念とより個体に近い下位概念があります。これらの概念を内包で書換えてしまうと下位概念の内包は上位概念の内包を基に上位概念に含まれない内包を付与したものとして表現できます。このことはある概念に新しい「属性」を与えることでその概念の「下位概念」が構築できることを意味します。この操作がオブジェクト指向プログラミングでの「**継承**」に該当します。

この継承という考えは非常に自然な考え方です。実際、ある新しい動物を発見したときに、その動物が何に属するといった系譜が創られるでしょう。ところで、その動物の調査が進むにつれて新しい知見が得られると旧来の分類を基にして新しい分類が行われるでしょう。これと同様に扱うべきデータをあるオブジェクトの実体化として記述したとしても、のちにデータの理解が深まることで、そのデータがより細かく分類されることはそう

いると $(\exists\varphi).\psi x. \equiv_x .\varphi!x.$ と記述され、「任意の命題には扱い易い言い換え(パラフレーズ)が存在する」という意味の都合の良い公理です。

珍しいことではありません。このことは最初に大きく分類したクラスをより下位のクラス、すなわち、サブクラスへとさらに細かく分割することに相当しますが、この細分化は上位のクラスにない値やメソッドを追加することで行われます。このことは最初のクラス構築が間違っていない限り、システムの大枠を変更することなしに自然に拡張が行えることを意味します。ただし、この継承を上手く行うためには系統立った分析が必要になることは言うまでもありません。この分析を誤れば継承が自然に行うことのできないシステムになりますかねません。ここで継承関係が一子相伝的な継承であれば、継承関係が直線的な関係になるために属性やメソッドが何処から引き継がれたか探すことが容易ですが、実際の継承は複数のクラスからの継承を含む複雑なものになるでしょう。それに加えて経済的な側面も考えなくてはなりません。実際、あまりにも複雑怪異な継承関係は扱う側にとっても不要な混乱を招くだけでなく、メソッドや属性の検索という観点からも不利になります。たとえば、クラスを小分けにし過ぎるとどうなるでしょうか？具体的には「猫」から個体の「みけ」に至るまでに「三毛猫」が間に一つだけの場合と、「アジアの猫」、「東アジアの猫」、「日本猫」、「三毛猫」が入る場合を考えてみましょう。ここで、飼い猫の「みけ」が持つ「猫の属性」や「猫の習性」を知りたくなったとき、最初の継承関係であれば「三毛猫」を間に一つ挟む程度で済むことが、後者の継承関係では「アジアの猫」、「東アジアの猫」と「日本猫」の三つのクラスがあつて検索の手間が増えています。このように細かくすればよいというものではありません。また、属性やメソッドの検索順位の定め方次第で新しいクラスの属性やメソッドが反映されなくなる懼れもあります。この問題については「C3 MRO」といった手法で改善が図られていますが、最初のクラスの分析が非常に重要なことは言うまでもありません。

2.3 判断と推論

2.3.1 判断

アリストテレスに始まる伝統的論理学は主語と述語の関係の考察がその中心にあります。伝統的論理学の命題は「主語」、すなわち「主辞」と「述語」、すなわち「賓辞」、あるいは「客語」、そして、これらを結びつける「繋辞」の三つで構成するために「名辞論理学」とも呼ばれます。なお、フレーゲから始まる現代の論理学は命題の真偽を基に命題の考察を行うために「命題論理学」と呼ばれます^{*31}。

ここで「命題の判断」とは、主辞（主語）と賓辞（述語）の持つ概念が一致するか不一致

*31 古代ギリシャのストア派の論理学も命題論理学でしたが、伝統的論理学と同様に「すべて」と「存在する」に対応する量化詞が欠落しています。量化詞は19世紀末にフレーゲが函数概念と同時に論理学に導入しています。

であるかを断定することです。この本では命題の主辞を S 、賓辞を P と表記し、命題を $S - P$ と表記します。ただし、表記 $S - P$ の記号 “-” には判断の種類が記入されます。ここで判断の種類はカントによって量、質、関係、様相の4つの範疇のグループに分類されて計12個の判断があります。

■量：主辞の外延の大きさに関する判断です。

量に関する判断

-
- | | |
|------|--------------------------|
| 全称判断 | ： S すべての S は P である |
| 特称判断 | ： ある S は P である |
| 単称判断 | ： S は P である |
-

単称判断は全称判断の特殊な例として考えられるため、実質的に全称判断と特称判断の二つです。

■質：肯定と否定に関する判断です。

質に関する判断

-
- | | |
|------|------------------|
| 肯定判断 | ： S は P である |
| 否定判断 | ： S は P でない |
| 無限判断 | ： S は非 P である |
-

ここで「非 P 」について説明しておきましょう。ある概念「 A 」に対して概念「非 A 」を概念「 A 」の「**矛盾概念**」と呼びます。たとえば、「動物」という類概念の中の種概念「猫」に対して「非猫」が猫以外の動物の種概念を指し、動物という類概念は「猫」と「非猫」という種概念に明確に二分されるため、「猫」と「非猫」の間に他の動物の種概念は存在しません。このようにある概念 A が包含される類で、概念 A と共に個体を一切持たない概念が矛盾概念「非 A 」です。この本では「非 A 」と記載する他に \bar{A} とも記載します。この質の判断で、無限判断は肯定判断の特殊な例として考えられるために肯定判断と否定判断の二種類が質の判断の代表として挙げられます。

■関係：主辞と賓辞の関係に関する判断です。

関係に関する判断

-
- | | |
|------|-----------------------------------|
| 断言判断 | ： S は P である |
| 仮言判断 | ： もし A が B であれば S は P である |
| 選言判断 | ： S は A であるか B であるのかどちらかである |
-

関係の判断は基本形として「 A ならば B 」、つまり、「 A は B である」の断言判断の形

式になります。そのため、この関係では断言判断をその代表とすることができます。

■**様相:** 命題の確実性に関する判断です。

様相に関する判断

実然判断 : S は P である

蓋然判断 : S は P に違いない

必然判断 : S は P でないはずがない

実然判断の「 S は P である」を判断の骨子として考えられるために実然判断をその代表にすることができます。

以上から判断の代表を纏めると「**A:全称肯定判断 (Universal Affirmative)**」, 「**E:全称否定判断 (Universal Negative)**」, 「**I:特称肯定判断 (Particular Affirmative)**」, 「**O:特称否定判断 (Particular Negative)**」に判断を纏められます。このことを以下にまとめておきましょう:

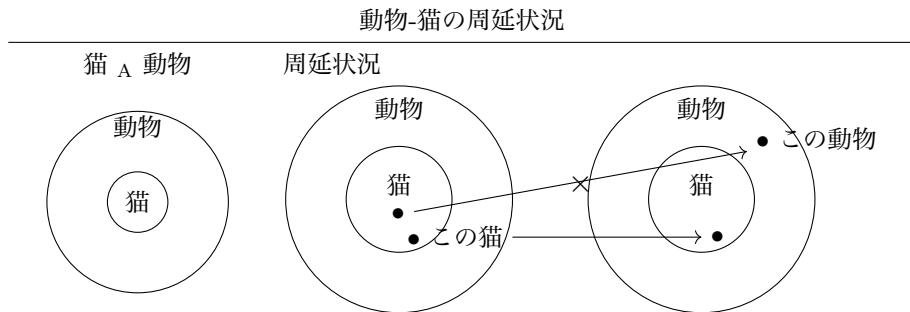
AEIO

- | | | |
|-------------|-----------------------|---------|
| (A) 全称肯定判断: | 「すべての S は P である」 | $S_A P$ |
| (E) 全称否定判断: | 「すべての S は P ではない」 | $S_E P$ |
| (I) 特称肯定判断: | 「ある S は P である」 | $S_I P$ |
| (O) 特称否定判断: | 「ある S は P ではない」 | $S_O P$ |

これら「**A**」, 「**E**」, 「**I**」, 「**O**」はラテン語の動詞AFFIRMO (私は肯定する) と NEGO (私は否定する) に由来する中世の論理学者が付けた略称です。そして, $S_A P$ 等の表記は、これら AEIO を表記する式の主辞と賓辞の周延の状況を示す表記です。

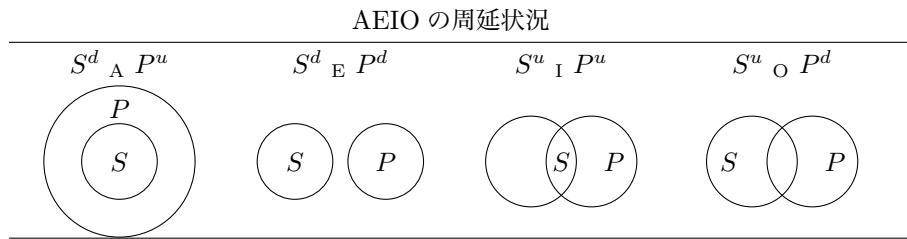
ここで「**周延 (distribution)**」は中世のスコラ哲学で発展した「**代表 (suppositio)**」に由来する概念で、主辞(主語) S に対応する概念が指示する個体と賓辞(述語) P に対応する概念が指示する個体との対応(指示)の関係の状況を示すものです。つまり、ある概念が「**周延**」されている状況は命題に起因する指示関係がその概念に属する全ての個体で成立するとき、すなわち、関係が全ての個体に対して「**分配**」されるとき、「**不周延**」である状況は、指示関係がその概念に属する個体全てで成立しないとき、すなわち、分配され得ないときです。このことを言い換えると、命題 $S - P$ を $\text{Pred}(S, P)$ と表記し、各概念に属する個体全てに函数 $\text{Pred}(\ , P)$ と $\text{Pred}(S, \)$ を分配したときに、これらを評価した値が常に真的ときが周延されている場合で、真であるとは限らないときが不周延の場合です。古典的な方法なら、主辞「 S 」と賓辞「 P 」に「**この**」という言葉を付加することで判断で

きます。たとえば、「すべての猫は動物である」という判断で、「この猫は動物である」と主張できるために「猫」は周延されていますが、「すべての猫はこの動物である」と一般的に主張できないために「動物」はこの命題に関しては不周延です。判断 $S - P$ において概念 S が周延されているときに S^d と表記し、そうでないときに S^u と表記します。この周延関係は図示も可能で、ベン図に類似した「オイラー図」が使われます。このオイラー図はベン図と同様に概念の外延を閉じた領域で表現しますが、それらの領域の交差で概念間の関係の可能性を表現します。つまり、ベン図と違って場合分けで共通元が存在する場合があれば領域を交差させ、そうでないときに交差させません。以下に「すべての猫と動物である」という命題の周延関係を図示しておきましょう：



この例では左側に周延の様子を示し、右側にその周延の根拠をベン図を使って示しています。左側の矢印で示した二つの命題「この猫は動物である」、「猫はこの動物である」に関しては、「この猫は動物である」は対応関係が成立しますが、「猫はこの動物である」で「猫」以外の動物を指示したときに不成立になることから対応関係は全体的に不成立です。このことから「猫」は周延され、「動物」は不周延になります。このことを記号で表示すると 猫^d , 動物^u であり、判断が全称肯定判断 (A) であることから「 $\text{猫}^d \wedge \text{動物}^u$ 」とまとめて表記できます。

現在、周延はスコラ哲学風に代表理論で説明するよりも概念の外延を用いて説明されます。この場合は概念 S の外延が概念 P の外延に包含されるときに概念 S は概念 P に周延されていると呼び、逆に概念 S が概念 P に包含されていないときを不周延と呼びます。ただし、考え方自体は先程の図示のような外延とその間の写像を考える形になります。では、AEIO の周延状況を以下オイラー図で示しておきましょう：



全称判断であれば主辞は周延され、否定判断であれば賓辞が周延されることがわかります。なお、アリストテレスの論理学では周延について明瞭に述べてはいませんが、日常語で「すべての S は P である ($\varepsilon\iota\nu\alpha$)」と記述するところを「 P は S のすべてにある ($\mu\nu\alpha\rho\xi\nu$)」と一種の人工語として判断を記述しており、 S が周延されている様子が判り易い記述になっています。そして、アリストテレス自身は線分を用いて状況を図示していたようです [4]。また、この周延関係は基本トポスの定義で必要な「**部分対象分類子** (subobject classifier)」にそのものが現れます。

この主辞と賓辞の周延を利用して、判断の意味を変えないように判断を変形する方法、要するに命題を言い換えを構成する方法があります。以下に換質法と換位法と呼ばれる判断の変形の方法について述べましょう。

2.3.2 換質法

「**換質法**」はもとの判断と同じ意味になるように判断の質を変更する方法です。具体的には主辞はそのまま賓辞の「**矛盾概念**」で賓辞を置き換え、さらにもとの判断が肯定判断であれば否定判断に、否定判断であれば肯定判断に置き換える方法です。以下に例を挙げておきましょう：

全ての惑星は天空を移動する	\rightarrow	全ての惑星は天空で停止していない
全ての神は死すべき存在ではない	\rightarrow	全ての神は不老不死である
ある生徒は常識がある	\rightarrow	ある生徒は非常識ではない
ある東海道新幹線は「こだま」では ない	\rightarrow	ある東海道新幹線は「のぞみ」か「ひ かり」である

これらの例の補足をしておきましょう。最初の例では「天空を移動」の矛盾概念は「天空を停止」です。同様に「死すべき存在」の矛盾概念は「不老不死」です。つぎの生徒の例では、生徒を「常識がある」と「非常識である」の二種類の生徒にバッサリと分類して述べています。そして最後の東海道新幹線では「のぞみ、ひかり、こだま」があるために「こだま」の矛盾概念は「のぞみ、ひかり」になります。

AEIO の換質法による変形を以下にまとめておきます。ここで賓辞 P に対する \bar{P} は賓辞 P の概念の矛盾概念である「非 P 」を示します：

換質法による AEIO		
A :	$S^d_A P^u \rightarrow S^d_E \bar{P}^u$	
E :	$S^d_E P^d \rightarrow S^d_A \bar{P}^u$	
I :	$S^u_I P^u \rightarrow S^u_O \bar{P}^d$	
O :	$S^u_O P^d \rightarrow S^u_I \bar{P}^u$	

このように換質法は判断の意味を変えることのない「**言い換え**」の方法の一つです。ここで注意すべきことは矛盾概念が日常語の「**反対**」ではないことです。たとえば、「不味いラーメン」の矛盾概念は「旨いラーメン」ではありません。実際、「不味いラーメン」の矛盾概念である「非(不味いラーメン)」には「旨いラーメン」だけではなく「普通のラーメン」や「不味くはないがそんなに美味しくもないラーメン」といった微妙なものがあるためで、このように日常言語との違いに注意を払う必要があります。

2.3.3 換位法

「**換位法**」は主辞と賓辞の位置を入れ替えて、もとの判断と同じ意味を持つ判断を構築する方法で、換質法と同様にもとの判断の意味を変えない「**言い換え**」を構成する方法^{*32}です。ただし、この手続では主辞と賓辞の周延に注意を払う必要があります。つまり、換位によって本来の判断で周延されていたものが新しい判断で不周延になることは許容され、不周延であったものを周延にすることは許容されません。このことに注意して AEIO で換位法を適用してみましょう。

■**全称肯定判断の場合**：周延が $S^d_A P^u$ するために主辞と賓辞の位置を入れ替えが可能です。ただし、「すべての S は P である」から「ある P は S である」と全称肯定から特称肯定で置換えられます。この置換を「**限定置換**」と呼びます。なお、全称肯定判断の特殊な例である「すべての理性的動物は人間である」から換位で「すべての人間は理性的動物である」になるように主辞と賓辞の概念の外延が一致する「**同一判断**」に対しては主辞と賓辞の単純に入れ替えが可能です。また、このような主辞と賓辞の単純な入れ替えを「**単位置換**」と呼びます。

■**全称否定判断の場合**：周延が $S^d_E P^d$ と主辞も賓辞もともに周延しているために単純に入れ替え、すなわち単位置換が可能です。つまり、「すべての S は P でない」から「す

*32 ここでの「換位」に対応するギリシャ語「ἀντιστρέψειν」の意味はものの順位や方向を逆転・反対することです [4]。

べての P は S でない」にすることができます。

■特称肯定判断の場合: 周延が $S^u \text{ I } P^u$ と主辞も賓辞もともに周延していないために単位置換が可能です。つまり、「ある S は P である」から「ある P は S である」にすることができます。

■特称否定判断の場合: 周延が $S^u \text{ O } P^d$ となり、 S^u を賓辞の位置に持つて行くときに否定判断のために S^d にならなければならぬため、この判断の換位はできませんが、特称否定判断であっても、換質法で特称肯定判断に変形できます。

これらの結果を以下の表にまとめておきましょう:

換位法による AEIO			
A : $S^d \text{ A } P^u$	\rightarrow	$P^u \text{ O } P^u$	
E : $S^d \text{ E } P^d$	\rightarrow	$P^d \text{ E } P^d$	
I : $S^u \text{ I } P^u$	\rightarrow	$P^u \text{ I } P^u$	
O : $S^u \text{ O } P^d$	\rightarrow	なし	

これら換質法と換位法を交互に使って、判断の意味を変えずに判断を変形できますが、この変形は判断が特殊否定判断 (I) になった時点で停止します。

2.3.4 三段論法 (Syllogism)

換質法と換位法は、与えられた命題の変形操作によって同じ意味の命題になることを利用して、より判断しやすい命題で置き換える操作でした。この操作による判断を「直接推論」と呼びます。伝統的論理学にはこの直接推論に加え、「三段論法 (Syllogism)」と呼ばれる推論の形式があります。この三段論法は二つの前提となる判断と一つの結論の合計三つの命題から構成されます。そして、各命題の判断の種類から「定言三段論法」、「仮言三段論法」と「選言三段論法」の三種類に分類できます。

ここでは定言三段論法について解説しましょう。定言三段論法は3つの概念 S, M, P に対し、 S と M, P と M の関係から S と P の関係を求める推論です。ここで概念 S と P の仲立ちをする概念 M を「媒概念」と呼びます。また、概念間の関係は「判断」になりますことから、定言三段論法は三つの判断から構成され、媒概念の判断における位置、つまり、主辞か賓辞であるかで、次の4つの「格 (figure)」に分類されます:

定言三段論法の格

第一格	第二格	第三格	第四格
$M - P$	$P - M$	$M - P$	$P - M$
$S - M$	$S - M$	$M - S$	$M - S$
$S - P$	$S - P$	$S - P$	$S - P$

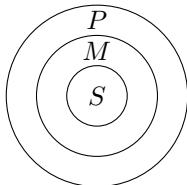
ここで格として4つ挙げていますが、アリストテレスは一、二、三格を挙げており、第四格はガレノスが形式整備のために補完したものです。この表の判断はA, E, I, Oの何れかになりますが、格によっては使えない判断の組合せがあります。ここで典型的な定言三段論法の例を示しておきましょう：

定言三段論法の例

- 大前提：人間は死すべき存在である
 小前提：ソクラテスは人間である
 結論：故にソクラテスは死すべき存在である

この例では概念 S として「ソクラテス」、概念 P として「死すべき存在」、そして、概念 M として「人間」の三つの概念があります。そして、「ソクラテス」は「人間」に包含され、「人間」は「死すべき存在」に包含されており、概念の外延に関しては $S \subset M \subset P$ という包含関係が成立し、概念 S と P の包含関係が概念 M を介することで明瞭になることで判断ができます。

この概念の外延の関係から、概念 S を「小概念」、概念 P を「大概念」、(媒) 概念 M を「中概念」と呼びます。そして、三段論法の前提についても大概念に関わる前提を「大前提」、小概念に関わる前提を「小前提」と呼びます。



ここで定言三段論法を構成する概念 S, P, M に対しては以下の公理があります：

三段論法の公理

- 概念 S, P がそれぞれ概念 M に一致するときに概念 S と P も一致する。
- 概念 S, P のどちらか一方のみが概念 M に一致し、もう一方が一致しないときに概念 S と P は一致しない。
- 概念 S, P がともに概念 M に一致しないときは概念 S, P の関係は不定である。

これらの公理から次の6個の規則と派生する3個の規則の計9規則が得られ、これらの規則を利用して三段論法の格での判断の組合せが定まります：

三段論法の規則

1. 三段論法は 3 個の概念から構成され、これら 3 個の概念に限定される。
2. 三段論法は 3 個の判断のみで構成され、3 個の判断に限られる。
3. 媒概念は二つある前提判断のどちらかで必ず周延されていなければならない。
4. 前提で一度も周延されていない概念を結論で周延させてはならない。
5. 前提が二つともに否定のときは結論が得ることができない。
6. 前提が共に肯定のときは結論も肯定、前提の一方が否定のときに結論は否定である。

■規則 1: 名辞として一致していても文脈上で異なる概念があることに注意しなければなりません。たとえば、「病気」にしても病院で治療を行うべき疾病なのか、単なる悪癖めいた趣向のことかもしれません。このように概念に曖昧さがあるときは概念が実質的に 4 個になり、規則 1 に反することになります。この誤謬を「**媒概念曖昧の誤謬**」と呼びます:

媒概念曖昧の誤謬の例

病気 (=疾病) は休養を必要とする
私の怠惰は病気 (=悪癖) である
ゆえに、私の怠惰には休養が必要である

■規則 2: ここで述べている判断は三段論法を構成する上で必要な判断であり、ある結論を得るために必要な判断のことではありません。実際の推論では、複数の三段論法や直接推論を組み合わせて行うために判断が三個に限定されるとは限りません。

■規則 3: 三段論法は基本的に概念の概念の包含関係に基づく推論であるために、全ての概念が不周延であれば包含関係が不明慮になって結論が得られません。すなわち、概念 S と P の間を媒概念 M が取り持つことで S と P の包含関係が明瞭になることを利用しているため、媒概念 M が不周延であれば、概念 M が上手く概念 S と概念 P の間を上手く取り持っていない状況になります。この状況を「**媒概念不周延の誤謬**」と呼びます:

媒概念不周延の誤謬

すべての男の子は人間である
すべての女の子は人間である
ゆえに、すべての男の子は女の子である

この例では概念「男の子」と概念「女の子」の仲立をする媒概念「人間」が「男の子」と「女の子」の双方から不周延であるために間違った結論になります。

■規則 4: 前提にて不周延な概念は媒概念との関係上、部分的な合致に留まり、全体の合致や周延関係があることを主張できないためです。小前提にのみに現れる概念が不周延で、それを結論で周延されたときに「**小前提不当周延の誤謬**」、大前提のみに表れる概念が不周延で、それを結論で周延させたときに「**大前提不当周延の誤謬**」と呼びます：

小前提不当周延の誤謬

すべての男の子はゲーム好きある
すべての男の子はサッカーファンである
ゆえに、全てのサッカーファンはゲーム好きである

これは「小前提不当周延の誤謬」の例になります。実際、ここでの小概念は「サッカーファン」、大概念が「ゲーム好き」、媒概念が「男の子」であり、まず、小前提で「サッカーファン」は周延されていませんが、結論では周延されています。ところでサッカーファンは不周延、つまり、男の子以外のファンもいる訳で、ここでの前提からゲーム好きと結論付けられません。

もう一つの例を挙げておきましょう：

大前提不当周延の誤謬

すべての OL はスイーツが好きである
このレストランの客はみんな OL ではない
ゆえに、このレストランの客はスイーツが好きではない

これも「大前提不当周延の誤謬」の例です。実際、この例で小概念は「このレストランの客」、大概念は「スイーツ好き」で媒概念が「OL」です。そして、大前提では「スイーツ好き」が不周延になりますが、結論が全称否定判断であるために、今度は大概念が周延されています。そのために間違った結論になっています。

■規則 5: 三段論法の公理 3 の小概念と大概念が媒概念に一致しないときに小概念と大概念の周延関係が定まらないことから得られます。この規則に反したときに「**否定二前提の誤謬**」と呼びます。

■規則 6: 三段論法の公理 1 と 2 から得られる規則で、結論が肯定であるべきときに否定したときに「**不当否定の誤謬**」、逆に否定であるべきときに肯定したときに「**不当肯定の誤謬**」と呼びます。この規則 6 からは次の三つの系が得られます：

規則 6 から派生する系

1. 前提が共に特称判断のときは結論が得られない。
2. 前提の一方が特称判断であれば結論も特称判断になる。
3. 大前提が特称判断で小前提が否定であれば結論が得られない。

これらの公理、公理に基づく規則や系から、定言三段論法の格に置くことのできる判断に制約が入ります。実際、4つの格があり判断が AEIO の4種類があるために単純計算では $4^3 = 64$ 通りになるものが、上述の規則と系によって限定されます。

■第一格の場合: 大前提が $M - P$ 、小前提が $S - M$ であり、媒概念が前提で周延しなければならないことから、大前提が全称判断、そうでなければ小前提が否定判断でなければなりません。ところで小前提が否定判断のときに結論も否定判断になり、そのためには P も周延されることになります。前提で P が周延されるためには大前提が否定判断でなければなりませんが、このときは前提が二つとも否定になって「否定二前提の誤謬」になるため、小前提是肯定判断でなければなりません。のことから AAA, AII, EAE, EIO のいずれかになることが判ります。

■第二格の場合: 大前提が $P - M$ 、小前提が $S - M$ であることから、媒概念 M が周延されるためには大前提か小前提のどちらか一方が否定判断でなければなりません。ここで前提の一つが否定判断になるときに結論は否定判断になるために P は周延されます。したがって大前提が全称判断でなければなりません。のことから AEE, AOO, EAE, EIO になります。

■第三格の場合: 大前提が $M - P$ 、小前提が $M - S$ で、媒概念 M が周延されるために前提が全称肯定か全称否定のいずれかでなければなりません。まず、大前提が全称肯定判断 (A) であれば P は不周延のために結論は否定判断にはなり得ません。そのため小前提是肯定判断でなければなりません。さらにこのときに S は不周延になるために結論は特称肯定判断 (I) になります。また、大前提が全称否定判断 (E) であれば、小前提是否定二前提の誤謬を避けるために肯定判断に限定されます。これらのことから S が不周延、 P が周延されるため結論は特称否定判断 (O) になります。最後に小前提が全称肯定判断 (A) のとき、大前提が全称判断の場合は先程の考察に該当するために大前提が特称判断の場合だけを考察すればよいことになります。ここで大前提が特称肯定判断 (I) であれば S, P 共に不周延の為に結論は特称肯定判断 (I) になります、そして、大前提が特称否定判断 (O) であれば S は不周延、 P が周延であるために結論は特称否定判断 (O) になります。これらのことから AAI, AII, EAO, EIO, IAI, OAO があります。

■第四格の場合: 大前提が $P - M$ 、小前提が $M - S$ で、媒概念 M が周延されるためには大前提が否定判断であるか、小前提が全称判断でなければなりません。まず、大前提

が全称否定判断 (E) のときに小前提は肯定判断でなければならないために S は不周延, P が周延されることになり, 結論が特称否定判断 (O) になることがわかります. また, 大前提が特称否定判断 (O) であれば P が不周延になりますが, 二つの前提の一方が否定判断であれば結論が否定判断になることと, そのときに P が周延されていなければならないために大前提が特称否定判断 (O) であることはあり得ません. 小前提が全称肯定判断 (A) のとき, S は不周延であるために結論は特称判断になります. ここで大前提が全称肯定判断 (A) のときと特称肯定判断 (I) のときは結論は特称肯定判断 (I) になります. そして, 小前提が全称否定判断 (E) のときは S が周延され, 結論も否定判断になるために P も周延されていなければなりません. このことから大前提と結論の双方が全称否定判断 (E) になります. これらのことから, EAO, EIO, AAI, IAI, AEE があることが判ります.

以上の考察から次の表が得られます:

認められる判断の組み合わせ			
第一格: AAA(Barbara)	EAE(Celarent)	AII(Darii))	EIO(Ferioque)
第二格: EAE(Cesare)	AEE(Camestres)	EIO(Festino)	AOO(Baroco)
第三格: AAI(Darati)	IAI(Disamis)	AII(Datisi)	EAO(Felapton)
	OAO(Bocardo)	EIO(Ferison)	
第四格: AAI(Bramantip)	AEE(Camenes)	IAI(Dimaris)	EAO(Fesapo)
	EIO(Fresison)		

この表で括弧内は中世の学生が暗記するために使った単語で, 子音を外すと格が現われるという仕組です. この暗記では Syllogismus という詩が元ネタです.

定言三段論法の大前提に使える命題は, 当然のことながら「明らかに真であると判断できるもの」でなければ「帰納的に求められるもの」でなければなりません. ところでイデアや概念といった普遍の存在を認めてしまえば存在含意を充すため, この推論を行う際の障害がなくなります. ところが存在が不確かな仮説では, どのような結論が出ても不思議がありません. 実際, イスラム哲学においてアッバース朝の公認神学であった「ムアタズィラ (Mu'tahzilah)」と呼ばれる超合理主義派は自らを「正義と神の唯一性の提唱者」と自称していた程で, 彼等は三段論法を駆使してともすれば異端的な結論を導出していたそうです [6]. それに加えて「神の人格表現の否定」によりクルアーン (コーラン) で述べられた神の人間的表現を字義通りではなく一種のお比喩として捉え, 神を知識や理性と見なしました^{*33}. 彼らは「哲学こそが全て, 宗教は一般大衆向けの幼稚な哲学」という考え方

^{*33} 神を νοῦς や Λόγος とみなすために同時代の神学者からは「『神よ!』と呼びかけるのではなく, 「知恵よ!」と呼びかけば良いではないか」と皮肉 られている程です.

を持ち、やがて、「正統派」によってムアタズィラの著作が根絶させられるという憂き目にあっています。この様子は19世紀以降、ヨーロッパ諸国の軍事力に圧倒された結果、世俗的な社会改革を行うものの宗教的保守派や原理主義、そして改革を受けられないと大衆によって再三、妨げられ、改革の失敗後に極端な復古が生じるというイスラム教諸国でよく見られる動向と類似していなくありません^{*34}。

2.4 集合論について

2.4.1 集合論言語について

「それが何であるか?」という問に対する説明規定が概念で、その概念で説明され得るもの集まりが外延ですが、「**それ自身でないもののあつまり**」という命題には外延が存在しません。どのような命題にも外延が存在しているという前提でフレーゲが開始した論理主義は厳密な数学の基礎を与えるかのように見えましたが、この命題から矛盾が生じて呆気なく破綻してしまいました。集合論の創始者のカントール (Cantor) は、素朴集合論から派生する逆理をその体系の豊かさと捉えていたようですが、この論理主義の失敗から「**外延**」という「**命題を充すもののあつまり**」と「**集合**」との間に境界線が必要との認識が生じ、公理的集合論が生まれます。この公理的集合論はツエルメロ (Zermelo) の公理系を基にフレンケル (Frankel) の公理等を追加した公理系と、それらの公理とは独立した「**選択公理**」と呼ばれる重要な公理があり、これらの公理の組み合わせで Z, ZF や ZFC 等と略記された公理系を基に集合論が構築されています。そして、この公理系を語る必要がありますが、この集合論にはその体系で扱う対象を語るために「**言語**」があり、それが「**集合論言語**」と呼ばれる言語です。この言語の記号系を以下に示します：

集合論で用いる記号系

1. 基本述語: “=”, “ \in ”
2. 変項: x, y, z, u, w, \dots
3. 論理記号: “ \vee ”, “ \wedge ”, “ \neg ”, “ $\neg\neg$ ”, “ \equiv ”, “ \exists ”, “ \forall ” $\forall x$
4. その他の記号: “(”, “)”, “,”

ここでは元が集合に属するという意味で用いる記号 “ \in ” と対象の同一性を示す記号 “=” の他は論理式の論理和 “ \vee ”, 論理積 “ \wedge ”, 否定 “ \neg ” と含意 “ $\neg\neg$ ”, それと量化詞の記号で「**全て**」に対応する “ \forall ” と「**存在する**」に対応する “ \exists ”, 最後にその他の記号として論理式のグループ化を行う括弧 “(” と “)”, それに区切記号の “,” が記号系に含まれます。またこの

^{*34} グーテンベルクの印刷術が西欧諸国で宗教改革に大きく関与したのと同様に、インターネットが現在のイスラム教国の原理主義にエネルギーを与えている点は実に皮肉なことです。もちろん、親 (=権威) に対する若者の反発という古典的な要因もありますが。

本では「 a を b で定義する」ことを記号 “ $\stackrel{\text{Def.}}{\equiv}$ ” を導入して ‘ $a \stackrel{\text{Def.}}{\equiv} b$ ’ と表記します。それから記号 “ \equiv ” を同値性を意味する記号として以下で定義します:

$$A \equiv B \stackrel{\text{Def.}}{=} (A \supset B) \wedge (B \supset A)$$

これらの記号を用いて集合論の論理式を次の形成規則で定義します:

論理式の形成規則

1. $x = y$ と $x \in y$ は集合論の論理式である。
2. A, B を集合論の論理式とするとき, $A \vee B$, $A \wedge B$, $A \supset B$, $\neg A$, $A \equiv B$, $\exists x A(x)$, $\forall x A(x)$ も集合論の論理式である。
3. 上記の方法で構成されたもののみが集合論の論理式である。

この論理式の形成規則を持つ系を「**集合論言語**」と呼び, \mathcal{L} と表記します。この形成規則は帰納的であり, 論理式の具体的な表記は §3 の §3.3 で述べる BNF 記法に基く表記で表現可能です。なお, この論理式の形成規則は, 集合論で扱う論理式の形成方法について述べたものであって, 論理式の意味や意義について述べたものでも, どのような論理式が集合論の体系に受け入れられるかを述べたものでもなく, 集合論の公理系がそれらを規定します。

2.4.2 集合論の公理系

集合論言語 \mathcal{L} を使って集合論の公理系を記述し, 公理を解説しましょう:

集合論の公理系

- | | |
|-----------|--|
| A1 外延公理 | $\forall x \forall y (\forall z (z \in x \equiv z \in y) \supset x = y)$ |
| A2 対集合公理 | $\forall x \forall y \exists z (\forall u \in z \equiv (u = x \vee u = y))$ |
| A3 和公理 | $\forall x \exists y \forall z (z \in y \equiv \exists u (z \in u \wedge u \in x))$ |
| A4 署集合公理 | $\forall x \exists y \forall z (z \in y \equiv z \subseteq x)$ |
| A5 空集合公理 | $\exists x \forall y \neg(y \in x)$ |
| A6 無限集合公理 | $\exists x (\emptyset \in x \wedge \forall y (y \in x \supset y \cup \{y\} \in x))$ |
| A7 置換公理図式 | $\forall x \forall y \forall z (\phi(x, y) \wedge \phi(x, z) \supset y = z)$
$\supset \exists u \forall y (y \in u \equiv \exists(x \in u \wedge \phi(x, y)))$ |
| A8 正則性公理 | $\neg(x = \emptyset) \supset \exists y (y \in x \wedge y \cap x = \emptyset)$ |
| A9 選択公理 | $\forall x \in u (\neg x = \emptyset) \wedge \forall x, y \in u (\neg x = y \supset x \cap y = \emptyset)$
$\supset \exists v \forall x \in u \exists t (t \in x \wedge t \in v)$ |

■**外延性公理** (Axiom of extensionary): 外延から集合が一意に定まることを保証する公理です。なお、集合の外延の記述は $\{a, b, c, d\}$ のように括弧 $\{\}$ の中に区切記号 “,” とする元の列を記載します。このときに括弧 $\{\}$ 内の列の順番と無関係に集合が一意に定まります。

■**対公理** (Axiom of pairing): 集合 x, y を成分とする「**対集合**」の存在を保証する公理です。ここで、集合 x, y の対集合を $\{x, y\}$ と表記します。特に $\{x, x\}$ を $\{x\}$ と表記して「**1-要素集合 (シングルトン, singleton)**」と呼びます。この対集合 $\{x, y\}$ は集合 x と y をその成分として持つことを意味するだけで、集合 x と集合 y の順序等の関係について何も述べていません。そこで、

$$\langle x, y \rangle \stackrel{\text{Def.}}{=} \{\{x\}, \{x, y\}\}$$

で集合 x, y の順で順序を持つ「**順序対**」と呼ばれる集合 $\langle x, y \rangle$ を定義します。なお、順序対では $x = y$ でない限り $\langle x, y \rangle = \langle y, x \rangle$ ではありません。そして、成分が 3 以上の順序対は帰納的に構成できます：

順序対の構成方法

$$\begin{aligned} \langle x_1, x_2 \rangle &\stackrel{\text{Def.}}{=} \{x_1, \{x_1, x_2\}\} \\ \langle x_1, x_2, \dots, x_n \rangle &\stackrel{\text{Def.}}{=} \langle x_1, \langle x_2, \dots, x_n \rangle \rangle \quad n > 2 \end{aligned}$$

■**和集合公理** (Axiom of union set): 「**集合族**」(=集合の集合) x の成分になる集合の成分を全て含む集合の存在を保証する公理です。この公理から保証される集合を $\cup x$ と表記し、「**和集合**」と呼びます。また、対集合 $\{x, y\}$ の和集合は特別に $x \cup y \stackrel{\text{Def.}}{=} \cup\{x, y\}$ で式 $x \cup y$ を定め、この集合 $x \cup y$ を「**集合 x, y の和集合**」と呼びます。

■**幂集合公理** (Axiom of power set): この公理に現われる記号 “ \subseteq ” は

$$a \subseteq b \stackrel{\text{Def.}}{=} \forall x(x \in a) \supseteq x \in b \vee x = b$$

で定義される記号で、幂集合公理の意味は集合 x の任意の成分を外延として持つ集合の存在の保証です。この公理と外延性公理から唯一存在する集合を「**幂集合**」と呼び、集合 x の幂集合を $\mathcal{P}(x)$ と表記します。

■**空集合公理** (Axiom of empty set): 元を持たない集合の存在を保証する公理です。この公理と外延公理から唯一存在する集合を「**空集合**」と呼び、記号 “ \emptyset ” と表記します。

■**無限集合公理** (Axiom of infinity set): 無限集合の創り方を定める公理で、 v が集合のときに $\emptyset \cup \{v\}$ が集合になることを保証します。ここで空集合 \emptyset と無限集合公理から

$\emptyset \cup \{\emptyset\}$ が集合になることが保証されます。さらに $\emptyset \cup \{\emptyset \cup \{\emptyset\}\}$ も集合になることが無限集合公理から保証されます。このように空集合 \emptyset から開始して、この処理を繰り返すことで

$$\emptyset, \emptyset \cup \{\emptyset\}, \emptyset \cup \{\emptyset \cup \{\emptyset\}\}, \dots$$

という集合の無限列が構成できますが、この無限列が自然数になります：

自然数の定義

$$\begin{aligned} 0 &\stackrel{\text{Def.}}{=} \emptyset \\ 1 &\stackrel{\text{Def.}}{=} \emptyset \cup \{\emptyset\} (= \{0\}) \\ 2 &\stackrel{\text{Def.}}{=} \emptyset \cup \{\emptyset \cup \{\emptyset\}\} (= \{0, 1\}) \\ 3 &\stackrel{\text{Def.}}{=} \emptyset \cup \{\emptyset \cup \{\emptyset \cup \{\emptyset\}\}\} (= \{0, 1, 2\}) \\ \dots &\dots \dots \\ n+1 &\stackrel{\text{Def.}}{=} \emptyset \cup \{\emptyset \cup \{\emptyset \cup \{\emptyset \cup \dots\}\}\} (= \{0, 1, 2, \dots, n\}) \\ \dots &\dots \dots \end{aligned}$$

空集合 \emptyset が自然数の 0, $\emptyset \cup \{\emptyset\}$ が自然数の 1 に対応し、以降、集合の無限公理で認められた集合の生成規則にしたがって自然数が続々と生成されます。このように空集合公理と無限集合公理を含む公理系では自然数を体系内に包含しています。

さらに「超限順序数」を上述の方法で構成した集合全ての和集合として定義します：

超限順序数

$$\omega \stackrel{\text{Def.}}{=} \{0, 1, 2, 3, \dots\}$$

自然数 a, b に対して $a < b \stackrel{\text{Def.}}{=} a \in b$ で記号 “ $<$ ” を導入し、同様に記号 “ \leq ” を $a \leq b \stackrel{\text{Def.}}{=} a \in b \vee a = b$ で定義することで自然数に「大小関係」を導入できます。さらに自然数 a に対して $a + 1$ を $a + 1 \stackrel{\text{Def.}}{=} \emptyset \cup \{a\}$ で定め、この $a + 1$ を a の「後続」、あるいは「後者」と呼びます^{*35}。この自然数については順序数で再度触れます。

■置換公理図式 (Axiom schema of replacement)：式中の $\phi(x, y) = \phi(x, z) \supset y = z$ を $F(x) = y$ で置換すると集合 x の函数 F による像も集合になるという公理であると同時に、集合そのものに制約を入れる公理です。この公理はフレンケルが導入しましたが、ツエルメロが入れていた本来の公理は「分出公理 (Axiom of displacement)」と呼ばれる次の公理です：

*35 自然数の後者関係についてはフレーベルの「概念記法」[26] ではじめて厳密に述べられています。

分出公理 (Axiom of displacement)

$$A7' \quad \forall x \exists y \forall u (u \in x \equiv (u \in x \wedge \phi(u)))$$

分出公理の意味は任意の命題が外延を持つとは限らず、既存の集合から指定された命題を充す集合が存在するという意味で、置換公理図式からも導けます。実際、置換公理図式 A8 の $\phi(x, y)$ を $\psi(x) \wedge x = y$ で置換えることで分出公理 A7' が直ちに得られ、この分出公理から得られる集合を $\{u \in x : \phi(u)\}$ と表記します。そして、この分出公理から幾つかの重要な集合の生成方法が定義できます。まず、集合 x, y に対して $\{u \in x : u \in y\}$ で得られる集合を $x \cap y$ と表記し、集合 x と y の「**共通集合**」と呼びます。それから $\{\langle u, v \rangle : u \in x \wedge v \in y\}$ で得られる集合を $x \times y$ と表記し、集合 x と y の「**直積集合**」と呼びます。

ここで命題 $\phi(x)$ の外延 $\{x : \phi(x)\}$ はその元がある集合の元であるとの保証がないために分離公理から集合であると断言できません。命題の外延を「**類**」、あるいは「**クラス (class)**」と呼び、「**集合**」と区別します。オブジェクト指向の「**クラス**」が「**クラス**」と呼ばれるのも複数の「**述語**」に対応する「**属性値**」や「**メソッド**」から構成されるものの、それらが定める外延がとある「**集合**」から切り出したものとは限らないためです。では素朴集合論で問題となつた「**ラッセルの逆理**」をもう一度考えてみましょう。分出公理によつて、あらかじめ集合として認められたものから命題 $x \notin x$ を充す x を取り出さなければなりませんが $x \notin x$ より自分自身を包含しない集合が構成できません。そのために、この命題の外延は集合にならず、集合論の体系から排除できます。

分出公理は逆理の排除という目的では有効ですが、この公理をフレンケルが置換公理図式で置換えた理由として「**大きな集合の生成ができない**」ということに尽きます。ここでは「選択公理と数学」[21] で紹介されている函数の例を挙げておきましょう：

まず、最初に函数 f を

$$\begin{array}{lll} f(0) & = & \omega \\ f(1) & = & \mathfrak{P}(\omega) \\ \dots & \dots & \dots \\ f(n+1) & = & \mathfrak{P}(f(n)) \\ \dots & \dots & \dots \end{array}$$

で定めます。このとき函数 f の値域 $\text{rng}(f)$ は：

$$\text{rng}(f) \stackrel{\text{Def.}}{=} \{\omega, \mathfrak{P}(\omega), \mathfrak{P}^2(\omega), \dots\}$$

によって与えられますが、この値域 $\text{rng}(f)$ が集合になることが分出公理から導出できま

せん。ここで置換公理を認めると函数による集合の像も集合になることが保証されるため、その値域 $\text{rng}(f)$ が集合になります。このように新たな集合を作り出せる置換公理の方が単に集合から集合を取り出すことで集合としての制約を加える分出公理よりも強力な公理であることが理解できるでしょう。

■正則性公理 (Axiom of regularity): この公理によって論理式 $a \in a$ の外延が集合から排除されて集合と集合の元が区別されます。その結果、 $\dots, x_3 \in x_2, x_2 \in x_1, x_1 \in x_0$ を充す**集合の底なしの無限列**: 「 $\dots, x_3, x_2, x_1, x_0$ 」も排除されます。このような底なしの無限列があると困ることを説明しておきましょう。空集合公理と無限公理から自然数と大小関係が導入できますが、正則性公理があれば $\dots \in x_2 \in x_1 \in x_0$ となる集合の列 x_i は底なしの無限列にならないために必ず $x_n \in \dots \in x_2 \in x_1 \in x_0$ を充す集合 x_n が存在し、さらに有限列になることが判ります。このことが自然数の列に必ず最小値が存在することに対応し、その自然数の性質に反する集合の無限列の存在を気にすることなしに順序数が導入できます^{*36}。また関係 \in に対する無限降下列が存在しないことは公理 A8':

無限降下列の非存在性

A8' 無限降下列 $\dots \in u_2 \in u_1 \in u_0$ が存在しない

とすることができます。この「**無限降下列の非存在性公理 A8'**」と「**正則性公理 A8**」の間には $A8 \supset A8'$ が成立しますが、その逆の $A8' \supset A8$ が成立するためには次の「**選択公理**」が必要になります [21]。

■選択公理 (Axiom of choice) 空集合と異なる集合からは成分を取り出すことができるという公理で、後述の ZFC 公理系の“C”に該当します。この選択公理は他の集合論の公理から独立した公理で、この公理なしでも「数学」を構築できます。この選択公理は便利な一方で、厄介な逆理が幾つか導きだせることが知られています。その逆理の一つの「**バナッハ-タルスキ (Banach-Tarski) の逆理**」を紹介しておきましょう：

バナッハ-タルスキの逆理

3 次元ユークリッド空間 \mathbb{R}^3 の有界集合 A, B を適当な同数個の区画に分割する：

$$\begin{cases} A = A_1 \cup A_2 \cup \dots \cup A_n \\ B = B_1 \cup B_2 \cup \dots \cup B_n \end{cases}$$

すると各 A_i と B_i ($1 \leq i \leq n$) を合同にできる。

たとえば、ゴルフボールの表面を適当に分割し、それらを貼り合せるだけで地球が覆える

*36 底無しさ加減は落語の「頭山」のオチに通じますが、「**自分の頭にできた池に本人が飛び込む**」という行為をとともに考えると、それこそ「**底なし**」の状況になるために、この噺には「**オチがない**」とも言えます。

ということを主張しています。牛の皮程もない蜜柑の皮で、皆どころか世界征服も可能と女王ディドーも大喜びな話です³⁷。さすがにこの定理は日常的な常識から大きく外れたものですが、選択公理自体はそれを認めたときの御利益が圧倒的に大きな公理です。なお、選択公理を認めなければ任意の自然数の部分集合が最小元を持つことを利用します。

ここで A1 から A9 までの公理系の組み合せ表を示しておきましょう：

——集合論の公理系——

Z	:	A1	A2	A3	A4	A5	A6	A7'	A8
ZC	:	A1	A2	A3	A4	A5	A6	A7'	A8 A9
ZF	:	A1	A2	A3	A4	A5	A6	A7	A8
ZFC	:	A1	A2	A3	A4	A5	A6	A7	A8 A9

通常の集合論の公理系として用いられるのが「**ZFC 公理系**」です。この公理系は表からも判るようにツェルメロ・フレンケルの公理系 (ZF) に選択公理 (C) を追加した公理系です。

2.4.3 順序数

ZFC 公理系にて順序数を次で定義します。

——順序数の定義——

$$\begin{aligned} \text{Trans}(u) &\stackrel{\text{Def.}}{=} \forall x, y(x \in u \wedge y \in x \supset y \in u) \\ \text{Ord}(\alpha) &\stackrel{\text{Def.}}{=} \text{Trans}(\alpha) \wedge \forall x, y \in \alpha(x \in y \vee x = y \vee y \in z) \end{aligned}$$

最初の述語 $\text{Trans}(u)$ は集合 u が推移的であることの定義です。ここで述語 $\text{Trans}(u)$ の意味するところは x が集合 u の元であり、 y が x の元であれば y も集合 u の元になることです。ここで記号 “ \in ” を記号 “ $<$ ” で置換えると「 $x < u$ かつ $y < x$ ならば $y < u$ 」が得られ、このことから通常の大小関係で見られる推移律に対応すること容易に判るでしょう。次に述語 $\text{Ord}(\alpha)$ を使って集合 α が順序数であることを定義しています。この述語 $\text{Ord}(\alpha)$ の意味するところは、まず、集合 α が推移的で、それから集合 α に属する任意の x, y に対して $x \in y$, $y \in x$ か $x = y$ の何れかの関係が成立することです。ここでも記号 “ \in ” を記号 “ $<$ ” で置換えると順序数 α に対して $x < \alpha$, $y < \alpha$ になる $x, y \in \alpha$ に対して $x < y$, $y < x$ か $x = y$ の何れかの関係が成立すること、つまり、集合 α が全順序集合であることを意味しています。たとえば、自然数全体の集合 $\omega = \{0, 1, 2, 3, \dots\}$ の元 u は $\text{Trans}(u)$ を充すために推移的で、さらには $\text{Ord}(u)$ を充すために順序数になります。そし

*37 牛の皮で覆えるだけの土地が与えられるという条件で牛の皮を細かく切って取り囲んで得た場所から発展したというカルタゴの建国神話があります。

て、この順序数の定義からはさまざまな集合の無限列からも順序数が得られることが判ります。たとえば置換公理で紹介した $\{\omega, \mathfrak{P}(\omega), \mathfrak{P}^2(\omega), \dots\}$ も順序集合で、また、 ω は自然数を含む順序数の中で最小の順序数です。

それから任意の二つの順序数 α, β に対しては、その包含関係から $\alpha \in \beta$, $\alpha = \beta$ か $\alpha \in \beta$ の何れか一つが成立します。ここで順序数では関係 \in を大小関係 $<$ で置換えます。つまり、 $\alpha \in \beta$ を $\alpha < \beta$ と表記します。さらに順序数 α に対して $\alpha + 1$ を $\alpha \cup \{\alpha\}$ で定義し、この $\alpha + 1$ を順序数 α の「後続」、あるいは「後者」と呼びます。そして、ある順序数の後者にならない 0 以外の順序数 α を「極限数」と呼び、 $\alpha \in \text{Lim}$ と表記します。極限数の例として ω を挙げておきましょう。では次に順序数全体 OR を定義しましょう：

順序数全体

$$\text{OR} \stackrel{\text{Def.}}{=} \{\alpha : \text{Ord}(\alpha)\}$$

この順序数全体 OR は大き過ぎるために ZFC では集合ではなくクラス(類)になります。実際、この OR は推移的で、また、 \in に関して全順序になります。ここで OR が集合であれば $\text{OR} \in \text{OR}$ になって OR の後者 $\text{OR} + 1$ を考えることができますが推移率から $\text{OR} \in \text{OR} + 1$ 、一方で OR は順序数の全体なので $\text{OR} + 1 \in \text{OR}$ になって矛盾が生じます。これが素朴集合論で「**プラリ=フォルティの逆理**」と呼ばれる逆理です。ただし、この素朴集合論上の逆理も ZFC では正則性公理によって、「OR は集合でない」という定理になります。

2.4.4 モデルと宇宙

ここで M を空集合 \emptyset と異なる集合、あるいはクラスとします。さらに M 上で前述の集合論言語 \mathcal{L} が定められているとしましょう。このことを $\langle M, \in \rangle$ と表記し、集合論言語 \mathcal{L} の「 \in -構造」、「 \in -モデル」、あるいは単に「**モデル**」と呼びます。さらに M のことを「(集合論の) 宇宙(universe)」と呼びます。それから集合論言語 \mathcal{L} の文 φ が M の元に対して成立するときに $\langle M, \in \rangle$ を φ の「**モデル**」と呼び、 $\langle M, \in \rangle \models \varphi$ 、あるいは簡潔に $M \models \varphi$ と表記します。また、モデル M 上で文 φ が成立しないことを $M \not\models \varphi$ と表記します。

このモデル M は集合論言語 \mathcal{L} の文 φ の意味を判断する上での文脈に相当します。ちなみに日常の文でも文脈によって、その意味が真であったり偽となったりすることがあります。たとえば、ある人達の会話で「彼はイケメン」という話が出たとき、その会話をしている人達にとっては「彼」が誰なのかは自明なことですが、この人達と無関係な人にとって「彼」が誰を指すのか不明なために真偽の判断ができないものです。これはモデルでも

同様で、モデル M で文 φ の意味が真であったとしても別のモデル N では偽となることがあります。ところが、文 $A \supset A$ 、日常語なら「 A は A である」のように文脈と無関係に常に真になる文もあります。このように文脈とは無関係に常に真となる文のことを「**恒真式**」あるいは「**トートロジー (tautology)**」と呼びます。

2.5 函数と関係

2.5.1 函数について

論理学に函数概念を最初に導入した人はフレーゲです。フレーゲによると函数の本質は「**不飽和**」であると述べています。この不飽和の意味は、たとえば、 x^2 といった函数は 2^2 と異なり変数 x に値が設定されない限り、その値を持ちません。つまり、函数の変数に値を「**付与**」しなければ、それ自体が値を持つことはなく、この意味で「**変数に値を充填すべきもの**」です。さらにフレーゲは形式的な函数の表記を行っています。この表記はチャーチの λ -表記法に似た表記で、函数の表示でメタ変数を用いるものです。たとえば「与えられた二乗の数を返す函数」は x^2 という表記ではなく、メタ変数 ξ を導入して ξ^2 と表記し、二変数の函数であれば $f(x, y)$ と $f(y, x)$ が異なることから変数のタプルの順番、つまり、「**項の位置**」を導入し、第1変数に ξ 、第2変数に ζ というメタ変数を割り当て、函数 f を $f(\xi, \zeta)$ と記述します。ちなみに λ -表記では $\lambda(x, y).f(x, y)$ になります。また、三変数以上の多変数函数についてはカーリー化に類似した方法で二変数函数の話に還元できるとし、特に二変数函数を「**関係**」と呼んでいます。さらには高階函数をも考察し、高階函数の変数(=函数名)を f 等のドイツ文字で表記しています。ところで、函数は、その構造に関心がない限り、一般的にはメタ記号を用いた表記は用いずに ‘ $f(x)$ ’ や ‘ $g(x, y, z)$ ’ のように f, g 等の函数名と $(x), (x, y, z)$ のような変数を指示するタプル、つまり、変数の列を括弧 “()” で括ったものの結合として表記します。この本では、函数それ自体を明記する目的では λ -式表記を用い、‘ $\lambda(x, y).f(x, y)$ ’ のように表記しますが、明記する必要がないときは変数として x, y, z, \dots とアルファベットのうしろ側を用い、‘ $f(x.y)$ ’ と表記します。

2.5.2 関係について

関係は True, False といった真理値を返す二変数函数です。たとえば、関係 $y = x^2$ は変数 x と y の間に x の二乗が y と等しいという関係を示し、その値は真理値を持つ函数であることを示しています。関係の表記は二項の間に函数名を配置する中值表記が用いられるため、ここでは a と b の間の関係を ‘ $a R b$ ’ と表記します。以下に代表的な二項関係を纏めておきましょう：

重要な二項関係

1. 反射律: $x R x$
2. 対称律: $(x R y) \supset (y R x)$
3. 反対称律: $((a R b) \wedge (b R a)) \supset (a = b)$
4. 推移律: $((x R y) \wedge (y R z)) \supset (x R z)$
5. 全順序律: $(a R b) \vee (b R a)$

ここで重要な関係を二つ挙げておきます。最初に挙げる関係は「**同値関係**」と呼ばれる関係です：

同値関係

関係 ‘R’ が同値関係であるとは次の条件を充たすときです：

1. 反射律: $x R x$
2. 対称律: $(x R y) \supset (y R x)$
3. 推移律: $((x R y) \wedge (y R z)) \supset (x R z)$

同値関係は実は非常に身近な関係です。たとえば分数にその関係が見られます。そのことを確認するために $\frac{a}{b} \sim \frac{n}{m}$ という関係 “~” を次で定めましょう：

$$\frac{a}{b} \sim \frac{n}{m} \stackrel{\text{Def}}{=} a \cdot m - b \cdot n = 0$$

さて、二つの分数が与えられたときにそれらが等しいとは双方を約分すると同じ分数になるときです。実際、分数 $\frac{a}{b}$ と $\frac{c}{d}$ が分数 $\frac{n}{m}$ に約分できるとしましょう。このことは自然数 h, k が存在して $a = h \cdot n, b = h \cdot m, c = k \cdot n, d = k \cdot m$ であることを意味します。さて $a \cdot d - c \cdot b$ を計算してみましょう：

$$\begin{aligned} a \cdot d - c \cdot b &= (h \cdot n) \cdot (k \cdot m) - (k \cdot n) \cdot (h \cdot m) \\ &= h \cdot k \cdot n \cdot m - h \cdot k \cdot n \cdot m \\ &= 0 \end{aligned}$$

と、この結果から $\frac{a}{b} = \frac{c}{d}$ であれば $\frac{a}{b} \sim \frac{c}{d}$ になることが判りました。実際は $a \cdot m - b \cdot n = 0$ より $a \cdot m = b \cdot n$ の両辺を $b \cdot m$ で割ると $\frac{a}{b} = \frac{n}{m}$ になるため、約分をして同じ分数が得られることと関係 “~” を充たすことは同値です。

分数 $\frac{a}{b}$ はより正確には自然数と自然数から 0 を除いたものの対、すなわち $\mathbb{N} \times (\mathbb{N} \setminus \{0\})$ として表現できます。しかし、 $(1, 2) \neq (2, 4)$ であっても $\frac{1}{2} = \frac{2}{4}$ であるために、分数は「**自然数と自然数から 0 を除いたものの対**」そのものではありません。このことは分数が自然数と自然数から 0 を除いたものの対を関係 “~” で分類したものであることを意味します。この分数のように集合 S を同値関係 ~ で分類したものを「**同値類**」と呼び、同値類の集合

を商集合と呼び S/\sim と記述します。また、同値類から選出した元 a のことを「**代表**」と呼びます。ここで分数の話に戻すと分数 $\frac{1}{2}$ の同値類が $\left\{x : x \sim \frac{1}{2} \wedge x \in \mathbf{N} \times (\mathbf{N} \setminus \{0\})\right\}$ で、分数 $\frac{1}{2}$ はこの同値類の代表です。

次に集合 S の重要な二項関係として「**順序関係**」を挙げておきます。この順序関係には、その性質から「**前順序 (preorder)**」、「**半順序 (partial order)**」、「**全順序 (total order)**」の三種類があります：

前順序

集合 S の関係 R が反射律と推移律を充たすときに**前順序 (preorder)**と呼びます：

1. 反射律: $x R x$
2. 推移律: $((x R y) \wedge (y R z)) \supset (x R z)$

半順序

集合 S の関係 R が前順序であり、さらに反対称律を充たすときに**半順序**と呼びます：

1. 反射律: $x R x$
2. 推移律: $((x R y) \wedge (y R z)) \supset (x R z)$
3. 反対称律: $((a R b) \wedge (b R a)) \supset (a = b)$

前順序と半順序では $a R b, b R a$ のどちらも成立しない $a, b \in S$ が存在することもあります。このときに a と b は「**比較不能 (incomparable)**」と呼びます。つぎの全順序では、任意の $a, b \in S$ に対して $a R b, b R a$ のいずれかが必ず成立します：

全順序

集合 S の関係 R が半順序、かつ全順序律を充たすときに「**全順序**」と呼びます：

1. 反射律: $x R x$
2. 推移律: $((x R y) \wedge (y R z)) \supset (x R z)$
3. 反対称律: $((a R b) \wedge (b R a)) \supset (a = b)$
4. 全順序律: $(a R b) \vee (b R a)$

集合の包含関係 “ \subseteq ” が半順序、整数の大小関係 “ \leq ” が全順序になります。実際、集合 $A = \{a, b, c\}$ の幂集合 $\mathfrak{P}A$ で包含関係 “ \subseteq ” は $\{a\} \subseteq \{b, c\}$ にならないために全順序関係になりません。また、整数は数直線上に並び、 $a \leq b$ であれば整数 a が数直線上で整数 b の左側に配置されるために大小関係 \leq が全順序であることが判ります。

2.6 代数的構造について

数学的対象から構成される集合には何らかの代数的な構造、つまり、演算が充たすべき規則を体系的にまとめた性質があります。この性質を算数の話から進めましょう。まず、算数で最初に扱う数は自然数 $1, 2, 3, \dots$ です。この自然数の計算には足算、引算、掛算と割算があります。ここで足算と掛算は割算と引算に比べて非常に機械的な操作で、新しい自然数を生成する能力がありますが、引算や割算はそうではありません。実際、割算 “ \div ” では小学生が分数を習うまで割り切れない数が存在するために

$$1 \div 2 = 0 \text{ あまり } 1$$

と商と剩余を併記したものを答とし、 $1 \div 2 = \frac{1}{2}$ と書きません。つまり、 $1 \div 2$ で新しい数を生成しても、それが自然数であるとは限らず、自然数でなければ受け入れ先がないために商と剩余の両方を記した計算結果になります。そこで、自然数の割算で商のみを結果として採用すると足算、掛算と同様に二つの自然数から自然数を対応させる写像になります。

さて、これらの性質をより普遍的なもので言い換えてみましょう。そこで対象の自然数を集合 S 、足算や割算といった記号を記号 “ $*$ ” と表記し、この記号を「**演算子**」、それから演算子 “ $*$ ” の影響を受ける集合 S の元を「**被演算子**」と呼びます。それから集合と演算の対 $(S, *)$ で表記し、足算、掛算の何れのことを話題にしているか明瞭にします。すると、ここで話題にしている演算の大きな性質は集合 S の二つの元から新たな集合 S の元を生成する能力で、この演算 “ $*$ ” が集合 S の対 $S \times S$ から S への写像になっていることです。この性質は「**閉じている**」と呼ばれる演算の性質で、この集合 S と閉じた演算 “ $*$ ” の対 $(S, *)$ を「**マグマ (magma)**」と呼びます：

——マグマの定義——

集合 S と演算 “ $*$ ” が任意の $a, b \in S$ に対して次の条件を充たすとき、 $(S, *)$ を「**マグマ (magma)**」と呼ぶ。

- 演算 “ $*$ ” が閉じていること: $a * b \in S$.

この定義から自然数の集合 \mathbb{N} の足算 “ $+$ ”，掛算 “ \times ”，割算 “ \div ” といった計算処理は全て閉じた演算で、したがって $(\mathbb{N}+)$, (\mathbb{N}, \times) と (\mathbb{N}, \div) はマグマになります^{*38}。ところで、これらの演算の詳細を眺めてみましょう。まず、足算 “ $+$ ”，掛算 “ \times ” に共通する性質に $(1 + 2) + 3 = 1 + (2 + 3)$, $(3 \times 4) \times 5 = 3 \times (4 \times 5)$ という性質があります。しかし、割算 “ \div ” は $(12 \div 6) \div 2 \neq 12 \div (6 \div 2)$ です。つまり、式の括弧を動かすと結果が違います

^{*38} (\mathbb{N}, \div) は商のみを返す演算とします。

が、この括弧は演算の順番に対応し、最初に $a * b$ を計算して c との演算を計算する方法の $(a * b) * c$ と $b * c$ を計算して a との演算を計算する方法の $a * (b * c)$ が等しくなるかどうかという状況に対応します。この ‘ $(a * b) * c = a * (b * c)$ ’ という関係は「**結合律**」と呼ばれる重要な関係式です。ところで日常の言葉で括弧 “()” がなければ微妙な解釈の違いが生じることがあります。たとえば「太った猫と犬」という文を「(太った猫)と犬」と解釈するか「太った(猫と犬)」[30] と解釈するかで意味が異なります*39。集合が結合律を充足すということは、このような曖昧さも排除した、演算の順番に依存せずに一意に演算結果が定まるということを意味します。その意味で、割算は本質的に足算や掛算と異なる演算であることを示しています。そこで、閉じた演算を持ち、結合律を充たすという二つの性質を持つ集合に新しい概念を導入しましょう。これが「**半群 (semigroup)**」と呼ばれる概念です：

半群の定義

集合 S と演算 “ $*$ ” が任意の $a, b, c \in S$ に対して次の条件を充たすとき、 $(S, *)$ を半群 (semigroup) と呼ぶ。

- 演算 “ $*$ ” が閉じていること: $a * b \in S$.
- 結合律を充たすこと: $a * (b * c) = (a * b) * c$.

自然数 \mathbb{N} に対して $(\mathbb{N}, +)$ と (\mathbb{N}, \times) が半群という概念で纏められますが、 (\mathbb{N}, \div) はマグマであっても半群でないために割算が足算や掛算と本質的に異なる演算として分類できます。このように個々の数学的対象を普遍性を持った性質で区分することが数学的構造を入れることに他なりません。そして、この区分は同じ集合であっても演算によって入る構造が異なります。実際、自然数 \mathbb{N} もその二項演算を足算 “+”，割算 “ \div ” のいずれかにするかで半群、あるいはマグマと異なる概念に分類されます。

さらに足算 “+” と掛算 “ \times ” という演算には面白い性質があります。これは $1 + 2 = 2 + 1 = 3$ や $2 \times 3 = 3 \times 2 = 6$ と任意の $a, b \in S$ に対して $a * b = b * a$ と演算子 “ $*$ ” の両側の集合 S の元を入れ替えても同じ結果になるという性質です。このように二項演算子で左右の被演算子を入れ替えても演算結果が等しくなる演算子の性質を「**可換 (commutative)**」と呼びます。ところで割算 “ \div ” は $4 \div 2 \neq 2 \div 4$ であるために可換ではありません。また、引算 “ $-$ ” も $1 - 2 \neq 2 - 1$ であるために可換にはなりません。このように可換でない演算子の性質を「**非可換 (noncommutative)**」と呼びます。

$(\mathbb{N}, +)$ の 0, (\mathbb{N}, \times) の 1 は共に面白い性質を持っています。まず、0 については

*39 「太った」の参照範囲 (スコープ) が猫だけなのか、猫と犬の両方なのかどうかが分からない文の構造のためです。

$a + 0 = 0 + a = a$, 1 については $1 \times a = a \times 1 = a$ を充たしています。これらの関係式を演算“*”， $0, 1$ を元 u で置き換えると任意の $a \in \mathbb{N}$ に対して $u * a = a * u = a$ という関係式が得られます。このように集合 S の閉じた演算“*”で任意の $a \in S$ に対して $a * u = u * a = a$ を充たす集合 S の元 u を「**単位元**」と呼びます。そして、単位元を持つ半群を「**単系（モノイド, monoid）**」と呼びます。したがって、 $(\mathbb{N}, +), (\mathbb{N}, \times)$ は単系であり、 0 が $(\mathbb{N}, +)$ の単位元、 1 が (\mathbb{N}, \times) の単位元です。

中学年になると分数が現れます。ここで分数 \mathbb{Q}_+ は分母と分子が 0 以外の自然数の数で $\frac{n}{m}$ と表記され、約分という操作が入った数です。この約分という操作は分数 $\frac{n}{m}$ の分子 n と分母 m が共通の約数 a を持つときに m と n を a で割ったもので置換えます。また、分母が 1 のときは分子のみを、つまり、整数として表記します。この約分は $m = a \times b$, $n = a \times c$ のときに $\frac{n}{m} = \frac{c}{b}$ とする関係^{*40}と言い換えることができます。それから分数は演算“ \times ”に対して 1 が単位元になります。また $a, b, c \in \mathbb{Q}$ に対して $a \times b \in \mathbb{Q}_+$, $a \times 1 = 1 \times a = a$, $(a \times b) \times c = a \times (b \times c)$ が成立するために演算“ \times ”は \mathbb{Q}_+ で閉じた演算で、 1 がその単位元で結合律も成立するために (\mathbb{Q}_+, \times) は単系になります。

さらに分数 \mathbb{Q}_+ の元には興味深い性質があります。ここで $a \in \mathbb{Q}_+$ であれば 0 と異なる二つの自然数の対 (m, n) で $a = \frac{n}{m}$ を充すものが存在します。この自然数の対 (m, n) に対して分数を $b = \frac{m}{n}$ で定めます。このときに整数の掛け算“ \times ”の可換性から $a \times b = \frac{n \times m}{m \times n} = \frac{m \times n}{n \times m} = b \times a = 1$ になることが判ります。このように分数 \mathbb{Q}_+ の元には掛け合せることで 1 になる元が存在します。この分数 \mathbb{Q}_+ の例のように単位元 u を持つ单系 $(S, *)$ で $a \in S$ に対して $a * b = b * a = u$ を充たす元 b のことを a の「**逆元（inverse）**」と呼び、 a^{-1} と表記します。また、逆元を持つ a を「**正則（regular）**」と呼びます。ここで挙げた (\mathbb{Q}_+, \times) の全ての元は正則で、このように全ての元が正則になる单系を特に「**群**」と呼びます：

群の定義

- $(S, *)$ は半群である。
- $(S, *)$ は単位元 u を持つ。
- $(S, *)$ の元は全て正則である。

では、 $(\mathbb{N}, +)$ と (\mathbb{N}, \times) は群でしょうか？ $(\mathbb{N}, +)$ では $a + b = 0$ になる元は双方が 0 のときだけで、 (\mathbb{N}, \times) でも $a \times b = 1$ になる元は双方が 1 のときだけです。したがって、これらの单系は群にはなりません。これらの单系が群になれない理由は任意の元に逆元が

^{*40} 同値関係と呼ばれる関係になります。

存在しないためです。だから逆元を追加してしまえば群になります。実際、 $(\mathbb{N}, +)$ に負の数を導入して整数 \mathbb{Z} を構築すれば、この整数 $\mathbb{Z}, +$ は群になります。また、 (\mathbb{N}, \times) に分数を導入して (\mathbb{Q}_+, \times) にすると群になります。なお、演算が可換な群のことを「**可換群 (commutative group)**」と呼び、さらにこの可換群の演算を記号 “+”，単位元を 0，そして、 a の逆元を $-a$ と表記して「**加法群 (additive group)**」と呼ぶことがあります。また、加法群でない群を「**乗法群 (multiplicative group)**」と呼び、演算を記号 “*”，単位元を 1, a の逆元を a^{-1} と表記します。これらの呼び名は自然数 \mathbb{N} や整数 \mathbb{Z} が足算 “+” と掛け算 “×” のような二つの演算を持つ数学的対象で、それらの演算を区別するときに用いられます。

ところで整数 \mathbb{Z} は足算と掛け算という二つの演算を持ち、演算 “+” なら可換群ですが、演算 “×” は演算 “+” の単位元 0 を除く集合 $\mathbb{Z} - \{0\}$ は単系で、群にはなりません。それから足算と掛け算が混在する計算で次の性質を充たします：

1. $a \times (b + c) = a \times b + a \times c$
2. $(a + b) \times c = a \times c + b \times c$

この性質は左右の掛け算 “×” を仲立ちにして被演算子を分配しているために「**分配律**」と呼ばれる性質です。この整数のように二つの演算 “+” と “*” を持つ、演算 “+” では可換群、もう一方の演算 “*” で可換群の単位元 0 を除くと半群としての構造を持つ集合 $(S, +, *)$ を「**環 (ring)**」と呼びます：

環の定義

- $(S, +)$ は単位元 0 を持つ可換群である。
- $(S - \{0\}, *)$ は半群である。
- $(S, +, *)$ は分配律を充す：

$$\begin{aligned} 1 \quad a * (b + c) &= a * b + a * c \\ 2 \quad (a + b) * c &= a * c + b * c \end{aligned}$$

さらに乗法 “*” にも単位元が存在するときに「**単位的環**」、乗法 “*” も可換である環を「**可換環 (commutative ring)**」、逆に可換環でない環を「**非可換環 (non-commutative ring)**」と呼びます。可換環の代表的なものが整数 \mathbb{Z} で、環としての構造を明確にすることは「**整数環**」と呼びます。この他に実数係数の多項式から生成される「**多項式環**」と呼ばれる環も非常に重要です。環の定義では乗法 “*” で群である要請はありませんが、乗法でも群になる環のことを「**斜体**」、乗法が可換な斜体を「**体 (field)**」と呼びます。斜体で代表的なものが n 次の実数成分の正方行列の集合 $M(n)$ で、体の代表的な例としては有理数 \mathbb{Q} 、実数 \mathbb{R} や複素数 \mathbb{C} が挙げられます。

この環に次いで重要な対象が「**環上の加群**」で「**R-加群**」で、先程の環は同じ集合で閉じた二つの演算でしたが、ここで積演算“*”を別の集合からの加法群への作用で置き換えたものになります。たとえば、次の分数計算はどうでしょうか？

$$4 \times \frac{5}{16} - 2 \times \frac{2}{5} \times \frac{15}{12}$$

この計算は整数と分数を含む計算になっています。これがR-加群の一つのモデルになります。実際、この式を

$$4 \times \left(\frac{5}{16} \right) + (-2) \times \left(\frac{2}{5} \times \frac{15}{12} \right)$$

と書き換えてみましょう。この式は左から環である整数 \mathbb{Z} の元を加法群である有理数 \mathbb{Q} の元と掛け合わせたものの和の計算を行っています。ここで有理数 \mathbb{Q} の元に整数 \mathbb{Z} の元を左側から掛けたものは有理数 \mathbb{Q} の元で、マグマで見られる閉じた演算に類似した状況です。そこで、整数を左側から有理数に掛けるという操作を整数 \mathbb{Z} の有理数 \mathbb{Q} への「**左から作用**」と呼びます。これを一般化して、環 $(R, +, *)$ の元 r による加法群 $(A, +)$ の元 a への左側の作用を $r * a$ 、右側の作用を $a * r$ と表記します。

次に整数と有理数との作用では、たとえば $(2 \times 3) \times \frac{2}{5} = 2 \times \left(3 \times \frac{2}{5} \right)$ から判るように整数側で積演算して有理数に作用させたものと、有理数に近い側から作用させていったものの結果が一致する性質があります。この性質は有理数と整数の積が有理数の分子との積になり、このときに双方が全て整数であることから結合律を充すために成り立つ性質です。これも作用の性質としてまとめると、左作用であれば $(r_2 * r_1) * a = r_2 * (r_1 * a)$ 、右作用であれば $a * (r_1 * r_2) = (a * r_1) * r_2$ を充たします。

また、この左右の作用には環側と加法群側の加法に対する分配律が成立します：

左作用に関する分配律

$$\begin{aligned} L_1. \quad r * (x + y) &= r * x + r * y \\ L_2. \quad (r + s) * x &= r * x + s * x \end{aligned}$$

右作用に関する分配律

$$\begin{aligned} R_1. \quad (x + y) * r &= x * r + y * r \\ R_2. \quad x * (r + s) &= x * r + x * s \end{aligned}$$

さて、整数が有理数に作用するとき、整数の掛算“×”の単位元1は有理数に手を加えない恒等写として作用しています。実際、 $1 = 1 \times 1 = 1 \times \dots \times 1$ であるために恒等写として作用するか、常に0に写す零写像のいずれかであるべきですが、零写像では何も面白く

ないため、我々が考察すべき作用では単位元 1 が恒等写像であることが好都合です。このように分数の式から作用という概念に到達しました。

このような環からの作用を持つ加群のことを「**環上の加群 (module)**」、「**R-加群**」と呼びます。なお、作用の方向によって左右の区別があります。これらの R-加群の定義を以下にまとめておきましょう：

左 R-加群の定義

- 集合 $(A, +)$ は可換群である。
- 環 R と集合 A には写像 $* : R \times A \rightarrow A$ が存在する。
- $r * (x + y) = r * x + r * y$
- $(r + s) * x = r * x + s * x$
- $(r * s) * x = r * (s * x)$
- $1 * x = x$

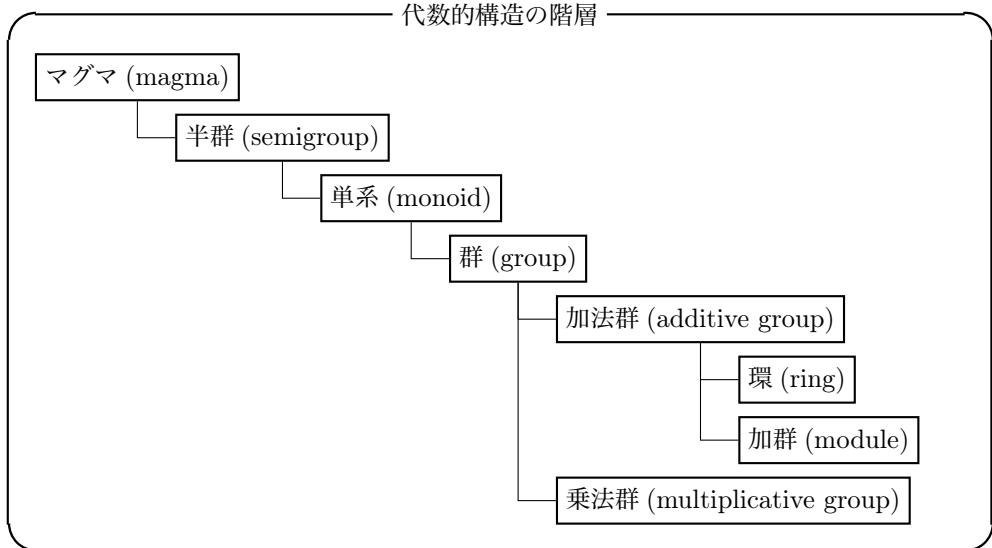
右 R -加群の定義

- 集合 $(A, +)$ は可換群である。
- 環 R と集合 A に写像 $* : A \times R \rightarrow A$ が存在する。
- $(x + y) * r = x * r + y * r$
- $x * (r + s) = x * r + x * s$
- $x * (s * r) = (x * s) * r$
- $x * 1 = x$

この左右の R -加群の定義で、この定義では環 R の積と和と R -加群の係数環 R の作用に対応する積と加法群としての和の演算子を同じ記号を使っています。もし、右からも左からも作用するときは「**両側 R-加群**」と呼びます。また、環 R が可換環で、左右からの環 R の作用が一致するとき ($r * x = x * r$) に R -加群と呼びます。

この R -加群には先程の整数 \mathbb{Z} の乗法 \times による作用を持つ加法群 \mathbb{Q} 、環を実数 \mathbb{R} 、作用を通常の積 \times による n 次の実ベクトル空間が挙げられます。また、 n 次のベクトル空間も R -加群の一例です。実際、 n 次の(実)ベクトルの空間 $\text{Vect}(n) = \{(v_1, \dots, v_n) : v_1, \dots, v_n \in \mathbb{R}\}$ とし、係数環として実数 \mathbb{R} の作用を各成分単位の積、つまり、 $v \in \text{Vect}(n)$, $a \in \mathbb{R}$ に対して $a \cdot v = (a \cdot v_1, \dots, v_n)$ で定めると、この作用は左右の R -加群の条件を充たすためです。

さて、ここまでにマグマ、半群、群、環、体、そして、加群といった代数的構造を説明しましたが、発生順に並べた樹形図として以下にまとめておきましょう：



この樹形図は §5.4.2 にて SageMath の数学的オブジェクト SageObject に関連して再度触ることにします。

2.7 圈 (Category)

2.7.1 はじめに

ここからは数学の「圈」について解説します。この「圈」は英語で「Category」が対応し、哲学用語ではアリストテレスの「カテゴリー (κατηγορία)」、その日本語訳として「範疇」が対応します。このカテゴリーを最初に扱ったアリストテレスの著作「範疇 (カテゴリー) 論」は「**真実を探求するための道具**」としての「**道具 (オルガノン (ὄργανον))**」と呼ばれる著作群の筆頭に置かれ、哲学を学ぶ上で最初に読まれるべき書物とされていましたが [1]、この圏論も数学の対象を語ることに関連するだけではなく、数学を研究する上の道具として扱うという意味で類似した立場にあります。実際、MacLane[45] は「Category」という言葉はアリストテレスとカントの「カテゴリー論」、「functor」はカルナップ (Carnap) の著作に由来すると述べています。この圏論はそもそも代数的位相幾何学のホモロジーと呼ばれる数学的対象の計算から現れたものですが、現在では、数学全般、さらには哲学にも関連するものとなっています。

2.7.2 メタグラフについて

圏の定義を行う前に、圏の定義に必要な用語を説明しなければなりません。そのためには「メタグラフ」という概念と関連する用語を導入します。さて、このメタグラフは下記の性質を持つ対象と矢(射)で構成されます:

メタグラフ (metagraph)

- **対象:** A, B, C, \dots
- **矢 (射) :** f, g, h, \dots
- **始域 (domain) と終域 (codomain) :** 矢は始域と終域と呼ばれる二つの対象の関係であり, 矢 f の始域を $\text{dom } f$, 終域を $\text{cod } f$ と表記する.
- **矢の表記:** 矢 f に対して $A = \text{dom } f$, $B = \text{cod } f$ とするとき
 $f : A \rightarrow B$, あるいは $A \xrightarrow{f} B$ と表記する.

この定義で対象の同一性を示す記号として記号 “=” を用います. ここでメタグラフの対象は集合の元, 矢は写像を抽象化した形式的な定義です. 実際, 対象とそれらの間の二項関係である矢の存在が前提であっても, それらが具体的にどのようなものであるか言及されておらず, それらのあつまりが集合になるかどうか言及がありません.

つぎに幾つかの記号を導入しておきます. まず, メタグラフ \mathcal{C} の対象のあつまり, すなわち「類 (クラス, class)」を $\text{Ob}\mathcal{C}$, 同様にメタグラフ \mathcal{C} の矢の類を $\text{Arr } \mathcal{C}$ と表記します. そして, メタグラフ \mathcal{C} の矢の始域になり得る対象で構成される類を \mathcal{C}_0 , 終域になり得る対象で構成される類を \mathcal{C}_1 と表記します. 同様に対象 A を始域, 対象 B を終域とするメタグラフ \mathcal{C} の矢から構成される類を $\text{Hom}_{\mathcal{C}}(A, B)$, $\mathcal{C}(A, B)$, あるいは $\text{Hom}(A, B)$ と表記します. そして, 後述の函手との関連で, $\text{Hom}(A, B)$ を $\text{H}_A(B)$ や $\text{H}^B(A)$ とも表記します. なお, メタグラフ \mathcal{C} の矢 f が対象 A を始域, 対象 B を終域とする矢のときに記号 “∈” を使って $A, B \in \mathcal{C}$, $f \in \mathcal{C}$ と表記したり, より詳細に $A \in \mathcal{C}_0$, $B \in \mathcal{C}_1$, および $f \in \text{Hom}_{\mathcal{C}}(A, B)$ と表記することで対象や矢のメタグラフ \mathcal{C} への包含関係を表示します.

ここでメタグラフ \mathcal{C} の矢 $f : A \rightarrow B$ はメタグラフ \mathcal{C} の二つの対象 A, B に順序を含めた関係を与え, この矢を \xrightarrow{f} と表記することで矢 $f : A \rightarrow B$ から $A \xrightarrow{f} B$ へと図式化が行えます. また, この図式化によって矢 f が対象 A と B を繋ぐ機能を持つものとしての性格が明瞭になります. その結果, メタグラフの矢は伝統的形式論理学での「繋辞 (copula)」と同様の機能を持つものと考えることができます. 繋辞は命題「 A は B である」の中の「... は... である」のように主語と述語の二項間を取り持ち, 命題を構成します [5]. また, 論理学への函数概念の導入はフレーゲ (Frege) が「概念記法」で行っていますが, 变数が二個以上の函数を特に「関係 (relation)」, 变数の位置を「項位置」と呼び, 関係の第一引数を ξ , 第二引数を ζ と表記しています. メタグラフの矢はこの関係をより抽象化し, その機能を明瞭にするために図式化をさらに推し進めたものになっています.

2.7.3 矢について

さて、「メタグラフ」に含まれる「グラフ」という言葉から「函数のグラフ」等の「グラフ」を連想される方も多いかと思います。函数グラフは点 x における函数 f の値 $f(x)$ を XY 平面上の点 $(x, f(x))$ として描いたもので、座標の表記で最初の成分が X 座標、つぎの成分が Y 座標と座標を構成する対の順序に重要な意味があります。そこで座標 $(x, f(x))$ を集合論言語 \mathcal{L} の順序対 $\langle x, f(x) \rangle$ として記述するとグラフ全体は $\{\langle x, f(x) \rangle : x \in A\}$ で外延として記述され、対象 A が ZFC 公理系の集合であれば、このグラフも置換公理図式から集合になります。さて、メタグラフの矢 $A \xrightarrow{f} B$ に対しても対象 A, B がともに集合であれば集合 $\{(x, y) : x \in A \wedge y \in B \wedge f x = y\}$ が得られます。ここで空集合 \emptyset を始域とするメタグラフの矢 f は空集合 \emptyset でなければならないことがグラフの外延から判ります。この実例として後述の Python のオブジェクト None 型が挙げられます。実際、None 型は空集合 \emptyset を始域とする矢と同様の働きを持っています。

次に「矢の合成」と呼ばれる矢の生成操作について解説しましょう。ここでの矢の合成は写像の合成の抽象化で、まず、 $\text{dom } f = \text{cod } g$ を充たす「合成可能対」と呼ばれる矢の順序対 $\langle f, g \rangle$ を考えます。それから、合成可能対で構成される類を $\text{Arr}\mathcal{C} \times_{\text{Ob}\mathcal{C}} \text{Arr}\mathcal{C}$ と表記して「合成可能類」と呼び、 $\text{Arr}\mathcal{C} \times_{\text{Ob}\mathcal{C}} \text{Arr}\mathcal{C}$ に属する順序対 $\langle f, g \rangle$ に対して $f \circ g$ を始域と終域をそれぞれ $\text{dom } f, \text{cod } g$ になる矢に対応させる操作とします。もちろん、この操作の可能はメタグラフで保証されませんが、この操作が可能なときに得られる矢を矢 f, g の「合成」と呼びます。ここで 3 個の矢 $f : C \rightarrow D, g : B \rightarrow C, h : A \rightarrow B$ の合成として $f \circ (g \circ h) = \langle f, \langle g, h \rangle \rangle$ と $(f \circ g) \circ h = \langle \langle f, g \rangle, h \rangle$ がそれぞれ構築できますが、これら一致するかどうかは一般的に正しいと言えません。これらの矢の合成が一致すること、すなわち、 $f \circ (g \circ h) = (f \circ g) \circ h$ を充すことを「結合律」と呼びます。そして、これらの矢が結合律を充たすときには合成の順序を問わないために括弧 “()” が不要になって $f \circ g \circ h$ と表記できます。

2.7.4 図式 (diagram) について

メタグラフ \mathcal{C} の対象と対象間の結合律を充す矢の類を「図式 (diagram)」と呼びます。図式は対象を頂点 (vertex)、矢の向きを持つ辺 (edge) の平面グラフとして図示することができます。たとえば、矢 $A \xrightarrow{f} B$ と矢 $B \xrightarrow{g} C$ の合成矢 $A \xrightarrow{g \circ f} C$ を図示すると

$$A \xrightarrow{f} B \xrightarrow{g} C$$

になります。なお、図式で一部の対象や矢を省略した表記もあります。たとえば、 $\text{cod } f_i =$

$\text{dom } f_{i+1}$ を充たす矢の列 $f_1, \dots, f_i, f_{i+1}, \dots, f_n$ を「**矢の道 (path)**」と呼び

$$\bullet \xrightarrow{f_1} \bullet \cdots \bullet \xrightarrow{f_i} \bullet \xrightarrow{f_{i+1}} \bullet \cdots$$

と省略記号“...”を含めて図示します。図式は対象、矢と省略記号“...”から構成され、図式の可視化は対象を頂点 (vertex), 矢を辺 (edge) とし、これらに省略記号“...”で構成される2次元グラフになります。また、辺を辿ることは矢の合成を行うことに対応し、図式を構成する矢が結合律を充たすために、これらの合成は一意に定まります。このことからも図式で矢の合成に由来する曖昧さを排除するためには矢が結合律を充たさなければなりません。また、図式中のある対象を始域とする複数の矢の道が存在し、それらの経路の何れを通っても道に対応する矢の合成が一致するときに「**可換図式**」と呼びます。

ここで重要な可換図式を以下に示しておきます：

$$\begin{array}{ccc} A & \xrightarrow{f} & B \\ & \searrow h & \downarrow g \\ & C & \end{array} \qquad \begin{array}{ccc} C & \xrightarrow{h} & A & \xrightarrow{\begin{matrix} f \\ g \end{matrix}} & B \\ & & \downarrow & & \\ & & C & & \end{array}$$

左の図式は可換図式で最も重要な図式で対象 A から対象 C に向う道として合成矢 $g \circ f$ と矢 h の二つが存在し、これら二つの矢が一致することを意味します。この矢の合成 $g \circ f$ を矢 h の「**分解 (factorization)**」と呼びます。また、右の図式では対象 C から B に向かう道として $C \xrightarrow{h} A \xrightarrow{f} B$ と $C \xrightarrow{h} A \xrightarrow{g} B$ の2つがあり、それぞれに対応する矢の合成 $f \circ h$ と $g \circ h$ が一致することを示しています。また、メタグラフ \mathcal{C} の対象からそれ自身への矢が考えられますが、そのような矢の中で特に任意の $f \in \text{Hom}_{\mathcal{C}}(A, B)$, $g \in \text{Hom}_{\mathcal{C}}(B, C)$ に対して

$$\begin{array}{ccccc} A & \xrightarrow{f} & B & & \\ & \searrow f & \downarrow \text{id}_B & \swarrow g & \\ & B & & & C \end{array}$$

を可換、すなわち、 $\text{id}_B \circ f = f \circ \text{id}_B$ 、かつ、 $\text{id}_B \circ g = g \circ \text{id}_B$ を充たす矢 $B \xrightarrow{\text{id}_B} B$ を「**同一矢 (恒等矢)**」と呼びます。なお、対象 B の同一矢は 1_B とも表記されますが、 1_B は後述の終対象 1 と紛らわしいために同一矢の表記では id_B を用います。この可換図式の意味することを「**同一矢の公理**」と呼びます。この公理から各対象に同一矢が必ず一つだけ

存在することが保障されます。実際、対象 A に対して二つの同一矢 id_A と ι_A が存在するとき、この公理から直ちに $\text{id}_A = \iota_A$ が導き出せるためです。このことから同一矢と対象は一対一に対応するために対象と同一矢を同一視することができます。

2.7.5 単射, 全射, 同型

矢は写像を抽象化したものであるために矢にも「**単射 (mono)**」、「**全射 (epi)**」と「**同型 (iso)**」といった写像に類似した性質があります：

—— 単射, 全射, 同型 ——

- **単射 (mono)**：任意の矢 $h, g : C \rightarrow A$ に対して $f \circ h = f \circ g$ を充せば $h = g$ のとき、 $f : A \rightarrow B$ と表記します。
- **全射 (epi)**：任意の矢 $h, g : B \rightarrow C$ に対して $h \circ f = g \circ f$ を充せば $h = g$ のとき、 $f : A \rightarrow B$ と表記します。
- **同型 (iso)**： $g \circ f = \text{id}_A$ かつ $f \circ g = \text{id}_B$ を充す矢 $f : A \rightarrow B$ と $g : B \rightarrow A$ が存在するときに矢 f が「**可逆 (invertible)**」であると呼び、矢 g を f^{-1} 、対象 A, B の関係を $A \cong B$ と表記します。

矢 f が単射であれば $f \circ g = f \circ h$ ならば $g = h$ であるために左側の矢 f が除去可能 (cancelable)，すなわち、「**左簡約可能 (left cancellable) な矢**」であると呼びます。同様に矢 f が全射であれば $g \circ f = h \circ f$ から $g = h$ であるために右側の矢 f が除去可能，すなわち、「**右簡約可能 (right cancellable) な矢**」であると呼びます。

これら単射、全射、同型といった矢の性質は対象が集合であれば通常の写像の単射、全射、同型に対応します。実際、対象が集合であれば矢は通常の写像が対応するために単射で全射であれば同型になります。なお、圏の対象が集合であると限らないために勝手が異なり、矢 f が同型であれば矢 f は単射、かつ全射ですが、矢 f が単射、かつ全射であっても同型になるとは限りません。また、対象 $A, B \in \mathcal{C}$ が $A \cong B$ であるということは、二つの対象に双方からの対応関係が存在し、その関係によって対象の構造に一致が見られ、対象 A と B を同じ対象として見做すことができることを意味します。とはいえる $A \cong B$ であることは $A = B$ という二つの対象の同一性を保証するものではありませんが、この同型による関係で後述の始対象や終対象、対象の積や幂が定義されます。

分裂单射と分裂全射

- **分裂单射 (split mono)** : $g \circ f = \text{id}_A$ を充たす矢 $B \xrightarrow{g} A$ (左逆矢) が存在するときに矢 $A \xrightarrow{f} B$ を「**分裂单射 (split mono)**」と呼びます.
- **分裂全射 (split epic)** : $f \circ g = \text{id}_B$ を充たす矢 $B \xrightarrow{g} A$ (右逆矢) が存在するときに矢 $A \xrightarrow{f} B$ を「**分裂全射 (split epic)**」と呼びます.

矢 f が同型であることは、矢 f が分裂单射かつ全射であることと矢 f が分裂全射かつ单射であることと同値です.

2.7.6 圈について

メタグラフ \mathcal{C} が「**メタ圈**」であるとはメタグラフ \mathcal{C} の矢に対して矢の合成が可能で同一矢の公理と結合律を充すときです:

メタ圈 (metacategory)

メタグラフ \mathcal{C} が次の性質を充すときにメタ圈と呼ぶ:

- 矢の合成が可能である
- 同一矢の公理を充す
- 矢の合成について結合律を充す

メタグラフでは対象の間に矢という関係の存在のみが要請としてありました、メタ圈では矢の合成という演算が導入され、矢が結合律と同一矢の公理を充たすことから「**单系 (モノイド, monoid)**」の構造が加わり、さらに図式も一意に定まります。しかし、対象や矢のあつまり (類) が集合になるとは限りません。そこでメタグラフやメタ圈に対象と矢が構成する類が集合となるという要請を入れましょう。この要請を入れることで、対象や矢の類が集合として扱え、その結果、それらを扱う後述の函手や自然変換が通常の函数になることを意味します。この対象と矢の類がそれぞれ ZFC 公理系の集合になるメタグラフ \mathcal{C} を「**グラフ**」、そのようなメタ圈を「**圈**」と呼び、以後、これらを中心に考察します:

グラフ (graph) の定義

グラフ \mathcal{C} を次の性質を充すものとして定義します:

- 対象 A, B, C, \dots を包含する集合 \mathbf{O}
- 矢 f, g, h, \dots を包含する集合 \mathbf{A}
- 関数 $\text{dom}, \text{cod}: \mathbf{A} \rightarrow \mathbf{O}$
 $f \in \mathbf{A}$ に対し $\text{dom } f$ を始域, $\text{cod } f$ を終域と呼ぶ
- 矢 f の図式: 矢 $f \in \mathbf{A}$ に対し $A = \text{dom } f, B = \text{cod } f$ であれば
 f の図式は $f: A \rightarrow B$ あるいは $A \xrightarrow{f} B$ で与えられる

なお, グラフ \mathcal{C} の対象全体の集合 O を $\text{Ob } \mathcal{C}$, 同様に矢全体の集合 A の集合を $\text{Arr } \mathcal{C}$ と表記します^{*41}. このグラフでは新たな矢を生成する矢の合成と矢の合成に関する結合律, それと可換図式で表現される同一矢の存在についての言及はありません. 矢の合成に関するこれらの性質を加味したグラフが「圏」になります:

圏 (category)

グラフ \mathcal{C} が次の性質を充すときに圏と呼ぶ:

- 矢の合成を持つ
- 同一矢の公理を充す
- 矢の合成について結合律を充す

この圏の定義から判るように圏は必ずその対象に対応する同一矢を包含し, 同一矢の公理から同一矢は矢の合成で単位元としての働きを持ち, さらに矢の合成と結合律を充たすために矢の集合は単系の構造を持ちます. また, 対象の集合と矢の集合は集合論言語 \mathcal{L} という言語を持つことになり, このことが圏論の「豊潤さ」に係ってきます.

また, 圏 \mathcal{C} の部分圏」を以下で定義します:

部分圏 (subcategory)

対象と矢で構成された \mathcal{A} が次の 3 つの条件を充たすときに圏 \mathcal{C} の部分圏と呼ぶ:

- \mathcal{A} の対象 A は圏 \mathcal{C} の対象で, その恒等矢 id_A も \mathcal{A} の矢である.
- \mathcal{A} の矢 f は圏 \mathcal{C} の矢で, その始域 $\text{dom } f$ と終域 $\text{cod } f$ が \mathcal{A} の対象である.
- \mathcal{A} の二つの矢 $A \xrightarrow{f} B, B \xrightarrow{g} C$ の合成 $A \xrightarrow{g \circ f} C$ も \mathcal{A} の矢である.

ここで 部分圏の例として図式を捉えることもできます. 先程の図式の説明では対象の同一矢の存在について触れていませんが, 同一矢の存在も含めると図式を部分圏として見な

*41 グラフ \mathcal{C} を有向グラフと看做すとき $(\text{Ob } \mathcal{C}, \text{Arr } \mathcal{C}, \text{dom}, \text{cod})$ を「**箭 (quiver)**」と呼びます.

すことができます。また、圏 \mathcal{C} の部分圏 \mathcal{A} が圏 \mathcal{C} の全ての矢を包含しているとき、つまり、任意の $A, B \in \text{Ob}\mathcal{A}$ に対して $\text{Hom}_{\mathcal{C}}(A, B) = \text{Hom}_{\mathcal{A}}(A, B)$ を充たすときに部分圏 \mathcal{A} を圏 \mathcal{C} の「**充足部分圏 (full subcategory)**」と呼びます。たとえば群を対象、群準同型を矢とする圏 **Grp** とアーベル群を対象、群準同型を矢とする圏 **Ab** では、圏 **Ab** が圏 **Grp** の充足部分圏になりますが、対象を集合、矢を連続写像とする圏 **Set** と対象を集合、矢を全単射写像とする圏 \mathcal{C} であれば圏 \mathcal{C} は圏 **Set** の部分圏であっても充足部分圏になりません。

圏 \mathcal{C} の対象と矢に対し、対象については同一矢、矢については、その矢の始域と終域を入れ替える操作を考えます。つまり、圏 \mathcal{C} の対象 A はそのまま対象 A に対応させ、矢 $f : A \rightarrow B$ を矢 $f^{\text{op}} : B \rightarrow A$ で置き換える操作です。この操作によって二つの矢 $f : A \rightarrow B$ と $g : B \rightarrow C$ の合成 $g \circ f$ に矢 $(g \circ f)^{\text{op}}$ が対応し、矢の合成の方法から矢 $f^{\text{op}} \circ g^{\text{op}}$ になります。この操作 ${}^{\text{op}}$ で得られる圏を「**双対**」と呼びます。この操作 ${}^{\text{op}}$ は対象に対しては恒等矢で、矢に対しては、その矢の始域と終域を入れ替える操作で、矢の合成に対してはその合成が逆順になります。この双対によって圏 \mathcal{C} から新しい圏が構築され、この圏のことを圏 \mathcal{C} の「**双対圏**」と呼んで \mathcal{C}^{op} と表記します。また、 $\mathcal{C} = \mathcal{C}^{\text{op}}$ を充たす圏 \mathcal{C} を「**自己双対 (self-dual)**」と呼びます。

2.7.7 圈の例

最初に有限個の対象を持つ圏の例を挙げておきましょう：

- **0** = (なにもない圏)
- **1** = •. (一つの対象のみ、同一矢の他の矢を持たない圏)
- **2** = • → • (対象が二つで同一矢の他に矢が一つの圏)
 - → •
- **3** =  (対象が三つで同一矢の他の矢が三つの圏)
- **4** = • → • (対象が二つで同一矢と異なる矢が二つ)

ここでの **0** と **1** は圏の名称であり、後述の始対象 0 と終対象 1 と異なります。

つぎに矢が対象に対応する同一矢のみの圏は「**離散圏**」と呼ばれます。ここで対象 $A \in \mathcal{C}$ の同一矢 $A \xrightarrow{\text{id}_A} A$ に繫辞の「... は... である」を対応させると $A \xrightarrow{\text{id}_A} A$ は「 A は A である」と解釈できますが、このときに離散圏は任意の対象 $A \in \mathcal{C}$ に対して「 A は A である」としか主張できない圏です。これは犬儒派のアンティステネス (*Antisthenes*) の

主張する「一つの主語は一つの述語あるのみ」^{*42}と普遍を認めない立場に対応します。

その他の重要な圏の例を挙げておきましょう：

- **Set**: 対象が集合, 矢が通常の写像
- **Set_{*}**: 対象が基点付きの集合, 矢が基点を基点に写す写像
- **Grp**: 対象が群, 矢が準同型写像
- **Ab**: 対象が可換群 (アーベル群), 矢が準同型写像
- **Rng**: 対象が環, 矢が環準同型写像
- **CRng**: 対象が可換環, 矢が環準同型写像
- **Field**: 対象が体, 矢が体準同型写像
- **Vect_K**: 対象が係数体 K のベクトル空間, 矢が準同型写像
- **Top**: 対象が位相空間, 矢が連続写像
- **Top_{*}**: 対象が基点付きの位相空間, 矢は基点を保つ連続写像
- **Cat**: 対象が圏, 矢が後述の函手
- \mathcal{C}^{op} : 圏 \mathcal{C} の双対圏

これらの集合, 圏と位相空間の対象は「**小集合 (small set)**」, 「**小圏 (small category)**」, 「**小位相空間 (small topological space)**」と呼ばれ, これらの「小」は「**グロタンディーク宇宙 (Grothendieck Universe)**」との関係を表します。すなわち, グロタンディーク宇宙は圏の対象全てと後述の対象の演算結果を含む大きな類で U と表記し, 対象は集合であって, グロタンディーク宇宙の元として包含されるために「**小**」, その一方でグロタンディーク宇宙そのものはそれ自身の元として包含されないために「**大**」と呼ばれます。また, 圏 \mathcal{C} が「**局所的に小さい**」とは任意の 対象 $A, B \in \mathcal{C}$ に対し, その矢の集合 $\text{Hom}(A, B)$ が小集合になるときで, 同様に「**小さい**」とは対象全体の集合 \mathcal{C}_0 と矢全体の集合 \mathcal{C}_1 の双方が小集合になるときです。

2.7.8 グロタンディーク宇宙について

さきほどの圏の例で, 対象が「**小集合**」等と小が付くものを示しました。ここでの大小はグロタンディーク宇宙との関係で決まると言いました。ここではもう少し細かく説明することにしましょう。まず, 「**グロタンディーク宇宙**」に包含される対象なら小で, 宇宙そのものは大となります。このグロタンディーク宇宙は集合の公理系に類似する公理を充す対象の集合です。圏の場合, 対象や矢の類は集合を構成し, それらの集合に対してグロタンディーク宇宙にて次の演算が許容されています:

^{*42} 形而上学 [2] 5 卷 29 章, 1024b34

—— グロタンディーク宇宙で許容される演算 ——

$$\text{対集合 } \{ \} \quad \{u, v\} \stackrel{\text{Def.}}{=} \{(x, y) : x \in u \wedge y \in v\}$$

$$\text{順序対 } () \quad \langle u, v \rangle \stackrel{\text{Def.}}{=} \{\{u\}, \{u, v\}\}$$

$$\text{直積 } \times \quad u \times v \stackrel{\text{Def.}}{=} \{\langle x, y \rangle : x \in u \wedge y \in v\}$$

$$\text{幂集合 } \mathfrak{P} \quad \mathfrak{P}(u) \stackrel{\text{Def.}}{=} \{v : v \subset u\}$$

$$\text{和集合 } \cup \quad \cup u \stackrel{\text{Def.}}{=} \{x : x \in u\}$$

これらの演算は ZFC 公理系であれば問題なく充される集合の演算処理で、以下に示す性質が成立する集合 U を「**グロタンディーク宇宙 (Grothendieck Universe)**」と呼びます：

—— グロタンディーク宇宙 U が充すべき性質 ——

- (i) $x \in u \in U \supset x \in U$
- (ii) $u \in U \wedge v \in U \supset \{x, y\} \in U \wedge \langle x, y \rangle \in U$
- (iii) $x \in U \supset \mathfrak{P}(u) \in U \wedge \cup u \in U$
- (iv) $\omega \in U$
- (v) $a \in U \wedge b \subset U \wedge a \rightarrow b$ が上への写像 $\supset b \in U$

グロタンディーク宇宙 U 自体は ZFC 公理系の「**正則公理**」によって U の元として含まれることはありません。さらに、この宇宙 U が集合になるとも限りません。このように宇宙 U とその元には区分があり、この区分を対象が集合であれば宇宙 U の元を「**小集合**」、対象が圈であれば U の元を「**小圈**」と宇宙 U の元を、その元の型の頭に「**小 (small)**」を付けます。逆に宇宙 U は頭に「**大 (large)**」を付けます。たとえば対象が集合であれば「**大集合**」、圈であれば「**大圈**」といったあんばいです。

2.7.9 始対象と終対象

重要な圈の対象として「**始対象 (initial object)**」と「**終対象 (terminal object)**」を定義します。これらの対象はのちのトポスの定義で必要になります：

—— 始対象と終対象 ——

- 圈 \mathcal{C} の対象 A が「**始対象 (initial object)**」であるとは任意の対象 $B \in \mathcal{C}$ に対して矢 $f : A \rightarrow B$ が一意に存在するときである。このとき始対象 A を 0 、矢 f を 0_A と表記する。
- 圈 \mathcal{C} の対象 B が「**終対象 (terminal object)**」であるとは任意の対象 $A \in \mathcal{C}$ に対して矢 $f : A \rightarrow B$ が一意に存在するときである。このとき終対象 B を 1 、矢 f を $!_A$ と表記する。

これら始対象、終対象は圏によって存在するとも限りません。また、始対象や終対象が存在したとしても、それらが唯一であるとは限りませんが、同型矢で一意に定まり、始対象 0 と終対象 1 は後述の反変函手 op で終対象 1 と始対象 0 にそれぞれ写され、始対象 0 と終対象 1 が双対関係にあります。また、圏 **Set** で終対象 1 から対象 A への矢が小集合 A の要素を指示する写像になることから、これを一般化して圏 \mathcal{C} の始対象 1 から対象 $A \in \text{Ob}\mathcal{C}$ への矢を圏 \mathcal{C} の対象 A の「要素 (element)」と呼びます。

ここで始対象と終対象の例を挙げておきます。圏 **Set** の始対象 0 は空集合 \emptyset です。実際、 $A \in \text{Ob}\text{Set}$ に対して \emptyset から A への矢として $\emptyset \xrightarrow{\emptyset} A$ のみが存在するためです^{*43}。圏 **Rng** の始対象 0 は整数 \mathbf{Z} 、一方で圏 **Field** には始対象が存在しません。次に **Set**, **Grp**, **Ab** と **Rng** の各圏の終対象は成分が一つの集合、すなわち、「**シンゲルトン (singleton)**」ですが、圏 **Field** には始対象も終対象も存在しません。また、圏 **Set** では矢 $1 \rightarrow A$ が集合 A の成分を定めます^{*44}。このように始対象と終対象の双方が存在する圏が存在する一方で始対象と終対象のどちらも存在しない圏があります。

2.7.10 函手

圏 \mathcal{C} と圏 \mathcal{D} に対して、圏 \mathcal{C} の対象を圏 \mathcal{D} の対象に、同時に圏 \mathcal{C} の矢を圏 \mathcal{D} の矢に対応付けることができます。さらに圏では対象や矢の類が集合になるため、これらの対象と矢単位の対応付けは写像になります。また、矢の写像については圏 \mathcal{C} の同一矢 id_A を圏 \mathcal{D} の同一矢 id_B に写し、矢の合成もそれらの像の合成になると都合が良くなります。すなわち、圏 \mathcal{C} の二つの矢 $A \xrightarrow{f} B$ と $B \xrightarrow{g} C$ の合成 $A \xrightarrow{g \circ f} C$ が写像 $F : \mathcal{C} \rightarrow \mathcal{D}$ で $F(g \circ f) = Fg \circ Ff$ 、あるいは $F(g \circ f) = Ff \circ Fg$ のどちらかに写せは良いでしょう。ここで前者は矢の始域と終域をそのまま写し、矢の合成の順序も保つ矢の写像、後者は矢の始域と終域を入れ替え、矢の合成の順序が反転する写像になっています。ここで矢の写像が前者の性質を持つ函手を「**共変函手 (covariant functor)**」、後者の性質をもつものを「**反変函手 (contravariant functor)**」と呼びます。

最初に共変函手の定義を示しておきましょう：

^{*43} Python の None オブジェクトはここでの \emptyset と同様の働きをします。

^{*44} 圏 **Set** では新プラトン主義者が聞けば喜びそうな「一者からの流出」によって万物は定義されていると言えます。

共変函手 (covariant functor) —

圏 \mathcal{C} から圏 \mathcal{D} の写像 F で以下の性質を充すものを「**共変函手**」と呼ぶ:

- $C \in \text{Ob}\mathcal{C}$ に対し $F C \in \text{Ob}\mathcal{D}$
- 圈 \mathcal{C} の矢 $A \xrightarrow{f} B$ に対して圏 \mathcal{D} の矢 $F A \xrightarrow{F f} F B$ が対応
- $F \text{id}_A = \text{id}_{FA}$
- $F(g \circ f) = F g \circ F f$

共変函手のことを単に**函手 (functor)**とも呼びます。この本でも誤解がない限り、単に函手と呼ぶときは共変函手のことを指します。この共変函手の例として圏 \mathcal{C} の対象や矢をそれ自身に対応させる「**恒等函手 (identity functor)**」 $\text{id}_{\mathcal{C}} : \mathcal{C} \rightarrow \mathcal{C}$ が挙げられます。つぎに圏 **Grp** から圏 **Set** への函手で、群の演算に関する性質、つまり代数的構造をすっかり無視して単純に集合として写す函手があります。この函手のように始域の圏が持っている群といった集合に付加されていた「**構造を忘れてしまう函手**」を「**忘却函手 (forgetful functor)**」と呼びます。忘却函手は環の圏 **Rng**、位相空間の圏 **Top** や可換群の圏 **Ab** から小集合の圏 **Set** への函手等があります。

つぎに反変函手の定義を示しておきます:

反変函手 (contravariant functor) —

圏 \mathcal{C} から圏 \mathcal{D} の写像 F で以下の性質を充すものを「**反変函手**」と呼ぶ:

- $C \in \text{Ob}\mathcal{C}$ に対し $F C \in \text{Ob}\mathcal{D}$
- 圈 \mathcal{C} の矢 $A \xrightarrow{f} B$ に対して圏 \mathcal{D} の矢 $F B \xrightarrow{F f} F A$ が対応
- $F \text{id}_A = \text{id}_{FA}$
- $F(g \circ f) = F f \circ F g$

反変函手の例として、対象はそれ自身に写し、矢の始域と終域を入れ替える双対 ${}^{\text{op}} : \mathcal{C} \rightarrow \mathcal{C}^{\text{op}}$ が挙げられます。実際、函手 ${}^{\text{op}}$ によって対象 $A \in \text{Ob}\mathcal{C}$ と同一矢はそのままで、矢 $A \xrightarrow{f} B$ は $B \xrightarrow{f^{\text{op}}} A$ に、 $B \xrightarrow{g} C$ との合成矢 $A \xrightarrow{g \circ f} C$ は $C \xrightarrow{f^{\text{op}} \circ g^{\text{op}}} A$ に写されるためです。この双対 ${}^{\text{op}}$ に対しては「**双対原理 (duality principle)**」と呼ばれる原理があります。これは S を圏 \mathcal{C} の対象と矢に関して真である命題 S の矢の向きを逆にした命題 S^{op} が双対圏 \mathcal{C}^{op} で成立し、逆に、双対圏 \mathcal{C}^{op} で成立する命題 S は圏 \mathcal{C} で成立することです。双対函手 ${}^{\text{op}}$ による圏 \mathcal{C} と圏 \mathcal{C}^{op} 上の命題との代表的な対応関係を以下にまとめておきます:

圏 \mathcal{C} 上での命題	圏 \mathcal{C}^{op} 上での命題
$A \in \text{Ob}\mathcal{C}$	$A^{\text{op}} = A \in \text{Ob}\mathcal{C}^{\text{op}}$
$A \xrightarrow{f} B$	$B \xrightarrow{f^{\text{op}}} A$
$A = \text{dom } f$	$A = \text{cod } f^{\text{op}}$
id_A	$\text{id}_A^{\text{op}} = \text{id}_A$
$g \circ f$	$(g \circ f)^{\text{op}} = f^{\text{op}} \circ g^{\text{op}}$
$f: \text{mono}$	$f^{\text{op}}: \text{epi}$
$A: \text{始対象}$	$A: \text{終対象}$

また双対函手の性質で $\mathcal{C}^{\text{op op}} = \mathcal{C}$ になります.

圏 \mathcal{C} から \mathcal{D} への函手 $F: \mathcal{C} \rightarrow \mathcal{D}$ は集合から集合への写像であるために通常の写像の議論が可能で、函手が单射、全射、同型になる場合を考えられます。まず、函手 F に対して $GF = \text{id}_{\mathcal{C}}$ と $FG = \text{id}_{\mathcal{D}}$ を充す逆向きの函手 $G: \mathcal{D} \rightarrow \mathcal{C}$ が存在するときに函手 F を「**同型 (isomorphism)**」と呼び、圏 \mathcal{C}, \mathcal{D} に同型な函手が存在するときに $\mathcal{C} \cong \mathcal{D}$ と表記します。また、函手 F が「忠実 (faithfull)」であるとは函手 F が矢の集合に関して单射になる場合です。具体的には共変なら写像 $\text{Hom}_{\mathcal{C}}(A, B) \xrightarrow{F} \text{Hom}_{\mathcal{D}}(FA, FB)$ が、函手 F が反変なら写像 $\text{Hom}_{\mathcal{C}}(A, B) \xrightarrow{F} \text{Hom}_{\mathcal{D}}(FB, FA)$ が单射になるときで、同様に、この写像 F が全射になるときに「**充满 (full)**」と呼びます。また、圏 \mathcal{C} から圏 \mathcal{D} の函手 F が「**稠密 (dense)**」であるとは、任意の対象 $B \in \text{Ob}\mathcal{C}$ に対して $FA \cong B$ を充す対象 $A \in \text{Ob}\mathcal{D}$ が存在するときです。そして、圏 \mathcal{C} から \mathcal{D} への函手 F が稠密、忠実、充满であるときに圏 \mathcal{C} と \mathcal{D} を「**同値 (equivalent)**」であると呼び、 $\mathcal{C} \simeq \mathcal{D}$ と表記します。なお、この圏の関係 “ \simeq ” は同値関係になります。

圏 \mathcal{C} から小集合の圏 **Set** への函手を圏 \mathcal{C} 上の「**集合値函手**」と呼びます。この集合値函手の共変、反変函手で重要な例を挙げておきましょう。圏 \mathcal{C} の対象 A から B の矢の類 $\text{Hom}_{\mathcal{C}}(A, B)$ とその双対 $\text{Hom}_{\mathcal{C}}(B, A)$ は圏の性質から集合になります。このときに $H_A: B \rightarrow \text{Hom}_{\mathcal{C}}(A, B)$ と $H^A: B \rightarrow \text{Hom}_{\mathcal{C}}(B, A)$ は圏 \mathcal{C} 上の集合値函手で、 H_A が圏 \mathcal{C} から圏 **Set** への共変函手、 H^A が圏 \mathcal{C}^{op} から圏 **Set** への反変函手です。なお、集合値函手 F が $F = H_A$ あるいは $F = H^A$ となる対象 $A \in \text{Ob}\mathcal{C}$ が存在するときに函手 F を「**表現可能 (representable)**」と呼びます。

2.7.11 自然変換

圏 \mathcal{C} から圏 \mathcal{D} への二つの函手の間に「**自然変換**」と呼ばれる対応関係が入ることがあります。まず、二つの函手 $F, G: \mathcal{C} \rightarrow \mathcal{D}$ が与えられたとき、圏 \mathcal{C} の矢 $A \xrightarrow{f} B$ の始域

と終域になる対象 $A, B \in \text{Ob}\mathcal{C}$ は函手 F によってそれぞれ $FA, FB \in \text{Ob}\mathcal{D}$ に写され, また, 矢 f 自体も圏 \mathcal{D} の矢 $FA \xrightarrow{Ff} FB$ に写されます. これは函手 G も同様で, 対象は $GA, GB \in \text{Ob}\mathcal{D}$ に矢は $GA \xrightarrow{Gf} GB$ へとそれぞれ写されます. それから函手 F と函手 G で写される対象 FA と GA , また, FB と GB の関係が考えられます. この FX から GX への対応関係を α_X と表記しましょう. この対応関係からは圏 \mathcal{C} の矢 $A \xrightarrow{f} B$ の始域 A と終域 B に写像 α による関係がそれぞれあるために $Gf \circ \alpha_B$ と $\alpha_A \circ Ff$ が等しいことは自然な要請になります. この写像 α を自然変換と呼びます:

自然変換 (natural transformation)

二つの函手 $F, G : \mathcal{C} \rightarrow \mathcal{D}$ に対して右下の図式を可換にする写像 α を「**自然変換**」と呼び, $\alpha : F \rightarrow G$ と表記する:

$$\begin{array}{ccc} A & & FA \xrightarrow{\alpha_A} GA \\ \downarrow f & & \downarrow Ff \\ B & & FB \xrightarrow{\alpha_B} GB \end{array}$$

左側の図式が矢 $A \xrightarrow{f} B$ で右側が矢 f に関する自然変換の可換図式になります. また, 圈 \mathcal{C} から圏 \mathcal{D} への函手 F から G への自然変換のあつまりを $\text{Nat}(F, G)$ と表記します. ここで自然変換と函手との間には「**米田の補題**」と呼ばれる非常に有名な補題があります:

米田の補題

圏 \mathcal{C} とその上の集合値函手 F に対して $\theta(\eta) = \eta_A(\text{id}_A)$ で定められる写像 $\theta : \text{Nat}(\text{H}_A, F) \rightarrow F(A)$ は全単射になる.

$\eta \in \text{Nat}(\text{H}_A, F)$, $A \xrightarrow{f} B$ とするときに η は自然変換であるために図式

$$\begin{array}{ccc} A & & \text{H}_A(A) \xrightarrow{\eta_A} F(A) \\ \downarrow f & & \downarrow H_A(f) \\ B & & \text{H}_A(B) \xrightarrow{\eta_B} F(B) \end{array}$$

が可換になります. ところで $f \in \text{H}_A(B)$ で, f の自然変換 η_B による像 $\eta_B(f)$ は $f = \text{H}_A(f)(\text{id}_A)$ であることと, この可換図式から $\eta(\text{H}_A(f)(\text{id}_A)) = F(f)(\eta_A(\text{id}_A))$. このことから写像 θ による η の像是 $\eta_A(\text{id}_A)$ で決定づけられ, 以上から写像 θ は単射になります. つぎに θ が全射であることを示しましょう. そのために $a \in F(A)$ に対して写像 $a_B^* : \text{H}_A(B) \rightarrow F(B)$ を $a_B^*(f) \stackrel{\text{def}}{=} F(f)a$ で定めます. ここで a^* が自然変換になることを示すために $g \in \text{H}_B(C)$ に対して図式

$$\begin{array}{ccc}
 B & \xrightarrow{\quad H_A(B) \quad} & F(B) \\
 \downarrow g & \downarrow H_A(g) & \downarrow F(g) \\
 C & \xrightarrow{\quad H_A(C) \quad} & F(C)
 \end{array}$$

が可換になることを確認できれば十分です。そのためには $F(g)(a_B^*(f)) = a_C^*(H_A(g)(f))$ を示さなければなりません。ここで

$$F(g)(a_B^*(f)) = F(g)(F(f)(a)) = F(g \circ f)(a)$$

であり、また

$$a_C^*(H_A(g)(f)) = a_C^*(g \circ f) = F(g \circ f)(a)$$

になります。したがって a^* は自然変換であり、また、 $\theta(a^*) = a$ であるために写像 θ が全射であることが判り、以上から米田の補題が証明できます。なお、 $\text{Nat}(H^A, F) \cong F(A)$ についても $H^A(B) = \text{Hom}_{\mathcal{C}}(B, A) = (\text{Hom}_{\mathcal{C}}(A, B))^{\text{op}}$ から同様に証明できます。

また、函手と自然変換から圏を構成することができます。すなわち、対象を圏 \mathcal{C} から圏 \mathcal{D} への函手、矢をそれらの間の自然変換とすることで圏が構成できます。この方法で定まる圏を $\mathcal{D}^{\mathcal{C}}$ と表記します。この構成による圏で重要なものに圏 $\text{Set}^{\mathcal{C}^{\text{op}}}$ を挙げておきましょう。この $\text{Set}^{\mathcal{C}^{\text{op}}}$ の対象である集合値の反変函手を「前層 (presheaf)」と呼びます。また、函手 $Y : \mathcal{C} \rightarrow \text{Set}^{\mathcal{C}^{\text{op}}}$ を $A \in \text{Ob}\mathcal{C}$ に対しては $Y(A) = H^A$, $A \xrightarrow{f} B$ と $C \xrightarrow{g} A$ に対しては $Y(f)_{\mathcal{C}}(g) = f \circ g$ で定めると、函手 Y は \mathcal{C} から $\text{Set}^{\mathcal{C}^{\text{op}}}$ への充満な埋め込みになります。この函手 Y を「米田の埋め込み (Yoneda embedding)」と呼びます。

ここで自然変換を使って「図式 (diagram)」を定義しなおします:

— J 型の図式 (diagram) の定義 —

圏 \mathcal{C} , J で定まる圏 \mathcal{C}^J の対象 $D : J \rightarrow \mathcal{C}$ を圏 \mathcal{C} 上の J 型 (J -type) の「図式 (diagram)」, さらに $i \in \text{Ob}J$ に対して $D_i \in \text{Ob}\mathcal{C}$ を D_i と表記し、図式 D の「頂点 (edge)」と呼びます。また、圏 J を図式の「添字圏 (index category)」, あるいは「シェーマ (schema)」と呼びます。

この定義を図式化したもの以下に示しておきます:

$$\begin{array}{ccc} J & & i \xrightarrow{u} j \\ \downarrow D & & \downarrow D \\ \mathcal{C} & & D_i \xrightarrow{Du} D_j \end{array}$$

この定義から図式は圏 \mathcal{C} から圏 J で指示される「添字」を使って抜き出された対象と矢から構成された部分圏であるとも言い換えることができます。ここで特殊な図式として「定図式 (constant diagram)」と呼ばれる図式を定めておきましょう。この定図式 $\Delta_J : \mathcal{C} \rightarrow \mathcal{C}^J$ は対角函手とも呼ばれ、 $\Delta_J(C)$ は添字圏の対象全てを圏 \mathcal{C} の対象 C に、矢は全て C の同一矢 id_C に写す函手です。この定図式は錐と余錐の定義で用いられます。

2.7.12 コンマ圏

既存の圏と函手を使って新しい圏を構成することができます。まず、 b を圏 \mathcal{C} の対象を固定します。それから対象 b を始域とする矢 $b \xrightarrow{f} c$ を $\langle f, c \rangle$ と表記し、「対象 b の下の対象」と呼びます。それから二つの対象 b の下の対象 (f, c) と (g, d) が与えられたときに圏 \mathcal{C} の矢 $c \xrightarrow{h} d$ で $g = h \circ f$ を充たす

$$\begin{array}{ccc} & b & \\ f \swarrow & & \searrow g \\ c & \xrightarrow{h} & d \end{array}$$

は対象 b の下の対象 (f, c) を始域とし (g, d) を終域とする矢になり、これらを使って新たな圏が構成されます。このようにして構築される圏を「対象 b の下の圏」と呼び $(b \downarrow \mathcal{C})$ と表記します。また、次に述べるスライス圏の双対であることから「コスライス (coslice) 圏」とも呼びます。

それからこの圏の双対を考えることができます。このときに対象は $c \xrightarrow{f} b$ で与えられ、この対象を「対象 b の上にある対象」と呼んで対 $\langle c, f \rangle$ と表記します。また、矢は先程と同様に次の可換図式が成立する矢 h で与えられます：

$$\begin{array}{ccc} c & \xrightarrow{h} & c' \\ f \searrow & & \swarrow f' \\ & b & \end{array}$$

これらの対象と矢で構成される圏を「対象 b 上の圏」と呼び $(\mathcal{C} \downarrow b)$ と表記します。また、この圏は特に「スライス (slice) 圏」と呼ばれて \mathcal{C}/b と表記されます。さらに二つの

圏 \mathcal{C}, \mathcal{D} に対して函手 $S : \mathcal{D} \rightarrow \mathcal{C}$ が与えられたときに対象 $b \in \mathcal{C}$ を固定します。それから $d \in \mathcal{D}$ に対して矢 $f : b \rightarrow Sd$ を $\langle f, d \rangle$ と表記します。つぎに $\langle f, d \rangle$ と $\langle f', d' \rangle$ を考えます。ここで矢 $d \xrightarrow{h} d'$ で以下の可換図式が成立するものを $\langle f, d \rangle$ と $\langle f', d' \rangle$ の矢 h とします：

$$\begin{array}{ccc} & b & \\ f \swarrow & & \searrow f' \\ Sd & \xrightarrow{Sh} & Sd' \end{array}$$

こうすることで $\langle f, d \rangle$ を対象、矢を $\langle f, d \rangle \xrightarrow{h} \langle f', d' \rangle$ とする圏が構築できます。この圏を $(b \downarrow S)$ と表記し、 b 上の S の圏と呼びます。また、この圏 $(s \downarrow S)$ の双対を $(S \downarrow s)$ と表記します。

これらを一般化したものが「コンマ圏」です。この圏を構築するために、次の圏と函手：

$$\mathcal{C} \xrightarrow{T} \mathcal{C} \leftarrow \mathcal{D}$$

を考え、 $d \in \mathcal{D}, e \in \mathcal{C}$ に対し矢 $Te \xrightarrow{f} Sd$ を $\langle e, d, f \rangle$ と表記し、この三対を対象とします。次に対象 $\langle e, d, f \rangle$ と $\langle e', d', f' \rangle$ の間の矢を定義しなければなりませんが、この矢は次で定められます。まず $e \xrightarrow{h} e'$ と $d \xrightarrow{k} d'$ を次の図式：

$$\begin{array}{ccc} Te & \xrightarrow{Th} & Te' \\ \downarrow f & & \downarrow f' \\ Sd & \xrightarrow{Sh} & Sd' \end{array}$$

を可換にする矢 $e \xrightarrow{h} e', d \xrightarrow{k} d'$ から新たに $\langle e, d, f \rangle$ から $\langle e', d', f' \rangle$ への矢 $\langle h, k \rangle$ を定め、これらの対象と矢で構成される圏を「コンマ圏」と呼び、 $(T \downarrow S)$ 、あるいは (T, S) と表記します。

このコンマ圏が上述の圏を一般化したものであることを確認しておきましょう。まず圏 \mathcal{C} に終対象 1 が存在するとき、函手 T をこの終対象 1 から圏 \mathcal{C} の対象 b の函手とします。するとこの函手 T は対象 b そのものとして考えることができます。そして対象 $\langle e, d, f \rangle$ も $\langle 1, d, f \rangle$ となります（実質的には $\langle d, f \rangle$ であり、矢 $\langle h, k \rangle$ も $\langle \text{id}, k \rangle$ で、この矢が充たすべき可換図式も）。

$$\begin{array}{ccc}
 T1 = b & \xrightarrow{\text{id}} & T1 = b \\
 \downarrow f & & \downarrow f' \\
 Sd & \xrightarrow{Sh} & Sd'
 \end{array}$$

と圏 $(b \downarrow S)$ の可換図式と実質が違わないことが判ります。同様に函手 S を圏 \mathcal{C} の同一矢とすると圏 $(b \downarrow \mathcal{C})$ が得られます。

2.7.13 普遍矢

アリストテレスの論理学で「普遍」であるとは、命題の主語と述語の関係で、主語の取り替えが効く述語になり得る性質を持つことでした。たとえば、「みけは猫である」、「たまは猫である」や「三毛猫は猫である」という命題の「猫」という類概念は「みけ」や「たま」といった個体概念や、「三毛猫」といった種概念の述語になるために普遍です^{*45}。また、「みけは三毛猫である」や「たまは三毛猫である」ことから「三毛猫」も普遍ですが、「猫は三毛猫である」とは通常は言わないので^{*46}、「三毛猫」は「猫」よりも下位の普遍です。この「三毛猫」という概念は「猫」という概念よりも下部の概念で、この「猫」という類概念を「毛並み」という種差で分類しようとする意図で導入された概念で、猫は毛並みによって「白猫」、「黒猫」、「虎猫」、「三毛猫」,... と分類できます。ところで、集団の分類とは、その集団の構成員の特徴で纏めた集団を作ることですが、この特徴を種差とすると類概念を種差に対応する種概念で分割することになります。ここで、類概念を種概念で分割したときに、ある種から典型的な構成員を種概念の代表として一つ取り出し、他と種概念と比較することはよくあることです。このときに種の代表とその種の構成員は、その分類の観点から「同じもの」として見なされます。このことを整理すると、まず、集合 T をある観点で複数の部分集合 S_1, \dots, S_n に分類します。そして、 $a \in S_i$ を部分集合 S_i の代表として $[a]$ と表記するときに $a, b \in S_i$ であれば $[a] = [b]$ 、 $S_i \neq S_j$ のときに $a \in S_i$, $b \in S_j$ であれば $[a] \neq [b]$ とすることに対応します。ところで、個体を比較するときはその代表を含めて比較します。たとえば、「「みけ」は典型的な三毛猫で、それと比べて三毛猫の「たま」は...」といった風にです。このように個体と代表を比較したり、個体間の関係と代表間の関係を述べたりと、その個体のグループ分けは、その集団の属性や関係も含めて考察することになります。このことから、単なる集合ではなく、圏で考えることは至って自然なことです。

^{*45} 道具で “universal” は方向が自由自在であること、さまざまな状況に対応できることや取替えが効くという意味で用いられています。

^{*46} 「猫は三毛猫(が一番、私は好き)である」等、文脈を含めたときは別です。

そこで、母体の集合に対応する圏 \mathcal{C} と、その部分圏 \mathcal{D} を考えます。この圏 \mathcal{D} は $A, B \in \text{Ob } \mathcal{D}$ が $A \cong B$ のときに $A = B$ を充たすものとします。このように同値な対象が等しいものに限定される圏 \mathcal{D} を「骨格的 (sketal)」と呼びます。先程の猫の分類では、猫の種の集合 $\{\text{三毛猫, 白猫, 黒猫, 虎猫, …}\}$ が対応します。そして、圏 \mathcal{D} が骨格的で、圏 \mathcal{D} から圏 \mathcal{C} への函手 F が忠実、かつ充足で、さらに任意の $B \in \text{Ob } \mathcal{C}$ に対して $FA \cong B$ を充たす $A \in \text{Ob } \mathcal{D}$ が存在するときに圏 \mathcal{D} を圏 \mathcal{C} の「骨格 (skelton)」と呼びます。また、この骨格に似た概念で「同値な函手」があります。これは函手 $F : \mathcal{C} \rightarrow \mathcal{D}$ で、任意の $B \in \text{Ob } \mathcal{D}$ に対して $FA \cong B$ を充たす $A \in \text{Ob } \mathcal{C}$ が存在するときです。このような函手 F が存在するときに圏 \mathcal{C} と圏 \mathcal{D} が「同値 (equivalent)」と呼び $\mathcal{C} \simeq \mathcal{D}$ と表記します。先程の猫の分類の話で、代表の集合と全体の集合は同値で、代表の集合は全体の集合の骨格になります。このことを猫の毛並みによる分類で確認してみましょう。

まず、「猫全体の集合」を \mathcal{C} 、「猫の分類の集合」をここでは $\mathcal{D} = \{\text{三毛猫, 白猫, 黒猫, 虎猫, 斑猫, その他}\}$ とします。このときに \mathcal{C} から \mathcal{D} への写像として包含写像から誘導される写像 F , \mathcal{D} から \mathcal{C} への写像として代表を返す写像 G が考えられます。具体的には F は「みけ」を [みけ] に写す写像で, G は [みけ] を「みけ」に写す写像にそれぞれ対応します。ここで A と B が猫 ($A, B \in \mathcal{C}$) のときに恒等矢 1 に対応する関係を「 A は A である」, $A, B \in \mathcal{C}$ の関係 ' $A \xrightarrow{\text{同じ毛並み}} B$ ' を「 A は B と同じ毛並み」, ' $A \xrightarrow{\text{違う毛並み}} B$ ' を「 A は B と違う毛並み」, 写像 $F : \mathcal{C} \rightarrow \mathcal{D}$ によって「[A] は [B] である」と「[A] と [B] は異なる」にそれぞれ写されます。また、猫全体の集合 \mathcal{C} 内の毛並みに関する関係は猫の毛並みによる分類 \mathcal{D} でも同様の関係があり、逆に \mathcal{D} での関係も \mathcal{C} に対応する関係があります。したがって、 \mathcal{C}, \mathcal{D} を圏とみなすときに写像 F, G は函手になり、特に \mathcal{D} は \mathcal{C} と同型 $\mathcal{C} \simeq \mathcal{D}$ になります。

ここで毛並みの分類については次の可換図式が得られます:

$$\begin{array}{ccc}
 \text{みけ} & \xrightarrow{\text{同じ毛並み}} & \text{たま} = G \text{ 三毛猫} \\
 & \searrow \text{違う毛並み} & \downarrow \text{違う毛並み} \\
 & & \text{しろ} = G \text{ 白猫}
 \end{array}
 \quad
 \begin{array}{c}
 \text{三毛猫} = [\text{たま}] \\
 \downarrow \text{同類ではない} \\
 \text{白猫} = [\text{しろ}]
 \end{array}$$

この例は左側の可換図式が猫全体:圏 \mathcal{C} での話で、右側が猫の分類:圏 \mathcal{D} での話です。この \mathcal{C} と \mathcal{D} の双方を繋いでいるものが函手 G です。まず、函手 G は、対象に関しては猫の種からその代表を返す函数で、圏 \mathcal{D} の矢「違う毛並み」は圏 \mathcal{C} でも「違う毛並み」に写されます。これらの図式の面白いところは、左側の「みけ」と「しろ」に関する矢が現われると、可換になるようにその種の代表である「たま」から「しろ」への矢が一意に存

在し、その矢はさらには種の間の矢と対応関係があることです。さらに「みけ」から「たま」への矢では注目すべき性質があります。これは函手 F によって圏 \mathcal{C} の矢: 「同じ毛並み」は圏 \mathcal{D} の同一矢へと写される矢であり、「同じ毛並み」はある種への帰属を示す矢で、ここでの猫の毛並みの分類を意図した矢であるということです。つまり、〈三毛猫、同じ毛並み〉という対は、猫全体の集合のある一群を「三毛猫」と呼ばれる概念に「同じ毛並み」という矢で纏めることを意味する対になっています。また、「みけ」から「しろ」への矢は「しろ」が帰属する集団に「みけ」が帰属しないことを意味する矢で、〈白猫、違う毛並み〉という対は、猫全体の集合を「白猫」と呼ばれる概念に対して「違う毛並み」という矢で帰属しないことを意味する対になっています。そして、これらの対を結ぶ種の間の矢として「同類ではない」が一つ存在しています。この対は次の可換図式でも現われます：

$$\begin{array}{ccc}
 \text{みけ} & \xrightarrow{\text{同じ毛並み}} & \text{たま} = G \text{ 三毛猫} \\
 & \searrow \text{同じ毛並み} & \downarrow 1_{\text{猫}} \\
 & & \text{たま} = G \text{ 三毛猫}
 \end{array}
 \qquad
 \begin{array}{ccc}
 \text{三毛猫} = [\text{たま}] & & \\
 \downarrow 1_{\text{三毛猫}} & & \\
 \text{三毛猫} = [\text{たま}] & &
 \end{array}$$

この図式は同類の「みけ」と「たま」の毛並みに関する矢の可換図式ですが、この図式についても、「みけ」と「たま」の間の矢に対応する種の間の矢が一意に存在しています。これらのこと整理すると、ある圏 \mathcal{C} の対象 A と圏 \mathcal{D} の対象 R に対して、矢 $A \xrightarrow{u} GR$ が存在し、任意の矢 $A \xrightarrow{f} GD$ に対して $Gf' = f$ を充たす圏 \mathcal{D} の矢 $R \xrightarrow{f'} D$ が一意に存在しています：

$$\begin{array}{ccc}
 A & \xrightarrow{u} & GR \\
 & \searrow f & \downarrow Gf' \\
 & & GD
 \end{array}
 \qquad
 \begin{array}{ccc}
 R & & \\
 \downarrow f' & & \\
 D & &
 \end{array}$$

ここで注目すべきことは左辺の圏 \mathcal{C} に対して \mathcal{D} という構造を函手 G を使って与えている形になっているということです。つまり、矢 u は圏 \mathcal{C} の対象と、圏 \mathcal{D} で与えられる構造に関わる概念への帰属を与える関係であり、さらに \mathcal{C} での関係には「イデア界」に対応する \mathcal{D} の対象間の関係が一意に定まることです。

この特徴は圏論では「普遍矢」として昇華されています：

普遍矢 (universal arrow) —

函手 $G : \mathcal{D} \rightarrow \mathcal{C}$, 対象 $C \in \mathcal{C}$ とするときに A から G への普遍矢とは次の性質を見たす対 $\langle R, u \rangle$ のことである:

- $R \in \text{Ob } \mathcal{D}, C \xrightarrow{u} GR$ とする
- 任意の $D \in \text{Ob } \mathcal{D}, C \xrightarrow{f} CD$ から対 $\langle D, f \rangle$ を定める
- $Gf' \circ u = f$ を充すただ一つの矢 $R \xrightarrow{f'} D$ が存在する

これは次の可換図式で表現することができます:

$$\begin{array}{ccc} C & \xrightarrow{u} & GR \\ \downarrow 1_C & & \downarrow Gf \\ C & \xrightarrow{f} & GD \end{array} \qquad \begin{array}{c} R \\ \downarrow f' \\ D \end{array}$$

ここでコンマ圏 (comma category) $\langle C, G \rangle$ を考えると普遍矢 u はコンマ圏の始対象になります.

さて数学の普遍とはどのようなものでしょうか? たとえば, 位相幾何学では被覆空間というものがあります. これは底空間 B と全空間 C と呼ばれる二つの位相空間が存在し, 被覆写像と呼ばれる連続写像 $p : C \rightarrow B$ で任意の $x \in B$ に対し, その開近傍 $U_x \subset B$ で $p^{-1}(U_x)$ が互いに共通部分を持たない C の可算個の開集合 $\tilde{U}_i, i = 1, 2, \dots$ の和集合となる場合です. この被覆空間に対して普遍被覆空間という空間を考えることができます. まず, $q : D \rightarrow B$ を B の被覆空間とします. それから $p : C \rightarrow B$ を B の被覆空間とすると, 被覆写像 $f : D \rightarrow C$ が存在し, $p \circ f = q$ となるときに q を普遍被覆空間と呼ぶのです. そして, この関係は次の可換図式で表現することができます:

$$\begin{array}{ccc} D & \xrightarrow{f} & C \\ & \searrow q & \downarrow p \\ & & B \end{array}$$

次に Mac Lane の本 [45] に出ている例を挙げておきましょう, まず集合の圏 **Set** と体 K を係数とするベクトル空間の圏 **Vect** $_K$ を考えます. そして, 函手 U をベクトル空間の圏から集合の圏への函手とします. この函手 U は対象については, ベクトル空間を, その演算を忘れることで, 単に集合に写すというもので, その矢も単純にベクトル空間の線形写像の対応関係をそのまま集合の写像としての対応関係に置換えた矢とみなすだけです. 次に $X \in \text{Set}$ を基底とし, 係数体を K とするベクトル空間を V_X と記述すると圏 **Set** の

対象 $X, U(V_X)$ 間の矢 $X \xrightarrow{j} U(V_X)$ が定まります。つぎに任意の $W \in \mathbf{Vect}_K$ に対し、 $U(W)$ を考えて X の元を $U(W)$ に対応させることで矢 $X \xrightarrow{f} U(W)$ を構成することができます。それからこの f の対応関係を基に体 K について線形になるように拡張することで新たに \mathbf{Vect}_K の矢 $V_X \xrightarrow{f'} W$ を構成することができます。以上の結果から次の可換図式が得られます：

$$\begin{array}{ccc} X & \xrightarrow{j} & U(V_X) \\ f \searrow & \downarrow Uf' & \downarrow f' \\ & U(W) & W \end{array}$$

2.7.14 圈の演算

圏 \mathcal{C} の対象について積や幂を定めることができます。最初の対象の積について述べましょう：

対象の積

下記の条件を充す \mathcal{C} の対象 X を $A \times B$ と表記し、対象 A, B の積と呼ぶ：

- \mathcal{C} の二つの矢 $X \xrightarrow{\pi_1} A$ と $X \xrightarrow{\pi_2} B$ が存在
- \mathcal{C} の任意の対象 C と C から A, B への二つの矢 $C \xrightarrow{f} A, C \xrightarrow{g} B$ について $f = \pi_1 \circ h, g = \pi_2 \circ h$ を充す矢 $C \xrightarrow{h} X$ が一意に存在する。この矢 h を $\langle f, g \rangle$ と表記する。

圏 \mathcal{C} の任意の二つの対象に対して積が存在するときに圏 \mathcal{C} のことを「積を有する圏」と呼ぶ。

圏の積を可換図式として表現することができます：

$$\begin{array}{ccccc} & & C & & \\ & f \swarrow & \downarrow & \searrow g & \\ A & \xleftarrow{\pi_1} & A \times B & \xrightarrow{\pi_2} & B \end{array}$$

この可換図式で示す対象の積は対象 C から積 $A \times B$ への矢が一意に存在するという性質を利用したもので、 $A \rightarrow B$ という命題を考えたときに複数の主語の述語になるという「普遍性」に関わる定義です^{*47}。ただし、この普遍性は対象の一意性、つまり、 $A \times B$ が

^{*47} この普遍性は後述の極限に関連します。

一意に存在することを保障するものではありません。実際、対象 P についても $A \times B$ と同様の性質があれば

$\begin{array}{c} P \\ \downarrow \langle \pi'_1, \pi'_2 \rangle \\ A \times B \\ \uparrow \langle \pi_1, \pi_2 \rangle \\ P \end{array}$
 $\begin{array}{c} A \times B \\ \downarrow \langle \pi_1, \pi_2 \rangle \\ P \\ \uparrow \langle \pi'_1, \pi'_2 \rangle \\ A \times B \end{array}$

から $\langle \pi_1, \pi_2 \rangle \circ \langle \pi'_1, \pi'_2 \rangle = \text{id}_P$, $\langle \pi'_1, \pi'_2 \rangle \circ \langle \pi_1, \pi_2 \rangle = \text{id}_{A \times B}$ が得られるために $A \times B \cong P$ が判り、積自体は iso な対象で一意に定まることを意味します。なお、圏を小集合の圏 Set とするときに対象の積は $A \times B = \{\langle x, y \rangle : x \in A \wedge y \in B\}$ と小集合のデカルト積になりますが、圏 \mathcal{C} の対象を順序数、矢を \leq とするときの $A \times B$ は A と B のどちらかより後者にある対象で与えられ、対象の積が対象の成分の順序対に類似したものになることは限りません。なお、圏 \mathcal{C} の対象 A_1, A_2 に対して部分圏 $\mathcal{C}|_{A_1, A_2}$ を $A_1 \xleftarrow{f_1} B \xrightarrow{f_2} A_2$ を充たす対象から構成される圏として定めます。すると $A_1 \times A_2$ はこの部分圏 $\mathcal{C}|_{A_1, A_2}$ の終対象として考えることができます。

対象の積は次の性質を充すことが容易に判ります：

————— 対象の積の性質 —————

- 可換性: $A \times B \cong B \times A$
- 結合律: $A \times (B \times C) \cong (A \times B) \times C$

これらの性質から任意の対象の積が存在する圏 \mathcal{C} の対象の集合 $\text{Ob}\mathcal{C}$ には iso による関係が入るもののが可換な積の構造が入ります。また、対象の積が結合律を充すことから、3 個以上の対象の積は $A_1 \times A_2 \times \dots \times A_n$ と括弧を外した表記、あるいは $\prod_1^n A_i$ と表記し、特に対象 A の n 個の積 $A \times \dots \times A$ を A^n と表記します。さらに圏 \mathcal{C} に終対象 1 が存在するときに終対象 1 は演算 \times の単位元としての働きがあります：

————— 終対象との積 —————

$$1 \times A \cong A \times 1 \cong A$$

これらのこととは積の可換性と以下の可換図式から判ります：

$$\begin{array}{ccccc}
 & & A & & \\
 & \swarrow !_A & \downarrow \langle !_A, id_A \rangle & \searrow id_A & \\
 1 & \xleftarrow{!_{1 \times A}} & 1 \times A & \xrightarrow{\pi_2} & A \\
 & \downarrow \pi_2 & & \nearrow id_A & \\
 & & A & &
 \end{array}
 \quad
 \begin{array}{ccccc}
 & & 1 \times A & & \\
 & \swarrow !_A & \downarrow \pi_2 & \searrow \pi_2 & \\
 1 & \xleftarrow{!_{1 \times A}} & A & \xrightarrow{id_A} & A \\
 & \downarrow \langle !_A, id_A \rangle & & \nearrow \pi_2 & \\
 & & 1 \times A & &
 \end{array}$$

まず、左の可換図式から $\pi_2 \circ \langle !_A, id_A \rangle = id_A$ が判ります。また、右の図式で 1 が終対象のために矢 $A \rightarrow 1$ が一意に存在することから $1_{1 \times A} = !_A \circ \pi_2$ 、このことから $\langle !_A, id_A \rangle \circ \pi_2 = id_{1 \times A}$ であることが判り、 $1 \times A \cong A$ が証明できます。

同じ対象の積 $A \times A$ を考えます。このときに A から $A \times A$ への矢が一意に定まります。この矢を「対角矢 Δ_A 」と呼びます。この対角矢 Δ_A の可換図式を次に示します：

対角矢 Δ_A

以下の可換図式を充す矢 $\langle id_A, id_A \rangle$ を特に対角矢と呼び記号 Δ_A で表記される。

$$\begin{array}{ccccc}
 & & A & & \\
 & \swarrow id_A & \downarrow \Delta_A & \searrow id_A & \\
 A & \xleftarrow{\pi_1} & A \times A & \xrightarrow{\pi_2} & A
 \end{array}$$

この対角矢 Δ_A は後述の特性写像 δ_A で重要です。

ここで対象の積の双対は「直和」、あるいは「双対積 (coproduct)」と呼ばれ、 $A \amalg B$ 、あるいは $A + B$ と表記します。この直和の定義を以下に記しておきましょう：

対象の直和

下記の条件を充す \mathcal{C} の対象 X を $A \amalg B$, あるいは $A + B$ と表記し, 対象 A, B の直和と呼ぶ:

- \mathcal{C} の二つの矢 $A \xrightarrow{i_1} X$ と $B \xrightarrow{i_2} X$ が存在
- \mathcal{C} の任意の対象 C と A から C , B から C への二つの矢 $A \xrightarrow{f} C, B \xrightarrow{g} C$ について $f = h \circ i_1, g = h \circ i_2$ を充す矢 $X \xrightarrow{h} C$ が一意に存在する.

この直和の可換図式は直積の可換図式の双対です:

$$\begin{array}{ccccc}
 & & C & & \\
 & f \nearrow & \downarrow & \swarrow g & \\
 A & \xrightarrow{i_1} & A + B & \xleftarrow{i_2} & B
 \end{array}$$

また, 直和は積と双対であるために次が成立します:

対象の直和の性質

- 可換性: $A + B \cong B + A$
- 結合律: $A + (B + C) \cong (A + B) + C$

積と同様に対象の直和が結合律を充すことから, 3 個以上の対象の積は $A_1 + A_2 + \dots + A_n$ と括弧を外した形や $\amalg_1^n A_i$ と表記します. さらに圏 \mathcal{C} に始対象 0 が存在するときに始対象 0 は演算 $+$ の単位元としての働きがあります:

始対象との直和

$$0 + A \cong A + 0 \cong A$$

積の場合と同様に iso による関係が入りますが, 対象の直和によって圏 \mathcal{C} の対象の集合 $\text{Ob}\mathcal{C}$ に单系の構造が入ります.

つぎに積を有する圏では, その対象の積から圏の「矢の積」も定義することができます:

矢の積

\mathcal{C} の二つの矢 $A \xrightarrow{f} B$ と $C \xrightarrow{g} D$ に対し $\langle f \circ \pi_1, g \circ \pi_2 \rangle : A \times C \rightarrow B \times D$ を $f \times g$ と表記して矢 f, g の積と呼ぶ

この矢の積は, 対象の積の可換図式の特殊な例として考えられ, 以下の可換図式として示すことができます:

$$\begin{array}{ccc}
 A & \xrightarrow{f} & B \\
 \uparrow \pi_1 & & \uparrow \pi_1 \\
 A \times C & \xrightarrow{f \times g} & B \times D \\
 \downarrow \pi_2 & & \downarrow \pi_2 \\
 C & \xrightarrow{d} & C
 \end{array}$$

さらに積を有する圏 \mathcal{C} では対象の幕を定義できるものがあります:

対象の幕

積を有する圏 \mathcal{C} の対象 A, B と矢 $C \times A \xrightarrow{g} B$ に対し、以下の図式を可換にする圏 \mathcal{C} の対象 B^A と矢 $B^A \times A \xrightarrow{\text{ev}} B$ と $C \xrightarrow{\hat{g}} B^A$ が存在し、さらに \hat{g} が一意的に存在するときに、 \mathcal{C} の対象 B^A のことを幕と呼ぶ:

$$\begin{array}{ccc}
 B^A \times A & & \\
 \uparrow \hat{g} \times 1_A & \searrow \text{ev} & \\
 C \times A & \xrightarrow{g} & B
 \end{array}$$

ここで矢 ev のことを「評価 (evaluation)」、矢 \hat{g} を矢 g の「転置 (transpose)」と呼びます。さらに対象の幕では $\text{Hom}(C \times A, B) \cong \text{Hom}(C, B^A)$ が成立します。実際、 $g \in \text{Hom}(C \times A, B)$ に対して幕 B^A の定義から一意に $\hat{g} \in \text{Hom}(C, B^A)$ が存在するため $\text{Hom}(C \times A, B)$ から $\text{Hom}(C, B^A)$ への写像が存在します。そして、 $h \in \text{Hom}(C, B^A)$ に対して $g = \text{ev}(h \times 1_A)$ と定めることで $g \in \text{Hom}(C \times A, B)$ が得られますが、幕の定義から g の転置 \hat{g} が一意に定まるために $\hat{g} = h$ 、したがって、 $\text{Hom}(C \times A, B)$ から $\text{Hom}(C, B^A)$ への写像が全単射であることが判ります。

2.7.15 等化と余等化について

ここでは最初に「等化 (イコライザー, equalizer)」を次で定義します:

等化 (equalizer)

二つの矢 $A \xrightarrow[\substack{g \\ f}]{} B$ に対し、対象 C と矢 $C \xrightarrow{e} A$ が存在し、 $f \circ e = g \circ e$ を充し、さらに以下に示す可換図式にて $f \circ k = g \circ k$ であるときに $k = e \circ h$ を充す矢 $D \xrightarrow{h} A$ が存在して一意に定まるとき、矢 e を矢 f と矢 g の等化と呼ぶ：

$$\begin{array}{ccccc} & D & & & \\ & \downarrow h & \searrow k & & \\ C & \xrightarrow{e} & A & \xrightarrow[\substack{f \\ g}]{} & B \end{array}$$

ここで等化 e は必ず mono になります。実際、 h, k を対象 D から 対象 C の矢で、それが $e \circ h = e \circ k$ を充すときに二つの図式：

$$\begin{array}{ccc} \begin{array}{ccc} D & & \\ \downarrow h & \searrow e \circ h & \\ C & \xrightarrow{e} & A & \xrightarrow[\substack{f \\ g}]{} & B \end{array} & \quad & \begin{array}{ccc} D & & \\ \downarrow k & \searrow e \circ k & \\ C & \xrightarrow{e} & A & \xrightarrow[\substack{f \\ g}]{} & B \end{array} \end{array}$$

を充しますが、ここで矢 e が等化のために $h = k$ 、したがって、矢 e が mono であることが判ります。

また等化の双対を「余等化 (コイコライザー, coequalizer)」と呼びます：

余等化 (coequalizer)

二つの矢 $A \xrightarrow[\substack{g \\ f}]{} B$ に対し、対象 C と矢 $B \xrightarrow{e} C$ が存在し、 $f \circ e = g \circ e$ を充たし、以下に示す可換図式にて $f \circ k = g \circ k$ であれば $k = e \circ h$ を充す矢 $B \xrightarrow{h} D$ が存在して一意に定まるときに矢 e を矢 f と矢 g の余等化と呼ぶ：

$$\begin{array}{ccccc} & & D & & \\ & & \downarrow h & & \\ & & \nearrow k & & \\ A & \xrightarrow[\substack{f \\ g}]{} & B & \xrightarrow{e} & C \end{array}$$

余等化は等化の双対であるために、mono の双対である epic になることがただちに判ります。

2.7.16 引き戻しと押し出し

ここでは「**引き戻し (pull back)**」と「**押し出し (push out)**」について述べます。この引き戻しと押し出しは互いに双対の関係にあります。ここでは最初に引き戻しについて述べることにしましょう：

引き戻し (pull back)

$A \xleftarrow{g'} P \xrightarrow{f'} B$ が $A \xrightarrow{f} C \xleftarrow{g} B$ の「**引き戻し**」であるとは、左下の可換図式を充し、任意の対象 $E \in \mathcal{C}$ に対して右下の図式が可換図式になるような矢 $E \xrightarrow{h} A$ と $E \xrightarrow{k} B$ が存在し、さらに矢 $E \xrightarrow{l} P$ が一意に存在するときである。

ここで圏 \mathcal{C} が終対象 1 を持つときに対象 C を終対象 1 で置換えると引き戻しの対象 P が対象の積 $A \times B$ になります。また、圏 \mathcal{C} の矢が通常の写像の圏 **Set** のときに引き戻しの対象 P は $A \times_C B = \{(x, y) : (x \in A) \wedge (y \in B) \wedge (f x = g y)\}$ と外延として記述することができます。このように引き戻しは特殊な積としても考えることができます。

引き戻しの性質の性質を幾つか挙げておきます：

- $A \xrightarrow{f} C \xleftarrow{g} B$ に対する引き戻し $A \xleftarrow{g'} D \xrightarrow{f'} B$ に対し、 f が mono であるときに f' も mono になります：

$$\begin{array}{ccc} P & \xrightarrow{f'} & B \\ g' \downarrow & & \downarrow g \\ A & \xrightarrow{f} & C \end{array}$$

- 次の可換図式において、右側の四角と外側の四角の図式がそれぞれ引き戻しであれ

ば左側の四角の図式も引き戻しになります、また、右側と左側の図式が引き戻しになるときに外側の四角の図式も引き戻しになります：

$$\begin{array}{ccccc} P & \xrightarrow{g'} & Q & \xrightarrow{h'} & D \\ f' \downarrow & & f \downarrow & & f'' \downarrow \\ A & \xrightarrow{g} & B & \xrightarrow{h} & C \end{array}$$

さて、圈 **Set** で引き戻しがどのようなものか考えてみましょう。つまり $A \xrightarrow{f} C \xleftarrow{g} B$ を充す集合 $A, B, C \in \text{Set}$ に対しては $A \times_C B = \{(a, b) \in A \times B : f(a) = g(b)\}$ を考えると

$$\begin{array}{ccc} A \times_C B & \xrightarrow{\pi_A} & B \\ \pi_B \downarrow & & \downarrow g \\ A & \xrightarrow{f} & C \end{array}$$

は引き戻しになりますが、ここで

$$\begin{array}{ccccc} D & \xrightarrow{k} & A \times_C B & \xrightarrow{\pi_A} & B \\ l \searrow & & \downarrow \pi_B & & \downarrow g \\ h \swarrow & & A & \xrightarrow{f} & C \end{array}$$

をよくよく考えると次の集合の積と等化が現われます：

$$\begin{array}{ccc} A & \xleftarrow{\pi_B} & A \times_C B & \xrightarrow{\pi_A} & B \\ h \nearrow & & l \downarrow & & k \searrow \\ & D & & & \end{array} \quad \begin{array}{ccccc} A \times_C B & \xrightarrow{e} & A \times B & \xrightarrow{\frac{f \circ \pi_A}{g \circ \pi_B}} & C \\ l \downarrow & & h \times k \searrow & & \\ & D & & & \end{array}$$

ここで $h \times k$ は $d \in D$ に対して $(h(d), k(d)) \in A \times B$ を対応させる写像です。この例は集合の圈 **Set** で考えたことですが、ここで対象の積と等化が現われていることから引き

戻しには対象の積と等化が関係していることが予想できます。実際、対象の積、等化と引き戻しについては以下の関係があります：

—— 積、等化と引き戻しの関係 ——

圏 \mathcal{C} の任意の二つの対象について積が存在し、また任意の二つの矢に対してもその等化が存在するときに、任意の $A \xrightarrow{f} C \xleftarrow{g} B$ に対して引き戻し $A \xleftarrow{g'} D \xrightarrow{f'} B$ が存在する。

この証明は、まず任意の $A \xrightarrow{f} C \xleftarrow{g} B$ に対して次の等化を考えられます：

$$\begin{array}{ccccc} D & \xrightarrow{e} & A \times B & \xrightarrow{\pi_A} & B \\ & & \downarrow \pi_B & & \downarrow g \\ & & A & \xrightarrow{f} & C \end{array}$$

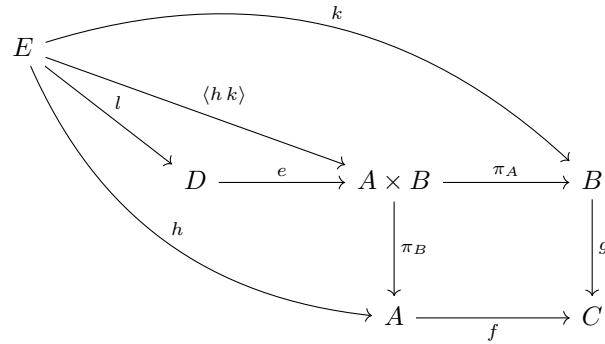
このときに

$$\begin{array}{ccc} D & \xrightarrow{\pi_A \circ e} & B \\ \downarrow \pi_B \circ e & & \downarrow g \\ A & \xrightarrow{f} & C \end{array}$$

が引き戻しになることを示せば十分です。さて、先程の等化について次の可換図式を考えます：

$$\begin{array}{ccccc} E & \xrightarrow{k} & A \times B & \xrightarrow{\pi_A} & B \\ \swarrow h & \searrow \langle h, k \rangle & \downarrow \pi_B & & \downarrow g \\ D & \xrightarrow{e} & A \times B & \xrightarrow{\pi_A} & B \\ & & \downarrow \pi_B & & \downarrow g \\ & & A & \xrightarrow{f} & C \end{array}$$

ここで $A \xleftarrow{\pi_A} A \times B \xrightarrow{\pi_B} B$ が積であることから一意に $E \xrightarrow{\langle h, k \rangle} A \times B$ が存在することがわかります。すると今度は e が等化であることから一意に $E \xrightarrow{l} D$ が存在することになります：

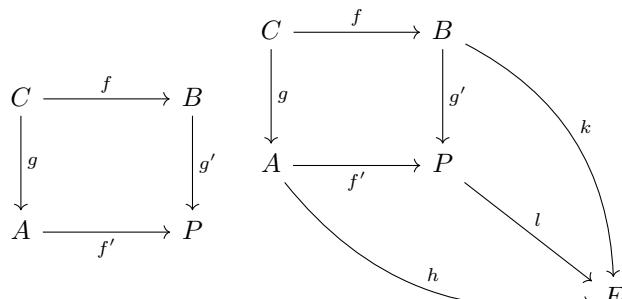


のことから $A \xleftarrow{\pi_A \circ e} D \xrightarrow{\pi_B \circ e} B$ が $A \xrightarrow{f} C \xleftarrow{g} B$ の引き戻しになっていることがわかります。

「引き戻し」の双対に「押し出し (push out)」があります:

押し出し

$A \xrightarrow{f'} P \xleftarrow{g'} B$ が $A \xleftarrow{g} C \xrightarrow{f} B$ の「押し出し」であるとは左下の可換図式に対し、任意の対象 $E \in \mathcal{C}$ に対して右下の図式が可換図式になるような矢 $E \xleftarrow{h} A$ と $E \xleftarrow{k} K$ が存在し、さらに矢 $P \xrightarrow{l} E$ が一意に存在する場合である:



この押し出しの定義は引き戻しの定義の矢の向きが逆になったもの、すなわち、引き戻しの双対であることに注目して下さい。そのために対象 C が始対象 0 であれば押し出しの対象 P は対象 A, B の積 $A \times B$ の双対の $A \amalg B$ になり、圏 \mathcal{C} が小集合の圏 Set のときは $A \times_C B$ の双対である $A \amalg_C B$ になります。

極限と余極限

引き戻しと押し出し、それと等化と余等化は互いに双対の関係にあります。この直接的なものの見方に加えて別の見方をすることができます。ここで極限と余極限という概念を導入しましょう。まず、圏 \mathcal{C} に対して図式 $D : J \rightarrow \mathcal{C}$ に対して「錐 (cone)」を次で定めます：

錐 (cone) の定義

圏 \mathcal{C} のある対象 C から図式 D の各対象 D_i への矢の集合 $\{C \xrightarrow{\xi_i} D_i\}$ で、図式 D の矢 $D_i \xrightarrow{\delta_{ji}} D_j$ に対して $\xi_j = \delta_{ji} \circ \xi_i$ を充すときに図式 D 上の「錐 (cone)」と呼びます。このときに対象 C を「錐の頂点」と呼びます。

ここで $\xi_j = \delta_{ji} \circ \xi_i$ の意味は次の図式

$$\begin{array}{ccc} & C & \\ \xi_i \swarrow & & \searrow \xi_j \\ D_i & \xrightarrow{\delta_{ji}} & D_j \end{array}$$

が可換になることを意味します。また、錐の定義として定図式 $\Delta_J(C)$ を用いると、 $\Delta_J(C)$ から D への自然変換 ξ として与えられます：

$$\begin{array}{ccc} i & & C \xrightarrow{\xi_i} D_i \\ \downarrow u & \parallel & \downarrow D(u)=\delta_{ji} \\ j & & C \xrightarrow{\xi_j} D_j \end{array}$$

$$\Delta_J(C)(u)=\text{id}_C$$

次に「余錐 (cocone)」を定義します：

余錐 (cocone) の定義

図式 D の各対象 D_i から圏 \mathcal{C} のある対象 C への矢の集合 $\{D_i \xrightarrow{\zeta_i} C\}$ で、図式 D の矢 $D_i \xrightarrow{\delta_{ji}} D_j$ に対して $\zeta_j = \delta_{ji} \circ \zeta_i$ を充すときに図式 D 上の「余錐 (cocone)」と呼びます。このときに対象 C を「余錐の頂点」と呼びます。

錐のときと同様に $\zeta_j = \delta_{ji} \circ \zeta_i$ の意味は次の図式

$$\begin{array}{ccc} D_i & \xrightarrow{\delta_{ji}} & D_j \\ \zeta_i \swarrow & & \searrow \zeta_j \\ C & & \end{array}$$

が可換になることを意味します。また、錐の定義として定図式 $\Delta_J(C)$ を用いると、 D から $\Delta_J(C)$ への自然変換 ζ として得られます：

$$\begin{array}{ccccc} i & & D_i & \xrightarrow{\zeta_i} & C \\ \downarrow u & & \downarrow D(u)=\delta_{ji} & & \parallel \Delta_J(C)(u)=\text{id}_C \\ j & & D_j & \xrightarrow{\zeta_j} & C \end{array}$$

このように錐や余錐では $\Delta_J(C)$ の対象が全て C であるためにこれらの可換図式を

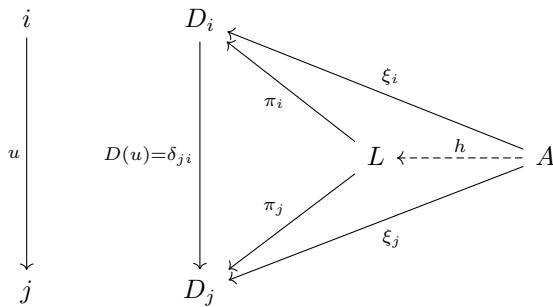
$$\begin{array}{ccc} i & & D_i \\ \downarrow u & & \downarrow Du \\ C & \begin{array}{c} \nearrow \pi_i \\ \searrow \pi_j \end{array} & D_j \\ j & & \end{array} \quad \begin{array}{ccc} D_i & & D_i \\ \downarrow Du & & \downarrow Du \\ D_j & \begin{array}{c} \nearrow \rho_i \\ \searrow \rho_j \end{array} & C \end{array}$$

と対象 C を纏めた図式で錐と余錐を表記することにします。

図式の錐と余錐が定義できたところでようやく極限と余極限の定義の準備が整ったことになります。まず、図式 D の「極限 (limit)」は図式 D の錐を使って定義されます：

— 極限 (limit) の定義 —

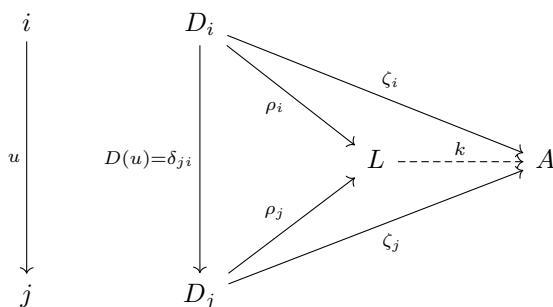
圏 \mathcal{C} の対象 L が図式 $D \in \mathcal{C}^J$ の「**極限 (limit)**」であるとは L を頂点とする図式 D の錐 π が存在し、任意の図式 D の錐 ξ に対して次の図式が可換になるような圏 \mathcal{C} の矢 $A \xrightarrow{h} L$ が一意に存在するときで、図式 D の極限を $\lim_{\leftarrow} D$ 、あるいは $\lim D$ と表記します。



極限の双対を「**余極限 (colimit)**」と呼びますが、余極限の定義では余錐が用いられます：

— 余極限 (colimit) の定義 —

圏 \mathcal{C} の対象 L が 図式 $D \in \mathcal{C}^J$ の「**余極限 (colimit)**」であるとは L を頂点とする図式 D の余錐 ρ が存在し、任意の図式 D の余錐 ζ に対して次の図式が可換にするような圏 \mathcal{C} の矢 $L \xrightarrow{k} A$ が一意に存在するときで、図式 D の余極限を $\text{colim}_{\rightarrow} D$ 、あるいは $\text{colim} D$ と表記します。



極限を使って引き戻しと押し出し、対象の積と等化を解釈し直すことができます。最初に引き戻しは図式 D を $\{\bullet \rightarrow \bullet \leftarrow \bullet\}$ に対応するものとします。この図式に対する

錐 π の極限が引き戻しになります、押し出しはその双対の余極限になります。次に対象の積と等値については、まず、添字圏 J を対象が $1, 2$ 、矢が同一矢のみの圏 $\{1, 2\}$ とし、図式 $D \in \mathcal{C}^J$ に $\lim_{\leftarrow} D$ が存在し、図式 D の錐 ξ に対して

$$\begin{array}{ccccc} & & A & & \\ & \swarrow \xi_1 & \downarrow h & \searrow \xi_2 & \\ D_1 & \xleftarrow{\pi_1} & \lim_{\leftarrow} D & \xrightarrow{\pi_2} & D_2 \end{array}$$

を充たす矢 h が一意に定まります、この可換図式は $\lim_{\leftarrow} D \cong D_1 \times D_2$ であることを意味します。つぎに対象を $1, 2$ 、同一矢以外の矢を $1 \xrightarrow{f} 2$ と $1 \xrightarrow{g} 2$ とする圏 \Downarrow を添字圏とし、図式 $D \in \mathcal{C}^{\Downarrow}$ に対してその極限 $\lim_{\leftarrow} D$ が存在するときに $\lim_{\leftarrow} D$ を頂点に持つ錐 π が D から $\lim_{\leftarrow} D$ への自然変換であることから

$$\begin{array}{ccc} D_1 & \xrightleftharpoons[F=Df]{G=Dg} & D_2 \\ \swarrow \pi_1 & & \searrow \pi_2 \\ \lim_{\leftarrow} D & & \end{array}$$

が可換、すなわち、 $F \circ \pi_1 = G \circ \pi_1$ 、さらに D の錐 ξ に対しても $F \circ \xi_1 = G \circ \xi_2$ を充し、図式の極限の定義から次の可換図式を充たす矢 h が一意に存在しますが

$$\begin{array}{ccccc} & & F=Df & & \\ & \nwarrow \pi_1 & \downarrow G=Dg & \nearrow \pi_2 & \\ D_1 & \xrightleftharpoons[\xi_1]{\lim_{\leftarrow} D} & \lim_{\leftarrow} D & \xrightleftharpoons[\xi_2]{\lim_{\leftarrow} D} & D_2 \\ & \uparrow h & & & \end{array}$$

この可換図式を次の可換図式で置き換えることができます:

$$\begin{array}{ccccc} \varprojlim D & \xrightarrow{\pi_1} & D_1 & \xrightarrow{\begin{array}{c} F=Df \\ G=Dg \end{array}} & D_2 \\ h \uparrow & \nearrow \xi_1 & & & \\ A & & & & \end{array}$$

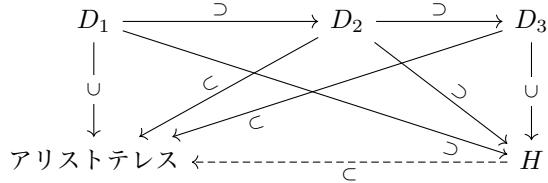
つまり, $\varprojlim D \xrightarrow{\pi_1} D_1$ が等化として得られます. また, これらの双対を考えることで余極限があれば余積と余等化も同様に得られます. 以上から, 圈 \mathcal{C} に (余) 極限が存在するときに (余) 積と (余) 等化が存在することが判ります.

この余極限については興味深い事例が挙げられます. 「アベラールとエローイーズ」で有名な 12 世紀の哲学者アベラールは普遍についてこのように興味深いことを述べています ([32] pp. 43-44).

- 普遍は名指し作用できさまざまなものをある仕方で意味表示する.
- 以上のことは意味表示の対象から生じる概念を構成することではなく, 個々のものに到達する概念を構築することでなされる.
- このようにして「人間」という音声は「人間である」という共通の理由で人間を名指すのであって, この共通の原因を有することが普遍である.

ここでの「普遍」は「ソクラテスは人間である」, 「アリストテレスは人間である」といった複数の対象の述語になる能力のことです. また, 「人間であるという共通の原因を有すること」をアベラールは事態と呼んでいます. ちなみにアベラールが挙げている人間という概念は古来から「動物である」, 「理性的である」と「死すべき存在である」と述べられています. アリストテレスの事物の定義は「類と種の関係で述べること」で, ポルフュリオスの樹で示される階層構造を持ちます. したがって, 定義付けることは階層を持つことからある種の図式としての構造を持ちます. そこで, $D_1 = \{ \text{動物である} \}$, $D_2 = \{ \text{動物である, 理性的である} \}$, $D_3 = \{ \text{動物である, 理性的である, 死すべき存在である} \}$ とし, 矢をつとして図式 $\mathcal{D} = D_1 \xrightarrow{\supset} D_2 \xrightarrow{\supset} D_3$ を考えます. この図式が「個々のものに到達する概念の構築」に対応します. それから $H = \{ \text{人間である} \}$ としましょう. すると「人間は動物」であり, 「人間は, 動物で理性的」であり, 「人間は動物で理性的で死すべき存在」なので, 余錐 $F = \{ H \longrightarrow D_i, i \in \{1, 2, 3\} \}$ が構築できます. この図式 \mathcal{D} に対しては, 「ソクラテス」, 「プラトン」や「アリストテレス」といった人々も余錐になります. この余錐になることが「人間であるという事態」に対応します. なぜなら $H = \operatorname{colim}_{\longrightarrow} D$ に

なるためで、実際、



と $H = \{ \text{人間である} \}$ から人間である事態にある個体への矢 “ \circlearrowleft ” が一意に定まることから「人間であること」が図式 \mathcal{D} の余極限として得られます。

圏 \mathcal{C} の任意の(有限)図式が極限を持つときに圏 \mathcal{C} のことを「(有限) 完備 ((finite-)complete)」と呼びます。同様に圏 \mathcal{C} の任意の(有限)図式が余極限を持つときに圏 \mathcal{C} を「(有限) 余完備 ((finite-)co-complete)」と呼びます。ここで図式の完備性については次の定理が知られています:

完備性と積、等価の関係

圏 \mathcal{C} が(余)完備であることは、圏 \mathcal{C} に対象の(余)積と(余)等価が存在するときに限る。

この定理の前半は先程の図式の極限から積や等化の導出に対応するため、実際に示さなければならぬことは後半の積と等化から極限が構成できることです。そこで圏 \mathcal{C} を積と等化が存在する圏と仮定し、与えられた図式 $D \in \mathcal{C}^J$ の全ての成分で構成される積を考えます。つまり、図式 D の全ての頂点の積 $\bar{D} = \prod_{i \in \text{Ob } J} D_i$ とその射影 $\bar{D} \xrightarrow{\pi_i} D_i$ 、それと図式 D の全て辺の終点の積 $\underline{D} = \prod_{f \in \text{Arr } J} D_{\text{cod } f}$ とその射影 $\underline{D} \xrightarrow{\rho_i} D_i$ を定めます。ここで \bar{D} と \underline{D} が対象の積であるために矢 $i \xrightarrow{u} j \in \text{Arr } J$ に対して次の図式を可換にする圏 \mathcal{C} の矢 f, g が一意に存在します:

$$\begin{array}{ccc} \bar{D} & \xrightarrow{f} & D \\ \pi_j \searrow & & \downarrow \rho_j \\ & & D_j \\ \end{array} \quad \begin{array}{ccc} \bar{D} & \xrightarrow{g} & \underline{D} \\ Du \circ \pi_i \searrow & & \downarrow \rho_j \\ & & D_j \\ \end{array}$$

ここで圏 \mathcal{C} で等化が存在することから図式 $\bar{D} \xrightleftharpoons[g]{f} \underline{D}$ の等化を $A \xrightarrow{h} \bar{D}$ とします。次

に $\pi \circ h$ が A を頂点とする図式 D の錐であることを示しましょう。そのためには以下の図式が可換図式であることを示さなければなりません；

$$\begin{array}{ccc} & A & \\ \swarrow_{\pi_i \circ h} & & \searrow^{\pi_j \circ h} \\ D_i & \xrightarrow{Du} & D_j \end{array}$$

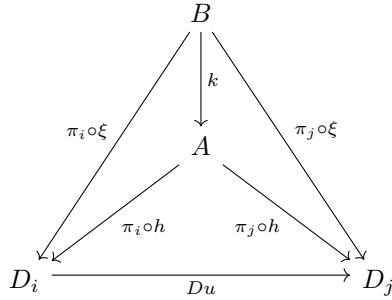
ここで矢 f に関わる可換図式から $\pi_j \circ h = \rho_j \circ f \circ h$, $A \rightarrow \bar{D}$ が等化であることから $f \circ h = g \circ h$ より $\pi_j \circ h = \rho_j \circ f \circ h = \rho_j \circ g \circ h$. 矢 g に関わる可換図式から $\rho_j \circ g \circ h = D_u \circ \pi_i \circ h$. 以上から $\pi_j \circ h = D_u \circ \pi_i \circ h$. したがって、自然変換 $\pi \circ h$ が図式 D の錐になることが判ります次に $A \xrightarrow{h} \bar{D}$ が等化であるために $f \circ \xi = g \circ \xi$ を充す矢 $B \xrightarrow{\xi} \bar{D}$ が存在するときに次の図式が可換になる矢 $B \xrightarrow{k} A$ が一意に存在します：

$$\begin{array}{ccccc} A & \xrightarrow{h} & \bar{D} & \xrightleftharpoons[f]{g} & D \\ \uparrow k & \nearrow \xi & & & \\ B & & & & \end{array}$$

そこでも $\pi \circ \xi$ が図式 D の錐になることを示しましょう。そのためには次の図式

$$\begin{array}{ccc} & B & \\ \swarrow_{\pi_i \circ \xi} & & \searrow^{\pi_j \circ \xi} \\ D_i & \xrightarrow{Du} & D_j \end{array}$$

が可換図式であることを示せば十分です。ここで $A \xrightarrow{h} \bar{D}$ が等化であることから $D_u \circ \pi_i \circ \xi = D_u \circ \pi_i \circ h \circ k$, これと $\pi \circ h$ が図式 D の錐であることから $D_u \circ \pi_i \circ h \circ k = \pi_j \circ h \circ k = \pi_j \circ \xi$, したがって、自然変換 $\pi \circ \xi$ が対象 B をその頂点とする図式 D の錐になります。それから次の図式



が可換図式であることを示せば十分ですが、ここで $\pi_i \circ \xi = \pi_i \circ h \circ k$ と $\pi_j \circ \xi = \pi_j \circ h \circ k$ は $A \xrightarrow{h} \bar{D}$ が等化であることから、 $Du = \pi_i \circ h = \pi_j \circ h$ は $\pi \circ h$ が図式 D の錐であることから、また、 $Du = \pi_i \circ \xi = \pi_j \circ \xi$ も $\pi \circ \xi$ が図式 D の錐であることから判ります。これらのことから対象 A が図式 D の極限になり、以上から、積と等化が存在する圏には極限が存在すると言えます。また、ここでの証明の双対を考えることで同様に余積と余等化が存在するときに余極限が存在し、また、その逆も言えます。

このように圏 \mathcal{C} が完備であることと、圏 \mathcal{C} に積と等化が存在することが同値であることが示せましたが、圏 \mathcal{C} が引き戻しと終対象を持つことと完備であることも同値です。実際、圏 \mathcal{C} が完備のときに図式 $\{\}$ の極限が終対象 1、図式 $A \leftarrow 1 \rightarrow B$ の極限が積 $A \times B$ になります。また、その逆は $A \xrightarrow{\langle f,g \rangle} B \times B \xleftarrow{\Delta_B} B$ の引き戻し：

$$\begin{array}{ccc}
 E & \xrightarrow{f \circ e = g \circ e} & B \\
 \downarrow e & & \downarrow \Delta_B \\
 A & \xrightarrow{\langle f, g \rangle} & B \times B
 \end{array}$$

から $A \xrightarrow[g]{f} B$ の等化 $E \xrightarrow[e]{f} A$ が得られるために引き戻しと終対象を持つ圏 \mathcal{C} は積と等化を持つことになります。以上から圏 \mathcal{C} が完備であることが判ります。なお、矢 $B \xrightarrow{\Delta_B} B \times B$ は「**対角矢**」と呼ばれる矢です。

2.8 トポス (Topos)

2.8.1 アリストテレスのトポスとの関連

「トポス (Topos)」はアリストテレスの著作「トピカ (Topica, τόποι)」^{*48}に由来し, ここで「Topo」は位相幾何学 (Topology) の “Topo” と同義の「場所」を意味する言葉です. なお, アリストテレスのトポスに多義性があるために日本語に翻訳されていませんが, 弁論の主題に適した論証を探し出す「場所」としての性格を有しています. また, アリストテレスの トピカで論じられているトポスは偶有性に関するもので 103 個, 類に関するもので 81 個, 特有性に関するものが 69 個, 定義に関するものが 84 個と全部で 337 個のトポスが挙げられています ([3] の註を参照). ここで述べる圏論のトポスはそれに似た働き, つまり, 判断の枠組を与える場所としての働きをします. 以下, トポスがどのように判断の枠組を与えるかを見ましょう.

2.8.2 部分対象分類子 (subobject classifier)

終対象 1 を有する圏 \mathcal{C} にトポスを導入するためには「部分対象分類子」と呼ばれる圏 \mathcal{C} の対象 Ω が必要です. この部分対象分類子は終対象 1 からの矢 \top を伴い, 与えられた対象の分類の判断基準, 要するに, ものごとの「あれかこれか」の判断に関わる対象です:

部分対象分類子の定義

圏 \mathcal{C} の終対象 1 からの矢 \top を伴った対象 Ω が次の性質を充たすときに「部分対象分類子 (object classifier)」と呼びます:

- 圏 \mathcal{C} には終対象 1 が存在する.
- 圏 \mathcal{C} の単射である矢 $A \xrightarrow{f} B$ に対して「特性矢」と呼ばれる矢 $B \xrightarrow{\chi_f} \Omega$ が一意に存在し, 次の図式が引き戻しになる.

$$\begin{array}{ccc} A & \xrightarrow{!_A} & 1 \\ \downarrow f & & \downarrow \top \\ B & \xrightarrow{\chi_f} & \Omega \end{array}$$

この図式の意味を小集合の圏 Set で説明しておきましょう. このとき, 対象 A, B の関係は $A \subset B$ として考えられます. それから Ω を $\{\text{True}, \text{False}\}$ の二つの真理値の集合と

^{*48} τόπος の複数形です.

しましょう。次に \top は $A \xrightarrow{!_A} 1$ が一意に存在するために单射で、 1 を True に写すときに図式が可換であることから部分集合 A の元は合成写像 $\chi_f \circ f$ によって全て True に写され、集合 B の像 $f(B)$ 以外の元で構成される集合 $B - f(A)$ は写像 χ_f によって全て False に写されます。このことは χ_f が対象 A とその他の対象を区分する写像として動作し、区分するための特徴付けを行う写像になるために矢 χ_f を「**特性矢**」と呼ぶ理由になります^{*49}。また、部分対象分類子 Ω に付随する矢 $1 \xrightarrow{\top} \Omega$ が分類時の判断基準を与えています。また、対象 A と対象 B との間の矢が单射であることから対象 A は部分対象と呼ばれる対象になります。周延関係を含めて考慮するときは、対象 A が周延されていると言えます。

ここで重要な特性矢として対角矢 δ_A の特性矢 $\delta_A (= \chi_{\Delta_A})$ を挙げておきます：

対角矢 Δ_A の特性矢 δ_A

圏 \mathcal{C} における以下の可換図式を充す矢 δ_A を「**対角矢 Δ_A の特性矢**」と呼び、“ $=_A$ ”とも表記する：

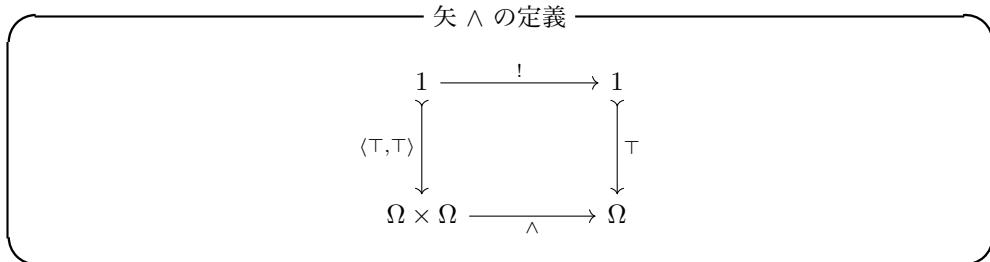
$$\begin{array}{ccc} A & \xrightarrow{!_A} & 1 \\ \downarrow \Delta_A & & \downarrow \top \\ A \times A & \xrightarrow{\delta_A} & \Omega \end{array}$$

この図式の例として圏 Set にて部分対象分類子 Ω を $\{\text{True}, \text{False}\}$, $\top 1 = \text{True}$ の場合を考えてみましょう。このとき $A \times B$ は集合 A, B の積集合になり、 $\delta_A(a, b)$ の意味、すなわちその値は $b = a$ ならば True で、そのときに限ることを意味します。ここで $a =_A b \stackrel{\text{Def}}{=} \delta_A(a, b)$ と定義すると集合 A の二つの元の同値性を判断する演算子 “ $=_A$ ” が定義されることを意味します。ちなみにフレーゲは真理値について彼の概念記法で $\vdash \mathfrak{a} = \mathfrak{a}$ すなわち、 $\forall x(x = x)$ を真 (True), 概念記法で $\vdash \mathfrak{a} \neq \mathfrak{a}$ すなわち、 $\forall x(x \neq x)$ を偽 (False) として定義していますが[27]、ここでの同値性 “ $=_A$ ” の定義で真理値集合に該当する部分対象分類子 Ω が与えられていなければならないために、フレーゲの真理値の定義が妥当でないことが分かります。実際、真理値 True と $\forall x(x = x)$ の値が一致することが主張できても演算子 “=” 自体が真理値に依存するために $\forall x(x = x)$ を真理値 True の定義にできないためです。ところで、アリストテレスによると真理値は「**真や偽は命題の状態を示すもの**」で、「**存在するものを存在すると言い、あるいは存在しないものを存在しないと言うこと**」が真で、「**存在するものを存在しないと言い、あるいは存在しない**

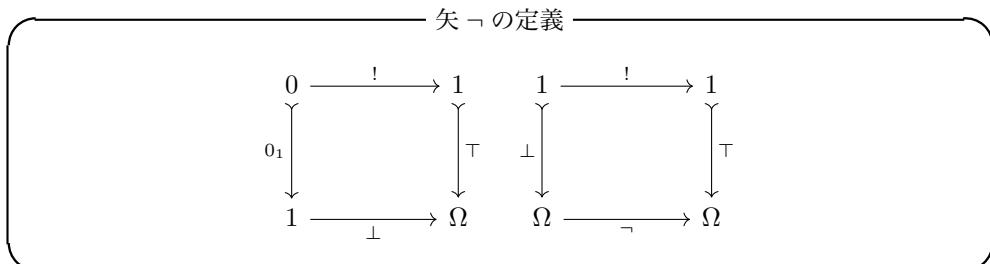
^{*49} 機械的学習はこの特性矢を具体的に構築するための手続になります。

「もの Ω が存在する」と言うこと」が偽である ([2]11b27) と述べていることと、部分対象分類子の可換図式における \top の機能がアリストテレスが真理値について語っていることと符合していることは非常に興味深いことです。

それから圏 \mathcal{C} に部分対象分類子 Ω と付随する矢 $1 \xrightarrow{\top} \Omega$ が存在するときに連言 “ \wedge ” も同値性 “ $=_A$ ” と同様に定義できます:



これを小集合の圏 \mathbf{Set} で考えるとき、部分対象分類子 Ω を $\{\text{True}, \text{False}\}$, $\top 1 = \text{True}$ とするときに矢 “ \wedge ” は $(\text{True}, \text{True})$ のみを True に写す写像として解釈され、まさに連言としての性質が見られ、同値性と同様に「命題の状態を表すもの」になっています。さらに圏 \mathcal{C} に始対象 0 が存在するときに否定 \neg が定義できます:



この \neg の定義は可換図式を二つ必要とします。まず、左側の可換図式が矢 $1 \xrightarrow{\perp} \Omega$ を定義する図式で、右側の図式か矢 \perp を用いて矢 $\Omega \xrightarrow{\neg} \Omega$ を定義する図式になります。この図式の意味を小集合の圏 \mathbf{Set} で部分対象分類子 $\Omega = \{\text{True}, \text{False}\}$ として説明しましょう。まず、矢 \perp の意味ですが、圏 \mathbf{Set} で始対象 $0 = \emptyset$, 終対象 $1 = \{*\}$ であるために $0_1 = \emptyset$, $\perp \emptyset = \text{True}$, $\perp 1 = \text{False}$ になります。したがって右の可換図式から $\neg \text{False} = \text{True}$, $\neg \text{True} = \text{False}$ であることが判ります。

このように圏 \mathcal{C} に終対象 1 が存在して部分対象分類子 Ω が存在すれば、「あれかこれか」という判断に対応する特性写像があり、それによって連言も定義され、さらに圏 \mathcal{C} に始対象 0 も存在すれば否定も定義ができることになります。と、このように一階の論理式を構成する上で必要なものは \forall と \exists といった量化子を除いて揃うことになります。そして次に述べるトポスは一階の論理式が定義できる圏です。

2.8.3 基本トポス

「**基本トポス (elementary topos)**」を次の定義します:

—— 基本トポスの定義 ——

1. 圈 \mathbf{E} には終対象 1 が存在する.
2. 任意の対象 $A, B \in \mathbf{E}$ に対して積 $A \times B \in \mathbf{E}$ が存在する.
3. 任意の対象 $A, B \in \mathbf{E}$ に対して幕 $B^A \in \mathbf{E}$ が存在する.
4. \mathbf{E} には部分対象分類子 Ω が存在する.

1., 2., 3. を充す圏を「**デカルト閉圏 (Cartesian Closed Category)**」と呼び、「CCC」と略記します。また、基本トポスのすべての条件を充たすときに始対象 0 , 直和, 押し出しが存在することが知られています。このことから基本トポスであれば前述の「**あれかこれか**」といった判断に加え、その同一性や連言、否定も定義可能なことから一階の論理式が構築可能です。

—— トポスの定義 ——

1. 圈 \mathbf{E} には終対象 1 が存在する.
2. 圈 \mathbf{E} の任意の対象からなる $A \rightarrow C \leftarrow B$ に対してその引き戻しが存在する.
3. 圈 \mathbf{E} の任意の対象 A, B に対し、その幕 B^A が存在する.
4. 圈 \mathbf{E} には部分対象分類子 Ω が存在する.

トポスになる圏として代表的なものとして小集合から構成される圏 \mathbf{Set} がありますが、部分対象分類子 Ω が存在するということは、任意の対象を分類し得るということを意味し、引き戻しの存在からその分類に普遍性を持つことを意味します。機械学習では「**あれかこれか**」を分類させる函数を学習によって構成させていますが、そもそもそのような函数が存在するものでなければ学習自体が無意味なことです。ところがトポスであれば、「**あれ**」や「**これ**」を包含する部分対象分類子に対する特性写像の構築が可能になるために学習自体に意味があります。

2.8.4 自然数の扱いについて

ここではトポス \mathbf{E} での自然数の扱いについて述べます。最初に「**自然数対象 (Natural Number Object)**」を定義しましょう:

自然数対象 (NNO)

N をトポス \mathbf{E} の対象, $1 \xrightarrow{\xi} N$ と $N \xrightarrow{\sigma} N$ をトポス \mathbf{E} の矢とするときに任意の $1 \xrightarrow{g} A \xrightarrow{h} A$ となる対象と矢に対し, 次の図式を可換にする矢 $N \xrightarrow{f} A$ が一意的に存在するときに N を自然数対象 (Natural Number Object) と呼びます.

$$\begin{array}{ccccc} & & N & \xrightarrow{\sigma} & N \\ 1 & \nearrow \xi & \downarrow f & & \downarrow f \\ & \searrow g & \xrightarrow{h} & A & \xrightarrow{h} A \end{array}$$

NNO とペアノの公理系の対応では対象 N が自然数 \mathbf{N} , 矢 $1 \xrightarrow{\xi} N$ が自然数を指示する操作, $N \xrightarrow{\sigma} N$ が指示された自然数に対して後者関係にある自然数を与える操作, すなわち, $\lambda x.(x + 1)$ に対応します. また, 帰納法の原理は ξ と σ であらかじめ取り込んだ形になっています. さらに自然数対象を持つトポス \mathbf{E} の任意の対象と矢 $A \xrightarrow{g} B \xrightarrow{h} B$ に対して次の可換図式を充たす矢 $A \times N \xrightarrow{f} B$ が一意に存在します:

$$\begin{array}{ccccc} & & A \times N & \xrightarrow{\text{id}_A \times \sigma} & A \times N \\ A & \nearrow \zeta & \downarrow f & & \downarrow f \\ & \searrow g & \xrightarrow{h} & B & \xrightarrow{h} B \end{array}$$

なお, $\zeta = \langle \text{id}_A, \xi_N \rangle$, ここで, 矢 ξ_N は $A \xrightarrow{!_A} 1 \xrightarrow{\xi} N$ です. このことは次の手順で確認することができます. 最初に $A \xrightarrow{g} B \xrightarrow{h} B$ が与えられたときに以下の可換図式を充たす矢 $h' : B^A \rightarrow B^A$ が存在します:

$$\begin{array}{ccccc} & B^A \times A & \xrightarrow{\text{ev}} & B \\ h' \times \text{id}_A \downarrow & \swarrow h \circ \text{ev} & & \downarrow h \\ B^A \times A & \xrightarrow{\text{ev}} & B & & \end{array}$$

この矢 h' は具体的には $h \circ \text{ev}$ の転置: $(\widehat{h \circ \text{ev}})$ として一意に与えられます. さらに $A \xrightarrow{g} B$ に対しては次の可換図式が成立します:

$$\begin{array}{ccc}
 & B^A \times A & \\
 g' \times \text{id}_A \uparrow & \searrow h \circ \text{ev} & \\
 1 \times A & \xrightarrow{g \circ \pi_2} & B \\
 \downarrow \pi_2 & \nearrow g & \\
 A & &
 \end{array}$$

ここで矢 g' は $h \circ \text{ev}$ の転置: $(\widehat{g \circ \pi_2})$ です。これらから $1 \xrightarrow{g'} B^A \xrightarrow{h'} B^A$ が得られますが、トポス \mathbf{E} が自然数対象 N を持つことから次の図式を可換にする矢 $N \xrightarrow{k} B^A$ が一意に存在します:

$$\begin{array}{ccccc}
 & N & \xrightarrow{\sigma} & N & \\
 \xi \nearrow & \downarrow k & & \downarrow k & \\
 1 & \searrow g' & & & \\
 & B^A & \xrightarrow{h'} & B^A &
 \end{array}$$

この矢 $N \xrightarrow{k} B^A$ を転置とする矢 $A \times N \xrightarrow{f} B$ を次で与えます:

$$\begin{array}{ccc}
 A \times N & \xrightarrow{f} & B \\
 \text{id}_A \times k \downarrow & \nearrow \text{ev} & \\
 A \times B^A & &
 \end{array}$$

これらの可換図式から矢 f が求める矢であることが判ります。この図式で $A = N$ とすると、和 “ $+_{\mathbf{E}}$ ” が次で定義できます:

$$\begin{array}{ccccc}
 & N \times N & \xrightarrow{\text{id}_N \times \sigma} & N \times N & \\
 \zeta \nearrow & \downarrow +_{\mathbf{E}} & & \downarrow +_{\mathbf{E}} & \\
 N & \searrow \text{id}_N & & & \\
 & N & \xrightarrow{\sigma} & N &
 \end{array}$$

この可換図式で矢 ζ は $\langle \text{id}_N, \xi_N \rangle$ のことです。

2.9 トポスの基本定理

トポスの基本定理

- 圈 \mathcal{E} がトポスであり、 B が \mathbf{E} の任意の対象であるときにはコンマ圏 $(\mathbf{E} \downarrow B)$ もトポスになる。
- A, B をトポス \mathbf{E} の任意の対象、矢 $A \xrightarrow{f} B$ とするときに二つのコンマ圏 $(\mathbf{E} \downarrow A)$ と $(\mathbf{E} \downarrow B)$ の間に函手 $f^* : (\mathbf{E} \downarrow B) \rightarrow (\mathbf{E} \downarrow A)$, $\Sigma_f : (\mathbf{E} \downarrow A) \rightarrow (\mathbf{E} \downarrow B)$ と $\Pi_f : (\mathbf{E} \downarrow A) \rightarrow (\mathbf{E} \downarrow B)$ が存在して $\Sigma_f \dashv f^* \dashv \Pi_f$ を充たす。

この基本定理の証明を順番に行いましょう。

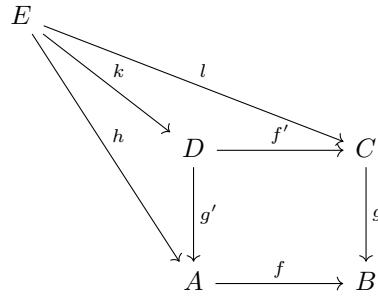
まず、圏 \mathbf{E} が基本トポスであればコンマ圏 $(\mathbf{E} \downarrow B)$ も基本トポスであることを示しましょう。そのためにはコンマ圏 $(\mathbf{E} \downarrow B)$ で終対象の存在、積の存在、幂の存在と部分分類子の存在を示さなければなりません。これらの事項を順番に確認しましょう。

■終対象が存在すること: 対象 B の同一矢 $B \xrightarrow{\text{id}_B} B$ は任意の $A \xrightarrow{f} B$ に対して次の図式が可換になります:

$$\begin{array}{ccc} A & \xrightarrow{f} & B \\ & \searrow f & \swarrow \text{id}_B \\ & B & \end{array}$$

このことから $B \xrightarrow{\text{id}_B} B$ がコンマ圏 $(\mathbf{E} \downarrow B)$ の終対象であることが判ります。

■積が存在すること: 圈 \mathbf{E} が基本トポスであるために引き戻しが存在します。そこで次の引き戻しを考えます:



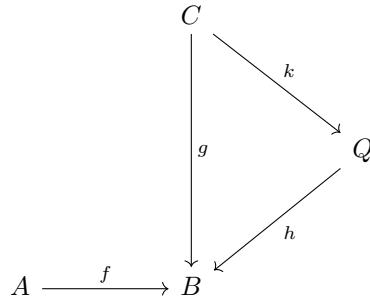
ところで、この引き戻しの可換図式はコンマ圏 $(\mathbf{E} \downarrow B)$ の対象の積の可換図式そのものになります。このときにコンマ圏 $(\mathbf{E} \downarrow B)$ の対象 $D \xrightarrow{f \circ g'} B$ がコンマ圏 $(\mathbf{E} \downarrow B)$ の対象 $A \xrightarrow{f} B$ と $C \xrightarrow{g} B$ の積になります。

■部分対象分類子の存在: \mathbf{E} が基本トポスであることから部分対象分類子 Ω が存在します：

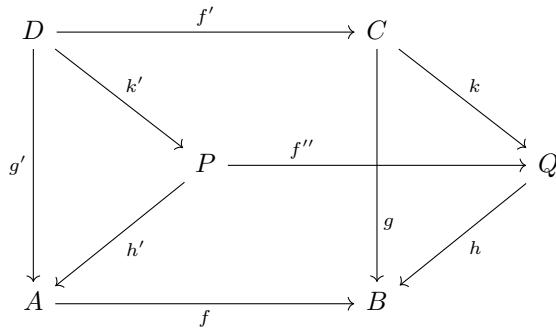
$$\begin{array}{ccc}
 A & \xrightarrow{!_A} & 1 \\
 f \downarrow & & \downarrow \top \\
 B & \xrightarrow{\chi_f} & \Omega
 \end{array}$$

$$\begin{array}{ccccc}
 A & \xrightarrow{f} & B & & \\
 \downarrow h & \searrow f & \swarrow \text{id}_B & \downarrow \top & \\
 C & \xrightarrow{g} & B & \xleftarrow{\pi_2} & \Omega \times B \\
 & \downarrow \chi_h & & & \downarrow
 \end{array}$$

■函手 f^* の存在: 任意の矢 $A \xrightarrow{f} B$ に対して次の図式を考えます：



このような対象 A, B, C, Q を考えると, h と g が $(E \downarrow B)$ の元であり, 矢 $C \xrightarrow{k} Q$ は g, h 間の矢になります. このときに \mathbf{E} が基本トポスであることから $A \xrightarrow{f} B \xleftarrow{g} C$ と $A \xrightarrow{f} B \xleftarrow{h} Q$ の引き戻し D, P が存在し, 次の図式が可換になります:



ここで $h, g \in \text{Ob}(\mathbf{E} \downarrow B)$ に対して $h', g' \in \text{Ob}(\mathbf{E} \downarrow A)$ が存在し, P が引き戻しであることから $g' \xrightarrow{k'} h'$ がただ一つ定まります. ここで $f^*h = h'$, $f^*g = g'$, $f^*k = k'$ とすることで函手 $f^* : \mathbf{E} \downarrow B \rightarrow \mathbf{E} \downarrow A$ が構築できます.

2.10 高階論理 λ -h.o.l. とトポス

基本トポスでは自然に言語を導入することができます。この言語は「圏論による論理学」[16] では「函数型高階論理 λ -h.o.l.」、「Toposes and Local Set Theories」[38] では「locat set language」，一般的には「Mitchell-Benabou Language」と呼ばれるトポスに内在的な言語です。ここでは基本トポスを小集合の圏 **Set** とし，部分対象分類子 Ω を真理値 {True, False} とする，いわゆるブーリアン・トポス **E** の事例を紹介します。なお，圏を小集合の圏 **Set** とするために対象は集合になりますが，函数型言語としては，その対象である集合を「**型 (type)**」と呼ぶために λ -h.o.l. の言語としての性格は型付きの言語になります。また，型が小集合であるために各小集合には集合論から定義される言語を既に保持していますが， λ -h.o.l. は対象としての小集合に限定されることなく大域的に定義される言語になります。

ここでは最初に高階論理 λ -h.o.l. について概要を述べ，それらがどのように基本トポスで実現されるか，つまり，自然に包含されるかを見て行きたいと思います。

2.10.1 高階論理 λ -h.o.l. について

函数型古典高階論理 λ -h.o.l. について説明しておきます。この函数型古典高階論理 λ -h.o.l. は型を持った λ 計算です。ここで型は λ -h.o.l. が扱う対象としての「**領域**」に対応し，これらの領域は意味や対象をのものを表現します。たとえば，数値計算を行うのであれば領域の一つは倍精度の浮動小数点数であり，もう一つは数値の等価性や大小関係の意味に対応する領域としての真理値が考えられます。そういった個体や真理値の領域に加え，さらには命題を評価する，すなわち，命題と真理値の領域の間の写像，個体同士の置き換えを行う写像も考えられます。ここでは領域 D の型が α のときにその領域の型が明示的になるように D_α と表記し，特に t は真理値の型を示すものとします。そして，領域 D_α から領域 D_β の写像全体で構成される領域は，その領域の型を $\langle\alpha\beta\rangle$ でその型を定めます。ただし，この写像の型の表記には次の規則を入れておきます：

函数の型の表記

1. $\langle\alpha, \beta\rangle$ を $\alpha\beta$ と略記してもよい。
2. $\langle\alpha, \langle\beta, \gamma\rangle\rangle$ を $\alpha\langle\beta\gamma\rangle$ や $\alpha\beta\gamma$ と略記してもよい。

まず，1. については単なる略記として認めるることは問題がないでしょう。2. の記号の表記については，記号 $\langle \rangle$ を二項演算子として見なしたときに右側の記号との結合が強いとする立場を探ることを意味しており，このような表記を「**右結合**」と呼びます。ここで写像

が通常の集合の写像であれば $f \circ (g \circ h) = (f \circ g) \circ h$ であるために, この結合の順序はそれほど問題にはなりません. ちなみに論理学に函数概念を最初に導入したのはフレーゲですが, 彼は函数を二項間の関係とみなしており, その場合, 項の場所に注目し, 三変数以上の函数を二項関係の延長として捉えています.

それから λ -h.o.l. の型については次のように定めています:

————— λ -h.o.l. の型 (type) —————

1. e は型である.
2. t は型である.
3. α, β が型であれば $\langle \alpha, \beta \rangle$ も型である.
4. 1., 2. と 3. で構成されたもののみが型である.

最初 1. で述べた型 e は言語対象での「**実在物 (entity)**」の集合を指し, 次の 2. によって型 t と別にあることを主張しています. そして, 2. の型 t が真理値の型であり, 高階言語 λ -h.o.l. で記載された式の意味を表現します. それから 3. は多変数函数で構成される型の存在と構築方法を述べています. この 3. に示すように型には実在物と真理値といった「静的」な側面だけではなく, 写像の合成によって新たな型を創り出す「動的」な側面を有することを主張しています. なお, 写像の型の表記は右結合 (右側の結合を優先する表記) を採用します.

次に λ -h.o.l. はこれらの領域間の言語でもあります. だから言語を構成するための基本的な記号を必要とします. そこで言葉を構成するために最低限必要な記号, つまり, 基本記号を以下で定めます:

————— λ -h.o.l. の基本記号 —————

1. 論理常項: $=_{\alpha}(\alpha t)$
2. 変項: $x_{\alpha}, y_{\beta}, z_{\gamma}, \dots$
3. 補助記号: $\lambda, (,)$

ここでの論理常項は同一領域の対象に対してその同一性を判断する函数です. その型は括弧を外した αat ですが, ここでは型を右結合で表記しているために $a, b \in D_{\alpha}$ に対して $((=_{\alpha}(\alpha t) a)b)$ になります. このことから論理記号 ‘ $=$ ’ を以下で定めることにします:

論理記号 ‘=’ の定義

$$= \stackrel{\text{Def.}}{=} \lambda x_\alpha. (\lambda y_\alpha. (=_{\alpha(\alpha t)} x_\alpha) y_\alpha)$$

高階論理 λ -h.o.l. はこれらの基本記号から次の項の定義に沿って生成される項から古典的論理学の \wedge, \neg, \supset といった論理記号が生成されます。

λ -h.o.l. の型付き項の定義

1. $x_\alpha, y_\alpha, \dots$ は型 α の項である。
2. $=_{\alpha(\alpha t)}$ は型 $\alpha(t)$ の項である。
3. $A_{\alpha\beta}, B_\beta$ に対し $(A_{\alpha\beta} B_\beta)$ は型 β の項である。
4. $(\lambda x_\alpha^i. A_\beta)$ は型 $\alpha\beta$ の項である。
5. 上記, 1. - 4. で構成されたもののみが項である。特に型 t の項を「式 (論理式, formula)」と呼ぶ。

この定義は高階論理 λ -h.o.l. の項と項の生成方法について述べたものになります。また、この項の定義で述べたように項の型が t のものを「論理式」と呼びます。この論理式を構成する上で必要な記号を幾つか定義しておきましょう：

論理記号と式の定義

1.	T	$\stackrel{\text{Def.}}{=} \lambda x_t. x_t = \lambda x_t. x_t$
2.	F	$\stackrel{\text{Def.}}{=} \lambda x_t. T = \lambda x_t. x_t$
3.	\neg_{tt}	$\stackrel{\text{Def.}}{=} (\lambda x_t. (F = x_t))$
4.	$\wedge_{t\langle tt\rangle}$	$\stackrel{\text{Def.}}{=} \lambda x_t. \lambda y_t. (\lambda f_{t\langle tt\rangle}. (f_{t\langle tt\rangle} TT) = \lambda f_{t\langle tt\rangle}. (f_{t\langle tt\rangle} x_t y_t)))$
5.	$\supset_{t\langle tt\rangle}$	$\stackrel{\text{Def.}}{=} \lambda x_t. (\lambda y_t. (x_t = (x_t \wedge_{t\langle tt\rangle} y_t)))$

ここで T, F といった真理値が右辺の式で定められるというよりは、むしろ、言語に含まれる真理値がどのように対応するかを定めたものです。

次に同値性の判断を行う ‘=’、論理式の否定 ‘ \neg ’、論理式の連言 ‘ \wedge ’ と含意 ‘ \supset ’ を高階論理 λ -h.o.l. を使って定義しています。

論理式の定義

1.	$\neg A_t$	$\stackrel{\text{Def.}}{=} \neg_{tt} A_t$
2.	$A_t \wedge B_t$	$\stackrel{\text{Def.}}{=} ((\wedge_{t\langle tt\rangle} A_t) B_t)$
3.	$A_t \supset B_t$	$\stackrel{\text{Def.}}{=} ((\supset_{t\langle tt\rangle} A_t) B_t)$

2.10.2 高階論理とトポスとの関係

ここでは高階論理 λ -h.o.l. とトポス \mathbf{E} との関係を見ることにします。そこで高階論理 λ -h.o.l. の型とトポス \mathbf{E} の対象との対応関係を以下にまとめておきましょう：

型と対象の対応関係		
型 e	→	対象 E
型 t	→	対象 Ω
型 $\langle \alpha, \beta \rangle$	→	対象 B^A

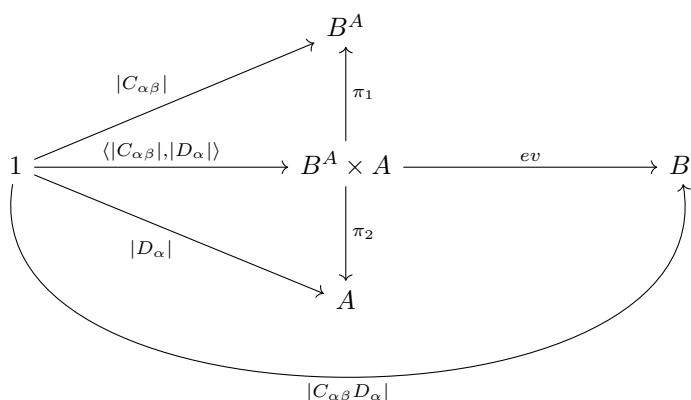
このように λ -h.o.l. とトポス \mathbf{E} の対象との対応付けを行うことができます。トポス \mathbf{E} の対象 a に対してその終対象 1 からの矢 $1 \rightarrow a$ が a の成分を定めることになります。そして、我々の考察は実際の「もの」というよりはその「もの」を指し示す「名辞」であり、その意味では圏の対象そのものというよりは名辞としての矢が項に対応しなければなりません。また、トポスには対象の積、幂や矢の積が存在するといった性質もあります。これらのことを利用して λ -h.o.l. の項 a とトポス \mathbf{E} の矢 $|a|$ との対応付けを行ってみましょう。

■変項 x_α : $A \xrightarrow{|x_\alpha|} A$. ここで $|x_\alpha|$ は id_A になります。

■定項 C_α : $1 \xrightarrow{|C_\alpha|} A$. これは集合の圏であれば終対象 1 から対象 A への矢がその A の成分を一つ定めることを利用したものです。

■項 $C_{\alpha\beta}$: $1 \xrightarrow{|C_{\alpha\beta}|} B^A$.

■項 $C_{\alpha\beta}D_\alpha$: $1 \xrightarrow{|C_{\alpha\beta}D_\alpha|} B$. ここで $|C_{\alpha\beta}D_\alpha| = ev \circ \langle |C_{\alpha\beta}|, |D_\alpha| \rangle$ になります。なお、この可換図式を以下に示しておきます：



■項 $C_{\alpha\beta}x_\alpha$: $1 \times A \xrightarrow{|C_{\alpha\beta}x_\alpha|} B$. ここで $|C_{\alpha\beta}x_\alpha| = \text{ev} \circ \langle |C_{\alpha\beta}| \times |x_\alpha| \rangle$ になります. このことが判る可換図式を以下に示しておきます:

$$\begin{array}{ccccc}
 & & 1 & \xrightarrow{|C_{\alpha\beta}|} & B^A \\
 & \nearrow \pi_1 & & & \uparrow \pi_1 \\
 1 \times A & \xrightarrow{|C_{\alpha\beta}| \times |x_\alpha|} & B^A \times A & \xrightarrow{\text{ev}} & B \\
 & \searrow \pi_2 & & \downarrow \pi_2 & \\
 & & A & \xrightarrow{|x_\alpha|} & A \\
 & & \text{---} & & \text{---} \\
 & & |C_{\alpha\beta}x_\alpha| & &
 \end{array}$$

ただし, この図式では矢の積が判り易くなるように $1 \times A$ を射影 π_1, π_2 を使って分解したくどい説明になっています.

■項 $\lambda x_\alpha.D_\beta$: $|\lambda x_\alpha.D_\beta| = (|D_\beta| \circ \pi_1) : 1 \rightarrow B^A$ になります.

ここで $(|D_\beta| \circ \pi_1)$ は D_β の転置であることが次の可換図式から判ります:

$$\begin{array}{ccccc}
 & & B^A & \xleftarrow{\pi_1} & B^A \times A \\
 & & \uparrow & & \uparrow \pi_2 \\
 & & |\lambda x_\alpha.D_\beta| & & A \\
 & & \uparrow & & \uparrow |x_\alpha| \\
 1 & \xleftarrow{\pi_1} & 1 \times A & \xrightarrow{\pi_2} & A \\
 & & \downarrow \pi_1 & & \downarrow \text{ev} \circ (|\lambda x_\alpha.D_\beta| \times |x_\alpha|) \\
 & & 1 & & B \\
 & & \text{---} & & \text{---} \\
 & & |D_\beta| & &
 \end{array}$$

ここでも矢の積 $|\lambda x_\alpha.D_\beta| \times |x_\alpha|$ が判り易くなるように可換図を構築していますが, そのために $|\lambda x_\alpha.D_\beta|$ が $|D_\beta|$ の転置であることが判り難くなっています. そこで矢の積の

箇所を説明する部位を除いた可換図式を以下に示しておきましょう:

$$\begin{array}{ccc}
 & B^A \times A & \\
 | \lambda x_\alpha . D_\beta | \times | x_\alpha | \uparrow & \searrow \text{ev} & \\
 1 \times A & \xrightarrow{\text{ev} \circ (| \lambda x_\alpha . D_\beta | \times | x_\alpha |)} & B \\
 \downarrow \pi_1 & & \swarrow | D_\beta | \\
 1 & &
 \end{array}$$

この可換図式から $\text{ev} \circ (| \lambda x_\alpha . D_\beta | \times | x_\alpha |) = | D_\beta | \circ \pi_1$ より $| \lambda x_\alpha . D_\beta |$ が $| D_\beta | \circ \pi_1$ の転置であることが容易に判るでしょう.

■項 $\lambda x_\alpha . C_{\alpha\beta} x_\alpha$: $1 \times A \xrightarrow{|C_{\alpha\beta} x_\alpha|} B$. なお, 以下の可換式で $|C_{\alpha\beta} x_\alpha| = |\text{ev} \circ (| \lambda x_\alpha . D_\beta | \times | x_\alpha |)|$ になる結果を用いています:

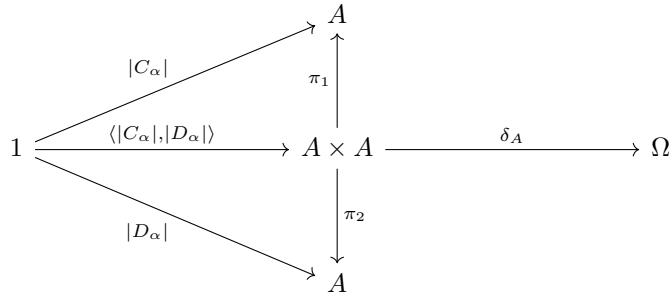
$$\begin{array}{ccccc}
 & B^A & & A & \\
 \leftarrow \pi_1 & B^A \times A & \xrightarrow{\pi_2} & A & \\
 | \lambda x_\alpha . C_{\alpha\beta} x_\alpha | = | C_{\alpha\beta} | \uparrow & | \lambda x_\alpha . C_{\alpha\beta} | \times | x_\alpha | \uparrow & & | x_\alpha | \uparrow & \\
 1 & \xleftarrow{\pi_1} & 1 \times A & \xrightarrow{\pi_2} & A \\
 & \curvearrowright | C_{\alpha\beta} x_\alpha | & & & \curvearrowright B
 \end{array}$$

この図式の骨子を取り出したものが次の可換図式になります:

$$\begin{array}{ccc}
 & B^A \times A & \\
 | \lambda x_\alpha . C_{\alpha\beta} | \times | x_\alpha | \uparrow & \searrow \text{ev} & \\
 1 \times A & \xrightarrow{\text{ev} \circ (| \lambda x_\alpha . D_\beta | \times | x_\alpha |)} & B
 \end{array}$$

これらの可換図式から $|\lambda x_\alpha.C_{\alpha\beta}| \times |x_\alpha| = |C_{\alpha\beta}| = |C_{\alpha\beta}x_\alpha|$ であることがわかります.

$$\blacksquare \text{項 } C_\alpha = D_\alpha : 1 \xrightarrow{\delta_A \circ \langle |C_\alpha|, |D_\alpha| \rangle} \Omega$$



ここではじめて判断の是非を問う項が出てきました. この場合は Δ_A の特性矢 δ_A を用いて命題の判断を行うことが明示されています.

2.10.3 λ -h.o.l. の解釈について

トポス \mathbf{E} には終対象、積や幂が存在するために前述のように λ -h.o.l. で扱う項とトポス \mathbf{E} の矢の対照が行えました. ただし、ここまででは項とトポスに対応関係があると主張するだけで、アリストテレスの言う「トポス」のように「判断のよりどころ」になるものとはまだ言えません. では一体、何が不足なのでしょうか？ 現時点では「等しいかどうか」という判断に対応する論理常項 “=” 以外に判断に関係するものがなく、言語としても項と項同士の合同性以外の判断ができないということです. つまり、少なくとも論理学が成立するためには項を繋ぐものがまだ必要です.

ここで現代の論理学の創始者と言えるフレーゲは含意 ‘ $A \supset B$ ’ を $\frac{}{} B$, 否定 ‘ $\neg A$ ’ $\frac{}{} A$

を ‘ $\frac{}{} a$ ’ とし、それと「Modus Ponens(MP)」と呼ばれる推論規則^{*50}、それと「すべての...」や「ある...」に対応する量化詞から「概念記法」と呼ばれる壯麗な論理学体系を構築しているのです. したがって高階論理 λ -h.o.l. が言葉であるためには含意、否定、量化詞と MP に相当する推論規則が必要になるでしょう. ところで高階論理 λ -h.o.l. の含意は次のように定義されています:

^{*50} ‘P である’ と ‘P ならば Q である’ から ‘Q である’ を導く推論規則です.

λ -h.o.l. の含意 \supset の定義

$$\supset_{t(t,t)} \stackrel{\text{Def.}}{=} \lambda x_t. \lambda y_t. x_t = (x_t \wedge y_t)$$

このように λ -h.o.l. では論理常項 “ $=_{t(t,t)}$ ” と連言 “ \wedge ” から含意が定義されるので、トポス \mathbf{E} 内でこれらが定義できていれば包含が定義可能であり、それから否定が定義できて MP に相当する推論もトポス \mathbf{E} にて問題なく成立するのであれば、フレーゲにならって言語をトポス \mathbf{E} にて構築できるということになります。

そしてフレーゲは量化詞 “ \forall ” を論理学に初めて概念記法で導入していますが、この高階論理 λ -h.o.l. では

$$x_\alpha D_t \stackrel{\text{Def.}}{=} \forall \lambda x_\alpha. D_t = \lambda x_\alpha. \mathbf{T}_t$$

で定義されます。この定義もトポス \mathbf{E} の下で

トポス \mathbf{E} における量化詞

$$\begin{aligned} & |\forall x_\alpha D_t| = |\lambda x_\alpha. D_t| = \lambda x_\alpha. \mathbf{T}_t | \\ = & \delta_{\Omega^A} \circ \langle |\lambda x_\alpha. D_t|, |\lambda x_\alpha. \mathbf{T}_t| \rangle \\ = & \delta_{\Omega^A} \circ \langle |D_t| \hat{\circ} \pi_1, |\mathbf{T}_t| \hat{\circ} \pi_1 \rangle \end{aligned}$$

になることがわかります。

$\mathbf{E} \models C_t$ について

ここで高階論理 λ -h.o.l. には 4 つの公理があります：

λ -h.o.l. の公理

- A.1 $(x_\alpha = y_\alpha) \supset (A_{\alpha t} x_\alpha = A_{\alpha t} y_\alpha)$
- A.2 $(A_{\alpha\beta} = \forall x_\alpha (A_{\alpha\beta} = B_{\alpha\beta}))$
- A.3 $(\lambda x_\alpha^i. A_\beta) B_\alpha = A_\beta [x_\alpha^i := B_\alpha]$
- A.4 $(A_{tt} T_t \wedge A_{tt} F_t) = \forall x_t A_{tt} x_t$

2.11 ニューラルネットワークについて

2.11.1 ニューロンモデル

ここでは図 Set の特性函数 χ の具体的な構築方法の一つについて簡単な考察をしたいと思います。

あれかこれかの判断は「脳」で行っていますが、この脳は「**ニューロン (神経細胞, neuron)**」と呼ばれる神経細胞で構成され、大人の脳で 100 億から 1000 億のニューロンがあるといわれています。このニューロンのおおよその構造は、「**細胞体 (Soma)**」から樹木の枝のように複雑な分岐を持つ「**樹状突起 (Dendrite)**」、通常は一本の太い「**軸索 (Axon)**」に大きく区分できます。

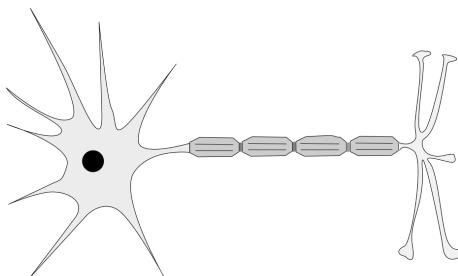


図 2.3 ニューロンの概略図

樹状突起は細胞体から樹状に出ており、細胞体の軸索小丘からは神経纖維を内包する軸索が伸びています。脊椎動物の場合、軸索は電気抵抗の大きなシュワン (schwann) 細胞が何重にも取り巻いてできた「**髓鞘**」、あるいは「**ミエリン**」と呼ばれる 2[mm] 程度の長さの鞘で覆われ、髓鞘と髓鞘の間にはランビエ絞輪と呼ばれる神経纖維の細胞膜が剥き出しになった部位があります。そして、軸索の末端付近でさらに枝分れし、この

枝状の末端を終末側枝と呼びます。これらの側枝の先端には「**シナプス (synapse)**」があり、この部位を介して他のニューロンの樹形突起と間接的に繋がります。ここでシナプスで直接、別のニューロンの樹状突起に繋るのではなく、ここには間隙があり、ニューロンで発生した電気信号が直接、他のニューロンに伝播することはありません。シナプスでは軸索からの電気信号を「**神経伝達物質**」と呼ばれるアセチルコリン、ノルアドレナリンやドーパミンなどの化学物質に置換え、これらの化学物質はシナプスを経由して樹状突起から細胞体に入ります。そして、これらの化学物質の影響で細胞体の内部電位が上昇して、ある閾値を越えた時点でパルス状の信号を軸索に送出します。この状態をニューロンが「**興奮**」、あるいは「**発火**」したと呼びます。この軸索をパルスが通過する際に、髓鞘がコンデンサとして働き、細胞体から伝播するパルスをランビエ絞輪から絞輪へと跳躍的に伝達することで高速・効率的に転送できます。イカのような無脊椎動物では軸索は髓鞘に覆わっておらず、高速に情報を伝達するために神経纖維が太くなっています。その結果、「**巨大神経纖維**」と呼ばれますが、最も速いヤリイカの伝達速度が 20[m/sec] 程度であり、人間では伝

達速度が $100[\text{m/sec}]$ 以上と人間の神経の伝達速度に及びません。この神経纖維の情報伝達の電気的な仕組については [18] を参照して下さい。

ニューロンの特徴としては以下の性質が挙げられます：

1. 空間的加法性：

n 個の樹状突起からの情報 x_i に重み係数 w_i を加えた線形和： $\sum_{i=1}^n w_i x_i$ で内部電圧が決定されます。この性質が特に重要で、後に述べる学習は、結局はここでの重み係数を試行錯誤で定める手法です。

2. 時間的加算性：

ニューロンに短い時間で反復刺激したり、複数の神経を刺激すると加重が生じ、それが閾値を越えると発火します。この現象はニューロン内部の電位 $u(t)$ が各信号 x_i と重み v_i の畳み込み (convolution)： $v_i * x_i(t) = \int_{-\infty}^t v_i(t-s)x_i(s)ds$ の総和で定まるとも言えます。

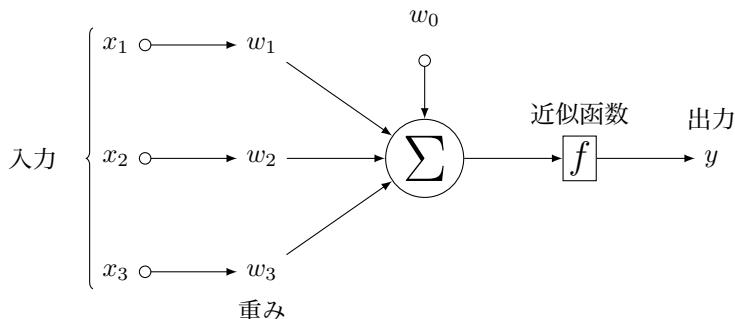
3. 非線形性：

樹状突起からの情報の大きさに出力が比例するという入力に対する線形性はありません。時間的加法性もあるために、その時点の入力信号の大きさに限定されず、内部電圧が閾値を越えた時点で信号が出力されます。

4. 二值性：

軸索からの信号出力は「信号がある」か「信号がない」をそれぞれ表現する 0 か 1 のいづれか一つの出力になります。つまり、神経の情報伝達には減衰はありません。

このニューロンを単純化した数学モデルとして表記したものが以下の図になります：



このニューロンは入力が x_1, x_2, x_3 の 3 個の入力であり、これらの入力とバイアス w_0 の線形和 $\sum_{i=0}^n w_i x_i$ を近似函数 f と合成したものとして表現されます。なお、 $x_0 = 1$ として

います。ここで近似函数 f として与える函数によってニューロンの数学モデルには静的デジタルモデルと動的デジタルモデルの二種類があります：

■静的デジタルモデル：(単位) ステップ函数 $T(t)$ で近似したものです：

$$T(t) = \begin{cases} 0 & t < 0 \\ 1 & t \geq 0 \end{cases}$$

ここで入力側の信号を x_1, \dots, x_n とするときにニューロンの空間加法性に対応する重み w_i を使って $T(w_0 + w_1x_1 + \dots + w_nx_n)$ として表現されます。

■動的デジタルモデル：シグモイド (sigmoid) 函数 $\sigma(t) = \frac{1}{1 + e^{-t}}$ を利用したものです。このときに入力側の信号を x_1, \dots, x_n とするときに空間的加法性に対応する重み w_0, \dots, w_n を使って $\sigma(w_0 + w_1x_1 + \dots + w_nx_n)$ で表現されます。シグモイド函数 σ を採用した理由は単純にそれが十分に滑らかな函数であるためです。ニューロン回路の計算でよりよい函数があれば単純にそれを利用します。

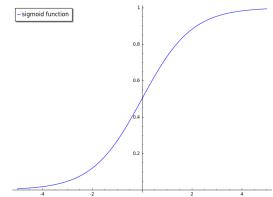


図 2.4 シグモイド函数

2.11.2 ニューロン素子

ニューロンは 0 か 1 のいづれかの値を出力とします。この性質を使って論理回路を構築することができます。このことに言及したのは MacCulloch と Pitts で、ここでは MacCulloch と Pitts の数理モデルについて述べます。まず、重要なことは、「論理式の形成規則」から量化詞を含まない論理式は論理式の否定、論理式の和と積から構築できます。このことから論理式の否定、和と積がどのようにニューロンモデルで表現できるかを決定できれば量化詞を持たない論理式が表現できることが判ります。

■否定： $\neg x$ は $T(-1 - x)$ で表現することができます。

■論理和： $x \vee y$ は $T(-1 + x + y)$ で表現することができます。

■論理積： $x \wedge y$ は $T(-2 + x + y)$ で表現することができます。

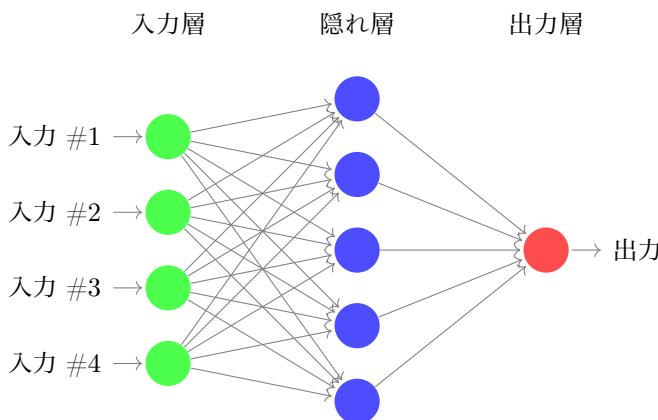
このように論理式の和、積と否定はニューロン素子一つで表現することができることから、量化詞を持たない論理式をニューロン素子を使って表現できることが判ります。さらに、量化詞を持たない論理式は、論理式の否定と積でグループ化し、それらの論理和として表現することができます。ここで論理式の否定と積のグループ化は一つのニューロン素子で表現可能であり、このことは論理和のニューロン素子を一つ加えることで、量化詞を持たない論理式が表現可能なこと、すなわち、二層のニューロン素子で任意の論理式が表現で

きることを意味します。

2.11.3 古典的パーセプトロン

古典的パーセプトロンは視覚による判断をモデルとしたフィードバック回路を持たない三層のニューロン素子で構成された回路です。入力側の第1層を「**感覚層**」、その次の第2層を「**連合層**」、そして出力側の第3層を「**反応層**」と呼びます。なお、第1層の感覚層が入力側にあるために「**入力層**」、第3層の反応層が出力側にあるために「**出力層**」、第2層の連合層は入力側からも出力側からも隠れた層になるために「**隠れ層**」とも呼ばれます。

この古典的パーセプトロンの例を挙げておきましょう：



この例では入力が4つ、入力層が4個のニューロン素子、隠れ層が5個のニューロン素子、そして出力側が1個のニューロン素子で構成されたパーセプトロンの例です。

ちなみに人間の脳の大脳皮質は性質の異なる6層のニューロンで回路が構成されていることが知られています。この古典的パーセプトロンは三層であり、三層以上の階層を持つニューロン回路に対する学習を「**深層学習 (deep learning)**」と呼びます。

2.11.4 学習

ステップ函数で近似された一つのニューロン素子 $\nu(x_1, \dots, x_n)$ は実数の重み w_0, \dots, w_n を使って $T(\sum_{i=0}^n w_i x_i)$ 、ここで $x_0 = 1$ と記述されます。ところで、 $B = \{0, 1\}$ とするときにニューロン素子は函数 $\nu : B^n \rightarrow B$ であり、その成分 $\zeta = (\zeta_1, \dots, \zeta_n) \in B^n$ を0か1の何れかに分類する函数です。ところで、ニューロン素子をステップ函数 T で近似しているために $\nu(\zeta) = 1$ になるのは $w_0 + \sum_{i=1}^n w_i \zeta_i \geq 0$ の

ときであり, $w_0 + \sum_{i=1}^n w_i \zeta < 0$ であれば $\nu(\zeta) = 0$ になります. ここで逆に入力と出力を定め, 重み w_i を求めることはできないでしょうか? これは δ -学習則と呼ばれる出力からの誤差を利用してベクトル v を調整するという方法で, いわゆる, 「**教師付き学習**」と呼ばれる機械学習になります.

第3章

Pythonについて

3.1 Python の概要

3.1.1 言語仕様としての Python

Python は一つの言語仕様であり、通常は C で記述された対話的処理言語 CPython を指しますが、他に Java による Jython, .Net Framework/Mono による IronPython, Python 自体^{*1}による PyPy, Python から C ライブラリを効率よく利用するために開発された Cython と環境や目的に応じた実装があります。SageMath は CPython 2.7 を基盤とする一方で、処理速度が重要視される C ライブラリの利用では Cython が用いられています。なお、Python には 2 系と 3 系が互換性の問題のために並立していますが、NumPy といった主要なパッケージの Python 2 向けの開発は停止するなど、Python 3 への移行が加速している状況です。

3.1.2 簡素化された構文

Python は構文が簡素で、Python 本体を小さくして機能拡張はモジュールで行います。Python の構文に関しては §3.5 に示すように変数宣言は不要、条件分岐は if 文、例外処理は try 文、反復処理は for 文と while 文と必要最低限の構文に抑えられ、その結果、常識的なプログラミングで妥当な結果が得られ、Perl^{*2} や Mathematica に見られる複雑な処理を一行に閉じ込める「超絶的技巧」を駆使したプログラミングではなく、均質的なプログラムに収斂されます。このことに加え、PEP-8^{*3}に代表されるプログラム記述のためのガイドラインがあり、それらのガイドラインにしたがってプログラムを記述しさえすれば、文書化も含めて、ある程度の品質のプログラムが得られるという実用本位の言語です。

3.1.3 構文要素としての字下げ

C 等の多くの言語でプログラムの構造の視覚的把握のために「インデント (indent, 字下げ)」が用いられていますが、Python の字下げは構文上、必要不可欠な要素です。その結果、Python のプログラムは 2 次元的な構造を持ち、視覚的にプログラムの構造の把握ができます^{*4}。この字下げは PEP-8 で欧文間隔 (Space) のみの 4 文字を字下げの単位とすることが推奨されています。たとえば、C の if 文は直線的に平面的に記述しても空白

^{*1} 正確には Python に幾つかの制約を加えた RPython です。

^{*2} 駱駝形のプログラム例 (camel code): <https://gist.github.com/cgoldberg/4332167>, 4 頭のラクダを ASCII アートで描きますが、プログラム自体が ASCII アートになっています。

^{*3} PEP(=Python Enhancement Proposal): Python 改善提案書

^{*4} 仕様として字下げを持つ言語に FORTRAN77 もありますが、FORTRAN77 はカード読取機の制約に由来し、プログラムの構造の可視化の意図はありません。

文字の Space, TAB や改行による字下げは C の構文上の積極的な意味を持たないためにプログラマーの意図とは別にプログラムとしての違いもなければ構文上の問題もありません。ところが Python ではクラスやメソッドの宣言、分岐や反復といった構文が複数の行で構成されるときに、その構文を構成する行に対して空白文字の Space や TAB を使って、それらの文字数と並びを含めた水準を揃えて字下げを行う必要があります。そのためには

```
if x == 0:  
    y = 1  
else:  
    y = 0
```

のように if 文内部の文 (ここでは ‘y = 1’ と ‘y = 0’) と if 文を構成する文節 if と else が同じ字下げの水準でなければなりません。より正確には文節の末端に記号 “:” があれば次の行から字下げを行うか、その次の行に記号 “:” が含まれないときのみに線的に記述することが許容されます。そして、字下げを伴う構文を終えると字下げの水準を一段階、元に戻します。したがって次の記述:

```
if x == 0: y = 1  
else: y = 0
```

は許容されても、記号 “:” を二つ以上含む線的な記述⁵:

```
if x==0: y = 1 else: y = 0
```

および、字下げの位置がチグハクでプログラムの構造が表現できていない記述:

```
if x == 0:  
    y = 1  
else:  
    y = 0
```

の双方は構文エラーになります。

3.1.4 文書文字列 (docstring)

Python にはプログラム内部に解説や例題等の文書を包含することでプログラム自体の文書性を高める工夫があります。このプログラムに埋め込まれた文字列を「文書文字列 (docstring)」と呼びます。プログラム中に文字列を組込む工夫は Python だけではなく、

⁵ ‘y = (1 if x==0 else 0)’ は可能です。

LISP や MATLAB 系言語でも見られ、MATLAB 系言語ではヘルプで表示されるオンラインマニュアルとして用いられています。Python でも文書文字列を函数定義の際に所定の位置に書き込むとオンラインヘルプで使えます：

```
>>> def add2(x):
...     u"""
...     2を足すよー
... """
...     return x+2
...
>>> help(add2)
```

Help on function add2 in module __main__:

```
add2(x)
    2を足すよー
>>> add2.__doc__
u'\n    2をたすよー\n    '
>>>
```

ここで“>>>”がインタプリタの通常のプロンプト、次に現れる文字の列“...”がPython の文が入力途上にあることを示すプロンプトで、ここでは函数 add2() を定義中であることを示しています。そして、函数 add2() を定義する文の def 節の次の三行が文書文字列を入力している箇所です。Python の文書文字列は三連続の二重引用符 ("")、あるいは三連続の单引用符 ('') で括られた文字の列で、さらに文書文字列のエンコーディングが UTF-8 であることを指示するために接頭辞“u”を配置しています^{*6}。このように所定の位置に置かれた文書文字列は函数 help() で表示できます。この例では定義した函数 add2() の文書文字列を函数 help() で表示していますが、Python ではオブジェクトの属性 __doc__ に記載した文書文字列が割り当てられます。そのことを add2.__doc__ で確認しています。

Python の文書文字列は三連続の二重引用符 ("") か三連続の单引用符 ('') で括られた文字列であるために一重の引用符であれば文書文字列内部に記載できます。また、オブジェクト内の文書文字列の閲覧には函数 help() が使えますが、Python の組込のオブジェクトのオンラインヘルプはモジュール pydoc であらかじめ整形されています。以下にインタプリタ上で函数 help() を使って函数 open() を調べた様子を示します：

Help on built-in function open in module __builtin__:

```
open(...)
```

^{*6} Python 3 では文字コードが UNICODE になります。

```
open(name[, mode[, buffering]]) -> file object
```

Open a file using the `file()` type, returns a file object. This is the preferred way to open a file. See `file.__doc__` for further information.
lines 1–7/7 (END)

この本文は函数 `open()` の文書文字列で、内容は ‘`__builtins__.open.__doc__`’ で直接確認ができます。なお、インタプリタで函数 `help()` で引数を指定せずに ‘`help()`’ と入力するとヘルプが起動してプロンプトが “`help>`” に切り替わり、この状態で調べたい事項を入力すれば函数 `help()` と同様の結果が得られ、ヘルプから抜けるときは ‘`quit`’ か ‘`q`’ の何れかを直接、入力するとインタプリタに戻ります。

Python の文書文字列は PEP-257 にその規約があり、さらに「**reST(reStructuredText)**」と呼ばれる「**組版指示 (markup) 言語**」で記述することが PEP-287 で提唱されています。reST は Markdown 同様の組版指示言語で、同類の HTML や LaTeX で見られるタグや命令が不要で、プレーンテキスト上で見出しや箇条書きをアスキーアート風に記述するために、その記述性と可読性が高いものです。そして、reST の文書は文書生成ツールの Sphinx^{*7}を使うことで LaTeX, HTML や PDF といった書式の文書に変換できます。このように Python はプログラムの文書化も重要視した仕様になっています。

3.1.5 クラスに基づくオブジェクト指向プログラミング言語

オブジェクト指向プログラミング言語では扱う対象全てを「**オブジェクト**」として捉えます。さらにクラスに基づくオブジェクト指向プログラミング言語であれば、扱うべき対象を抽象化して「**概念**」として捉え、「**クラス**」をオブジェクトを説明規定するもの、すなわち、**概念の内包**や**概念の外延**として表現します。ここで実際に扱うデータは現実のもの、あるいはそれに最も近接するものために「**概念の外延を構成する個体**」に相当し、クラスが実体化したものとして捉えられます。このクラスの実体化を「**インスタンス化 (instantiation)**」、インスタンス化したオブジェクトを「**インスタンス (instance)**」と呼び、インスタンスはそのクラスの成員になります^{*8}。それからクラスがあるクラスのインスタンスであるときに、この「**インスタンス化で得られたクラス**」を「**クラス・オブジェクト (class object)**」、「**クラスのクラス**」として、クラスの雛型に相当するクラスを「**メタクラス (metaclass)**」、実体化したオブジェクトを「**インスタンス・オブジェクト (instance object)**」と呼びます。

*7 <http://www.sphinx-doc.org/en/stable/index.html> と <http://docs.sphinx-users.jp/> を参照。

*8 ただし、クラスが集合になるとは限りません！

以下に最も簡単な Python でのクラスの定義を示しておきます。なお、この定義で生成されるオブジェクトの型が Python 2 で古典的クラス（旧スタイル）、Python 3 でクラスタイプ（新スタイル）になるという相違点がありますが、ここでの解説で差異はまだ露呈しません：

```
class TEST:
    pass
```

`pass` 文は「**何もしない**」ことを意味する文で、このクラスが名義的な定義であることを意味します。このクラスに対応するオブジェクトの生成は ‘`a = TEST()`’ で行なわれ、この際に名前 `a` への束縛も行われます。このクラス `TEST` は名義的で好き勝手ができます。その様子を以下に示しておきましょう：

```
>>> class TEST:
...     pass
...
>>> a = TEST()
>>> a.name = 'mike'
>>> a.weight = '10kg'
>>> a.age = '10years'
>>> a.name
'mike'
>>> a.weight
'10kg'
>>>
```

`TEST` クラスを定義したのちに ‘`a = TEST()`’ で実体化したオブジェクトを名前 `a` に束縛し、`a.pet`、`a.weigth` と `a.age` という属性に値を設定す処理は C の構造体に類似しています。ここで ‘`class TEST:`’ の末尾の記号 ‘`:`’ は文節の末尾を示す記号です。なお、名前 `a` に続く `name` や `age` は名前 `a` で参照されるオブジェクトの「**属性 (attribute)**」で、クラス `TEST` の属性ではありません。Python の属性の表記はリンネの二分法に対応し、先頭がインスタンスの名前、うしろが属性、これらの区切記号が記号 ‘`.`’ です。この例で示すように代入式では演算子 ‘`=`’ を用います。とくに Python の代入式ではタプルに対しても代入ができます：

```
>>> a = [1,2,3,4]
>>> x, y, z, w = a
>>> x, y, z, w
(1, 2, 3, 4)
>>>
```

ここでは変数のタプル ‘`x, y, z, w`’ にリスト ‘`[1, 2, 3, 4]`’ の内容を割当てています。この

割当は演算子 “=” の双方の被演算子の長さが等しいときにできます。

次にもう少し複雑なクラスを定めてみましょう。ここで定義するクラスは C の構造体により類似した構造です：

```
class TEST:  
    x = 1  
    y = 1
```

このクラスの定義では二つのクラス属性 x と y があり、それらの値として 1 を設定しています。実際に使ってみましょう：

```
>>> class TEST:  
....     x = 1  
....     y = 1  
....  
>>> a1 = TEST()  
>>> a1.x  
1  
>>> a1.y  
1  
xx z  
>>> a1.x = 128  
>>> a1.y = 0  
>>> a1.x  
128  
>>> a1.y  
0  
>>>
```

この例ではクラス TEST を定義し、「a1 = TEST()」でインスタンス化と同時に名前 a1 にオブジェクトを束縛させて属性を参照しています。最初の例と同様にインスタンス化したオブジェクトの属性値変更の影響で上位のクラスの属性変更は生じません。と、ここまで使い方ではクラスは C の構造体と同様です。

さて、クラスには属性だけではなくメソッドという機能があります。たとえば猫には「雨が降る前に顔を洗うような仕草をする」という習性があるために家の猫のミケがそのような仕草をしたときに洗濯物を取り込むことは妥当な行為ですが、犬のポチがその仕草をすることが洗濯物を取り込む理由になりません。この場合はミケやポチといったインスタンスが属するクラスがそれぞれ「猫」と「犬」と異なり、おまけに犬にそのような属性(習性)がないためです。クラスは我々が扱う対象が「何であるか」と「どのようなもので

あるか」という問に対する回答であり、それには何かの値だけではなく何等かの機能も当然、含まれ、その機能を表現したものがメソッドです。メソッドはクラスに結び付けられた函数として表現され、そのクラスのインスタンスであるか、そのクラスの派生クラスのインスタンスでなければ使えません。狸が葉っぱを乗せたらお侍さんに化けるからといって猫が顔を洗う仕草をすると三つ又の化け猫になる訳ではありません。メソッドひとつにしても、そのオブジェクトに作用するものから、オブジェクトに何等の副作用を与えないものもあります。

また、クラスに設定された値にせよ、そのクラスであれば一定値になるものや取り得る値が一意でないものや、ある値を持つということ自体がオブジェクトを特徴付けるもの、つまり、「**特有性**」である場合、あるいは値が程度を表現し、その値が設定されない状態も考えられる場合、つまり、「**偶有性**」である場合が考えられます。また属性の役割を考えると、そのクラスを特徴付ける種差や特有性のようにクラス単位で共通になる値が設定される属性、個々のインスタンスごとに異なる値が設定される属性の二種類が考えられます。ここで前者の属性のようにクラス全体で共通になる値が設定される属性を「**クラス変数**」、後者のようにインスタンスごとに異なる値が設定される属性を「**インスタンス変数**」と呼びます。なお、PythonにはC++のようなクラス変数やインスタンス変数に対する型宣言がないために変数の使い方で両者を区別します。たとえば、「**クモ**は足の数が8本」の「**足の数が8本**」はクモというクラスを特徴付ける属性の一つで、「**足の数**」がクラス変数です。また、「**犬**のニコは年齢が6ヶ月」の「**年齢**」は犬というクラスに付随する属性の一つで、個体（インスタンス）ごとに異なるため、こちらはインスタンス変数の例になります。具体的にはメソッド`__init__()`でオブジェクトの生成時に属性に対応する変数値がインスタンスごとに設定される属性がインスタンス変数、クラスで共通の値を持つべき変数がクラス変数と言えるでしょう。また、Pythonの属性には他の言語の「**公開 (public)**」や「**非公開 (private)**」といった概念がありません。そこで、Pythonでは非公開にしたい属性名に「`_`」を先頭に付けることで隠すことができます。さらに属性の設定、取得と削除に制約を設けたり、属性値の変更で依存関係にある属性の値が自動変更されるように属性の管理を行いたければ「**記述子 (ディスクリプタ)**」を利用します。ただし、クラスタイプ型でなければならないためにクラス`object`を継承する必要があります。

Pythonのメソッドにはクラス操作に関わる「**クラスメソッド (classmethod)**」とインスタンス操作に関わる「**インスタンスマソッド (instancemethod)**」と函数的なメソッドである「**静的メソッド (staticmethod)**」があります。そして、クラスメソッドと静的メソッドの定義では「**デコレータ**」を用い、クラスメソッドなら`@classmethod`、静的メソッドなら`@staticmethod`が定義で用いられます。これらのメソッドには定義の際の引数に違いがあり、クラスメソッドでは第1引数に‘`cls`’というクラスそれ自体に対応する

固定の名前があり、そのクラスの属性の参照が可能ですが、静的メソッドはクラスそれ自身を示す引数を持たないために参照するクラス名を直接指定する必要があります。そのためクラスメソッドでは指定した属性がそのクラスになければ継承関係でより上位のクラスへと遡る「動的 (dynamic) な参照」^{*9}が行われますが、静的メソッドでは名前で直接指定したクラスの参照に留まり、「属性の参照を勝手に遡らずに指定した範囲内で行われる」という意味で「静的 (static) な参照」になります。

では先程の TEST クラスのクラス変数に値を束縛し、それらの和を計算するインスタンスマソッドを追加したクラスの定義の例を挙げておきましょう：

```
class TEST:

    x = 1
    y = 1

    def wa(self):
        return self.x + self.y
```

この例ではクラス変数とインスタンスマソッドを属性を持つクラス TEST を定義しています。メソッドの定義自体は Python の函数定義と同様ですが、インスタンスマソッドの定義で操作すべきインスタンスを ‘self’ という名前で指示し、必ず第 1 引数に配置します。また、メソッド内部でインスタンスの属性の参照でも ‘self’ でインスタンスそれ自身を表現します。なお、ここで定義するインスタンスマソッド wa() はクラス属性 x と y の和を返却します。実際に動かしてみましょう：

```
>>> class TEST:
...     x = 1
...     y = 1
...     def wa(self):
...         return self.x + self.y
...
>>> a1 = TEST()
>>> a1.x = 10
>>> a1.y = 2
>>> a1.wa()
12
>>>
```

^{*9} MRO, あるいは C3 linearization がそのアルゴリズムとして用いられます。詳細は §3.10 を参照。

と、引数 self はメソッドの引数として現れません。ところで、定義したクラス等のオブジェクトや、オブジェクトのメソッドや属性に何があるかを調べる方法はないでしょうか？この目的に応えられる函数が組込函数 dir() です。以下に起動したてのインタプリタで函数 dir() を使った結果を示しておきます：

```
>>> dir()
['__builtins__', '__doc__', '__name__', '__package__']
>>> globals()
{ '__builtins__': <module '__builtin__' (built-in)>, '__name__': '__main__',
  '__doc__': None, '__package__': None}
>>> __name__
 '__main__'
>>>
```

組込函数 dir() は「**参照範囲（スコープ）内**」の名前のリストを返却する函数で、引数がなければ大域变数と参照可能なオブジェクトの名前のリスト、オブジェクトの名前が引数として与えられるとそのオブジェクトのメソッドや属性の名前のリストを返却します。この例では ‘dir()’ と引数なしでトップレベルのスコープ内の大域变数のリストを返却します。なお、大域变数の情報を返す函数には函数 globals() もあり、こちらは指定したスコープ内の大域变数の辞書を返却します。ところで大域变数 __name__ の先頭の文字 “_” には「**非公開（private）**」であるという意味があり、特に名前の先頭に文字列 ‘__’ を持つメソッドや属性は隠蔽されるべきオブジェクトの名前を意味します。そこで、この文字 “_” の働きを確認しておきましょう：

```
>>> dir()
['__builtins__', '__doc__', '__name__', '__package__']
>>> class test:
...     x = 1
...     __y = 1
...     __z = 1
...
>>> dir()
['__builtins__', '__doc__', '__name__', '__package__', 'test']
>>> a1 = test()
>>> dir()
['__builtins__', '__doc__', '__name__', '__package__', 'a1', 'test']
>>> a1.x
1
>>> a1.__y
1
>>> a1.__z
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
AttributeError: test instance has no attribute '__z'  
>>> dir(test)  
['__doc__', '__module__', '_test__z', '_y', 'x']  
>>> a1._test__z  
1
```

この例ではクラス test の属性として x, _y と __z を定め、クラス test のインスタンスを a1 で属性値の確認を行います。クラス test を定義したことで函数 dir() が返すリストに test が現れ、それから test のインスタンスとして a1 を生成したことで函数 dir() の結果に a1 が追加されます。次に a1 の属性を参照しますが、ここで名前の先頭に文字“_”が二つある属性 a.__z だけ参照ができません。このように名前の先頭に文字“_”が二つある属性を隠蔽していますが、函数 dir() でクラス test の中にある名前を見ると属性 __z だけに文字の列 ‘_test’ が先頭に置かれています。つまり、文字の列 ‘_’ を名前の先頭に持つ属性は文字列 _(クラス名) が先頭に置かれた名前に置換されて本来の属性名で参照できなくても a1._test__z で本来の属性 __z の値の参照ができます。さらに他の属性と同様に、その属性値の書換も可能です。このように Python の属性の隠蔽の方法は詮索すれば容易にわかるという不徹底な方法です。なお、ここでは函数 dir() を用いましたが似た処理を行う組込の函数 vars() があり、こちらはリストではなく辞書と呼ばれる型で返すという違いがあります。このように Python のクラスの属性に非公開 (private) という概念がありません。ただ見えなくするだけでは属性値の保護は望めず、属性値の型に関する検証もはありません。そこで、属性値の管理を行う必要があるときは 2 系の古典的クラスではなくクラス object を継承し、「記述子規定 (descriptor protocol)」と呼ばれる属性値の設定に関わる 3 個のメソッド __set__(), __get__() と __delete__() の上書きで行います。つまり、これらのメソッドは属性の設定、取得、削除の「束縛動作 (binding behavior)」に関わるメソッドで、これらのメソッドの上書きで束縛動作での振舞いを利用者の目的に適ったものにします。

ところで隠蔽の仕組を有する構造体モドキが使えるというだけではクラスの有難味がありません。オブジェクト指向プログラミング言語の大きな有難味の一つは「継承」と呼ばれる機能、つまり、既存のクラスを土台に新しいクラスを効率的に構成できる機能です。この機能を活用することで既存の資産を新たなシステムの開発に生かせます。そこで、自然数を拡張して有理数を構築することで後利益を体験してみましょう。

3.2 有理数を構築してみよう

3.2.1 有理数の表現

ここでは有理数を定義してみましょう。まず、整数とその算術演算は定義されていますが、整数からいきなり有理数に到達できません。まず、我々は既存の整数と有理数の違いを明瞭にしなければなりません。そのためには有理数が「何であるか」、「どのようなものであるか」を明確にしておく必要があります。まず、有理数は二つの整数 n, m を用いて n/m の書式で表現される数です。そこで整数対のクラスを定義します。この整数対は有理数全体で一定の値ではないために、整数対を格納する属性はクラス変数ではなくインスタンス変数で、インスタンス化の時点で整数対が定まるべきです。また、この整数対のインスタンス名を入力すると ‘n/m’ と分かり易い書式で表示されるべきです。これらインスタンスの初期化や表示、さらには比較や演算を定めるメソッドに「**特殊メソッド**」と呼ばれるメソッドがあり、これらを上書きすることで上記の目的が達成できます。これら特殊メソッドの詳細は §3.8 で述べますが、ここでは分母と分子の値の設定をインスタンス化の時点で行うためにインスタンス属性の初期化を行う特殊メソッド `__init__()`、インスタンスが割当てられた名前が入力されたときに表示内容を定める特殊メソッド `__repr__()` の二つを用いて整数対のクラス `PairOfInts` を以下で定義します：

```
class PairOfInts:  
    def __init__(self, numer, denom):  
        self.numer = numer  
        self.denom = denom  
    def __repr__():  
        return '%s/%s' %(str(self.numer), str(self.denom))
```

特殊メソッド `__init__()` の第一引数の `self` は対象そのもので、この変数はメソッドの定義では必須です。ただし、実際に利用するときに `self` は不要です。そのうしろの引数 `numer` と `denom` が実際のインスタンスの生成で必要な引数、つまり、インスタンス変数に束縛すべき値です。ここで二つのインスタンス変数の `numer` が分子、`denom` が分母に対応して ‘`a = PairOfInts(1,2)`’ で対象を生成します、この `PairOfInts()` の働きから `PairOfInts()` のことを「**構築子/コンストラクタ**」と呼びます。構築子で生成され、変数 `a` に束縛されたオブジェクトに関しては、`a.numer` で属性 `numer` の値、`a.denom` で属性 `denom` の値の参照や代入が行えます。それから二番目に定義されているメソッド `__repr__()` がインスタンスの表示に関わるメソッドで、オブジェクトを指示する名前

が入力されたときや print 文^{*10}でどのような表示を行うべきかを定め、ここでは出力書式を文字列 '%s/%s' で指定して 'a = PairOfInts(1,2)' でインスタンスを生成して名前 a をインタプリタに入力すると文字列 '1/2' が表示されることを意味します。

このことを実際に試してみましょう。ここではカレントディレクトリ^{*11}上に PairOfInts クラスの定義をファイル PairOfInts.py に記載して import 文でインタプリタに読み込みます。ここで Python に読み込んだファイルは「モジュール」と呼ばれるオブジェクトになり、複数のモジュールを一つにまとめたものを「パッケージ」と呼びます。それから 'from PairOfInts import PairOfInts' でクラス PairOfInts() を直接読み込みます：

```
>>> from PairOfInts import PairOfInts
>>> a = PairOfInts(1,2)
>>> a
1/2
```

この例では PairOfInts クラスを import 文で読み込み、そのインスタンスを生成して名前 a に束縛させて名前 a を入力すると表示 '1/2' を得ます。では、メソッド __repr__() がなければどうなるでしょうか？

```
>>> class TEST:
....     def __init__(self, numer, denom):
....         self.denom=denom
....         self.numer=numer
.... 
>>>
>>> b = TEST(1,2)
>>> b
<__main__.TEST instance at 0x4f607a0>
>>> [b.numer, b.denom]
[1, 2]
>>> repr('%s/%s' % (b.numer, b.denom))
"'1/2'"
```

ここでは PairOfInts クラスからメソッド __repr__() を除いた TEST クラスをインタプリタ上で直接定義し、それから生成したオブジェクトを名前 b のインスタンスとして割当てています。ここで名前 b を直接入力するとメソッド __repr__() が定義されていないために '<__main__.TEST instance at 0x4f607a0>' と表示されます。ちなみに '0x4f607a0' という値は組込函数 id() が返却するオブジェクトの識別値と一致

^{*10} Python 3 では「**函数**」に変更されます。

^{*11} os モジュールの函数 getcwd() で確認できます。変更は os モジュールの函数 chdir() の引数に経路を与えるべきです。

することから、オブジェクトの先頭番地であることが判ります。このようにメソッド`__repr__()`等で表示内容を指示していない限り、名前を入力してもオブジェクトの番地が返されるだけです。またメソッド`__repr__()`が組込函数の函数`repr()`に対応し、メソッド`__repr__()`が定義されていなくてもインスタンスの属性とそれらの書式を函数`repr()`に引き渡せば同様の表示を得られます。このように特殊メソッド`__repr__()`を記述することでインスタンスの公式の表示が定められます。逆に言えば、二つのオブジェクトの表示の一一致からオブジェクトの同一性が保証できないことを意味します。なお、ここで行った上位クラスのメソッドを下位のクラスで新たに定義しなおすことを「**上書き (override)**」、メソッドの引数の型が異なるときは上書きではなく「**多重定義 (overload)**」と呼びます。

3.2.2 有理数のクラスの構築

「**有理数とは何なのか**」という問への回答には、二つの整数対が有理数として等しいと判断できるかという判断基準と二つの有理数の比較手段が必要です。最も原始的な判断基準は分母同士と分子同士が等しければ良いというのですが、これだけでは不十分です。実際、 $1/2 = 2/4$ を自然数の対 $(1, 2)$ と $(2, 4)$ で見ると、双方は異なっていますが、有理数としてこれらは一致しなければなりません。同様に整数対 $(-a, -b)$ と (a, b) は有理数としては等しくなければなりません。この「**等しい**」という関係を「**与えられた自然数の対 (a, b) と (c, d) が $'ad - bc = 0'$ を満すとき**」と定めます。これで「**等しい**」ことの判断基準が一つ定まりました。そして、有理数は一意に整数対で表現されるべきです。実際、 $(a, -b)$ と $(-a, b)$ が等しい自然数であるため、分母に相当する整数が常に正となるように符号を付け直し、0 と異なる整数 c に対して $(a*c, b*c)$ であれば c を除去して (a, b) とすべきです。つまり、`RationalNumber` クラスは自然数の対のクラス `PairOfInts` にこれらの正規化操作を加えたクラスであるべきです。では、等しくないときはどうでしょうか？ここで二つの有理数が与えられたとき何ができるでしょうか？整数が保持する関係には与えられた二つの整数が等しいか、それともどちらかが大きいかということ、すなわち、大小関係という関係があり、この関係は有理数にもあります。ここで有理数に対して正規化を行っていれば常に分母は 0 以外の正整数することができるために ‘ $ad > bc$ ’ と ‘ $(a, b) > (c, d)$ ’ が同値です。これらを踏まえて `RationalNumber` クラスを構築しましょう：

```
from PairOfInts import PairOfInts
class RationalNumber(PairOfInts):
    def __gcd__(self):
        __a = self.numer
        __b = self.denom
        if __a == 0:
```

```
        if __b != 0:
            __b = 1
        elif __b == 0:
            __a = 1
        elif __a > __b:
            __d = __a / __b
            __r = __a - __d * __b
        else:
            __c = self.conv()
        return c.__gdc__()

def __cmp__(self, other):
    """
    有理数の合同性と大小関係を判別するメソッド
    """
    return cmp(self.numer * other.denom,
               self.denom * other.numer)

def conv(self):
    """
    逆元を返すメソッド
    """
    tmp = self.numer
    if tmp == 0:
        self.numer = 1
        self.denom = 0
    else:
        self.numer = self.denom
        self.denom = tmp

def rexpr(self):
    """
    有理数の正規化を行うメソッド
    """
    if self.denom < 0:
        self.denom = - self.denom
```

```

        self.numer = - self.numer
if self.numer == 0:
    self.denom = 1
elif self.denom == 0:
    self.numer = 1
else:
    tmp = gcd(self.numer, self.denom)
    self.numer = self.numer / tmp
    self.denom = self.denom / tmp

```

class 節にて class 新しいクラスの名前(既存のクラスの名前) と既存のクラスを引数として与えることで既存のクラスを継承する新しいクラスが定義できます。この手法を「**継承**」と呼びます。この例では最初に import 文で PairOfInts クラスを読み込み、次に RationalNumber クラスの定義を ‘class RationalNumber(PairOfInts):’ で開始し、RationalNumber クラスが PairOfInts クラスを継承することを宣言しています。ここで RatinalNumber クラスに整数対の正規化を行うために二つの整数の最大公約数を求めるメソッド __gcd__(), 逆数を求めるメソッド conv() を定義している他に大小関係を整数から引継ぐメソッドとしてあらかじめ用意された特殊メソッド __cmp__() を先程の ‘a/b > c/d’ を定めるために上書きする形で用います。これらのメソッドを定義することで大小関係の二項演算子 “>”, “<” と等価 “==” がこのクラスでも使えるようになります*12。

この新しいクラスを定義するときに基になった PairOfInts クラスを「**基底クラス (base class)**」、継承する側の RationalNumber クラスを「**派生クラス (derived class)**」と呼びます。また、継承関係を親子関係にたとえ、基底クラスを「**親クラス**」、派生クラスを「**子クラス**」、さらに継承関係を上下関係として捉えるときには基底クラスを「**スーパークラス (super class)**」、派生クラスを「**サブクラス (subclass)**」と呼びます。クラスは処理の対象が「**何であるか**」ということと、「**どのようなものであるか**」を語るものであり、対象への理解が深まることでより詳細に分類され、その結果、下層のサブクラスは上位のクラスよりもより一層、現実の事物に近いために具象性が増し、逆に上位のクラス程、下位のクラスの共通性を引き出すものであるためにより抽象的(普遍的)になります。この特性からクラスの分析は段階を踏んで詳細に分析をしなければ、将来的な拡張性に問題が生じる可能性を孕みます。

*12 ここでの「等価」はオブジェクトの値に対して「等価」であるかどうかを判断するメソッドで、オブジェクトの「同一性」を判断するメソッドではありません。

3.2.3 特殊メソッドによる四則演算の導入

この RationalNumber クラスにまだ足りないものがあります。それは四則演算です。四則演算を次で追加しましょう：

```
def __add__(self, other):
    """
    有理数の和を定義するメソッド
    """
    numer = self.numer * other.denom + \
             self.denom * other.numer
    denom = self.denom * other.denom
    c = RationalNumber(numer, denom)
    c.reexpr()
    return

def __sub__(self, other):
    """
    有理数の差を定義するメソッド
    """
    numer = self.numer * other.denom - \
             self.denom * other.numer
    denom = self.denom * other.denom
    c = RationalNumber(numer, denom)
    c.reexpr()
    return

def __mul__(self, other):
    """
    有理数の積を定義するメソッド
    """
    numer = self.numer * other.numer
    denom = self.denom * other.denom
    return RationalNumber(numer, denom)

def __truediv__(self, other):
```

```

u"""
有理数の商を定義するメソッド
"""

numer = self.numer * other.denom
denom = self.denom * other.numer
return RationalNumber(numer, denom)

```

最初の特殊メソッド`__add__()`が和演算子“+”，メソッド`__sub__()`が差演算子“-”，それからメソッド`__mul__()`が積演算子“*”，最後のメソッド`__truediv__()`が商演算子“/”にそれぞれ対応するメソッドです。ところで，四則演算は同じクラスの二つのオブジェクトに対する二項演算で，四則演算を表現するメソッドの引数にオブジェクト自体を参照することを意味する変数`self`に加え，同じクラスのもう一つのインスタンスを参照することを示す変数`other`があることに注意して下さい。

このように有理数を整数対として表現して四則演算を入れましたが，ここで注意すべきことは，その代数的構造（分配律，結合率等）や順序関係についてはまだ何も語られていないことです。比較については四則演算のように比較の特殊メソッドの上書きで済むでしょう。ところが代数的構造はそう簡単に導入できません。特殊メソッドの定義から分かるように`self`と`other`の二項の演算だけです。つまり，‘1 + 2’は処理ができるても‘1 + 2 + 3’の処理はできません。少なくとも‘(1 + 2) + 3’か‘1 + (2 + 3)’としなければダメです。この‘(1 + 2) + 3 = 1 + (2 + 3)’を保証し，‘1 + 2 + 3’と書けることを保証する性質が「**結合律**」と呼ばれる代数的な性質です。SageMathには代数的構造があらかじめ定義されているため，表現しようとする数学的対象に最も類似するクラスを継承すればより効率的に表現できます。このことがSageMathを用いる大きな利点です。

3.3 バックス・ナウア記法(BNF)について

Pythonの構文の解説では「**BNF**」と呼ばれる表記方法が用いられます。このBNFはどのように記号や文字を使ってPythonのオブジェクトや構文が記述されるかを表記する方法です。Python言語の簡潔過ぎるBNFの一覧はCPythonのソースファイルのGrammer/Grammerに記載されていますが，この本では「Python言語リファレンス」の記載に従って解説します。なお，PythonのプログラムはPython処理系の構文解析器によってZephyr ASDL(Abstract Syntax Description Language)^{*13}を使った表現(ASTs, Abstract Syntax Trees)で置換えられます。ASDLの書式はCPythonのソースファイル

^{*13} <ftp://ftp.cs.princeton.edu/techreports/1997/554.pdf>。ちなみにZephyrは西風の神であるとともに春の神でもあるΖεψυροςに由来します。

ル Parser/Python.asdl に記載されていますが、この ASDL は引数の型宣言を含む函数表現で、構文の構成手順を記載する BNF 以上に簡素化された書式です。

ここで「**バッカス・ナウア記法 (Backus-Naur form, BNF)**」と呼ばれる表記を拡張した「**EBNF**(Extended BNF)」はジョン・バッカス (John Backus) がプログラム言語 ALGOL *14の文法の説明で用いた記法をピーター・ナウア (Peter Naur) が改良したもので、その構文規則は次で与えられます：

非終端記号 $::=$ 定義₁ | 定義₂ | … | 定義_n

この式は演算子 “ $::=$ ” 左辺の非終端記号が右辺にある ‘定義_{i ∈ {1, …, n}}’ の何れか一つの定義が採用されることを意味します。したがって、生成規則がひとつであれば右辺は ‘定義₁’ だけになり、記号 “|” が不要になり、右辺が ‘定義₁|定義₂’ であれば ‘定義₁’ でなければ ‘定義₂’ で左辺が指示される表記であることを意味します。では、構文規則の左辺の「**非終端記号**」とは何でしょうか？この「**非終端記号 (Nonterminal Symbol)**」は「**形式文法 (Formal Grammar)**」の言葉で、演算子 “ $::=$ ” の右辺の定義にしたがって変化が生じ得る記号です。この記号は変数と同様の動作を行う記号であるために「**構文変数**」とも呼ばれます。関連して、「**終端記号**」は演算子 “ $::=$ ” の左辺の定義として現われる生成文法に従ったときに、それ以上の変化が生じない定数に対応する記号です。この終端記号はその性質から BNF では演算子 “ $::=$ ” の右辺のみに現われます。なお、本来の BNF は非終端記号を「(非終端記号)」と記号 “⟨ ⟩” で括ってその区別を明瞭にしますが、ここではその表記を用いません。

ここで BNF の例として数字の構成を挙げておきましょう。まず、「**数字**」は ASCII 文字の “0” から “9” です。これを BNF で表記するなら

BNF による数字の定義

数字 $::=$ “0” | “1” | “2” | “3” | “4” | “5” | “6” | “7” | “8” | “9”

になります。ここで ASCII 文字の “0” から “9” が終端記号であることに問題はないでしょう。実際、“0” から “9” の数字は数字を構成するときの素材です。では自然数の BNF はどうなるでしょうか？この場合は複数の非終端記号の定義行を組合せた表現：

*14 ALGOOrithmic Language:1950 年代に開発された Pascal の先祖になる言語。

BNFによる自然数の定義

自然数 ::= 数字 0以外の数字	自然数
数字 ::= "0" 0以外の数字	
0以外の数字 ::= "1" "2" "3" "4" "5" "6" "7" "8" "9"	

と表記できます。まず、「**自然数**」は0から9までの「**数字**」か「**0以外の数字**」と「**自然数**」を左から順番に並べて構成されるものと第一行で記述され、この定義は演算子“::=”の左右に「**自然数**」が現われる「**帰納的な定義**」になります。ここで「**数字**」は第二行目で「**"0"**」または「**0以外の数字**」から構成され、最後の行で「**0以外の数字**」は“1”, “2”から“9”までの数字と記述されます。と、この記述は「**自然数が何であるか?**」という間に、その意味を答えるものではありませんが、文字をどのように使って構築されたものであるかを示す記述であり、この構成手順を明確化したことで自然数であることの判断で機械的な処理が行えることを意味します。

このBNFは字句的な定義だけではなく構文の定義もできます。たとえば、論理学の「**命題論理式の定義**」は次のものです：

命題論理式の定義

- (1) 真理値の真 \top は命題論理式である。
- (2) 真理値の偽 \perp は命題論理式である。
- (3) 論理記号 P は命題論理式である。
- (4) A, B が命題論理式であれば $\neg A, A \wedge B, A \vee B, A \rightarrow B$ も命題論理式である。
- (5) 上の (1), (2), (3), (4) で構成されたもののみが命題論理式である。

この定義(4)の‘ $\neg A$ ’は命題論理式‘ A ’の否定, ‘ $A \wedge B$ ’は命題論理式‘ A ’と‘ B ’の論理積を取る操作, ‘ $A \vee B$ ’は命題論理式‘ A ’と‘ B ’の論理和を取る操作で、最後の‘ $A \rightarrow B$ ’は論理式の含意(A ならば B)を取る操作に対応します。ここで(5)に帰納的な定義(既存の要素を使って新たな命題を生成する手順)が入っていることに注目して下さい。では、この命題論理式の定義をBNFで書換えてみましょう：

命題論理式のBNF

命題論理式 ::= \top \perp 論理記号 \neg 命題論理式	
命題論理式 \wedge 命題論理式 命題論理式 \vee 命題論理式	
命題論理式 \rightarrow 命題論理式	

このBNFで、記号“::=”の右辺から順番に命題論理式の(1)から(4)が現われています。

そして、命題論理式の定義の(5)の帰納的な論理式の定義は BNF の構造で表現されています。このような「文の成り立ち」の表現にも BNF は使えます。

ところで、「Python 言語マニュアル」の BNF は、上述の BNF に対して正規表現を追加した「**拡張 BNF(EBNF)**」を採用しています。以下に拡張の要旨を纏めておきます：

EBNF の特徴

- 項目のグループ化は丸括弧“()”で行います。
- 文字リテラル(後述)は二重引用符(" ")で括ります。
- 角括弧 “[]” で括られた項目は 0 個か 1 個出現します。
- 順序を持つアルファベットや数字に対して “...” の直前の項目から開始して直後の項目のいずれか一つが現われます。
- 記号 “*” の直前の項目は 1 個以上出現します。
- 記号 “+” の直前の項目は 0 個以上出現します。

ここでの拡張は正規表現で見られる事項のグループ化や出現回数に関する指示とリテラル(=Python の文字列)の処理に関わります。以下に EBNF の具体例を幾つか示しておきます：

整数の EBNF

整数 ::= ["-"] 自然数

これは角括弧 “[]” を使った例で、角括弧 “[]” はあってもなくてもよい部位を角括弧 “[]” で括って表記しています。また、自然数が定義されているという暗黙の仮定がありますが、この EBNF は整数は自然数、あるいは自然数の頭に記号 “-” を追加したものという字句的な定義です。こうすることで整数の表記上の定義がより簡潔で明晰なものになります。

もう一つ、今度は「Python 言語リファレンス」にある例を示しておきます。なお、ここで定義している name は Python で重要な意味を持つ「名前(name)」ではなく、単に ‘name’ というものの EBNF です：

name の EBNF

name ::= lc_letter(lc_letter | "_")*

lc_letter ::= "a" ... "z"

Python の EBNF では文字を二重引用符 (" ") で括るために文字を “_”, “a”, “z” と二重

引用符で括った表記にします。一行目で name が lc_letter と記号 “_” で構成されることを示します。ここで ‘(lc_letter | "_")*’ は正規表現で ‘(...)*’ と括弧を使ってグループ化を行い、括弧内の表記が 0 回以上出現するという意味を持たせています。このことから name は lc_letter に対応する文字の列が必ず先頭に現われ、そのうしろに lc_letter か文字 “_” で構成された文字の列が続くことを意味します。では、lc_letter は何でしょうか？この定義が二行目で、lc_letter が文字 "a" から "z" までの小文字で構成されることを記号 “...” を用いて表記しています。ここでの表記は正規表現から外れた表記で、通常の正規表現であれば ‘a-z’ となるところですが、Python の EBNF で記号 “...” が正規表現の “_” に対応します。この BNF から name は ‘a’, ‘a_bc’ のように頭文字がアルファベット小文字、以後はアルファベット小文字や記号 “_” で構成された文字列であり、‘_a’ のように先頭がアルファベットでない文字列、‘a1’ や ‘A_v0.1’ のように BNF に記述のない文字が入った文字列は name に適合しません。このように EBNF を利用することで具体的にどのような記述がされ得るかより明瞭に表記できます。

3.4 Python の字句解析について

3.4.1 トークン (token) について

入力されたプログラムは Python 処理系の構文解析器 (parser) による字句解析の結果、トークンの列に変換されます。ここで「**トークン (token)**」は、自然言語の「**語彙素**」に相当し、プログラム内で意味を持つ最小単位で、「**NEWLINE**」、「**INDENT**」と「**DEDENT**」が行と文の構造に関わるトークンです。まず、NEWLINE は論理行の区切になるトークンの INDENT, DEDENT は INDENT と DEDENT のトークン対で §3.6 で説明する「**複合文 (compound statement)**」で用いられ、Python のコードの大きな特徴である字下げに関係します。この他のトークンは「**識別子 (identifier)**」、「**キーワード (keyword)**」、「**リテラル (literal)**」、「**演算子 (operator)**」と「**区切文字 (delimiter)**」に分類されます^{*15}。また、「**空白文字 (blank character)**」^{*16}とよばれる文字の中で「**欧文間隔 (Space)**」、「**水平タブ (TAB)**」と「**改ページ (FF, Form Feed)**」にはトークンを区切る作用があります。たとえば ‘ab’ と文字の間に Space を入れた ‘a b’ は前者が ‘ab’ の一つのトークン、後者が ‘a’ と ‘b’ の二つのトークンです。

^{*15} 演算子と区切文字は Parser/tokenizer.c にそれらの名前が定義されています。

^{*16} 空白文字と呼ばれる ASCII 文字には水平タブ (TAB), 垂直タブ (VT), 改行 (LF/NL), 改ページ (FF), 行頭復帰 (CR), 欧文間隔 (Space) の 6 文字あり、Python では文字列クラスのメソッド isspace() で空白文字かどうかが判断できます。なお、各言語の間隔文字は空白文字と区別します。

3.4.2 行構造について

トークンの列は「**NEWLINE**」を区切として複数の「**論理行**」に、論理行は「**物理行**」へと分解されます：

■**論理行 (logical line)**: 入力されたプログラムを分割したもので、先頭に「**字下げ/インデント (indentation)**」と呼ばれる ASCII 文字の **Space** や **TAB** による空白文字の列の末端に「**NEWLINE**」を持つトークンの列です。

■**物理行 (physical line)**: 論理行を行末端文字で分割した文字の列です。行末端文字は計算機環境で異なり、UNIX 環境では ASCII 文字の **LF(行送り)**、Windows 環境では **CR+LF**、MacOS では「**CR(復帰)**」ですが、Python のプログラムでは計算機環境を問わず C 同様に行末端文字として ASCII 文字 “**LF**” に対応するエスケープシーケン ‘\n’ *17 を用います。また、特殊な物理行に「**注釈**」、「**符号化宣言**」と「**空行**」があります：

- **注釈 (comment)**: 記号 “#” で開始して物理行の行末端文字を末端に持つ論理行です*18。なお、注釈は論理行を終らせる働きがあります。そのために後述の行継続で注意が必要です。
- **符号化宣言 (エンコーディング. encoding)**: 符号化宣言は、プログラムで用いる文字が属する文字コードを明示的に示すための宣言です。既定値の文字コードは Python 2 で ‘ascii’、Python 3 では ‘utf_8’ です。PEP-263 にその規定があり、注釈と同様に文字 “#” から開始して物理行の行末端文字で終える論理行とされ、プログラムの先頭の一行目か二行目に配置されて、次の正規表現:

————— 符号化宣言の正規表現 —————

coding[=:]\s*([-w.]+)

に適合します。たとえばプログラムの文字コードが UTF-8 であれば

```
# coding = UTF-8
```

あるいは

```
# coding : UTF-8
```

*17 エスケープシーケンスの詳細は 3.4.4 を参照。なお、日本語 Windows で用いられている文字コード: SHIFT_JIS(その実装の CP932) で記号 “\” が記号 “¥” で置換えられ、多くの書籍でこれらの記号を同一視していますが、UTF-8 等の文字コードで別記号のため、この本では記号 “\” を記号 “¥” で置換えません。したがって、日本語 MWWindows 環境では適宜、記号 “\” を記号 “¥” で読み直して下さい。

*18 記号 “#” は Python のリテラルに含まれません。

といった宣言行にします。ちなみに前者が GNU Emacs の符号化設定書式、後者が vim の符号化設定書式に適合します^{*19}。この符号化宣言でプログラムで用いられる文字リテラルを構成する文字がどの言語のどの符号化であるかが明示的に指定され、この宣言がなければプログラムに記載された文字は ASCII 文字として扱われます。なお、ここでの宣言は大域的なもので文字列単位でも文字コードが指定できます。

- **空行 (blank line):** Space, TAB, FF といった空白文字、あるいは注釈だけで構成された論理行です。これらの空行に字句解析で NEWLINE が生成されません。そして、これらの空行はプログラムの内容的な区切であっても、それ以上の意味は持ちません。

■**字下げ/インデント (indentation):** 論理行の先頭に配置された Space, TAB キーの個数で字下げの水準が計算され、その水準で入力文が纏められます。Python のプログラミング様式を規定する PEP-8 で字下げは一段 4 個の Space のみが推奨されています。実際、TAB キーと Space の判別は目視では困難で、それらが一貫した順序で並んでいないと構文解析器から例外:TabError が送出されます。また、Jupyter 等のノートブック形式のユーザ・インターフェイスで TAB キーに入力補完機能を持たせおり、このことからも Space のみが実用的です。

■**物理行の明示的/非明示的な分割:** 注釈と符号化宣言を除く物理行を複数の物理行に置換できます。明示的に物理行分割するときは行の継続を示す継続文字として記号 “\” を物理行の末尾 (行末端文字の直前) に置きます。ここで Python の構文解析器は継続文字直後の行末文字を削除して一つの物理行に変換しますが、継続文字 “\” に続けて註釈が追記できません。なぜなら、註釈で論理行が終わるために、註釈自体も継続文字を使って分割ができません。ここで改行の例外的な規則として丸括弧“()”，角括弧 “[]”，波括弧 “[{ }]” 内部で改行を行う際に継続行文字 “\” の併用を必要としません。同時にこれらの記号で括られたその内部で註釈を続けられます。

3.4.3 識別子とキーワードについて

「**識別子 (identifier)**」は「**名前 (name)**」に用いられ、名前を介してオブジェクト参照が行われます：

^{*19} vim は vi から派生したエディタで、GNU Emacs と vi はいわゆる Editor War の二大陣営です。

——識別子の EBNF ——

```

識別子 ::= (文字 | "_")(文字 | 数字 | "_")*
文字    ::= 小文字 | 大文字
小文字  ::= "a" ... "z"
大文字  ::= "A" ... "Z"
数字    ::= "0" ... "9"

```

識別子はアルファベット、ト数字と記号“_”のみで構成された ASCII 文字の列で、日本語の漢字“**三毛猫**”やいわゆる全角文字の“**A B C**”といった ASCII 文字以外の文字は上述の識別子の EBNF の文字に該当しないため、これらの文字を含む文字の列は識別子になりません^{*20}。ただし、以下の「**キーワード**」は式や文の構成要素であるために識別子として使えません：

——キーワードの一覧——

and	class	elif	finally	if	lambda	print	while
as	continue	else	for	import	not	raise	with
assert	def	except	from	in	or	return	yield
break	del	execr	global	is	pass	try	

識別子から上記のキーワードを除いたものが「**名前**」として使え、オブジェクトの参照は名前を介して行われます。そして、名前とオブジェクトとの対応付けが「**名前空間**」と呼ばれる仕組で、Python の連想配列である「**辞書 (dictionary)**」で実装されています。ここで何らのオブジェクトと対応関係がない名前でオブジェクトの参照が行われると例外:NameError が送出されます。

3.4.4 リテラルについて

「リテラル (literal)」は「**文字どおり**」、「**字義どおり**」を意味し、記号論理学では命題記号(原子論理式)や命題記号の否定といった論理式を構成する上で根本になる要素、プログラミングではコード内部で定数値となる文字列や数値といった値の記述を指します。Python のリテラルは「**文字列リテラル**」と「**数リテラル**」の二種類に分類できます：

——リテラルの分類——

```
リテラル ::= 文字列リテラル | 数リテラル
```

^{*20} Python 3 で文字コードとして UNICODE が採用された結果、漢字、キリル文字等の非 ASCII 文字も識別子として使えます。詳細は「PEP-3131 Supporting Non-ASCII Identifiers」を参照。

Python のリテラルは全て「**変更不能なデータ型**」で、オブジェクトの生成後に値を変更できません。

文字列リテラル

Python の「文字列リテラル」の EBNF を以下に示します:

文字列リテラルの EBNF

文字列リテラル	::= [接頭辞](短文字列 長文字列)
接頭辞	::= "r" "u" "ur" "R" "U" "Ur" "uR" "b" "B" "br" "Br" "bR" "BR"
短文字列	::= " , " 短文字列本文* " , " " " " 短文字列本文* " " "
長文字列	::= " , , " 長文字列本文* " , , " " , , , " 長文字列本文* " , , , "
短文字列本文	::= 短文字 エスケープシーケンス
長文字列本文	::= 長文字 エスケープシーケンス
短文字	::= 記号 "\\", 改行や引用符を除く文字
長文字	::= 記号 "\\" を除く文字
エスケープシーケンス	::= "\\" 任意の ASCII 文字

文字列リテラルは言語の文字列を拡張したものに相当し、文字列のエンコーディングを明示する接頭辞を配置できます。また、プログラムに文字のエンコーディング宣言が含まれざれば、リテラル自体にエンコーディングを示す接頭辞がなければリテラルを構成する文字は ASCII 文字として解釈され、接頭辞があればリテラルを構成する文字は、その接頭辞で指示されたエンコーディングの文字として処理されます。たとえば、文字列の先頭に “u” や “U” といった接頭辞で文字列リテラルが UNICODE 文字列型として扱われます。ただし、接頭辞と後続する文字列リテラルの間に空白文字を入れてはいけません。

Python の文字列には引用符の個数による型の区別があります。引用符に单引用符 (') と二重引用符 ("") があり、文字列はこれらの引用符の対で括られたオブジェクトです。ここで引用符が二種類存在することから "abc 'def' hij" のように文字列リテラル内部に別の引用符を用いた文字列リテラルを配置できます。そして、引用符で文字の列を一重で括るか、三重に括るかで異なります。まず、「**短文字列 (shortstring)**」は一重の引用符で括られた文字リテラルの型、「**長文字列 (longstring)**」は三重の引用符で括られた文字リテラルの型です。具体的には「' 三毛猫'」や「"虎猫"」が短文字列の例、「''' 三毛猫'''」と「"""虎猫"""」が長文字列の例です。なお、長文字列からは後述の「文書文字

列 (docstring)」が構成されますが、こちらはオンラインマニュアルとしての性格を持ち、文書文字列の中で改行や单引用符 (') や二重引用符 ("") が入れられるために、例題等を含む長い文書の記述ができます。ここで長文字列の先頭の引用符と同じ引用符を三回連続して配置すると長文字列が終了することに注意が必要です:

```
"""
だから

"こんな使い方"
や
‘こんな使い方’
それに改行をこんな風に複数入れたり

'''
引用符も長文字列を構成する際に用いた
引用符でなければ続けて三回使っても構いません'''

といったことが記入できます.
"""
```

と、单引用符や二重引用符、それと行末端文字を含む文字リテラルを一つの長文字列にします。なお、プログラムで改行の出力を表現するときは C 同様に出力したい文字を表現する文字列リテラルで改行を入れたい箇所に文字列 “\n”(正確にはエスケープシーケンスの改行 (LF)) を挿入して出力で文字列の改行ができます。この長文字列は MATLAB の m-file で函数のヘルプや例題の記述で用いられる文書文字列 (docstring) を拡張・強化したもので、長文字列を reStructuredText の書式で記述することで、プログラムの文書としての表現力が格段に向上します。

それから「エスケープシーケンス (Escape sequence)」は何らかの機能を表現するための文字の列です:

エスケープシーケンス

\\"	文字 “\”
\'	文字 (‘)
\"	文字 (“”)
\a	ベル (BEL)
\b	バックスペース (BS)
\f	改ページ (フォームフィード, FF)
\r	復帰 (CR)
\n	改行 (LF)
\t	水平タブ (HT)
\v	垂直タブ (VT)
\ooo	8進数 ooo に対応する ASCII 文字 (o は 0-7)
\xhh	16進数 hh に対応する ASCII 文字 (h は 0-f)
\0	NULL
\N{name}	name を Unicode 文字名とする Unicode 文字
\xxxxx	16ビットの 16進数値 xxxx を持つ Unicode 文字
\xxxxxxxx	32ビットの 16進数値xxxxxxxx を持つ Unicode 文字

ここで「UNICODE 文字名」は「Unicode Character Database」^{*21}に ASCII 文字で記載された Unicode 文字の名前です。たとえば文字 “[” の UNICODE 符号点は 005B, UNICODE 文字名は ‘LEFT SQUARE BRACKET’ です。なお、接頭辞 ‘r’, ‘R’ に続く文字列に含まれる記号 “\” はエスケープシーケンスの一部ではなく、通常の ASCII 文字として処理することを意味します。

数リテラル

数リテラルの EBNF を以下に示します:

数リテラルの EBNF

```
数リテラル ::= 整数リテラル | 長整数リテラル | 浮動小数点数リテラル | 虚  
数リテラル
```

Python 2 の数リテラルには「整数 (plain integer) リテラル」、「長整数 (long integer) リテラル」、「浮動小数点数リテラル」と「虚数リテラル」の 4 種類があり、整数リテラルと長整数リテラルが整数の表現、整数リテラルが符号付き 32bit 整数^{*22}、長整数は計算機

^{*21} <http://www.unicode.org/Public/UCD/latest/charts/CodeCharts.pdf> を参照。

^{*22} ±214743647 の範囲の数で、Python 2 で変数 sys.maxint に整数型の上限としてこの値が束縛されています。

の記憶容量に依存するものの任意桁数の整数を表現します^{*23}. そして, 浮動小数点数リテラルが実数の近似になります. ところで, 数学で重要な数に複素数がありますが, Python の複素数は浮動小数点数リテラルと虚数リテラルの和という式になるために数リテラルに該当しません.

以下に整数リテラルと長整数リテラルの EBNF を示します:

—— 整数リテラルと長整数リテラルの EBNF ——

長整数リテラル	$::=$	整数リテラル ("I" "L")
整数リテラル	$::=$	10 進表示 8 進数表示 16 進数表示 2 進数表示
10 進表示	$::=$	零以外の数字 数字* "0"
8 進数表示	$::=$	"0" ("o" "O") 8 進数 + "0" 8 進数 +
16 進数表示	$::=$	"0" ("x" "X") 16 進数 +
2 進数表示	$::=$	"0" ("b" "B") 2 進数 +
数字	$::=$	"0" 零以外の数字
零以外の数字	$::=$	"1" ... "9"
8 進数	$::=$	"0" ... "7"
16 進数	$::=$	数字 "a" ... "f" "A" ... "F"
2 進数	$::=$	"0" "1"

Python 2 の整数リテラルには 10 進数の他に 2 進数, 8 進数と 16 進数があり, 長整数のリテラルは末尾に “T” や “L” といった文字を配置しますが, 小文字 “t” は数字 “1” と紛らわしのために大文字 “L” の使用が推奨されています. なお, Python 3 では整数リテラルが長整数型のみに統一され,さらに SageMath では数オブジェクトを定義しなおすために, これらでは長整数末尾の “L” は不要です^{*24}. Python では整数型で処理していて整数型の最大値を超過したときは自動的に長整数型へ, 逆に長整数型で処理していて絶対値が整数型の最大値以下になったときに自動的に整数型へと変換されます.

次に浮動小数点数リテラルの EBNF を示します:

ますが, Python 3 では整数型が実質的に長整数型で統一されるために sys.maxint は廃止されています.

*23 整数值が計算機環境で処理できなくなるほど大きくなったときに例外:MemoryError が送出されます.
後述のシフト演算で容易に出せます.

*24 SageMath で Python2 の長整数リテラルを使う必要が生じたときのみ必要です.

浮動小数点数リテラルの EBNF

```

浮動小数点数リテラル ::= 小数表 | 指数表示
小数表示          ::= [10進表示] "." 10進表示 | 10進表示 "."
指数表示          ::= (10表示 | 小数表示) 指数部
指数部           ::= ("e" | "E") ["+" | "-"] 10進表示 +

```

浮動小数点数リテラルは符号を含みません。なぜなら符号を含むリテラルは-1との積演算の結果、すなわち式であるためで、この点は複素数も同様です。多くの言語では単精度と倍精度の二つの浮動小数点数の型を持ちますが、Pythonでは倍精度浮動小数点数の float 型のみです。また、特殊な数の表現として、Pythonには「無限大（'inf'）」と「非数（'nan'）」があります^{*25}。実際に利用するためには無限大であれば float('inf')、非数であれば float('NaN')、あるいは float('nan') でこれらのオブジェクトが生成できます。当然、これらの型は float 型です。

最後に虚数リテラルの EBNF を示しておきます：

虚数リテラルの EBNF

```

虚数リテラル ::= (浮動小数点数リテラル | 10進表示) ("j" | "J")

```

虚数リテラルは「浮動小数点数から構成されたリテラル」で、このことが虚数リテラルの性格を決定付けます。

3.4.5 演算子と区切文字について

以下のトークンは Python 組込の二項演算子として用いられます：

Python 組込の演算子

```

+ - * ** / // % << >> & | ^ ~
< > <= >= == != <>

```

ここで算術演算子は数リテラル、論理演算子は Boolean を返す演算子です。なお、演算子“<>”と演算子“!=”は同じ意味ですが、演算子“<>”は時代遅れの表記で、演算子“!=”の利用が推奨されています。また、演算子“&”と演算子“|”は Boolean に対しては論理積“and”，論理和“or”と同じ意味ですが、引数が整数型、あるいは長整数型のときに演算子“&”と演算子“|”は2進数表示したときのビット単位の積演算と和演算を行った結果を整数型、あるいは長整数型で返します。それに対して演算子“and”と“or”は双方の二進数

^{*25} inf, NaN の規定は IEEE 754 にあります。ここでの nan は qNaN が対応します。

表示の桁数が一致したときにビット単位の演算, そうでなければ演算子 “and” は右被演算子, 演算子 “or” は左被演算子を演算結果として返します. さらに演算子 “^” は排他的論理和 (XOR) の演算子です.

次のトークンは「区切文字 (delimiter)」として用いられます:

Python の区切文字 (delimiter)

括弧:	()	[]	{	}	@
区切記号:	,	:	.	'	=	;	
累算算術代入演算記号 (1):	+=	-=	*=	/=	//=	%=	
累算算術代入演算記号 (2):	&=	=	^=	>>=	<<=	**=	

上段は括弧の類で, 中段のコンマ “,”, コロン “:” は後述のスライス処理と呼ばれる配列の添字処理で使われます. そして, 下段の累算算術代入演算子は区切文字としても振舞います. また, 単引用符, 二重引用符, 記号 “#” と記号 “\” といった ASCII 文字は他のトークンの一部に用いられて特殊な意味を持ちます. 最後に “\$” と “?” は Python では用いられず, 文字リテラルと注釈以外に現われたときは無条件にエラーになりますが, IPython や Jupyter を Python のシェルとして使うときは記号 “?” に函数 help() を拡張した演算子としての働きを持たせているためにエラーになりません²⁶.

3.5 Python の式と文

3.5.1 式, 単純文と複合文について

「式 (expression)」は, 言語で何らかの意味を持つ最小の単位です. たとえば, ‘1’, ‘2’, ‘3’ のような数値, ‘True’, ‘False’ のような真理値, ‘1 + 2’, ‘math.sin(10)’ のような数式, ‘(1, 2, 3)’ や ‘[‘a’, ‘b’, ‘c’]’ のようなタプルやリスト, ‘{1, 2, 3}’ のような集合, さらには文字列等, プログラム言語で文を構成するための, いわば建物の煉瓦にあたる原始的な要素です. この式に対して「文 (statement)」は処理の手続を表現します. 文はプログラム言語によって大きく異なりますが, では文の構造から「単純文 (simple statement)」と「複合文 (compound statement)」に分類されます.

■単純文: 「一つの論理行に収めることのできる文」で, 例としては名前への拘束を行う代入文, モジュールの読み込む import 文が挙げられます. また, ‘x = 1’ のような変数にオブジェクトの束縛を行う代入式も「単純文」です.

²⁶ “??” でソースファイルを表示させる機能も追加されています.

■複合文: 「複数の論理行を必要とする文」で、複数の文節とその文節に対応する文の区画(ブロック)を持ちます。具体的には条件分岐や反復処理、関数やクラスの定義といったものが相当します。この複合文は字下げによる階層構造を持つことがあります。

3.5.2 式のEBNF

最初に式と式のリストのEBNFを示します:

式(expression) の EBNF

式のリスト	$::=$	式 ("," 式)* [","]
式	$::=$	一次語 演算式 if 式 lambda 式

■式のリスト: 区切文字がカンマ“,”の式の列の書式です。成分が2個以上の式のリストからはタプル型のオブジェクトが生成されます:

```
>>> True, True
(True, True)
>>> True, True,
(True, True)
>>> a=True, True,
>>> a
(True, True)
>>> type(a)
<type 'tuple'>
```

ここで示すようにタプル型のオブジェクトの生成のために丸括弧“()”は必須ではありません。この括弧“()”が必要になるときは空のタプルを生成するときとタプルをひとまとめにして全体を処理するときです。

■式(expression): 処理系での評価値としてのオブジェクトを返却するものが式です。Pythonの式はBNFに示すように「一次語」、「演算式」、「if式」と「lambda式」に大きくまとめられます。ここで一次語が最も原始的な式で、演算式は算術演算式や論理演算式、および比較演算式があります。そして、if式はif節を使った式の出力を伴い、lambda式は無名関数の構築で用いられます。

3.5.3 一次語(primary)

名前、属性参照、添字表記、呼出等とPython言語で基本になる表記です:

一次語の EBNF

```
一次語 ::= 原子要素 | 属性参照 | 添字表記 | スライス表記 | 呼出
```

一次語は Python の式の基本単位の原子要素、オブジェクトの名前、その名前から参照されるオブジェクトの属性への参照、またはオブジェクトが配列や辞書であればそれらの添字に対する値の参照で用いられる表記です。

原子要素 (atom)

Python の式を構成する基本単位で、単体で意味や機能を持ち得ます：

原子要素 (atom)

```
原子要素 ::= シングルトン | 識別子 | リテラル | 閉包
シングルトン ::= True | False | None
閉包 ::= 丸括弧形式 | リスト表現 | 集合表現 | 辞書表現
          | 文字列変換 | 生成式 | 産出原子式
```

「原子要素」は「シングルトン (singleton)」^{*27}、「識別子 (identifier)」、「リテラル (literal)」と「閉包 (enclosure)」の四種類に分類されます。まず、シングルトンは他と違つて固有の意味(値)と型を持ちます。識別子は「名前 (Name)」に用いられる文字に制限の入ったリテラルです。リテラルには「数リテラル」と「文字列リテラル」の二種類があり、具体的なデータを構成します。これらシングルトン、識別子とリテラルが最も基本的な原子要素です。つぎに「閉包」には「丸括弧式」、「リスト表現」、「辞書表現」、「集合表現」、「文字列変換」、「生成式」と「産出原子式」の七種類に分類されます。

■シングルトン (singleton): 識別子やリテラルと異なり固有の型と意味を持ちます。Python の真理値集合 Ω の元で真であることを意味する True、偽であることを意味する False、さらに何もないことを意味する None の 3 つがあり、これらの意味に加えて型を持ちます。まず、真理値集合 Ω の型は Boolean であり、None の型は None 型です。また、True には整数の 1、False には整数の 0 が整数値として付与されていますが、None には整数値が付与されていません。

■識別子 (identifier): 「名前 (Name)」に用いられ、その名前の参照で名前に束縛されたオブジェクトの値が返却され、名前にオブジェクトが束縛されていないときは例外:NameError が送出されます。また、クラスの属性名としての識別子で、その先頭に二つ

*27 「Python 言語リファレンス」にはシングルトンがありません。これはリファレンスの EBNF が「構文の構築」から記述されているためでしょう。しかし、この「シングルトン」という言葉が Python.asdl に含まれていること、さらにシングルトン単体で意味と型を持ち、通常のリテラルと意味合いが異なることから、ここでは原子要素に追加しています。

以上の記号“`_`”があり、末尾に二つ以上の記号“`_`”がないものは隠蔽されるべき名前とみなされ、名前空間内で別の名前に変換されます。「言語リファレンス」の例では、クラス名を `Hom`、属性名を ‘`__spam`’ のときに名前は ‘`__Hom__spam`’ に変換されます。直接、‘`__spam`’ で参照ができないという方法で属性の隠蔽が行われます。

■リテラル (literal): 文字列リテラルと数リテラルから構成されます:

リテラルの EBNF

```
リテラル ::= 文字列リテラル | 整数リテラル | 長整数リテラル | 浮動小数  
          | 点数リテラル | 虚数リテラル
```

Python 3 では長整数型に整数型が統合され、そのリテラルは Python 2 の整数リテラルです。

■閉包 (enclosure): Python のデータ表現で、識別子とリテラルと異なって書式を持つ原子要素です。ここで丸括弧形式、リスト表現、集合表現、辞書表現と文字列変換は特定の記号で囲むことで得られ、これらの書式が外延表現であるためにオブジェクトとして生成した時点で成分も定まり、生成式と産出式から生成されるオブジェクトの成分はメソッド `next()` の呼出で都度生成されます。なお、各閉包の生成で用いられる内包表現については §3.5.3 を参照してください。

■丸括弧形式: 一般的にタプルの構成で用いられる書式です:

丸括弧形式の EBNF

```
丸括弧形式 ::= "(" [式のリスト] ")"
```

括弧“()”はタプルの領域を明確にするために用いられますが、タプル型のオブジェクトの構築で必須のものであるとは限りません。実際、空集合 `Ø` を表現する空のタプルの書式は ‘`()`’ と丸括弧が必須ですが、空の式のリスト以外であれば丸括弧は不要です。たとえば ‘`1,2,3`’ でも ‘`(1, 2, 3)`’ でも同じタプルが生成されます。Python のタプルは後述のリスト型に類似していますが、リスト型と違って値が変更不可能なオブジェクトで、その生成も内包表現が使えずに式のリストから構築する方法しかありません。ちなみに内包表現を丸括弧“()”で括った書式は生成式の書式で、生成されるオブジェクトの型はジェネレータ型です。

■リスト表現: リスト型のオブジェクトを生成する表現で、タプルに似ていますがその生成で角括弧 “[]” は必須です:

リスト表現の EBNF

```
リスト表現 ::= "[" [式のリスト | 内包表現] "]"
```

リスト表現は「**式のリスト**」か「**内包表現**」を角括弧 “[]” で括った書式に分類され、前者の式のリストが具体的な成分の表記、内包表現では for 節が用いられます。リスト表現は集合と異なり空のリスト ‘[]’ を許容します。

■**集合表現**: 集合型のオブジェクトの生成で用いられ、波括弧 “{ }” が必須です:

集合表現の EBNF

```
集合表現 ::= "{" (式のリスト | 内包表現) "}"
```

式の評価は式のリストの左側から行われます。なお、空の式のリストや空集合 \emptyset を返す内包表現は集合表現では扱えません。実際、リテラル ‘{ }’ は集合型ではなく辞書型のオブジェクトで、空集合 \emptyset は空のタプル ‘()’、あるいは空のリスト ‘[]’ で表現します。

■**辞書表現**: 辞書型のオブジェクトの生成で用いられ、波括弧 “{ }” は必須です:

辞書表現の EBNF

```
辞書表現 ::= "{" 鍵データリスト | 辞書内包表現 "}"
鍵データリスト ::= 鍵データ ("," 鍵データ)*
鍵データ ::= 式 ":" 式
辞書内包表現 ::= 式 ":" 内包表現
```

辞書データは一般化されたリスト表現としての特性を持ちます。実際、リスト表現の添字が自然数に限定されますが、辞書データは一般的な式を添字を持つことができます。この添字になる式は鍵データの EBNF の左側の式です:

```
>>> a = {"Niko":1.95, "Mike":4.5}
>>> a["Niko"]
1.95
```

辞書型のオブジェクトの参照はリスト型のオブジェクトの参照と同様の書式になります。ただし、添字が鍵データで指示した式になる点がリスト型と異なります。

■**文字列変換**: 式のリストから得られるタプル全体を評価し、文字列型に変換します:

文字列変換の EBNF

```
文字列変換 ::= "()" 式のリスト "()"
```

ここで「式のリスト」として「空白文字」は許容されません:

```
>>> a =
File "<stdin>", line 1
    a =
      ^
SyntaxError: invalid syntax
```

■**生成式:** ジェネレータ(generator)型のオブジェクトを生成する式で、タプルのように丸括弧“()”を用い、この括弧“()”は必須です：

生成式の EBNF

生成式 ::= "(" 内包表現 ")"

リスト型の内包表現では内包表現をすべてを評価してオブジェクトが生成されますが、生成式からはジェネレータ型のオブジェクトが生成され、メソッド next() の呼出の都度、for 節が評価され、その for 節が返す式の値が返却されます：

```
>>> a = (i for i in range(0,5) if (lambda x:math.sin(2*x)>0)(i))
>>> a.next()
1
>>> a.next()
4
>>> a.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
```

生成式では内部表現の for 節に記載した標的リストに代入された値がメソッド next() を呼び出すことで送り返されます。

■**産出原子式:** 産出函数の生成で用いられる式で、函数内部のみで利用されます：

産出原子式の EBNF

産出原子式 ::= "(" 産出式 ")"
 産出式 ::= "yield' [式リスト]

yield 文を括弧“()”で括った書式になります。yield 文を持つ函数からジェネレータ型のオブジェクトが生成されます。ただし、yield 文と return 文を函数内部に混在させて使うことができません。ジェネレータ型のオブジェクトはメソッド next() を呼出す度に yield 文の式が評価されて値として返却されます：

```
>>> def neko(x):
...     for i in range(0,4):
...         yield x**i
...
>>> a = neko(2)
>>> type(a)
<type 'generator'>
>>> a.next()
1
>>> a.next()
2
>>> a.next()
4
>>> a.next()
8
>>> a.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
```

■map/filter オブジェクト: Python 2 の関数 map() と filter() は関数をリスト/タプルに作用させたリスト/タプルを返却する関数でしたが、Python 3 の扱いは生成式に類似した map オブジェクトと filter オブジェクトになります:

```
>>> b=map(lambda x:x**2,(1,2,3,4))
>>> type(b)
<class 'map'>
>>> b.__next__()
1
>>> b.__next__()
4
>>> b.__next__()
9
>>> b.__next__()
16
>>> b=map(lambda x:x**2,(1,2,3,4))
>>> c=list(b)
>>> c
[1, 4, 9, 16]
>>> d=filter(lambda x: x % 2, range(10))
>>> type(d)
<class 'filter'>
```

```
>>> d.__next__()
1
>>> d.__next__()
3
>>> d.__next__()
5
>>> d.__next__()
7
>>> d.__next__()
9
>>> list(filter(lambda x: x % 2, range(10)))
[1, 3, 5, 7, 9]
```

Python の内包表現

集合の内包表現はその集合の成分を列記せずに各成分が充すべき命題を記載して集合を定める方法です。たとえば、集合 S を ‘ $\{0, 1, 2, 3, 4, 5\}$ ’ と成分を列記する方法, ‘ $\{x : x \in \mathbf{N} \wedge x \leq 5\}$ ’ と論理式 ‘ $x \in \mathbf{N} \wedge x \leq 5$ ’ を使って集合 S の元が充たすべき性質を記載する方法と集合 S には二つの表現方法があり、前者が外延による集合の表現、後者が内包による集合の表現です。Python で用いられる内包表現は for 節で生成した対象を if 節で篩にかけて集合の成分のみを返却する処理を行います。

■内包表現: Python の内包表現の EBNF を次に示します:

内包表現の EBNF

内包表現	::=	式 for 節 if 節
for 節	::=	"for" 標的リスト "in" 論理式 [反復式]
if 節	::=	"if" 論理式 [else 反復式]
反復式	::=	for 節 if 節

「式 for 節」における「式」は「for 節」から生成される標的リストを変数として含む Python の式です。通常は標的リストに代入される値を式で利用しますが、「`1 for x in range(0,10)`」のように標的リストを定数のリストとする表記も可能です。この EBNF で本質的な箇所が「標的リスト "in" 論理式」で、標的リストで指示された変数への代入と密接に関係します。まず、演算子 “in” は左辺で指示されたオブジェクトが右辺のオブジェクトに包含されるかどうかを判断する演算子で、標的リストを ‘ x, y, \dots, z ’, 論理式を ‘ $A \vee, \dots, \vee Z$ ’^{*28} とするときに

*28 論理式は論理和 \vee の式として変形することができます。

$$((x, y, \dots, z)/\alpha \wedge \alpha \in A) \vee \dots \vee ((x, y, \dots, z)/\omega \wedge \omega \in Z)$$

を左の論理式から検証することに対応します。ここで $(x, y, \dots, z)/\alpha$ はオブジェクト α が標的リストに代入可能であれば True, そうでなければ False を返す論理式です。したがって, $(x, \dots, z)/\alpha \wedge \alpha \in A$ が True になるのは α 標的リストに代入可能, かつ, α が A で定められる集合の元であるときに限られます。この式の評価は, 最初の式が False と判断されると for 節はエラーを出力して処理を終えますが, 最初の式が True であれば, この最初の式の評価のみを実施して以後の論理式の評価を行いません。つまり, $((x, \dots, z)/\alpha \in A)$ の処理だけが行われ, この式 A は演算子 “or” を持たない式です。ここで $A = A_1 \wedge A_2 \wedge \dots \wedge A_n$ であれば $(x, \dots, z) \in A$ は $((x, \dots, z)/\alpha_1 \wedge \alpha_1 \in A_1) \wedge \dots \wedge ((x, \dots, z)/\alpha_n \wedge \alpha_n \in A_n)$ の処理になります。このときは式の左側から右側へと全ての式の評価が行われ, 一つでも False があれば $(x, \dots, z) \in A$ はエラーになり, 全て True であれば最後に評価した $(x, \dots, z)/\alpha_n$ がこの結果になります。このように詳細な論理式である必要性はなく, 標的リストへの代入が繰り返し実行可能な「**生成的な式**」であるべきで, そこで生成した値を if 節で篩にかける手法が妥当です。このような生成的な式には生成式, タプル, リスト, 集合や辞書といったオブジェクト, あるいはメソッド `__iter__()` や `__getitem__()` を持つクラスのインスタンスに限定されます。

内包表現で if 節の配置には二通りあり, 一つは for 節のうしろに配置して for 節で生成した成分を篩にかける方法, もう一つが if 節を前方に配置する方法です。前方に if 節を配置するときは else 節を有し, この else 節のうしろに for 節を置きます:

```
>>> [i for i in [1, 2, 3, 4]]
[1, 2, 3, 4]
>>> [i for i in [1, 2, 3, 4] if i%2==0]
[2, 4]
>>> [i if i%2==0 else -i for i in [1, 2, 3, 4]]
[-1, 2, -3, 4]
```

これらの例は最初のものが if 節を持たない内包表現の例, 次が if 節を篩として持つ内包表現の例, 最後が if 節と else 節を持つ内包表現の例で, 篭として持つときは if 節を for 節のうしろに, if-else を持つときは for 節をうしろに配置します。

属性参照

オブジェクトの属性参照で用いられる表記です:

属性参照の EBNF

```
属性参照 ::= 一次語 ". " 識別子
```

一次語でオブジェクトが指示され、識別子がオブジェクトの属性を指示します。

添字表記

オブジェクトの型が文字列リテラル、タプルやリスト等の列や辞書のときに成分の参照で用いられます：

添字表記の EBNF

```
添字表記 ::= 一次語 "[" 式のリスト "]"
```

MATLAB 系言語と異なり鍵括弧"["]"を使うことに注意が必要です。オブジェクトが列のときは「式のリスト」は整数値に限定され、列が1次元であれば0が列の先頭(左端)で、列の長さがnであれば'n-1'、あるいは'-1'が列の最後尾(右端)の成分を指示します：

```
>>> a = [1,2,3,4,5]
>>> a[0], a[1]
(1, 2)
>>> a[4], a[-1], a[-2]
(5, 5, 4)
>>>
```

この例では長さが5のリスト型のオブジェクトの成分取出を行っています。ここで添字が0でリストの先端の1、添字が4でリストの末端の5が返却されています。また、-1でリストの末端になります。この表記を利用すると列の長さが判らなくても最後尾から成分を取り出せます。列は後述のスライス表記による成分の取出しができます。そして、オブジェクトが辞書のときに「式のリスト」は辞書で用いられている鍵に限定されます。なお、集合型の成分の取出しは添字表記で行えません。

スライス表記

MATLAB 系言語で行列や配列の添字操作を、その表記方法も含めてタプル、リストや列で実現する表記です：

スライス表記の EBNF —

スライス表記	::=	単純スライス表記 拡張スライス表記
単純スライス表記	::=	一次語 "[" 短スライス表記 "]"
拡張スライス表記	::=	一次語 "[" スライスリスト "]"
スライスリスト	::=	スライス項目 ("," スライス項目)* [","]
スライス項目	::=	式 スライス本体 省略符号
スライス本体	::=	短スライス表記 長スライス表記
短スライス表記	::=	[下限] ":" [上限]
長スライス表記	::=	短スライス表記 ":" [刻幅]
上限	::=	式
下限	::=	式
刻幅	::=	式
省略符号	::=	"..."

スライス表記は MATLAB 系言語の配列処理と同様の表記ですが、MATLAB 系の言語では配列の添字が 1 から開始し、Python では C と同様に 0 から開始することに注意が必要です。また、MATLAB 系言語で添字の省略記号が “:” ですが、Python の省略記号は “...” であり、さらに長スライス表記で刻幅を短スライス表記のうしろに付けることが MATLAB 系言語の刻幅の表記と微妙に異なります。

Python のリストは 1 次元の配列としての性格を持ちます。多次元配列はリスト型では実現できず、パッケージ NumPy を必要とします。Python 本体には数学的定数や初等函数がほとんど含まれておらず、パッケージ NumPy で拡張する事実上の仕様になっています。ここで NumPy の配列 (ndarray 型) で添字の省略記号の持つ機能は、MATLAB 系言語の 1 次元的な省略よりも、Yorick の多次元配列の構造を省略表記する「ゴム添字 (rubber index)」がより近い機能を持っています：

```
>>> from numpy.random import *
>>> a = randn(100,100,100)
>>> a.shape
(100, 100, 100)
>>> a[...,2].shape
(100, 100)
>>> a[:, :, 2].shape
(100, 100)
>>> a[:, 2].shape
(100, 100)
>>>
```

この例では numpy の乱数モジュールを読み込み、関数 `randn()` で $100 \times 100 \times 100$ の乱数配列を生成します。省略記号によって省略された箇所の次元が保たれていることが判ります。この NumPy の詳細に関しては §4.4 を参照してください。

呼出

関数やメソッド等に処理を実行させるときに用いられる構文です：

呼出の EBNF

```

呼出      ::= 一次語 "(" [引数リスト [","] | 内包表現] ")"
引数リスト ::= 定位引数 ["," キーワード引数] ["," "*" 式]
              ["," キーワード引数] ["," "***" 式]
              | キーワード引数 ["," "*" 式] ["," "***" 式]
              | "*" 式 ["," キーワード引数] ["," "***" 式]
              | "***" 式
定位引数  ::= 式 ("," 式)*
キーワード引数 ::= キーワード事項 ("," キーワード事項)*
キーワード事項 ::= 識別子 "=" 式

```

引数リストの EBNF で「**定位引数**」、「**キーワード引数**」と「**式**」があり、「**定位引数**」は通常の関数やメソッドの引数が対応し、「**キーワード引数**」はオプションの引数、つまり、属性等の既定値の変更に用います。

3.5.4 演算式

Python の演算式は以下で分類されます：

演算式の EBNF

```
演算式  ::= 単項演算式 | 二項演算式 | 論理演算式 | 比較演算式
```

ここで「**単項演算式**」は被演算子が一つだけの演算子から構成される演算式、「**二項演算式**」はその値が真理値以外の値の二項演算子で構成される式、「**論理演算式**」は被演算子と演算結果が真理値になる演算式、そして、「**比較演算式**」は演算子として比較演算子を用いた式です。なお、ここで EBNF では構文の構成を記したもので、演算子が必要とする型について厳密に述べたものではありません。そのため詳細は各演算子の項目を参照してください。

単項演算式

式 -1 や $+1$ の演算子 “ $+$ ” や “ $-$ ” のように演算子のうしろに一つだけ被演算子を取る前置演算子で構成される式です:

単項演算式の EBNF

```

単項演算式 ::= 負符合式 | 正符合式 | ビット反転式
負符合式   ::= "-" ( 一次語 | "(" 演算式 ")" )
正符合式   ::= "+" ( 一次語 | "(" 演算式 ")" )
ビット反転式 ::= "~" ( 一次語 | "(" 演算式 ")" )

```

単項演算式には算術演算子の加法 “ $+$ ” や除法 “ $/$ ” に加え、2進数のビット反転演算子 “ \sim ” を先頭に置いた式があります。これらの演算子は整数、長整数、不動小数点数と虚数型の数オブジェクトの値を持つ被演算子を必要とします。

二項演算式

二つの被演算子を持つ演算式で、その値が真理値以外のものです。演算子は二つの被演算子を左右に配置する中值表現と呼ばれる書式になります:

二項演算式の EBNF

```

二項演算式 ::= 乗法演算式 | 加法演算式 | 置換演算式 | ビット演算式
乗法演算式   ::= ( 一次語 | "(" 演算式 ")" )
                  ( "*" | "/" | "//" | "%" )
                  ( 一次語 | "(" 演算式 ")" )
加法演算式   ::= ( 一次語 | "(" 演算式 ")" ) ( "+" | "-" )
                  ( 一次語 | "(" 演算式 ")" )
置換演算式   ::= ( 一次語 | "(" 演算式 ")" ) "***"
                  ( 一次語 | "(" 演算式 ")" )
ビット演算式   ::= ( 一次語 | "(" 演算式 ")" )
                  ( "&" | "|" | "^" | "<<" | ">>" )
                  ( 一次語 | "(" 演算式 ")" )

```

■乗法演算式: 積演算子 “ $*$ ”，商演算子 “ $/$ ”，整数置換演算子 “ $//$ ” と剰余演算子 “ $\%$ ” を持つ式です。これらの演算子は基本的に整数、長整数、浮動小数点数と複素数型、および Boolean をその被演算子として取ります。例外的に被演算子が正整数とリスト型のオブジェクトの一対のときはリスト型のオブジェクトの結合処理になります。また、Python 2

で演算子 “`/`” は被演算子が整数のときは結果も整数で、演算子 “`//`” と同じ結果になりますが、Python 3 で演算子 “`/`” を使った式は C と同様に浮動小数点数になります。このことは Python2 で演算子 “`/`” を使うときは被演算子のどちらかが浮動小数点数でなければ浮動小数点数の結果が得られないことを意味します。したがって、Python 2 では構築子 `float()` による型変換が必要な場合もあります。

■加法的演算式: 和演算子 “`+`” と減算演算子 “`-`” を持つ式です。このときの被演算子は数オブジェクトか Boolean ですが、文字列やリストの結合処理として演算子 “`+`” が利用可能です。

■幕演算式: Python で幕乗は演算子 “`**`” を用い、演算子 “`^`” は幕乗の演算子ではありません。ところで、SageMath では数学オブジェクトの幕乗演算子として演算子 “`^`” を用いていますが、完全な置き換えではありません。

■ビット演算式: 被演算子として整数、長整数型、あるいは Boolean を取り、2進数で表示に対して処理を行う演算子です。演算子 “`&`”, “`|`” と “`^`” は2進数表示した被演算子に対してビット単位で論理積 (AND), 論理和 (OR) と排他的論理和 (XOR) を行います。まず論理積は双方が 1 のときのみが 1 で他が 0, 論理和は双方が 0 のときだけが 0 で他が 1, 排他的論理和はどちらか一方が 1 のときだけが 1 で、それ以外は 0 を返す演算子です。また、演算子 “`<<`” と “`>>`” は被演算子を2進数で表現したときにビットを左右に桁を移動させる演算で、通常の算術演算子よりも優先順位が低くなっています。右被演算子が `sys.maxsize`^{*29} を越えたときは例外 `OverflowError` を送出し、負の数のときは例外 `ValueError` を送出します。

論理演算式

Boolean をその意味として持つ式を論理和、論理積、否定で処理する式です：

*29 最大の整数型の数です。

論理演算式の EBNF

```

論理演算式 ::= 一次語 | 論理和検証式 | 論理積検証式 | 否定式
              | 帰属検証式 | 比較演算式
論理和検証式 ::= ( 一次語 | 論理演算式 ) "or"
                  ( 一次語 | 論理演算式 )
論理積検証式 ::= ( 一次語 | 論理演算式 ) "and"
                  ( 一次語 | 論理演算式 )
否定式      ::= "not" 論理演算式
帰属検証式 ::= 式 "in" 式
比較演算式 ::= 式 ( "<" | ">" | ">=" | "<=" |
                  | "is" | "==" | "<>" | "!=" ) 式

```

これらの演算式によって Boolean 値が返却されます。なお、Boolean の True は整数の 1, False は整数の 0 と等価で、このことを利用した算術演算で if 文の代用になります。この手法は MATLAB 系の言語でも同様で、言語のオーバーヘッドを低減する手法として極めて有効です。なお、Python の内包表現で for 節で演算子 “in” と併用されますが、この場合は演算子 “or” や “and” でジェネレータ型のオブジェクトを結合した式が使えます。

■帰属検証式: 左辺の被演算子としての式が右辺の被演算子としての式の成分であることを検証する式です。基本的に右辺の式は閉包になります：

```

>>> 1 in (1, 2, 3)
True
>>> 1 in [2*i-1 for i in range(4)]
True
>>> 'neko' in 'mikeneko'
True
>>> import numpy as np
>>> 1 in np.arange(10)
True

```

帰属検証式ではタプルやリストといったデータを包含するものであれば使えます。そのためタプルやリスト、集合やハッシュ表以外の文字列や配列 (NumPy の ndarray) に対しても使えます。

■比較演算式: C の比較演算子と異なる優先順位を持ち、‘a > b > c’ のような複合的な表記が可能になっています。比較演算子による結果は True か False の Boolean になります。比較演算子には通常の大小関係の演算子と同値性を示す演算子 “==”，合同性を示す演算子としての演算子 “is” があります。比較演算式では幾らでも繋げることが可能ですが、ここで C や Fortran の比較の演算式は厳密に二項演算子で、2 以上のアリティを持つ演算子ではありません。

ませんが、Python では比較の演算子は 2 以上のアリティを持ち、さらに四則演算を表現する算術演算子のように扱えます。このときに四則演算の優先順位がないために式の左側の二項演算子から評価されます。具体的には比較演算の式が ‘ $a_1 \text{ op}_1 a_2 \text{ op}_2 \dots a_{n-1} \text{ op}_n a_n$ ’ であれば ‘ $(a_1 \text{ op}_1 a_2) \text{ and } (a_2 \text{ op}_2 a_3) \text{ and } \dots \text{ and } (a_{n-1} \text{ op}_n a_n)$ ’ の処理で置換えられ、左側の二項演算子の式から順番に実行されます。

if 式

if 節による条件分岐を含む式ですが、アリティが 3 の演算子としての性格も持ります：

if 式の EBNF

if 式 ::= 式 "if" 論理式 "else" 式

この構文は ‘ $x \text{ if } y \text{ else } z$ ’ の書式で y が `False` または 0 のときに z 、それ以外は x が返却されます。この構文の性格上、必ず `else` 節がなければなりません：

```
>>> a = 1
>>> 128 if a==0 else 256
256
```

lambda-式

Python の lambda 式は無名函数、いわゆるチャーチの λ -式を構成します：

lambda-式の EBNF

lambda-式 ::= "lambda" [パラメータリスト]: 式
 パラメータリスト ::= "(" 識別子 ("," パラメータリスト) ")"

函数定義の `def` 文、クラス定義の `class` 文のように記号 “`:`” のうしろに改行を入れてはなりません。「式」を括弧 “`()`” を使ってグループ化していれば改行を入れられますが、通常は一行の式を記述します：

```
>>> a = lambda(x):( x**2+
... x*2 +
... 1)
>>> a(1)
4
```

await 式

Python 3.5 から導入された式で、Python 2 にはありません。

3.5.5 単純文

「**単純文 (simple statement)**」は、その文全体を一つの論理行内に収めることができます。複数の単純文をセミコロン “;” で区切って続けられます：

—— 単純文の EBNF ——

```
単純文 ::= 式文 | 代入文 | 累積代入文
         | assert 文 | pass 文 | del 文 | print 文
         | return 文 | yield 文 | raise 文
         | break 文 | continue 文
         | import 文 | global 文 | exec 文 | nonlocal 文
```

なお、print 文と exec 文は Python 3 ではそれぞれ関数 print() と関数 exec() で置き換えられます。また、nonlocal 文は Python 3 で導入された文です。

■**式文**：式文は意味のある値を返さない関数で用いられます：

—— 式文の EBNF ——

```
式文 ::= 式のリスト
```

■**代入文**：名前とオブジェクトを関連付けるために用いられます：

—— 代入文の EBNF ——

```
代入文      ::= (標的リスト "=") + (式のリスト | 生成式)
標的リスト  ::= 標的 ("," 標的)* [","]
標的        ::= 識別子
             | "(" 標的リスト ")"
             | "[" 標的リスト "]"
             | 属性参照
             | 添字表記
             | スライス
```

代入文の「(標的リスト "=") + (式のリスト | 生成式)」は Python の内包表現で述べた for 節の標的リストに対する処理と本質的に同じです。つまり、標的リストは「式のリスト」や生成式が対応するオブジェクトは同じ長さのオブジェクトのリストでなければなりません。たとえば、「標的リスト」を ‘ x, y, \dots, z ’、右辺の「式のリスト」を ‘ a, b, \dots, k ’、生成式が対応するオブジェクトを m とすると、これらの長さは一致し、 $(x, y, \dots, z)/(a, b, \dots, k)$

と $(x, y, \dots, z)/m$ が True にならなければなりません。ここで記号 “/” は左辺の式に右辺の式が代入可能であるかどうかの判断です。

■**assert 文**: プログラム中にデバッグ用の仮定を仕掛けるための手法を提供します:

assert 文の EBNF

```
assert 文 ::= "assert" 論理和検証式 ["," 論理和検証式式]
```

引数の式が一つの assert 文の ‘assert 式’ は以下の if 文と同等の機能を持ちます:

```
if __debug__:
    if not 式論理和検証式 raise AssertionError
```

また、引数が二つの assert 文:

assert 論理和検証式₁ , 論理和検証式₂

は以下の if 文と等価です:

```
if __debug__:
    if not $論理和検証式_1$: raise AssertionError($論理和検証式_2$)
```

このように assert の直後の論理和検証式を充すときに処理が行われ、それ以外で例外:AssertionError が生じます。

■**pass 文**: 文字通り「何もしない」文です:

pass 文の EBNF

```
pass 文 ::= "pass"
```

この文は構文的に何らかの文が必要であっても何も評価や実行を行いたくないときに用います。

■**del 文**: オブジェクトや属性の削除を行う文です:

del 文の EBNF

```
del 文 ::= "del" 標的リスト
```

標的リストに対する削除では、指定したリストの左端の対象で指示されるオブジェクトから右端の対象で指示されるオブジェクトへと再帰的にオブジェクトの削除を行います。

■**print 文**: オブジェクトや属性の値を標準出力に出力する文です。

print 文の EBNF

```
print 文 ::= "print" [(式 (", 式)* [","]) | "»" 式 [(", 式)+ [","]])
```

print 文は Python3 で函数 print() になるために扱いに十分に注意してください。

■return 文 函数やメソッドで明示的に値の返却を行うために用いる文で、return 文の引数に返却すべき値を記載します：

return 文の EBNF

```
return 文 ::= "return" [式のリスト]
```

return 文は yield 文と同一函数内部で混在できません。

■yield 文： 産出函数の定義で用いられます：

yield 文の EBNF

```
yield 文 ::= yield 式
```

yield 文は PEP-255 より導入された文で、return 文の代わりに yield 文を用いて定義された函数は産出函数と呼ばれ、その内部で return 文を記載できません。産出函数からジェネレータ型のオブジェクトが生成されます：

```
>>> def fibonacci():
...     a, b = 0, 1
...     while 1:
...         yield a
...         a, b = b, a + b
...
>>> a = fibonacci()
>>> a.next()
0
>>> a.next()
1
>>> a.next()
2
>>> a.next()
3
>>> a.next()
5
>>> type(fibonacci)
<type 'function'>
```

```
>>> type(a)
<type 'generator'>
```

この例はフィボナッチ数列を产出函数で定義したもので、メソッド next() で都度、計算処理が行われます。ジェネレータ型のオブジェクトはリストや配列のように全てのデータをメモリ上に持たずに、必要なときに都度、生成できる点で優れています。

■**raise 文**: 利用者が意図的に例外の送出を行うときに用います:

raise 文の EBNF

```
raise 文 ::= "raise" [式 ["," 式 ["," 式]]]
```

例外そのものについては §3.12.3 を参照してください。raise 文自体は try 文の try 節に記載し、raise 文等が送出した例外に対応した処理を except 節で受け取って処理します。あらかじめ生じ得る問題を例外として定義し、それに対する処理をプログラムに記述することで安定した処理が行えるようになります。

■**break 文**: for 文、while 文による反復処理から抜けるために用いられる文で引数が不要です:

break 文の EBNF

```
break 文 ::= "break"
```

■**continue 文**: 引数が不要な文で、for 文や while 文による反復処理の継続で用います:

continue 文の EBNF

```
continue 文 ::= "continue"
```

■**import 文**: モジュールやパッケージの読み込みで用いられる文です:

import 文の EBNF

```

import 文      ::=  "import" モジュール ["as" 名前]
                  (",," モジュール ["as" 名前])* 
                  | "from" 関係モジュール "import" 識別子
                  [ "as" 名前] (",," 識別子 ["as" 名前])* [ "," ] ")"
                  | "from" モジュール "import" "*"
モジュール     ::=  (識別子 ".")* 識別子
関係モジュール ::=  "." モジュール | ".+" 
名前          ::=  識別子

```

import 文で読み込まれたモジュールやパッケージは module 型のオブジェクトになり、その名前は as 節が未指定であれば import 文の引数であるモジュール、as 節があればその引数の識別子がオブジェクトの名前になります。ここで複数のモジュールを纏めて階層構造を入れたモジュールのことを「パッケージ (package)」と呼びます。

■**future 文**: 将来の Python のリリースで利用可能になるような構文や意味付けを行うための指示句です:

future 文の EBNF

```

future  ::=  "from" "__future__" "import" 機能 ["as" 名前]
            (",," 機能 ["as" 名前])* 
            | "from" "__future__" "import" "(" 機能 ["as" 名前]
            (",," 機能 ["as" 名前])* [ "," ] ")"

```

future 文はモジュールの先頭に置かなければなりません。ここで future 文よりも先行して置ける内容に、文書文字列、注釈、空行と他の future 文に限定されます。

■**global 文**: コードブロック全体で維持される宣言文で、後続の識別子を大域変数として扱うことを指示します:

global 文の EBNF

```

global 文  ::=  "global" 識別子 (",," 識別子)*

```

ここで global 文で宣言する名前はプログラムの中で global 文に先行して配置してはいけません。また、for 文での反復処理制御用の変数の名前、class 文によるクラスの定義や関数定義、import 文内で global 文で宣言した名前を仮変数として用いてもいけません。

■**exec 文**: Python コードの動的な実行に関する文です:

exec 文の EBNF

exec 文 ::= "exec" 識別子 ("," 識別子)*

3.6 複合文

複合文は複数の文節と、それに関わる一群の文から構成されます。複合文は複数行で記述されますが、一行に纏められて記載されることもあります：

複合文の EBNF

```

複合文      ::= if 文
              | while 文
              | for 文
              | try 文
              | with 文
              | funcdef 文
              | classdef 文
              | decorated 文
一揃いの文  ::= 文の列 NEWLINE
              | NEWLINE INDENT 文 + DEDENT
文          ::= 文の列 NEWLINE | 複合文
文の列     ::= 単純文 (";" 単純文)*[";"]

```

3.6.1 条件分岐に関連する複合文

■if 文：条件分岐を指示する文です：

if 文の EBNF

```

if 文      ::= "if" 式 ":" 一揃いの文
              ( "elif" 式 ":" 一揃いの文 )*
              ["else" ":" 一揃いの文]

```

Python で用意されている条件分岐は if 文のみです。

3.6.2 反復処理に関する複合文

■**while 文**: for 文と並び反復処理を行うために用いる文で、与えられた条件の真理値が True のときに while 文内部の処理を実行します:

while 文の EBNF

while 文 ::= "while" 式 ":" 一揃いの文

■**for 文**: while 文と並び反復処理を行うために用意された文です。for 文は式のリストから標的リストに含まれる束縛変数に値を引渡し、その値を用いて for 文内部の式の評価を行います:

for 文の EBNF

for 文 ::= "for" 標的リスト "in" 式 ":" 一揃いの文

内包表現で述べたように、標的リストに代入可能な形式のオブジェクトが式から出力されなければなりません。そして式は本質的にタプル、リスト、集合、辞書やジェネレータ型のオブジェクトです。

3.6.3 例外処理に関する複合文

■**try 文**: 例外処理のために用意された文です:

try 文の EBNF

try 文 ::= try 文 1 | try 文 2
 try 文 1 ::= "try" ":" 一揃いの文
 ("except" [式 [("as" | ",") 標的] ":" 一揃いの文]+
 ["finally" ":" 一揃いの文])
 try 文 2 ::= "try" ":" 一揃いの文
 "finally" ":" 一揃いの文

例外の詳細は §3.12.3 を参照してください。基本的に try 節の文を実行し、そこで送出された例外を except で受けて処理を行います。

3.6.4 隠蔽に関連する複合文

■with文: ブロックの実行をコンテキストマネージャで定義されたメソッドで覆うために用いられます:

with の EBNF

```
with 文      ::= "with" with の項目 (", with の項目)* ":" 一揃いの文
with の項目 ::= 式 ["as" 標的]
```

3.6.5 定義に関連する複合文

■函数定義: 函数やメソッドの定義のための文です:

函数定義の EBNF

```
函数定義      ::= "def" "(" [パラメータリスト] ")" ":" 一揃いの文
函数名        ::= 識別子
decorated     ::= decorators (クラス定義 | 函数定義)
decorators    ::= decorator+
decorator     ::= "@" 付点名 [ "(" [引数リスト] [","] ")"]
                  NEWLINE
付点名        ::= 識別子 ("." 識別子)*
パラメータリスト ::= (パラメータ定義 ".")*
                  ( "*" 識別子 [, "*" 識別子] | "***" 識別子
                    | パラメータ定義 [","] )
副リスト      ::= パラメータ (",, パラメータ)* [","]
パラメータ    ::= 識別子 | "(" 副リスト ")"
```

函数定義が函数を実行するのではなく、函数が呼び出されたときのみに函数本体が実行されます。また、函数定義によって局所的な名前空間で函数名に函数オブジェクトの束縛が行われます。

■クラス定義: class文でクラスの定義を行います:

クラス定義の EBNF

```
クラス定義 ::= "class" クラス名 [継承] ":" 一揃いの文  
継承       ::= "(" [式リスト] ")"  
クラス名     ::= 識別子
```

最初に継承リストがあればリストの評価を行います。ここで継承リストの各要素の評価結果はクラスオブジェクト、あるいはサブクラス可能なクラス型でなければなりません。それから実行フレーム内部にて局所名前空間と大域名前空間を用いてクラス内の変数への束縛が行われます。ここで実行フレームは無視されるものの局所名前空間は保持され、それから基底クラスの継承リストを用いてクラスオブジェクトが生成されて局所名前空間を属性値辞書として保存します。それから最後に局所名前空間でクラス名がクラスオブジェクトに束縛されます。次の節では、オブジェクト、名前空間や例外の詳細について述べます。

3.7 オブジェクトについて

3.7.1 オブジェクトの概要

Python のオブジェクトは「**識別値 (identity)**」、「**型 (Type)**」と「**値 (Value)**」をその属性として持ります。まず、「**識別値**」はオブジェクト生成時に定まって変更できないオブジェクト固有の値で、具体的にはオブジェクトのメモリ上の番地に対応する整数値です。「**型**」もオブジェクトの生成時に決定されて以後、変更できません。そして、型 (type) 自体もまた型です。そして、オブジェクトはその値が「**変更可能 (mutable)**」なものと、逆に「**変更不可能 (immutable)**」なものに分類できます。なお、変更不可能なオブジェクトでも別のオブジェクトへの参照を持つコンテナであれば参照先の変更で実質的に内容の変更ができます。この変更不可能なオブジェクトでは、その生成時に一定のメモリを割り当てられ、さらに組込の変更不可能なオブジェクトは全て「**要約可能 (hashable)**」という性質を持ちます。オブジェクトが「**要約 (ハッシュ) 可能, hashable**」であるとは、そのオブジェクトが生成されて、それが存在している間に一定の整数値、すなわち、「**要約値 (hash value)**」を持つことを意味します。この要約値は二つのオブジェクトが同じ値を持っているときに要約値が一致するという重要な性質があります。要約値はオブジェクトの持つ値が等しいことをオブジェクトの詳細を調べなくても判別できるようにするために値で、要約値を計算するメソッドが `__hash__()` です。なお、要約不可能なオブジェクトでは `__hash__` にオブジェクト `None` が割り当てられています。ここで `None` は空集合 `Ø` に対応するために空集合 `Ø` を始域に持つ写像も空集合 `Ø` になるという事実に合致します。

オブジェクトの参照で「**名前空間**」が使われます。この名前空間はオブジェクトと識別子の間に対応関係を与える仕組で、名前空間で扱われる識別子を「**名前**」と呼び、オブジェクトの識別値、型、値の参照は名前を介して行われます。たとえば、オブジェクト固有の値である識別値の参照は名前を引数に函数 `id()` で、オブジェクトの参照はその名前を引数にして函数 `type()` を使って調べられます。また、異なる二つの名前で指示されるオブジェクトの同一性の判断は演算子 “`is`” に二つの名前を引き渡すことで行えます。このように名前はオブジェクトに取り付ける名札であり、オブジェクトへのさまざまな参照は名前を介して得られます。この名前空間の実装では連想配列の辞書型オブジェクトが用いられます。

3.7.2 オブジェクトの生成と廃棄を行うメソッド

オブジェクトの生成を行う函数、メソッドを「**構築子/コンストラクタ (constructor)**」と呼びます。オブジェクトの生成では「**メモリの割当 (allocation)**」と属性値の設定等のオブジェクトの「**初期化 (initialization)**」が同時に行われます。逆にオブジェクトを消去するときに呼び出される函数やメソッドを「**消滅子/デストラクタ (destructor)**」^{*30}と呼びます。

Python では構築子に相当するメソッドに `__new__()` と `__init__()` の二つのメソッドがあり、メソッド `__new__()` でオブジェクト生成、メソッド `__init__()` で初期化を行います。メソッド `__new__()` が呼出されると自動的にメソッド `__init__()` も呼出され、これらのメソッド二つで構築子としての機能を持ち、特にメソッド `__init__()` が C++ の構築子に最も類似した働きをします。なお、Python ではクラスの定義で構築子に対応するこれらの二つのメソッドを記載する必要はありません。これらのメソッドがクラスに記載されていなければより上位のクラスのメソッドがそのまま継承されます。

Python の消滅子としてはメソッド `__del__()` が該当しますが、オブジェクトを直接破棄するメソッドではありません。Python でオブジェクトの破棄には「**GC(Garbage collection)**」が深く関わります。

3.7.3 GC(garbage collection)について

Python ではオブジェクトの生成や破棄でメモリの割当や解放が自動的に行われ、利用者が不要になったオブジェクトを直接破棄できません。また、オブジェクトが破棄されるためにはオブジェクトと名前との関係が途切れた状態、すなわち「**到達不能の状態**」になることが必要で、到着不能の状態でオブジェクトの破棄が許可され、やがて、「**GC(塵回**

^{*30} Java ではファイナライザ (finalizer) と呼びます。

収, garbage collection)」でオブジェクトの回収と破棄が行われます。

CPython の GC では「**参照カウント (reference counting)**」が採用されています。この方式は全てのオブジェクトに参照カウントと呼ばれる整数値を付与し、オブジェクトの生成時に参照カウントに 0 が設定されます。以降、オブジェクトがある名前に束縛されたり、他のオブジェクトからの参照があれば参照カウントに 1 を加え、その名前に別のオブジェクトが束縛されて参照関係が解消されると参照カウントから 1 を減じ、参照カウントが 0 になった時点でオブジェクトの破棄が許可されます。なお、消去子 `__del__(self)` の実行でオブジェクトが破棄されるのではなく、実際の働きは名前との参照関係を外すためにオブジェクトの参照カウントが 1 減じるだけです。オブジェクトの参照カウントはモジュール `sys` の函数 `getrefcount()` で確認できますが、函数 `getrefcount()` で調べた値は本来の参照カウントよりも 1 多くなっています。これは函数 `getrefcount()` による参照が参照カウントに追加されるためです。ここでは簡単な例で確認しておきましょう：

```
>>> import gc
>>> gc.set_debug(gc.DEBUG_STATS)
>>> class A():
...     def __del__(self):
...         print "Nyao"
...
>>> for i in range(5):
...     print i
...     a = A()
...     a = 1
...
0
Nyao
1
Nyao
2
Nyao
3
Nyao
4
Nyao
>>>
```

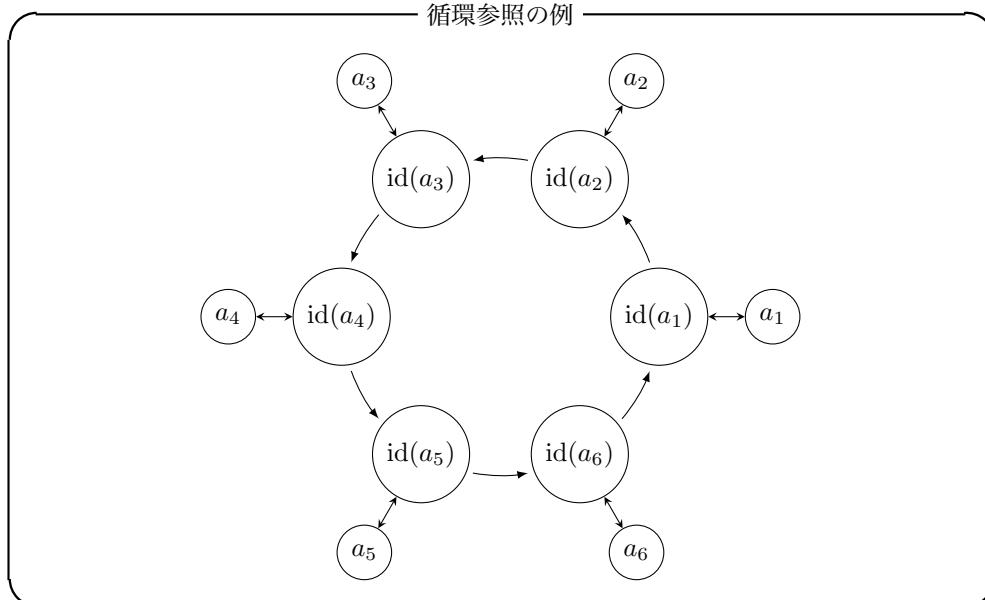
循環参照を持たずにオブジェクトの間の参照関係のみであれば、名前との参照関係が外れた時点でオブジェクトの参照カウントが 0 になってメソッド `__del__(self)` が呼び出されていることが判ります。しかし、参照カウントが 0 になってオブジェクトの破棄が許可されてもオブジェクトの総数が閾値に到達して GC が自動的に起動したとき、あるいは利用者

がモジュール `gc` の関数 `collector()` を起動したときにオブジェクトが破棄されます。この閾値はモジュール `gc` の関数 `get_threshold()` で確認できます:

```
>>> import gc
>>> gc.set_debug(gc.DEBUG_STATS)
>>> gc.collect()
gc: collecting generation 2...
gc: objects in each generation: 249 3282 0
gc: done, 0.0017s elapsed.
0
>>> gc.get_threshold()
(700, 10, 10)
>>>
```

この例では関数 `set_debug()` で統計量の出力を設定しているために関数 `collect()` でオブジェクトの実行で統計情報が表示されています。最後に実行した関数 `gc_threshold()` は GC を行う閾値を表示する関数で、ここで表示されたタプルの意味は、最初の 700 が第 0 世代、以降の 10, 10 が第 1, 第 2 世代の閾値です。この世代は到達可能なオブジェクトを乗り越えた GC の回数で分類したもので、最初に生成したオブジェクトを GC の 0 世代とし、第 1 世代が一度、GC を乗り越えた到達可能なオブジェクト、第 2 世代が二度以上 GC を乗り越えた到達可能なオブジェクトです。GC の自動実行は第 0 世代の総数が閾値を越えた時点で行われ、第 2, 第 1, 第 0 世代と古い順にオブジェクトの回収が行われます。

この参照カウント方式には大きな弱点があります。それは参照関係でオブジェクトを節点（ノード、node）、参照関係を線分（エッジ、edge）とすることで得られるグラフで考えると明瞭になります。たとえば $i \in \{1, \dots, 5\}$ のときにオブジェクト a_i が a_{i+1} を参照し、オブジェクト a_6 が a_1 を参照するといった自分の定義のために他者を参照し、その他者が最終的に自分を参照するために自分自身の参照が生じるという参照関係をグラフ化すると円環が現れます：



この例で名前 $a_{i,i \in \{1,\dots,6\}}$ に束縛されたオブジェクトを $\text{id}(a_i)$ と表記します。オブジェクトの表記に函数 $\text{id}()$ を用いた理由は函数 $\text{id}()$ で返却されるオブジェクトの識別値がオブジェクト固有の値になるためです。この例のように参照関係を辿ると最終的に自己に戻る参照を「循環参照 (circular reference)」と呼びます。この例のオブジェクトは名前を介して相互に参照があるために全てのオブジェクトと名前との参照関係が途絶えてもオブジェクト間の参照関係 (内側の矢印) が残ります。このことは各オブジェクトの参照カウントが 0 にならないことを意味し、参照カウントが 0 のオブジェクトを回収する方法では、これらの到達不能のオブジェクトが回収できません。

この弱点を克服する一つの手段がオブジェクトを世代別に分ける方法です。最初に生成直後のオブジェクトを第 0 世代とします。そして、第 0 世代のオブジェクト総数が閾値を超えるか、`gc` モジュールの函数 `collect()` を起動することで GC が起動します。この GC ではオブジェクトが世代毎の到達可能性を調べ、第 0 世代のオブジェクトで到達可能であれば第 1 世代、第 1 世代と第 2 世代のオブジェクトで到達可能なものは第 2 世代と世代の繰り上げます。その結果、到達不能なオブジェクトだけが残り、これらを回収すれば良いことになります。循環参照を行っている到達不可能なオブジェクトにメソッド `__del__()` が定義されていると、どちらのメソッド `__del__()` を利用すべきかを判断できなくなるために、これらのオブジェクトが GC で回収できなくなります。

このことを例で確認しておきましょう。GC がどのように実行されているかを観察するために `gc` モジュールの函数 `set_debug()` を使って設定を行います。ここで統計情報を出

力させるため gc.DEBUG_STATS の設定を行います。最初に循環参照があってもメソッド __del__() を持たないクラスの場合です：

```
>>> import gc
>>> gc.set_debug(gc.DEBUG_STATS)
>>> class A():
...     pass
...
>>> for i in range(200):
...     print i
...     a = A()
...     b = A()
...     a.x = b
...     b.y = a
...
0
1
— 略 —
116
gc: collecting generation 0...
gc: objects in each generation: 717 3282 0
gc: done, 460 unreachable, 0 uncollectable, 0.0002s elapsed.
117
— 略 —
199
>>>
>>> gc.collect()
gc: collecting generation 2...
gc: objects in each generation: 339 3496 0
gc: done, 336 unreachable, 0 uncollectable, 0.0023s elapsed.
336
>>> gc.collect()
gc: collecting generation 2...
gc: objects in each generation: 4 0 3469
gc: done, 0.0019s elapsed.
0
>>> gc.garbage
[]
>>>
```

閾値が 700 あるためにオブジェクトが 700 を越えた時点で最初の GC が行われ、この GC では第 1, 第 2 世代がまだないので第 0 世代のみです。最初の GC のあとでもオブジェクトの生成が行われていたために手動で 2 回、関数 collect() を実行して GC を実行し

ています。ここで最初の GC の実行で 112、最後の GC の実行で 0 と表示されていますが、これは回収したオブジェクトの総数です。GC で回収できないオブジェクトの情報は garbage にリストとして蓄えられ、このリストの長さが回収不能なオブジェクトの個数に一致します。この例では gc.garbage を入力して空リストが返却されているために回収できないオブジェクトが発生していないことが分かります。したがって、ここでの循環参照は上手くオブジェクトの回収ができています。次にメソッド __del__() を有するクラスで同じことをしてみましょう：

```
>>> import gc
>>> gc.set_debug(gc.DEBUG_STATS)
>>> class A():
...     def __del__(self):
...         print "Nyao"
...
>>> class B():
...     pass
...
>>> for i in range(200):
...     print i
...     a = A()
...     b = B()
...     a.x = b
...     b.y = a
...
0
1
2
— 略 —
114
gc: collecting generation 0...
gc: objects in each generation: 714 3282 0
gc: done, 452 unreachable, 452 uncollectable, 0.0001s elapsed.
115
— 略 —
198
199
>>> >>> gc.collect()
gc: collecting generation 2...
gc: objects in each generation: 347 3951 0
gc: done, 344 unreachable, 344 uncollectable, 0.0023s elapsed.
344
>>> len(gc.garbage)
199
```

この例は先程のクラス A にメソッド `__del__()` を追加し、もう一つのクラス B は消滅子を持たないクラスとして定義しています。ここでクラス A に追加したメソッド `__del__()` は文字列 'Nyao!' を表示するだけです。ところが、この場合は GC で到達不能のオブジェクトを認識しても、メソッド `__del__()` を持たないクラス B のインスタンスの回収は行っても、クラス A のインスタンスについては参照カウントを 0 にしてメソッド `__del__()` を呼び出して占有していたメモリを解放しません。その結果、不要になつたクラス A のインスタンスが溜る一方になります。このことは最後に函数 `collect()` を実行した後で `gc.garbage` のリスト長が 199 であることから判ります。

これらの例は GC がいつ起動するかその様子を見たかったために多くのオブジェクトを生成しています。最後の例として最初の a_1, \dots, a_6 の相互参照で構成される循環参照の例を挙げておきましょう：

```
>>> import gc
>>> gc.set_debug(gc.DEBUG_STATS)
>>> class A(object):
...     def __del__(self):
...         print "Nyao!"
...
>>> a1 = A()
>>> a2 = A()
>>> a3 = A()
>>> a4 = A()
>>> a5 = A()
>>> a6 = A()
>>> a2.x = a1
>>> a3.x = a2
>>> a4.x = a3
>>> a5.x = a4
>>> a6.x = a5
>>> a1.x = a6
>>> a1 = a2 = a3 = a4 = a5 = a6 = 1
>>> gc.collect()
gc: collecting generation 2...
gc: objects in each generation: 279 3282 0
gc: done, 12 unreachable, 12 uncollectable, 0.0018s elapsed.
12
>>> len(gc.garbage)
6
>>>
```

この例では函数 `collect()` で GC を実行しても円環を構築する 6 個のオブジェクトが残さ

れ、メソッド`__del__()`が実行されることもありません。また、この例では全ての名前に`1`を割り当てて参照関係を解消していますが、これを函数`del()`で置き換えたとしても、名前とオブジェクトとの参照関係が切断されて参照カウントが`1`減らされるだけで、「**どちらの消滅子を使えばよいのか?**」という問題のために肝心の処理が行われません。このように名前との参照関係を外しただけでメソッド`__del__()`の内容が必ず実行されるとは限らず、また、インタプリタを終了したときにメソッド`__del__()`が実行される保証もありません。そのためにファイル等の外部リソースを参照するときはメソッド`__del__()`で解放するのではなく、適宜、`close()`等のリソースを解放するメソッドを使うべきです。

3.7.4 オブジェクトの値

Python のオブジェクトは値が「**変更可能 (mutable)**」なものと「**変更不能 (immutable)**」ものの二種類に分類され、この性質はオブジェクトの型で決定されます。

ここで簡単な例を示しておきましょう：

```
>>> a = b = []
>>> id(a)
3073670732L
>>> b.append(128)
>>> a is b
True
>>> id(a)
3073670732L
>>> id(b)
3073670732L
>>> c = [128]
>>> id(c)
3073670700L
```

この例では`'a = b = []'`でリスト型のオブジェクト`'[]'`を生成して名前`a, b`に同時に束縛させています。そのために名前`a`と`b`で参照されるオブジェクトが共通のためにその識別値が一致します。ところでオブジェクト`'[]'`は変更可能なオブジェクトのために`b.append(128)`で名前`b`で参照されているオブジェクトに`'128'`を追加できます。ところで、このオブジェクトを名前`a, b`の双方で参照しているために名前`a`で評価しても当然、`'128'`が追加されたものが表示されます。このことは`'a is b'`^{*31}で調べても`True`が返却され、オブジェクト固有の識別値を返却する函数`id()`で双方が同じ値になることからも判り

^{*31} 二項演算子`"is"`はオブジェクトが「同一」であるかどうか、二項演算子`"=="`はオブジェクトの「値が等しい」かどうかを判断する違いがあります。

ます。次に名前 c にリスト [128] を束縛させてみます。すると名前 a, b, c で参照しているオブジェクトの値は一致しますが、名前 c のオブジェクトの識別値は名前 a, b のオブジェクトのそれと異なっています。だから、名前 a, b で参照されるオブジェクトと名前 c で参照されるオブジェクトは別物ですが、その値は [128] で一致します。次に変更不能な型のオブジェクトの場合はどうなるでしょうか？そこで、変更不能な型のオブジェクトである数リテラルの例で確認しましょう：

```
>>> c = d = 128
>>> print id(c),id(d)
148353652 148353652
>>> c = 256
>>> print id(c),id(d)
148356068 148353652
>>> print c, d
256 128
```

この例では最初に ‘c = d = 128’ でオブジェクト 128 を名前 c, d に束縛させています。この時点で函数 id() の結果から名前 c と d で参照されるオブジェクトの識別値が一致するために名前 c, d ともに同じオブジェクトを参照していることが分かります。次に名前 c に ‘c = 256’ でオブジェクトの束縛を行なうと、名前 c からの参照を解消し、新たにオブジェクト 256 を生成して名前 c に束縛させます。ところが名前 d で参照されるオブジェクトにこの影響が及ばないために前回の変更可能なオブジェクトの例と異なり、名前 d で参照されるオブジェクトは最初の ‘128’ のままで。このことは識別値を函数 id() の結果からも判ります。ここでさらに名前 d に別のオブジェクトを束縛させると名前 d に束縛されていたオブジェクト 128 への参照が途切れ、オブジェクト 128 への参照が名前 c と d 以外に存在しなければオブジェクト 128 は到着不能の状態であり、いずれ、GC で回収されます。

Python のオブジェクトには他のオブジェクトへの参照を持つものがあります。このようなオブジェクトのことを「**コンテナ (container)**」と呼びます。Python のコンテナには「**集合**」、「**タプル**」、「**リスト**」と「**辞書**」があります。コンテナは値の変更不能な型であっても変更可能な型のオブジェクトへの参照が行われていれば参照先のオブジェクトの値の変更に伴ってコンテナの実質的な値が変化します。要するにコンテナはアパートみたいなもので、住人が入替っても部屋番号はそのままのために部屋番号を介して住人への問合せができるが、コンテナの構造を変えることはアパートの改築に相当し、こちらは建物自体が以前のものと異なってしまいます。コンテナの例としてタプルとリストを使って違いを確認しておきましょう：

```
>>> a=[1]; b=[2]; c=[3]
>>> l1 = [a, b, c]
>>> t1 = (a, b, c)
```

```
>>> l1
[[1], [2], [3]]
>>> t1
([1], [2], [3])
>>> id(l1)
26005416
>>> id(t1)
25557832
>>> c.append(128)
>>> id(l1)
26005416
>>> id(t1)
25557832
>>> t1
([1], [2], [3, 128])
>>> l1
[[1], [2], [3, 128]]
>>> t1[2]=[128]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> l1[2]=[128]
>>> l1
[[1], [2], [128]]
>>> id(l1)
26005416
>>> c = [129]
>>> print l1, t1
[[1], [2], [128]] ([1], [2], [3, 128])
```

この例では三成分のリストとタプルを生成して中身の入れ替えを行っています。タプルは変更不可能で、リストのように成分を直接変更できないものの、コンテナであるために参照しているオブジェクトが変更可能であれば、そのオブジェクトの値を変更することで結果として中身の変更ができます。ところで最後に名前 `c` にオブジェクト `[129]` を束縛させた場合はリストやタプルの側の旧来の参照が新しい参照に切り替わることがないために、そのまま名前 `c` に束縛されていたオブジェクト `[3, 128]` への参照が継続されます。つまり、オブジェクトの生成で名前を用いた場合、その名前に対応するオブジェクトが参照されることで新しいオブジェクトとの参照関係が発生しますが、名前はそのオブジェクトの名札であって入れ物でないため、名前に別のオブジェクトを束縛したからといって以前構築したオブジェクトが更新されることはありません。

3.7.5 オブジェクトの型

Python の組込のオブジェクトの型には次のものがあります:

オブジェクトの型				
None	NotImplement	Ellipsis	数	列
集合	対応付け集合	呼出可能型	モジュール	クラス
クラスインスタンス	ファイル	内部型		

None, NotImplemented と Ellipsis は Python の組込の定数で、名前と値が一致し、同じ型のオブジェクトを持たないオブジェクトです。Python 組込の定数に布尔型の True, False と `__debug__` がありますが、これら、None, NotImplemented と Ellipsis は条件文で True や False のいずれかと同じ働きをする真理値として扱えます。ただし、真理値と違って数オブジェクトではないために演算ができません。そして、数は後付けになりますが、抽象基底クラス (ABC, Abstract Base Class) で実装された numbers と前述の布尔型を含みます。この抽象基底クラス numbers の具象クラスとして整数の int 型と long 型、実数の float 型、複素数の complex 型が表現されます^{*32}。数は Scheme の「**数値塔 (Numerical Tower)**」にしたがって複素数に実数が、実数に整数が含まれるという状況を、複素 数を基盤に実数を、実数を基盤に整数を載せた塔にたとえ、塔を逆さに組み立てられないのと同様に上部と下部構造の入替ができません。なお、数オブジェクト以外のオブジェクトはその生成に式や定義文を必要とするため、それらの EBNF は §3.5.2 で述べます。

■**None:** LISP の nil に似た値を持つオブジェクトの型で、そのオブジェクトが意味のある値を持たないことを指示します。None は組込の名前 None で参照され、その値は None そのもので if 文等の条件分岐で真理値 False として扱われますが False と同値ではありません。そして、この型を持つオブジェクトは None 以外に存在しません。このことを簡単な例で確認しておきましょう:

```
>>> def neko(x,y):
...     return None
...
>>> neko(1,2)
>>> zz = neko(1,2)
>>> zz
>>> type(zz)
<type 'NoneType'>
```

*32 PEP 3141: 「A Type Hierarchy for Numbers」参照。

```
>>> xx=None
>>> zz is xx
True
```

この例では `None` を返す函数 `neko()` を定義していますが、この函数の返却値を名前 `zz` に割り当てています。同時に名前 `xx` にも `None` を割当てていますが、演算子 “`is`” で両者の同一性を調べると `None` 型を持つオブジェクトは一つのみ存在するために両者の識別子が一致し、そのため `True` が返却されます。このオブジェクト `None` の挙動は（小）集合で構成される圈 **Set** で空集合 \emptyset を始域とする矢 \emptyset と同様の性質です。実際、圈 **Set** の矢は通常の写像が対応し、 $A, B \in \text{Obj } \mathbf{Set}$ に対して矢 $A \xrightarrow{f} B$ が存在したときに f を順序対の集合 $\{(x, f(x)) | x \in A\}$ と見なせます。ここで $a = \emptyset$ であれば $x \in \emptyset$ になる x が存在しないために $f = \emptyset$ となって \emptyset は圈 **Set** の対象であると同時に矢になります。このようにオブジェクト `None` は空集合 \emptyset と同様の性質を持ちます。

■NotImplemented: この型は单一の値しか持たず、この値を持つオブジェクトは `NotImplemented` のみです。条件文では真理値 `True` として扱われますが、`True` と同一ではありません。この `NotImplemented` の値は `NotImplemented` そのものです。

■Ellipsis: 配列処理で用いられる拡張スライス構文にて配列の添字全体を示すリテラル ‘...’ で構成されるオブジェクトが持つ型です。Ellipsis は省略を意味する型で、`None` のように何もないことを意味する型と異なります。この型を持つオブジェクトは `Ellipsis` のみで、その値は `Ellipsis` そのものです。条件文で真理値 `True` と同じ働きをしますが `True` と同一ではありません。この `Ellipsis` を用いたスライスの例を以下に示しておきます：

```
>>> from numpy import arange
>>> a = arange(16).reshape(2,8)
>>> a
array([[ 0,  1,  2,  3,  4,  5,  6,  7],
       [ 8,  9, 10, 11, 12, 13, 14, 15]])
>>> a[0, ...]
array([0,  1,  2,  3,  4,  5,  6,  7])
>>> a[1, ...]
array([ 8,  9, 10, 11, 12, 13, 14, 15])
```

ここで示すスライス操作は配列要素の取り出し操作で MATLAB 系言語でお馴染の添字操作です。例では最初に NumPy パッケージから函数 `arange()` の読みを行い、それから名前 `a` で参照される NumPy の一次元配列を生成してメソッド `reshape()` で 2×8 の配列へと大きさを変換しています。この配列は 2 次元で、話を簡単にするために 2×8 の行列として話を進めましょう。さて、それから二つの拡張スライス操作 ‘`a[0, ...]`’ と ‘`a[1, ...]`’ を行っていますが、これらの処理は行列の一行目と二行目の成分に相当する配列の取り出

しを行っています。つまり、

$$a = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \end{pmatrix}$$

に対して

$$\begin{aligned} a[0, \dots] &\Rightarrow \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{pmatrix} \\ a[1, \dots] &\Rightarrow \begin{pmatrix} 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \end{pmatrix} \end{aligned}$$

という処理です。この拡張スライス操作に現われるリテラル ‘...’ は該当する添字の取り得る値の全てを意味する MATLAB の記号 ‘:’ に相当する省略記号で、この記号が Ellipsis です。ただし、Python や MATLAB の記号 ‘:’ は 1 次元の添字の省略記号ですが、この Ellipsis は一次元に限定されず、Yorick の「ゴム添字 (rubber index)」のように多次元である点で大きく異なります。この Ellipsis は明記された箇所以外の添字が配列の大きさに応じて「省略」されていることを示すオブジェクトで、「何もない」ことを示す None とは異なります。実際、スライス操作で 1 次元的に記号 ‘:’ を用いた操作で始点や終点を省略した表記が可能ですが、実際は None を記載したものと同値です：

```
>>> a = [0, 1, 2, 3, 4]
>>> a = [:4]
[0, 1, 2, 3]
>>> a = [None:4]
[0, 1, 2, 3]
```

このように None が Ellipse が意味する「省略」ではなく、「何もない」ことを意味していることが明瞭になるかと思います。

なお、Ellipsis は Python 2 では式や文を構成するオブジェクトで、独立したオブジェクトではありません。実際、Python 2 のインタプリタ上で ‘...’ のみを入力するとエラーになりますが、Python 3 ではオブジェクトとしてリテラル ‘...’ が許容され、Ellipsis 型のオブジェクトが生成されます。

■数 (numbers.Number): 数リテラルで生成され、算術演算や組込の算術関数から返却されるオブジェクトです。数の EBNF は数リテラルを参照してください。また、オブジェクトとしての数は変更不能のオブジェクトです。この数オブジェクトには整数、浮動小数点数と複素数の型に分類することができます。ここで numbers.Number はモジュール numbers で定義されたクラス Number という意味で、このクラス Number を継承する形で整数、実数と複素数が抽象基底クラスを用いて順序付けられています：

- 整数型 (plain integer)：整数リテラルで生成され、32bit 符号付き整数 (-214783648 から 2147483647 まで) が扱えます。この型のオブジェクトの構築子は int() で、函

数 `type()` で調べると `int` が返却されます。なお、演算結果や整数リテラルの入力が整数型の範囲を越えると自動的に後述の長整数型に切替えられます。このことは構築子 `int()` を使って長整数型のオブジェクトを整数型に変換するときも同様で、整数型の範囲を超えるときは長整数型のオブジェクトが自動的に生成されます^{*33}。

- 長整数型 (`long integer`): 長整数リテラルで生成され、計算機のメモリに依存する任意桁数の整数が扱えます。そして、このクラスのオブジェクトの構築子は `long()` で、函数 `type()` でこの型を調べると `long` が返却されます。なお、Python 3 で整数型は実質的に長整数型に統合され、そのリテラルは整数型のリテラルで、呼び名も整数型 (`int`) になります。また、抽象基底クラスの順序付で、このクラスは整数型と共にクラス `Integral` の具象化クラスになります。
- ブール型 (`boolean`): 真であることを意味する `True` と偽であることを意味する `False` の二つの真理値から構成されます。この型のオブジェクトの構築子は `bool()`^{*34} で、函数 `type()` でこの型を調べると `bool` が返却されます。なお、構築子 `bool()` はリテラル `0`, `None` と `False` をブール型の `False` に変換し、それ以外のほとんどをブール型の `True` に変換します。また、四則演算を含む算術演算で `True` が整数型の `1`, `False` が整数型の `0` として扱われ、これを利用してブール型のオブジェクトとの積をうまく使って `if` 文を使わない式の記述ができます。たとえば `a, b` が数オブジェクトのときに式 ‘`a * True + b * False`’ の結果は `a` になります。この処理は MATLAB 系の言語で `if` 文を減らすことによって言語のオーバヘッドを減らし、処理の高速化を図るために使われます。
- 実数型 (`numbers.Real`): 浮動小数点数リテラルで生成される倍精度の浮動小数点数の型です。Python は单精度の浮動小数点数の型を数リテラルに持ちません。この型のオブジェクトの構築子は `float()` で、函数 `type()` で調べると `float` が返却されます。なお、構築子 `float()` で変換できるリテラルは整数、長整数、浮動小数点数とブール型で、ブール型の `True` は `0.0`, `False` は `1.0` に変換されます。また、特殊な数として無限大 (`'inf'`) と非数 (`'NaN'`) も `float('inf')` や `float('nan')` で生成できます。なお、抽象基底クラスによる順序付では、このクラスはクラス `Real` の具象化クラスとされ、クラス `Real` はクラス `Complex` の子クラスになります。
- 複素数型 (`numbers.Complex`): 浮動小数点数リテラルと虚数リテラルを演算子 “`+`” で結合することで生成されます。構築子は `complex()` で、函数 `type()` で調べると `complex` が返却されます。また、複素数 `z` の実部は `z.real`, 虚部は `z.imag` で取り出

*33 数値処理言語によっては補数表現のために正負が入れ替わることがありますが、Python では自動的に長整数型に切り替わります。なお、Python 3 では整数リテラルを持つ長整数型に統合されます。なお、抽象基底クラスの順序付で、このクラスは長整数型と共にクラス `Integral` の具象化クラスです。

*34 Bool 型は 19 世紀の英国の数学者 George Boole(1815-1864) に由来しますが、「Bool」となぜか最後の“e”省略で表記されます。

せますが、複素数型の性質上、これらは実数型、すなわち、倍精度の浮動小数点数です。そのために近似の数であることに注意が必要で、そのこともあって虚部が 0.0 の複素数を構築子 `int()`, `long()` や `float()` で変換することができません。なお、抽象基底クラスによる順序付では、このクラスはクラス `Complex` の具象化クラスで、クラス `Complex` がクラス `Number` の子クラスです。

SageMath では高速処理のために、これらの数のクラスは全て SageMath の数のクラスで置換えられます。詳細は §5 を参照して下さい。

■列 (sequence): 列は自然数 N からオブジェクトへの対応があるオブジェクトです。すなわち、列 S はある自然数 $N \in \mathbb{N}$ をその上限とし、自然数 $i \leq N$ に対してオブジェクト $S(i)$ を対応させ、0 が列の先頭、 N が列の末尾に対応し、列の濃度 $N + 1$ を「**列の長さ**」と呼びます。Python の列の長さは函数 `len()` で調べられ、成分の指定は ‘`a[2]`’ のように列の名前の直後の括弧 “[]” の中に対応すべき整数を与えることで行います。

列に対する特殊な操作に「**スライス操作**」と呼ばれる部分列の生成操作があります。この操作は MATLAB 系言語でお馴染の操作で、長さ n の列 `a` に対して $i < j$ を充す二つの正整数 $i, j \in \{0, \dots, n - 1\}$ を添字として `a[i : j]` で列 `a` の $i + 1$ 番目から j 番目の成分を持つ部分列を生成します。列の型によっては「**拡張スライス操作**」と呼ばれる刻幅指定のスライス操作が行えることもあります。たとえば、文字列 ‘123456789’ に対して刻幅 2 で先頭の文字から 8 番目までの文字で構成される部分列を取り出すときには ‘123456789’[0:8:2] で部分列 ‘1357’ を取り出することができます。つまり、[0:8:2] によって 0 から 8 までの間隔 2 の自然数の列 0, 2, 4, 6 が生成され、これらの自然数に相当する位置の文字が文字列 ‘123456789’ から取り出された部分文字列 ‘1357’ になります。また、列を逆向きに取り出すときは刻幅を負の整数にします。たとえば ‘123456’[5:2:-1] では 5 から開始して 2 を越えない間隔を-1 とする自然数の列 5, 4, 3, 2 に対応する文字列 ‘6543’ になります。この拡張スライス操作で用いられる文字リテラル “...” は Ellipsis 型のオブジェクトへの参照になります。また、スライス操作で添字の省略もできます。これは添字が列の端部を指定するときに限って可能です。たとえばリスト [1,2,3,4] で先頭から 3 成分を取り出したいときには `a[0:3]` と記述しますが、`a[:3]` と記述できます。

列は変更可能な型のオブジェクトと変更不能の型のオブジェクトの二種類に分類され、変更可能なものがリスト型と `ByteArrays` 型、変更不能なものが文字列型、UNICODE 文字列型とタプル型になります:

- リスト型 (list): 記号 “,” で区切られた任意の Python オブジェクトの列、すなわち、式のリストを角括弧 “[]” で括って生成されるオブジェクトの型です。この型の構

築子は `list()` で、この型のオブジェクトを函数 `type()` で調べると `list` が返却されます。

- **ByteArray 型:** 構築子 `bytearray()` で生成され、0 から 255 までの整数の列をバイナリ形式の列として蓄えられます。
- **文字列型 (string):** 接頭辞 ‘u’, ‘U’ を含まない文字列リテラルで生成される型です。構築子は `str()` で、函数 `type()` でオブジェクトを調べると `str` が返却されます。
- **UNICODE 文字列型:** 接頭辞 ‘u’, ‘U’ を含む文字列リテラルから生成される型です。構築子は `unicode()` で、この型のオブジェクトを函数 `type()` で調べると `unicode` が返されます。
- **タプル型 (tuple):** Python オブジェクトと記号 “,” を区切記号として並べた列、つまり、式のリストの型がタプルです。ただし、実用上、タプルの領域を明確にするためにリストの角括弧 “[]” ではなく、丸括弧 “()” で括ります。また、成分が一つのタプルは特に「**シングルトン (singleton)**」と呼ばれます。この型の構築子は `tuple()` で、この型のオブジェクトを函数 `type()` で調べると `tuple` が返されます。

■**集合 (set):** 有限個のオブジェクトから構成されるオブジェクトです。波括弧 “{ }” で成员を囲った書式になります。成员の間には順序や対応付けを持たないために列や対応付け集合と異なり、添字を用いた書式で成分の取出や参照ができません。なお集合の濃度は函数 `len()` で調べられます：

- **集合型 (set):** 変更可能な型で構築子 `set()` で生成されます。
- **FrozenSet 型:** 変更不能な型で構築子 `frozenset()` で生成されます。集合型と違って要約可能 (hashable) なために別の集合の成分や辞書の鍵にすることができます。

■**連想配列 (mapping):** LISP の「**連想リスト (association list, a-list)**」に相当するオブジェクトです。通常の列の成分取出は列 S の順位を表現する 0 から開始する自然数の部分集合を添字として指定することで行いますが、連想配列では添字集合が自然数の部分集合だけではなく Python の有限個のオブジェクトの集合とすることが可能、より簡単に言うならば添字として自然数以外のオブジェクトが使える配列です。また、連想配列 A の参照は添字の集合 I のときに $k \in I$ を使って $A[k]$ で行えます。また、連想配列の濃度は列と同様に函数 `len()` で調べられます。現時点で Python に組込まれている連想配列は「**辞書 (dict) 型**」のみです：

- **辞書型 (dictionary):** 変更可能な型で構築子は `dict()` です。また、添字集合のことを「**鍵 (キー)**」、添字集合の元を「**鍵値**」と呼びます。なお、辞書は名前空間の実装で用いられています。

■呼出可能 (callable): 呼出可能なオブジェクトの総称です。ここで呼出とは、形式的に名前等を函数の書式で扱うことで新たなオブジェクトの生成や既存のオブジェクトに対する処理操作のことです。呼出可能なオブジェクトかどうかは組込函数 `callable()` で判断できます。実装方法と挙動の違いから呼出可能のオブジェクトには以下のものがあります：

- ユーザ定義函数型 (user-defined function): 利用者による通常の函数定義から生成されるオブジェクトです。ここで「通常」は `yield` 文を内部に包含しないという意味で、`yield` 文を包含する利用者定義の函数は「**生成函数型**」になります。ユーザ定義函数型のオブジェクトの呼出は函数定義で用いた引数の列と同じ長さ^{*35}の列を引数にして行われ、任意の属性の設定や取得ができます。

```
>>> def count(x):
...     return x + 1
...
>>> count(1)
2
>>> callable(count)
True
>>>
```

- ユーザ定義メソッド型 (user-defined method): クラス、クラスインスタンス、それと `None` を一次語とし、記号“.”で任意の呼出可能なオブジェクトと結合させることで生成されるオブジェクトです。ここでの説明は `neko` という名前のクラスがあって、そのインスタンスマソッドに `CatchMouse()` があるときにそのインスタンスが `tama` であれば、一次語 `tama` と記号“.”との結合 `tama.CatchMouse()` でインスタンスマソッド `CatchMouse()` を呼び出せるという意味です。

ユーザ定義のメソッドの簡単な例を示しておきます：

```
>>> class TEST(object):
...     def __init__(self, val):
...         self.val = val
...     def dbl(self):
...         return self.val * 2
...
>>> a = TEST(3)
>>> a.dbl()
6
>>> callable(TEST)
```

^{*35} 引数の個数を函数のアリティ (arity) と呼びます。

```
True
```

```
>>>
```

メソッドの呼出はインスタンスの名前を一次語として記号“.”に続けてメソッド名を記載することで行えます。このときにメソッドの定義で用いた引数 `self` は除きます。なお、メソッド `__call__()` を用いると、インスタンスを函数と同様の書式の記載で、このメソッドの呼出が可能になります：

```
>>> class TEST(object):
...     def __init__(self, start):
...         self.count = start
...     def __call__(self):
...         count = self.count
...         self.count = count + 1
...         return count
...
>>> a = TEST(0)
>>> a()
0
>>> a()
1
>>> a.__call__()
2
>>> callable(TEST)
True
>>>
```

この例ではメソッド `__init__()` でインスタンス変数 `count` の初期化を行い、メソッド `__call__()` を実行すると `count` の値が `+1` されるというものです。このクラスのインスタンス `a` に対して ‘`a()`’ と函数の書式でメソッド `__call__()` の呼出が行われています。

- 生成函数型 (generator function): 内部に `yield` 文を持つ利用者定義の函数です。

```
>>> def genos(start):
...     count = start
...     while True:
...         yield count
...         count = count + 1
...
>>> a = genos(1)
>>> next(a)
1
>>> next(a)
```

```

2
>>> type(a)
<type 'generator'>
>>> callable(genos)
True
>>>

```

yield 文を有する函数で生成されるオブジェクトの型は generator で、函数 next() で生成型のオブジェクトから次の値を取り出すことができます。この生成函数と同様の処理をメソッドで行うことも可能で、そのクラスのインスタンスが次に述べる反復子型になります。

- 反復子型 (iterator): メソッド `__iter__()` とメソッド `__next__()` の一対を持つ利用者定義のクラスです:

```

>>> class TEST(object):
...     def __init__(self, start):
...         self.count = start
...     def __iter__(self):
...         return self
...     def next(self):
...         count = self.count
...         self.count = self.count + 1
...         return count
...
>>> a = TEST(1)
>>> next(a)
1
>>> next(a)
2
>>> callable(TEST)
True

```

- 組込函数型 (built-in function): 組込函数オブジェクトは C の函数のラッパーです。このような函数の例に組込函数 `dir()` や 函数 `math.sin()`^{*36}があります。
- 組込メソッド型 (built-in method): 組込函数を隠蔽したもので、C の函数に引き渡されるオブジェクトを何らかの非明示的な外部引数として持ちます。
- クラスタイプ型 (class type): クラスタイプ型のオブジェクトはインスタンスオブジェクトの生成で用いられ、このときに「**ファクトリクラス**」として振舞います。ここでメソッド `__new__()` の上書 (override) を行っても問題はありません。クラスを呼出したときの引数はメソッド `__new__()` に引渡され、このメソッドがクラ

^{*36} 標準モジュール `math` に包含される函数 `sin()` を指示する名前になります。

スタイル型のオブジェクトを返却するときにインスタンスオブジェクトの初期化メソッド`__init__()`に引数が渡されます。

- 古典的クラス型 (classic class): 呼出されたときに新たに「**クラスインスタンス型**」のオブジェクトが生成されますが、このオブジェクトはクラスタイプ型から生成されるインスタンスオブジェクトと別の型です。この呼出で用いられる引数はメソッド`__init__()`に引渡されるため、このクラスにメソッド`__init__()`がないときはクラスを引数なしで呼び出さなければなりません。
- クラスインスタンス型 (class instance): 古典的クラスのクラスにメソッド`__call__()`があるときに限って呼出可能型になります。

■モジュール (module): Python の文を記載したファイルを import 文で読込むことで生成されるオブジェクトです。インタプリタに読み込まれると、Python の文はコードオブジェクトに翻訳されて pyc ファイルに蓄えられます。このコードオブジェクトの実体はバイトコードと呼ばれるバイナリデータで、入力式や文と比べて計算機が処理し易い書式になっており、このバイトコードを Python 仮想計算機 (PVM) が処理します。また、複数のモジュールに階層構造を入れて管理できるようにしたものを「**パッケージ**」と呼びます。このパッケージの読み込みで生成されるオブジェクトの型も module 型です。

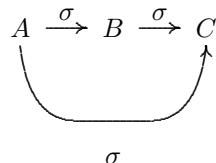
■クラス (class): Python 2 には古典的な「**クラスオブジェクト型**」と新しい様式の「**クラスタイプ型**」の二種類のクラスがあり、どちらも辞書で実装された名前空間を持ち、オブジェクトや各属性への参照で用いられます。なお、Python 3 はクラスタイプ型のみです。古典的クラス型は Python 2 で基底クラスとして object を継承しないときに生成され、クラスタイプ型と違ってメタクラスが扱えず、引数なしにメソッド super() で基底クラスへの属性等の参照ができません:

```
>>> class TEST1:  
...     pass  
...  
>>>  
>>> class TEST2(object):  
...     pass  
...  
>>>  
>>> type(TEST1)  
<type 'classobj'>  
>>> type(TEST2)  
<type 'type'>  
>>> a=TEST1()  
>>> b=TEST2()  
>>> type(a)
```

```
<type 'instance'>
>>> type(b)
<class '__main__.TEST2'>
```

ここで古典的クラスは函数 `type()` で `classobj` と表示されるクラスオブジェクト、そのインスタンスは函数 `type()` で `instance` と表示されるインスタンスオブジェクトです。ここで `object` を継承したクラスは函数 `type()` で `type` と表示されるクラスタイプ型で、そのインスタンスは `instance` ではなく、そのクラスのインスタンスになります。ここで `type(b)` の結果で '`__main__.TEST2`' とクラス名を含む文字列が返却されていることに注目して下さい。なお、函数 `type()` は動的にクラスタイプ型のオブジェクトが生成できます。

クラス間の関係で、継承する側のクラスを「**派生クラス**」、継承される側のクラスを「**基底クラス**」と呼びます。基底クラスは派生クラスよりも普遍的、すなわち上位のクラスです。この継承関係を親子関係にたとえるならば、基底クラスが「**親**」、継承する派生クラスが「**子**」にそれぞれ対応します。さらに、この継承関係を(素朴集合論の)集合の包含関係として捉えることも可能で、そのときに派生クラスが「**部分集合 (subset)**」にする「**サブクラス**」、基底クラスを「**スーパークラス**」と呼びます。さて、あるインスタンスがとあるクラスに包含されることと、あるクラスがとあるクラスを継承しているといった関係を矢で表現してみましょう。つまり、 $C_1 \xrightarrow{\sigma} C_0$ でクラス C_1 がクラス C_0 を継承したもの、すなわち C_0 の派生クラスであることを示します。このときに次の図式が可換になります：



この図式は3個のクラスの派生関係を示し、 $A \xrightarrow{\sigma} B$ かつ $B \xrightarrow{\sigma} C$ であれば $A \xrightarrow{\sigma} C$ となること、つまり、関係 $\xrightarrow{\sigma}$ は結合律を充します。ただし、反射律 $A \xrightarrow{\sigma} A$ は、クラス定義で自分自身が未定義の状態で自分自身を用いる循環的定義が許容されません*³⁷が、既存のクラスを上書きする形で自分自身を定義することは可能です。だから、反射律は循環的定義として不可でも、既存の類定義の更新として可能です。それから、下位のクラスの属性は新たにクラス変数に値の束縛を行うことで、そのクラスの属性値が更新されますが、上位の基底クラスの属性に遡及して値が更新されることはありません。逆に上位のクラスの属性を変更すると下位のクラスの属性の変更が生じます。

*³⁷ 参照すべきオブジェクトがなければ例外が発生します。

クラスには**特殊属性 (special attribute)** と呼ばれる属性があります。属性 `__name__` にクラス名, `__module__` にクラスを定義したモジュール名, `__dict__` にクラスの名前空間が入った辞書, `__bases__` に基底クラス名を収納したタプル, そして, `__doc__` にクラスを説明する文書文字列が束縛されています。

■クラスインスタンス: Python 3 で廃止された Python 2 の対象です。このクラスインスタンスは古典的クラスの呼出で生成される対象で、辞書で実装された名前空間を持ってるため最初の属性参照はここから開始します。属性参照にて辞書内で属性が見当らないものの、インスタンスのクラスに該当する属性名があるとき、そのインスタンスが含まれるクラス属性に検索領域が広げられます。このときの検索順序は基底クラスのタプルで、その左側のクラスから右側のクラスへと検索が行われます (MRO)。また、属性の代入や削除によってインスタンスの辞書が更新されますが、それに伴なうクラス辞書の更新はありません。

■ファイル (file): 開かれたファイルを表現するオブジェクトです。このファイルは組込函数 `open()`, `os.popen()`, `os.fdopen()`、および、`socket` オブジェクトのメソッド `makefile()` 等で生成されます。

■内部型 (internal type): インタプリタがその内部で用いる型で、これらの型の幾つかは利用者に公開されています。この内部型はインタプリタの仕様変更等で将来の変更が生じる可能性があります：

- **コードオブジェクト (code object):** 「バイトコード (bytecode)」を表現するオブジェクトで函数 `type()` で調べると `code` を返します。このバイトコードは式や函数のインタプリタ内部での表現で、バイナリ形式のオブジェクトです。Python 仮想計算機 (PVM) は入力された式や文を直接処理するのではなく、それらが翻訳されたバイトコードを処理しています。たとえば、函数のバイトコードはオブジェクトの属性 `__code__` に束縛されています。組込函数 `compile()` で Python の式をバイトコードに変換し、組込函数 `exec()` を使って処理することができます：

```
>>> a = compile('sum([1,2,3,4,5])', '', 'single')
<code object <module> at 00000000036E2A30, file "", line 1>
>>> type(a)
<type 'code'>
>>> exec a
15
>>> def pet():
...     print "dog and ", cat
...
>>> pet.__code__
```

```
<code object pet at 0000000003615130, file "<stdin>", line 1>
>>> type(pet.__code__)
<type 'code'>
>>> exec(pet.__code__, {'cat' : 'dog'})
cat and dog
```

ここでは式 ‘sum([1, 2, 3, 4, 5])’ を函数 compile() でバイトコードに変換したものと定義した函数 pet() の属性 __code__ に束縛されたバイトコードを函数 exec() で実行させています。このバイトコードの中身はモジュール dis の函数 dis() を使って確認できます：

```
>>> import dis
>>> dis.dis(a)
 1           0 LOAD_NAME               0 (sum)
 3 LOAD_CONST            0 (1)
 6 LOAD_CONST            1 (2)
 9 LOAD_CONST            2 (3)
12 LOAD_CONST            3 (4)
15 LOAD_CONST            4 (5)
18 BUILD_LIST             5
21 CALL_FUNCTION          1
24 PRINT_EXPR
25 LOAD_CONST            5 (None)
28 RETURN_VALUE

>>> dis.dis(pet)
 2           0 LOAD_CONST            1 ('dog and ')
 3 PRINT_ITEM
 4 LOAD_GLOBAL            0 (cat)
 7 PRINT_ITEM
 8 PRINT_NEWLINE
 9 LOAD_CONST            0 (None)
12 RETURN_VALUE
```

ここで表示されていることは PVM に引き渡す指示で、PVM はスタックマシンと呼ばれる与えられた命令を積み重ね (stack) て、それらの命令を逐次解釈する仕組みです。

import 文等でインタプリタに読み込まれたプログラムはバイトコードに翻訳され、そのバイトコードは ‘.pyc’ ファイルに蓄えられます^{*38}。一度 ‘.pyc’ ファイルが生成されると以後の import 文の読み込みで既存の ‘.pyc’ ファイルが利用されま

^{*38} バイトコードの詳細は次を参照のこと： https://www.ics.uci.edu/~brgallar/week9_3.html

す。そのため ‘.pyc’ ファイルの更新は函数 `reload()` によるファイルの再読込み ‘`python -m < ファイル名 ’` で ‘.pyc’ ファイルの更新を行う必要があります。また、バイトコードは異なる Python 仮想マシンでの互換性を保証するものではありません。

- **フレームオブジェクト (frame object):** 「実行フレーム (execution frame)」を表現するオブジェクトです。プログラム実行時に呼び出し可能 (callable) なオブジェクトが呼出され、そのオブジェクトの実行が終えた時点で呼出した側のプログラムで呼出前の実行状態を保存しているオブジェクトから取り出す仕組になっています。このときの実行状態が保存されているときのオブジェクトをフレームオブジェクトと呼びます。
- **トレースバックオブジェクト (traceback object):** 「例外スタックトレース (exception stacktrace)」を表現するオブジェクトです。この `traceback` オブジェクトは例外が発生した時点で生成され、例外ハンドラを検索してスタックを戻すときに、その戻ったレベル毎にトレースバックオブジェクトが、その時点の `traceback` の前に挿入されます。例外ハンドラに入るとスタックトレースをプログラムで利用できるようになります。
- **スライスオブジェクト (slice object):** 「拡張スライス構文」を表現するために用いられるオブジェクトです。この構文は MATLAB 系言語で用いられるベクトル成分の取出に類似し、配列処理で類似の機能を与えます。
- **静的メソッドオブジェクト (staticmethod object):** 組込の構築子 `staticmethod()` やデコレータ@`staticmethod` を使って生成されるオブジェクトです。インスタンスマソッドと違いインスタンス化しなくても利用可能です。また、メソッドの定義はクラスメソッドと異なり引数としてクラス自体 ‘cls’ を取りません。クラス変数の参照ではクラスやクラス変数を直接呼び出すことで行います。またこのことから派生クラスで静的メソッドが定義されたクラスのクラス変数の参照が行われます。このように「静的」には「**継承において動的なクラス変数の参照が行われない**」という意味があります。なお、静的メソッドはインスタンス変数を参照できません。
- **クラスメソッドオブジェクト (classmethod object):** 組込の構築子 `classmethod()` やデコレータ@`classmethod` を使って生成されるオブジェクトで、`staticmethod` と同様にインスタンス化しなくとも利用できます。メソッドの定義では第一引数に ‘cls’ を必要とし、そのため継承で静的ではなく動的にクラス変数の参照が行われますが、静的メソッドと同様にインスタンス変数の参照はできません。

静的メソッドとクラスメソッドの違いについて

静的メソッドとクラスメソッドの相違点を明確にするために簡単な例を以下に示しておきます:

```
>>>class A(object):
...     neko = 'mike'
...     def sm(x):
...         print "%s: neko = %s %s"%(A, A.neko, x)
...     def cm(cls,x):
...         print "%s: neko = %s %s"%(cls, cls.neko, x)
...     static_method = staticmethod(sm)
...     class_method = classmethod(cm)
>>>class B(A):
...     neko = 'tama'
>>>A.static_method(1)
<class 'A'>: neko = mike 1
>>>A.class_method(1)
<class 'A'>: neko = mike 1
>>>B.static_method(1)
<class 'A'>: neko = mike 1
>>>B.class_method(1)
<class 'B'>: neko = tama 1
```

この例ではクラス A とクラス A の派生クラスのクラス B を生成しています。さらにクラス A に静的メソッド static_method() とクラスメソッド class_method() を定義していますが、‘cls’ を静的メソッドは引数として取らず、クラスやクラス変数への参照ではメソッド内部でそれらを直接呼び出しています。それに対してクラスメソッドでは第一引数に ‘cls’ を取り、クラスやクラス変数への参照は ‘cls’ を介して行います。そして両者の結果の相違は定義したクラス A ではありませんが、派生クラス B で両者に違いが出てきます。まず、‘B.static_method(1)’ でクラス変数 neko の値は静的メソッドが定義されたクラス A のものが用いられています。またクラスも A のままであるが、クラスメソッド ‘B.class_method(1)’ ではクラスが B に切り替わっており、それに伴ってクラス変数の値もクラス B のものになります。

ここで定義では組込函数 classmethod() と staticmethod() を用いましたが、「**デコレーター (decorator)**」を使えば以下のように静的メソッドやクラスメソッドを定義することができます:

```
>>>class A(object):
...     neko = 'mike'
```

```
...     @static_method
...     def static_method(x):
...         print "%s: %s %s"%(A, A.neko, x)
...     @class_method
...     def class_method(cls,x):
...         print "%s: %s %s"%(cls, cls.neko, x)
```

このように静的メソッドやクラスメソッドの定義の直前にデコレータを配置することで定義できます。

3.8 特殊メソッド

3.8.1 特殊メソッドの概要

特殊メソッドは特定の演算や機能をクラスに実装するためにあらかじめ用意された一群のメソッドです。これはメソッドの「**上書き (override)**」を利用し、派生クラスに含まれない特殊メソッドは基底クラス側のものがそのまま用いられ、派生クラスで特殊メソッドを新たに定義することで基底クラスから継承されたメソッドの上書きが行われます。その結果、派生クラスで基底クラスのものと別の改変されたメソッドが利用できます。たとえば、同じクラスのオブジェクトの比較で**「拡張比較」**と呼ばれる一群のメソッドは、これから定義するクラスの二つのインスタンスが等しいかどうか、同一クラスのインスタンス間の大小関係を判断するメソッドで、これらのメソッドを上書きすることで整数や実数で用いられる等号“==”とその否定“!=”，あるいは大小関係(“<”，“>”，“<=”，“>=”)に新たな意味を持たせられます。ただし、ここでの拡張比較のメソッドは相反するメソッドのどちらか一方を上書きすることでもう一方が自動的に再定義されるものではありません。たとえば、等号“==”の否定は“!=”ですが、等号“==”を再定義しても、その否定の“!=”が自動的に定義されるものではありません。

以下、項目別に特殊メソッドを列記しますが、クラス名を‘cls’、オブジェクト名を‘obj’、それからメソッドの引数をまとめて‘args...’と表記します。また、メソッドの引数に‘[，args...]’とある場合は鉤括弧“[]”で括った引数がオプションであることを指示します。また、二項演算子の引数は同一クラスに属するオブジェクトが二つ必要になりますが、引数として一つは前述の‘self’、もう一つは‘other’を用います。そして、‘cls’、‘self’と‘other’は‘obj’と異なり固定です。

3.8.2 オブジェクトの生成と削除に関連するもの

ここで述べるメソッドは構築子と消却子に相当します。

■`obj.__new__(cls [, args...])` クラスのインスタンス生成で呼び出される静的メソッドです。インスタンス生成が要求されているクラス名 `cls` を第一引数に取り、残りのオプションの引数をメソッド `__init__()` に引渡してオブジェクトの初期化を行います。このメソッド `__init__()` と併せて C++ の構築子に似た動作をします。なお、メソッド `__new__()` がクラス `cls` のインスタンスを返却するときに限ってメソッド `__init__()` が呼出されます。

■`obj.__init__(self [, args...])` オブジェクトの初期化に関わるメソッドで、前述のメソッド `__init__()` と併せて C++ の構築子と同様の働きをします。ここで基底クラスがメソッド `__init__()` を持つときに、その派生クラスのメソッド `__init__()` がインスタンスの基底クラスで定義されている部分が初期化されるようにする必要があります。なお「**構築子は値を返却してはならない**」という制約があるためにメソッド `__init__()` がこの制約に反して値を返すようにしていると実行時に `TypeError` が発生します。

■`obj.__del__(self)` C++ の「**消滅子 (destructor)**」に似た動作を行うメソッドで、オブジェクトを削除するときに呼出され、引数は `self` 以外はありません。なお、Python のオブジェクトは到達不可能になった時点から回収され、C++ の消滅子のように不要になった時点ではありません。一般的に基底クラスがメソッド `__del__()` を持っているときは派生クラスのメソッド `__del__()` で明示的に基底クラスのメソッド `__del__()` を呼出してオブジェクトを消去するようにしなければなりません。

3.8.3 値の表示に関するもの

利用者に適切なオブジェクトの持つ属性や値といった情報が表示されるようにするためのメソッドです。これらのメソッドの引数は `self` のみです。

■`obj.__repr__(self)` 組込函数 `repr()` や文字列への変換時に呼出され、オブジェクトをフロントエンド側で表示する「**公式の表示**」の生成を行います。この章の有理数の定義にて整数対を有理数として表示する例のように、利用者にとって分かり易い表示や必要とされる情報の表示に変更できます。

■`obj.__str__(self)` 組込函数 `str()` と `print` 文^{*39}から呼出され、オブジェクトをフロントエンド側で表現する**非公式の文字列**の生成を行います。このメソッドの返却値は文字列オブジェクトでなければなりません。なお、メソッド `__repr__()` が定義されていれば、このメソッドが定義されていなくてもメソッド `__repr__()` が利用されます。

^{*39} 3.x では函数 `print()`.

3.8.4 値の比較に関するもの

クラスに「大小関係」を導入するためのメソッドで、これらの演算子は当然のことながら演算子を被演算子の間に配置する中值表現の二項演算子です：

比較の演算子			
演算	記号	Python の式	特殊メソッド
小なり	<	a < b	obj.__lt__(self, other)
以下	<=	a <= b	obj.__le__(self, other)
等しい	==	a == b	obj.__eq__(self, other)
等しくない	!=	a != b	obj.__ne__(self, other)
大なり	>	a > b	obj.__gt__(self, other)
以上	>=	a >= b	obj.__ge__(self, other)

これらの演算子の書き換えは他の演算子に影響しません。すなわち、大小関係を新たに定義したクラスに導入するために演算子“ \geq ”を定義しても、その否定として演算子“ $<$ ”が自動的に定義されません。大小関係の書き換えが一部でも生じた場合は全体を通して再定義が必要ですが、大小関係の演算子“ $<$ ”, “ $>$ ”, “ \leq ”, “ \geq ”の何れか一つと同値の演算子“ $==$ ”に相当するメソッドを定義していれば高階機能モジュール `functools` の函数 `functools.total_ordering()` を使って残りの比較演算子の自動定義が可能です。また、これらの演算子を書き換えていないときに用いられる比較の特殊メソッドが次の `obj.__cmp__()` です。

■`obj.__cmp__(self, other)` 二つの同一クラスのインスタンスを比較するときにインスタンスの識別子(整数型)を使って大小関係の判断ができます。

3.8.5 整数演算に関するもの

整数オブジェクトの二項演算に関する特殊メソッドを以下にまとめておきます：

整数演算子(二項演算子)

演算	記号	Python の式	特殊メソッド
和	+	a + b	obj.__add__(self, other)
差	-	a - b	obj.__sub__(self, other)
積	*	a * b	obj.__mul__(self, other)
商	/	a / b	obj.__truediv__(self, other)
商	//	a // b	obj.__floordiv__(self, other)
剰余	mod	a mod b	obj.__mod__(self, other)
冪	**	a ** b	obj.__pow__(self, other)

ここで Python 2. で演算子 “/” の二つの被演算子が整数のときに結果は整数であり, Python 3 では演算子 “//” が相当し, Python 3 の演算子 “/” は割り切れないときに浮動小数点数になります.

これらの整数演算に加えて整数オブジェクトには, 被演算子を2進数として表現したときの二項演算も加わります:

2進数整数演算子(二項演算)

演算	記号	Python の式	特殊メソッド
左シフト	<<	a << b	obj.__lshift__(self, other)
右シフト	>>	a >> b	obj.__rshift__(self, other)
論理積	&	a & b	obj.__and__(self, other)
排他論理和	^	a ^ b	obj.__xor__(self, other)
論理和		a b	obj.__or__(self, other)

3.8.6 浮動小数点数演算に関するもの

浮動小数点数オブジェクトの演算に関するメソッドを以下に示しておきます.

浮動小数点数演算子(二項演算子)

演算	記号	Python の式	特殊メソッド
和	+	a + b	obj.__radd__(self, other)
差	-	a - b	obj.__rsub__(self, other)
積	*	a * b	obj.__rmul__(self, other)
商	/	a / b	obj.__rtruediv__(self, other)
商	//	a // b	obj.__rfloordiv__(self, other)
剰余	mod	a mod b	obj.__rmod__(self, other)
幕	**	a ** b	obj.__rpow__(self, other)

これらの演算子は被演算子のどちらか一方が浮動小数点数のときに浮動小数点数を返します。演算子“/”は単純な割算の演算子ですが、演算子“//”は商の整数部分のみを浮動小数点数の型で返却する演算子で、演算子“/”を除く演算子は被演算子が整数型のときの演算子を自然に拡張した演算になっています。

3.8.7 累算算術代入演算に関するもの

累算算術代入演算子は二項演算子の一つで、演算子左辺の変数に演算子右辺の変数との計算結果を代入する演算子です。たとえば‘x += y’は‘x = x + y’と同値の表記で、演算子“+=”から指示される演算‘x + y’を実行し、演算子左辺の変数 x に代入するという操作になります。これら累算算術演算代入の特殊メソッドを以下にまとめて示しておきます:

累算算術代入演算子(二項演算子)

同値な式	記号	Python の式	特殊メソッド
a = (a + b)	+=	a += b	obj.__iadd__(self, other)
a = (a - b)	-=	a -= b	obj.__isub__(self, other)
a = (a * b)	*=	a *= b	obj.__imul__(self, other)
a = (a / b)	/=	a /= b	obj.__itruediv__(self, other)
a = (a // b)	//=	a // b	obj.__ifloordiv__(self, other)
a = (a mod b)	mod=	a mod= b	obj.__imod__(self, other)
a = (a ** b)	**=	a **= b	obj.__ipow__(self, other)

累算算術代入演算子では本来の演算子が演算子を構成する文字“=”の左側にあり、被演算子は整数、浮動小数点数といった数オブジェクトの型の制約はありません。

2進数累算算術代入演算子(二項演算)

同値な式	記号	Pythonの式	特殊メソッド
$a = (a <= b)$	$<=$	$a <= b$	<code>obj.__ilshift__(self, other)</code>
$a = (a >= b)$	$>=$	$a >= b$	<code>obj.__irshift__(self, other)</code>
$a = (a \& b)$	$\&=$	$a \&= b$	<code>obj.__iand__(self, other)</code>
$a = (a ^ b)$	$^=$	$a ^= b$	<code>obj.__ixor__(self, other)</code>
$a = (a b)$	$ =$	$a = b$	<code>obj.__ior__(self, other)</code>

2進数累算算術代入演算子も同様で、本来の演算子が演算子を構成する文字“=”の左側に現れます。

3.8.8 単項演算子

単項演算子では整数、浮動小数点数のメソッドの区別はありません：

単項演算子			
演算の意味	記号	Pythonの式	特殊メソッド
$a \rightarrow -a$	-	-a	<code>obj.__neg__(self)</code>
$a \rightarrow +a$	+	+a	<code>obj.__pos__(self)</code>
$a \rightarrow a $	<code>abs()</code>	<code>abs(a)</code>	<code>obj.__abs__(self)</code>
$a \rightarrow \tilde{a}$	\sim	$\sim a$	<code>obj.__invert__(self)</code>

`abs()`以外はオブジェクト名の左側に演算子を配置します。このときにSpaceやTABといった空白文字が演算子と被演算子の間にあっても問題ありません。ここで \tilde{a} 、すなわち、 $\sim a$ は a の二進数表現の全てのビットの反転を意味します。

3.8.9 属性値の取得と設定に関連するもの

属性値の取得や変更、削除について特殊メソッドをクラスに定義することで新たな意味付けを行うことができます。ここで説明するメソッドに現れる変数名で`name`を属性名、`value`をその値とします：

■`obj.__getattr__(self, name)` 属性の検索において`self`のインスタンス属性やクラスツリーでも検出されなかったときに呼出されます。このメソッドは計算された属性値か例外`AttributeError`を送出しなければなりません。なお、新スタイルクラスで実際に完全な制御を行う方法はメソッド`__getattribute__()`を参照してください。

■`obj.__setattr__(self, name, value)` 属性への値の束縛で呼出される特殊メソッドです。ここで `name` が属性名, `value` がその属性値です。なお、インスタンス側の属性に値を束縛させるときに ‘`self.name = value`’ とした場合は自己参照が生じるために行つてはなりません。そうではなく ‘`self.__dict__[name] = value`’ のようにインスタンス側の辞書に値を追加します。

■`obj.__delattr__(self, name)` 属性に束縛した値の削除を行います。このメソッドの実装は ‘`del obj.name`’ に意味のあるときに限定すべきです。

■`obj.__getattribute__(self, name)` クラス型がクラスタイプのみに対して利用可能なメソッドで、指定した属性の値を返却するメソッドです。なお、メソッド `__getattr__()` が実装されていれば例外 `AttributeError` が送出されない限り呼び出されません。このメソッドの実装では再帰的な呼出を防止するために必要な属性全てへの参照で ‘`obj.__getattribute__(self, name)`’ のように基底クラスのメソッドと同じ属性名で呼び出さなければなりません。

3.8.10 その他

■`obj.__hash__(self)` 要約(ハッシュ)値を返却するメソッドです。要約値は整数値であり、同じ値を持つオブジェクトであればそれらの要約値は一致しなければなりません。つまり、‘`a == b`’ が `True` であるなら ‘`a.__hash__() == b.__hash__()`’ も `True` になります。なお、要約値は整数値になりますが、-1 をエラーフラグとして予約済にしている関係上、内部計算で-1 が得られると-2 を返却する仕様になっています*⁴⁰。実際、整数オブジェクトでメソッド `__hash__()` は基本的にそれ自身を返却しますが、オブジェクトの値が-1 の場合のみ要約値として-2 を返却します。

このメソッドは組込の函数 `hash()`, `set()`, `frozenset()`, `dict()` のような要約値を用いたオブジェクト操作で呼出しが行われます。クラスがメソッド `__cmp__()` と `__eq__()` を持たないときは必ずメソッド `__hash__()` を定義する必要があります。というのもこれらのメソッドは `__hash__` を使って比較を行うためです。逆に比較のメソッド `__cmp__()` と同値性検証のメソッド `__eq__()` が定義されていても、メソッド `__hash__()` が定義されていなければ、そのインスタンスが要約可能 (hashable) にならないために辞書の鍵として使えません*⁴¹。なお、利用者定義のクラスには `__hash__()` メソッドが継承されおり、このメソッドは識別値 `id()` を使って定義されています。ユーザ

*⁴⁰ <http://effbot.org/zone/python-hash.htm> 参照

*⁴¹ より正確には要約可能であるかどうかの判別にメソッド `__hash__()` が呼び出されることで行つていているためです。したがってユーザー定義クラスにて要約値の不変性はプログラマが保証しなければなりません。

定義クラスを非要約可能 (unhashable) にするためには, ‘`__hash__ = None`’ にすることで行えます。また Python 3 では `__hash__` を未定義の状態で `__eq__()` を上書きすることで自動的に ‘`__hash__ = None`’ になります。

■`obj.__nonzero__(self)` 真理値テストや組込演算 `bool()`^{*42} の実現のために呼出されます。このメソッドは真理値の ‘True’(=真) か ‘False’(=偽), あるいはそれらと等価の整数 ‘1’(=真) か ‘0’(=偽) の何れかを返さなければなりません。このメソッドが定義されていないときはメソッド `__len__()` が呼出され, その結果が ‘nonzero’ であれば True, nonzero のときは False です。それからもしもメソッド `__len__()` と `__nonzero__()` の双方が実装されていなければ, そのクラスのインスタンスの真理値は全て ‘True’ とみなされます。

■`obj.__unicode__(self)` 組込函数 `unicode()` を実現するために呼出され, `unicode` オブジェクトを返却しなければなりません。このメソッドが定義されていないときは文字列リテラルへの変換が試みられ, その結果, 既定値の文字エンコードを用いて UNICODE 文字列に変換されます。

3.9 記述子 (descriptor)

ある特定の性質を持つオブジェクトが実装すべきメソッドを「規約 (Protocol, protocol)」と呼びますが, 「記述子」はその規約の一つで, クラスタイプ (`object` や `type` の派生クラス) のみに対応し, 属性の束縛に関わるメソッドです。記述子は「記述子規約 (descriptor protocol)」と呼ばれる 3 個のメソッド `__get__()`, `__set__()` と `__delete__()` の何れかが上書きされ, メソッド `__get__()` と `__set__()` の双方が定義されている「データ記述子」とメソッド `__get__()` のみが定義されている「非データ記述子」の二種類に大きく分類されます。さらにメソッド `__set__()` の呼出で例外: `AttributeError` が送出されるデータ記述子を「読み専用データ記述子」と呼び, この記述子を導入することで属性の管理が可能になります。

記述子の呼出は属性への参照がその基点になります。たとえばオブジェクト `a` に対して属性 `x` の参照は `a.x` で行いますが, このときに `a.x` が基点になります。ここで引数がどのように記述子に結合されるかはオブジェクト `a` がクラスのインスタンスであるか, あるいはクラスそのものであるかに依存します:

^{*42} 何故か `bool` でない!

記述子の呼び出し

- 直接呼出: 最も単純な呼出操作で ‘x.__get__(a)’ に変換されます.
- インスタンス束縛: クラスタイプのインスタンスに対する束縛で ‘a.x’ が ‘type(a).__dict__['x'].__get__(a,type(a))’ に変換されます.
- クラス束縛: クラスタイプのクラスに対する束縛で ‘a.x’ が ‘a.__dict__['x'].__get__(None,a)’ に変換されます.
- スーパークラス束縛: a が super のインスタンスのときに束縛 super(b, obj).m() を行うと最初に a, 次に b に対して obj.__class__.__mro__ を検索し, それから呼出: ‘a.__dict__['m'].__get__(obj,obj.__class__)’ で構築子を呼出します.

インスタンス束縛で構築子の呼出の優先順序は定義内容に依存します。そして構築子は上述の 3 つのメソッドの任意の組合せで定義されますが、ここでメソッド __get__() が定義されていないときに該当する属性の参照が行われると構築子オブジェクト自体が返却されます。前述のようにメソッド __set__ と __delete__ のどちらか一方が定義され、データ構築子、双方が定義されていなければ非データ構築子になります。組込函数 property() はデータ構築子として Python に実装されたもので、このときにインスタンスでは属性の上書きができません。その一方で staticmethod() と classmethod() を含む Python のメソッドは非データ構築子として定義され、そのためにインスタンスでメソッドを再定義され、インスタンスでメソッドの再定義や上書きができます。このことを利用して同じクラスのインスタンスでも個々の挙動に違いを持たせることができます。また属性検索でデータ記述子やインスタンスの属性辞書、非データ記述子の順番で検索が行われます。

3.9.1 記述子 (descriptor) の実装

■obj.__get__(self, instance, owner) クラスの属性、インスタンスの属性の参照で呼出されます。ここで ‘owner’ はオーナークラスで、instance は属性への参照を仲介するインスタンス属性が owner を介して参照されるときは ‘None’ になります。

■obj.__set__(self, instance, value) オーナークラスのインスタンス instance 上の属性を新たな値:value に束縛する際に呼出されます。

■obj.__delete__(self, instance) オーナークラスのインスタンス instance 上の属性を削除する際に呼出されます。記述子は組込函数 property() と似た動作です。

3.10 クラス属性の参照について

クラス属性の参照はクラス名が C で属性が x のときに $C.x$ で行えます。この参照の実体は $C.__dict__["x"]$ で、目的の属性がクラス名で指示したクラスに見当らなければ上位の基底クラスで参照が行われます。ここで属性検索は検索しているクラスになければ継承関係が一つ上のクラスへと遡るという「**継承の深さ**」が関わります。たとえば、 $C_0 \xrightarrow{\sigma} C_1, C_1 \xrightarrow{\sigma} C_2, \dots, C_{n-1} \xrightarrow{\sigma} C_n$ という継承関係からはクラス C_0 から開始してクラス C_n に至るという継承関係の分解図式 (Resolution): $C_0 \rightarrow \dots \rightarrow C_n$ が得られ、この図式に現われるクラスの順に属性やメソッドの検索が行われます。つまり、この分解図式に現われるクラスを左側から並べることで得られたリスト (C_0, C_1, \dots, C_n) の並び順がクラス C_0 の「**MRO(Method Resolution Order)**」と呼ばれるメソッドの検索順序で、 $\mathcal{L}(C)$ と記述します。また、この検索順序を求める処理のことを「**線形化 (linealization)**」と呼びます。この MRO を説明するために幾つかの言葉を定義しておきます。まず、検索順序はクラスのリスト: (C_1, \dots, C_n) で表現されていますが、これを語: $C_1 \dots C_n$ と表記することにします。ここで、リストの先頭にクラス C_0 を追加することは語の先頭に C_0 を追加することに対応し、この追加する操作を $C_0 + C_1 \dots C_n$ と表記します。次に語 $C_0 C_1 \dots C_n$ の先頭 C_0 を取り出す操作を head、先頭以外の残りの $C_1 \dots C_n$ を取り出す操作を tail と表記し、語 L に対して $\underline{L} \stackrel{\text{def}}{=} \text{head}(L), \underline{\underline{L}} \stackrel{\text{def}}{=} \text{tail}(L)$ と略記します。それから語 $B_1 \dots B_m$ と語 $C_1 \dots C_n$ が与えられたとき、語 $B_1 \dots B_m$ に含まれる各 $B_i (1 \leq i \leq m)$ を語 $C_1 \dots C_n$ から取り除いた語を $C_1 \dots C_n \setminus B_1 \dots B_m$ と表記します。たとえば、 $abcd \setminus bd$ は ac です。また、検索で一度出たクラスを再度検索する必要はありません。このことから語 ‘ $\dots W_{(i-1)} W_i W_{(i+1)} \dots W_{(j-1)} W_i W_{(j+1)} \dots$ ’ は語 ‘ $\dots W_{(i-1)} W_i W_{(i+1)} \dots W_{(j-1)} W_{(j+1)} \dots$ ’ と検索順序として一致します。このように後続の一一致する語を除いたものとの関係を ‘ $\dots W_{(i-1)} W_i W_{(i+1)} \dots W_{(j-1)} W_i W_{(j+1)} \dots$ ’ と表記します。そして関係 ‘ \searrow 」によって語 L はより短い語へと置換えることが可能で、この短縮化には下限があるため必ず極限が存在します。この語 L の関係 ‘ \searrow 」の極限になる語をここでは \underline{L} と表記します。この作用素 \mathcal{L} は次の性質を持ちます：

————— 作用素 \mathcal{L} の性質 —————

1. $\mathcal{L}(C) = C$
2. $\mathcal{L}(C_0(C_1)) = C_0 + \mathcal{L}(C_1)$
3. $L_1 \searrow L_0$ のとき $\mathcal{L}(L_1) = \mathcal{L}(L_0)$
4. $\mathcal{L}(C(B_1, \dots, B_n)) = C + \mathcal{M}(\mathcal{L}(B_1), \dots, \mathcal{L}(B_n))$

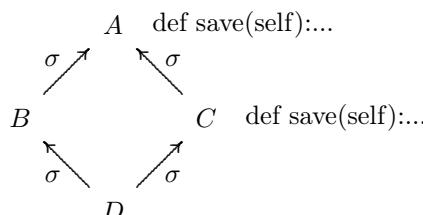
1. は継承関係を持たないクラス C のときに MRO は C のみであることを示します。つぎの 2. は $C_0 \xrightarrow{\sigma} C_1$ のとき、つまり、クラス C_0 が C_1 の派生クラスのときに最初にクラス C_0 を検索し、それからクラス C_1 の検索順序に従うということを意味し、つぎの 3. は関係 “\” で作用素 $*$ の値は不变であることを示し、最後の 4. は多重継承のときの処理です。次に作用素 \mathcal{M} を導入しておきましょう。この作用素 \mathcal{M} の働きは、継承関係を越る MRO を基本に多重継承のあるクラスで継承を示すタプルをそのまま用いて基底クラスの検索順序を入れるというものです。つまり、クラスの属性で基底クラスを示すタプルの左側から順番に先祖を辿る方法: 「**深さ優先、左から右の順番規則 (left-to-right depth-first rule)**」と呼ばれる規則になります。この操作を表現する作用素 \mathcal{L} は次の性質を持ちます:

作用素 \mathcal{M} の性質

- a. $\mathcal{M}(W) = \underline{W}$
- b. $\mathcal{M}(\dots, L, \dots) = \mathcal{M}(\dots, \underline{L}, \dots)$
- c. $\mathcal{M}(L_1, L_2, \dots, L_n) = \underline{L_1} + \mathcal{M}(L_2 \setminus L_1, \dots, L_n \setminus L_1)$

a. は検索順序を示す語 W 一つが引数であれば、関係 “\” の極限 \underline{W} を返すという性質です。そして次の b. は関係 “\” で作用素 \mathcal{M} の値は不变であることを示します。最後の c. は引数の最も左側にある経路を外に出し、その語に含まれるクラス名を他の引数から除去する処理方法を示しています。もし、引数の語に共通するクラス名がなければ \mathcal{M} は語に対する和になるだけです。

Python の古典的クラス型でメソッド検索で用いられる順序は MRO です。しかし、この手法は多重継承で有效地に動作しない問題があります。このことを次のクラス A, B, C, D の関係が次の菱形状になる図式を使って解説しておきましょう:



この図式は、クラス D はクラス B と C の双方を継承する多重継承の関係にあり、同時にクラス B と C はクラス A の派生クラスで、クラス A と C でメソッド `save` が定義されていることを表現しています。ここで各クラスが古典的クラス型のときにクラス B やクラス C でメソッド `save()` を利用しようとなればクラス A のものがそのまま用いられ、クラス C で上書きされたメソッド `save()` はそのままではクラス D で用いられません。ここで実際に MRO を計算してみましょう:

菱形状の継承関係の RMO

$$\begin{aligned}
 \mathcal{L}(B) &= BA \\
 \mathcal{L}(C) &= CA \\
 \mathcal{L}(D) &= D + \mathcal{M}(\mathcal{L}(B), \mathcal{L}(C)) \\
 &= D + \mathcal{M}(BA, CA) \\
 &= D + BA + \mathcal{M}(CA - BA) \\
 &= DBA + \mathcal{M}(C) \\
 &= DBAC
 \end{aligned}$$

このように古典的クラスの属性検索 (MRO) ではクラス D, B, A, C, A の順番で検索が行なわれますが、この属性検索では D, B の次のクラス A のメソッド `save()` が発見された時点で検索が終了し、その結果、クラス A で定義されたメソッド `save()` が用いられてクラス A の派生クラス C で再定義されたメソッド `save()` が用いられません。すなわち、より近い側のメソッドが用いられないという問題が生じます。そのためにはクラスタイプでは「**C3 MRO(Method Resolution Order)**」^{*43}と呼ばれる手法で検索が行われます。この C3 は多重継承にある場合に妥当な検出順序を提供するアルゴリズムで、最初に Dylan 言語に導入された手法です。この手法は先程の 3. の計算手順の作用素 \mathcal{M} を作用素 \mathcal{M}_{c3} で置換えて、次のようにまとめることができます：

C3 での作用素 \mathcal{L} の性質

1. $\mathcal{L}(C) = C$
2. $\mathcal{L}(C_0(C_1)) = C_0 + \mathcal{L}(C_1)$
3. $L_1 \searrow L_0$ のとき $\mathcal{L}(L_1) = \mathcal{L}(L_0)$
- 4'. $\mathcal{L}(C(B_1, \dots, B_n)) = C + \mathcal{M}_{c3}(\mathcal{L}(B_1), \dots, \mathcal{L}(B_n))$

作用素 \mathcal{M}_{c3} は次の性質を持ちますが、最初の a., b. は作用素 \mathcal{M} の場合と同様、また c. の計算手順は \mathcal{M} よりも階層を意識した検出方法に代ります：

作用素 \mathcal{M}_{c3} の性質

- a. $\mathcal{M}_{c3}(W) = W$
- b. $L_0 \searrow L_1$ のとき $\mathcal{M}_{c3}(\dots, L_0, \dots) = \mathcal{M}_{c3}(\dots, L_1, \dots)$
- c. \mathcal{M}_{c3} の計算は後述の方法で計算される。

\mathcal{M}_{c3} の計算手順を $\mathcal{M}(L_1, L_2, \dots, L_n)$ が与えられたときにどのように行われるかを纏めておきます：

^{*43} <https://www.python.org/download/releases/2.3/mro/> や PEP-253 を参照

\mathcal{M}_{c3} の計算手順

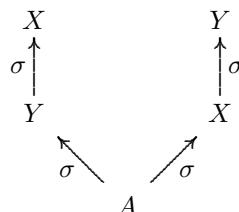
1. $i = 1$ とする.
2. $h_i = \overline{L}_i$ とする.
3. $j \neq i$ に対して $h_i \notin \underline{L}_j$ のときに $k \in (1, \dots, n)$ に対して $\overline{L}_k = h_i$ であれば, \mathcal{M}_{c3} の引数にある L_k を \underline{L}_k で置換し, $h_i + \mathcal{M}_{c3}(L_1, \dots, L_n)$ を作用素 \mathcal{M}_{c3} の結果として返却する.
4. $h_i \in \underline{L}_j (i \neq j)$ のとき $i \neq n$ ならば $i = i + 1$ として 2. に戻る. もし $i = n$ であればエラーを出力して処理を終える.

作用素 \mathcal{M}_{c3} は引数の語との間に共通するものが何もなければ最初の作用素 \mathcal{L} を拡張したものと同様に語の和として作用します. しかし, 共通する語が現われたときの処理がクラスの階層を合致させる働きになります. このことを先程の菱形状の継承関係で C3 MRO を計算することで確認してみましょう.

菱形状の継承関係の C3 RMO

$$\begin{aligned}
 \mathcal{L}(B) &= BA \\
 \mathcal{L}(C) &= CA \\
 \mathcal{L}(D) &= D + \mathcal{M}_{c3}(\mathcal{L}(B), \mathcal{L}(C)) \\
 &= D + \mathcal{M}_{c3}(BA, CA) \\
 &= D + B + \mathcal{M}_{c3}(A, CA) \\
 &= DB + C + \mathcal{M}_{c3}(A, A) \\
 &= DBCA
 \end{aligned}$$

C3 MRO では $DBCA$ と MRO の $DBAC$ と異なりクラス C の方が大本のクラス A よりも先に検索が行われるためにより新しいクラス C のメソッド `save()` が用いられて MRO よりも妥当な結果が得られます. さらに C3 MRO の長所は間違った継承関係が検出できることです. たとえば次の継承関係を想定しましょう:



この継承関係は $X \xrightarrow{\sigma} Y$ かつ $Y \xrightarrow{\sigma} X$ と, クラスの定義では相互参照的な関係, いわゆる循環的な定義であり, このような定義は Python では間違った定義です. しかし, MRO ではエラーではなく AYX が得られ, 一方の C3 MRO では

$$\begin{aligned}
 \mathcal{L}(Y) &= YX \\
 \mathcal{L}(X) &= XY \\
 \mathcal{L}(A) &= A + \mathcal{M}_{c3}(\mathcal{L}(Y), \mathcal{L}(X)) \\
 &= A + \mathcal{M}_{c3}(YX, XY)
 \end{aligned}$$

と計算が進むものの $\mathcal{M}_{c3}(YX, XY)$ の処理で YX の Y が XY に含まれるために YX の処理が行えず、今度は XY から X を取り出す処理に移ります。しかし、この X も前の YX に含まれるために XY からもクラスを取り出すことができずに作用素 \mathcal{M}_{c3} はエラーを出力しなければなりません。このように間違った継承関係の図式が与えられていてもも C3 MRO では適切な処理が行えることを意味します。

3.11 名前空間とスコープ

3.11.1 名前と名前空間

■名前 (name): オブジェクトの参照で用いられる識別子で、次の EBNF を持ります:

名前の BNF

名前 ::= [識別子 分離記号] 識別子

この名前への束縛 (binding) でオブジェクトと名前が結び付けられ、その結果、その名前の指示でオブジェクトへの参照が行われます。なお、参照すべき名前が名前空間に存在しないときに送出される例外が「**NameError**」、名前が名前空間に存在していても参照すべきオブジェクトが結び付けられていない変数を参照したときに送出される例外が「**UnboundLocalError**」です。

■名前空間 (name space): 統一的に名前を決定する手法です。Python は本体を小さくして必要に応じてモジュールで拡張する方式を採用していますがここで複数のモジュールを読込む必要があったときに階層なしに名前をそのまま展開すると、モジュール A で定義した函数 func() とモジュール B で定義した函数 func() と同名の函数が存在するために「**名前の衝突**」と呼ばれる事態になります。ここで Python 上に展開する名前にモジュールに依存する識別子を付けてモジュール単位で区別するという機械的な方法にするとどうでしょうか？この方法で名前の衝突を避けられるだけではなく、オブジェクトの検索も階層構造が入ることで検索範囲が狭まるために探し易くなります。このように名前の不用意な衝突を避け、オブジェクトの検索を行う範囲を定める仕組みが名前空間です。

■ブロック (block): プログラムで一つの実行単位になる区画であり、モジュール、クラスと函数定義はブロックです。そして、Python のシェルを経由して対話的に入力された

個々の命令もブロックです。

■**コードブロック (code block)**: スクリプトファイル, スクリプト命令, 組込函数 eval() や exec() に引き渡した文字列, 函数 input() から読み取られて評価される Python の文で構成され, これらコードブロックは「**実行フレーム (execution frame)**」上で実行されます。

■**スコープ (scope)**: 参照される名前の範囲のことで, Python のスコープにはモジュールの大域的なスコープと函数内部の局所的なスコープの二種類のみです. メソッドのコードブロックを含む拡張は行われません. その例としてリファレンスマニュアルでは生成子を一例として挙げています:

```
class A:  
    a = 42  
    b = list(a + i for i in range(10))
```

この例では名前 b に束縛する構築子 list() の引数が生成子 (generator) で, ここではリストの成員の型を for 節を使って表記するリストの内包表現による生成に対応します. そして, この生成子内部で名前 a への参照がありますが, 名前 a は構築子 list() のブロック外部にあるためにスコープ外になり, 名前 a への束縛が行われていてもエラーになります:

```
>>> class A:  
...     a=42  
...     b=list(a+i for i in range(10))  
...  
Traceback (most recent call last):  
File "<stdin>", line 1, in <module>  
File "<stdin>", line 3, in A  
File "<stdin>", line 3, in <genexpr>  
NameError: global name 'a' is not defined
```

名前がコードブロック内部で用いられているときに名前の参照で近傍のスコープを用いられます. ここで**ブロックの環境**とは, ある一つのコードブロック内で参照可能なスコープの全ての集合のことです.

名前があるブロック内部で束縛されているとき, その名前はそのブロックの局所変数になります. 名前がモジュールで束縛されているときは大域変数になります. そして, コードブロックで用いられていても, そのブロックで定義や束縛が行われていないときに, その変数は自由変数となります.

ここで名前の参照を行った際に, その名前に束縛されたオブジェクトがないと例

外:NameError が、局所変数で、名前が束縛されていない変数の参照を行ったときに例外:UnboundLocalError が創出されます。例外:UnboundLocalError は例外:NameError の派生クラスです。なお、`del` 文で指定された対象は、`del` 文の目的が対象の束縛の解除であるものの、束縛済みのものと見做されます。

`import` 文や代入文は、クラスや函数定義、モジュールレベル内で行われます。

`global` 文で指定された名前がブロック内にあるとき、その名前は名前空間の最上層で束縛された名前の参照を行います。

3.12 例外

3.12.1 例外の概要

「例外 (exception)」はプログラムの処理時に何らかの原因で生じた「異常」とされる状況で、プログラムの本質的な欠陥、開くべきファイルがないといった前提から外れた状況から生じた状況、さらには利用者側の理由 (受け取るはずのメールがまだ送られていない等) でプログラムを一時的に停止させて別の処理を実行させるために意図的に発生させる信号も含まれます。意図的に発生させた例外は「利用者定義例外 (user defined exception)」と呼ばれ、その生成には `raise` 文が用いられます。また、例外を分析して別の処理につなげる処理は「例外処理」と呼ばれ、Python では `try` 節で例外が生じ得る処理、`except` 節で対応する例外への処理、`finally` 節で後処理 (clean up) を記載します。なお、`finally` 節は `try` 文に一つだけ `try` 文の末端に記載可能で、`finally` 節の処理は `try` 文の最後に必ず実行されます。

3.12.2 BaseException クラスについて

例外のクラスが `BaseException` クラスです。この `BaseException` クラスの組込の派生クラスを以下に示します：

- `SystemExit`: フィル `sys.exit()` が送出する例外です。この例外が処理されなければスタックのトレースバックを全く表示することなくインタプリタが終了します。この関連値が通常の整数であればシステム終了ステータスをフィル `exit()` に渡して表示します。この値が `None` であれば終了ステータスは `0` になります。文字列のような他の型であれば、そのオブジェクトの値が表示されて終了ステータスが `1` になります。このクラスのインスタンスは属性 `code` を持ち、この値は終了ステータスまたはエラーメッセージ (既定値は `None`) に設定されています。なお、この例外は厳密に異

常ではないために BaseException の派生クラスです。ここで函数 sys.exit() は後処理 (try 文の finally 節) が実行されるようにするため、またデバッガが制御不能になるリスクを冒さずにスクリプトを実行できるようにするために例外に翻訳されます。即座に終了する必要があるときに、たとえば、函数 os.fork() を呼んだあとの子プロセス内で函数 os._exit() で実行できます。このクラスは BaseException を直接継承することで着実に呼び出し元に伝わりてインタプリタを終了させられます。

- KeyboardInterrupt: 利用者が Control-C または Delete を押したときに送出される例外です。割込の有無はインタプリタの実行中に定期的に調べられ、組込函数 input() や raw_input() が利用者の入力を待っている間に割込みキーを押しても、この例外が送出されます。Exception クラスの例外を処理するコードに誤って捕捉されないように、このクラスは BaseException クラスを継承しています。
- GeneratorExit: generator 型のメソッド close() が呼び出されたときに送出される例外です。この例外は厳密な意味で異常ではないために、一般の例外である Exception とは別のクラスです。
- Exception: 一般的な例外を表現します。このクラスの詳細は次の小節で述べます。

3.12.3 Exception クラスについて

Exception クラスは BaseException クラスの派生クラスです。利用者定義の例外クラスを構築するときは Exception クラスの派生クラスとして構築します。

Exception クラスの組込の派生クラスに StandardError, StopIteration と Warning があります。

StandardError クラスについて

StandardError クラスの派生クラスを以下に示しておきます:

Exception クラスの派生クラス			
ArithmetError	AssertionError	AttributeError	BufferError
EnvironmentError	EOFError	ImportError	LookupError
MemoryError	NameError	ReferenceError	RuntimeError
SyntaxError	SystemError	TypeError	ValueError

■ ArithmeticError: 算術演算処理で送出される例外で、その派生クラスに OverflowError, ZeroDivisionError と FloatingPointError があります:

- OverflowError: 算術演算の結果が表現できない大きな値になったときに送出されますが、Python の整数処理で生じることはほとんどなく、MemoryError が送出されるでしょう。C の浮動小数点演算の例外処理が標準化されていないために浮動小数点演算のほとんどが検査されません。

```
>>> try:
...     math.exp(10000)
... except OverflowError:
...     print "Too big!!"
...
Too big!
```

- ZeroDivisionError: 除算や剰余で 0 による除算が発生したときに送出されます。関連値はその演算における被演算子と演算子の型を示す文字列です：

```
>>> try:
...     1/0
... except ZeroDivisionError:
...     print "1/0!!!"
...
1/0!!!
```

- FloatingPointError: 利用している Python が ‘-with-fpectl’ オプションを有効にしてコンパイルされているときか、pyconfig.h ファイルに WANT_SIGFPE_HANDLER が定義されているときに限って利用可能な例外で、プロセッサで IEEE 754 浮動小数点エラーが発生したときに送出されます。モジュール fpectl はスレッドセーフではなく、利用上の危険もあるため、余程のことがない限り、この例外に関わることはないでしょう。

■BufferError: バッファ関連の処理に失敗したときに送出されます。組込の派生クラスはありません。

■AssertionError: assert 文が失敗したときに送出されます。組込の派生クラスはありません。

```
>>> x = 1
>>> try:
...     assert x>1
... except AssertionError:
...     print x
1
```

■**AttributeError:** 属性参照や代入が失敗したときに送出されます。なお、参照しようとしたオブジェクトが属性値の参照や属性値の設定を提供していないときは `TypeError` が送出されます。組込の派生クラスはありません。

```
>>> x = 1
>>> try:
...     x.test
... except AttributeError:
...     print x
1
```

■**EnvironmentError:** システム環境で生じる例外で、`IOError` と `OSError` を組込派生クラスとして持ちます。この型の例外はタプルで生成され、その長さが 2、あるいは 3 のタプルであれば第 1 成分はエラー番号に対応し、インスタンスの属性 `errno` で得られます。第 2 成分は属性 `strerror` で得られ、その値はエラーに関連するメッセージです。タプルに第 3 成分があるときは、この第 3 成分が属性 `filename` で得られます。これらのタプルの成分は属性 `args` からも得られますが、互換性のために `args` には第 1 成分と第 2 成分のみのタプルになります。なお、タプルの長さが 2, 3 以外のときに属性 `errno`, `strerror`, `filename` の値は `None` です。

- `IOError`: ファイルやメソッドによるファイルオブジェクト等の I/O 操作が「**ファイルが存在しない**」や「**ディスクに空き領域がない**」といった I/O に起因する理由で失敗したときに送出されます。
- `OSError`: ファイルやメソッドがシステムに起因する異常を返したときに送出されます。ここで属性 `errno` はモジュール `errno` に基づく数字のエラーコード、属性 `strerror` は C の函数 `perror()` で表示されるような文字列です。この `OSError` クラスには `VMSError` と `WindowsError` の二つの組込例外の派生クラスがあります:
 - ▲ `VMSError`: VMS でのみ利用可能で、VMS 特有のエラーが起こったときに送出されます。
 - ▲ `WindowsError`: MS-Windows 特有のエラー、あるいは、エラー番号が `errno` 値に対応しないときに送出されます。`winerrno`, `strerror` の値は Windows プラットフォーム API の函数 `GetLastError()` と `FormatMessage()` の返却値から生成され、`errno` の値は数値で表現された `winerror` の値を基にファイル `errno.h` から対応する値を取り出したものです。

■**EOFError:** ファイルの終端 (EOF) に到達したときに送出される例外です。なお、メソッド `file.read()` と `file.readline()` はこの例外ではなく空の文字列を返却します。組込の派生クラスはありません。

■**ImportError:** `import` 文でモジュールが見つけられなかったときや `from ... import` 文で指定した名前を取り込めなかったときに送出されます。組込の派生クラスはありません。

■**LookupError:** 連想配列 (mapping) や列 (sequence) で要素の指定で用いた鍵や添字が範囲外のときに送出される例外で、メソッド `codecs.lookup()` で直接送出されることもあります。`IndexError` と `KeyError` が組込の派生クラスです：

▲ **IndexError:** 指定した添字がその範囲を超えていたときに送出されます。なお、添字が整数でないときは例外 `TypeError` です。また、スライスの添字で添字の範囲内に収まるように自動調整されるために `IndexError` は送出されません。

▲ **KeyError:** 連想配列の鍵が存在する鍵集合内に含まれていなかったときに送出されます。

■**MemoryError:** プログラムの処理の過程でメモリが不足したときに送出される例外で、返却値はどの内部操作でメモリ不足が生じたかを示す文字列です。オブジェクトをいくつも消去すれば復旧可能なときもありますが、プログラムの暴走でも実行スタックの追跡結果が表示できるように、この例外が送出されます。組込の派生クラスはありません。

■**NameError:** 局所、大域の双方で指定した名前が存在しなかったときに送出されます。関連値は見つからなかった名前を含むメッセージです。`UnboundLocalError` が組込の派生クラスです：

▲ **UnboundLocalError:** 値が束縛されていない函数、メソッド内の局所変数への参照を行ったときに送出されます。組込の派生クラスはありません。

■**ReferenceError:** メソッド `weakref.proxy()` で生成された弱参照 (weak reference) プロキシを使って塵収集 (GC) で回収されたあとのオブジェクト属性を参照しようとしたときに送出されます。組込の派生クラスはありません。

■**RuntimeError:** 他のカテゴリに分類できない異常が検出されたときに送出されます。関連値は何が問題であるのかをより詳細に示した文字列です。`NotImplementedError` が組込みの派生クラスです：

- ▲ **NotImplementedError:** 未実装であることを示す例外です。利用者定義の基底クラスで抽象メソッドが派生クラスで上書きされることを要求するときに、この例外を送出しなくてはなりません。組込の派生クラスはありません。
- **SyntaxError:** Python の構文解析器が構文エラーを検出したときに送出される例外で、`import` 文、`exec` 文、組込関数 `eval()` と `input()`、初期化スクリプトの読み込みと標準入力で対話的な処理でも生じます。このクラスのインスタンスは例外の詳細に簡単にアクセスできるようにするために属性 `filename`, `lineno`, `offset`, `text` があります。例外インスタンスに対するメソッド `str()` はメッセージのみを返します。`IndentationError` が組込の派生クラスです：
- ▲ **IndentationError:** 字下の構文エラーに対応する基底クラスです。`TabError` が組込の派生クラスです：
 - ♡ `TabError`: 字下げで TAB と Space を一貫した順序で並べていないときに送出されます。組込の派生クラスはありません。
- **SystemError:** インタプリタが軽度の内部エラーを発見したときに送出されます。関連値は下位層の言葉でどのような問題があるのかを示す文字列です。Python の作者、インタプリタを保守している人にこの例外を報告してください。この際にインタプリタのバージョン (`sys.version`; 対話的セッションを開始した際にも出力されます)、正確なエラーメッセージ (例外の関連値)、そして、可能ならエラーを引き起こしたプログラムのソースコードを報告してください。組込の派生クラスはありません。
- **TypeError:** 関数やメソッド等にて引き渡されたオブジェクトで型の不整合が生じたときに送出される例外です。関連値は詳細を述べた文字列です。組込の派生クラスはありません。
- **ValueError:** 組込の演算や関数で、型は適合していても不適切な値を受け取ったときや `IndexError` のような詳細な例外では説明のできない状況で送出されます。`UnicodeError` クラスが組込みの派生クラスです：

`beginitemize`

- UnicodeError: 符号化/復号化で Unicode に関する異常が発生したときに送出されます。このクラスには異常を説明する次の属性があります:

- encoding: 異常を送出したエンコーディングの名前.
- reason: 異常を説明する文字列.
- object: 符号化/復号化しようとしたオブジェクト.
- start: オブジェクトの最初の無効なデータの添字.
- end: オブジェクトの最後の無効なデータの添字の次の整数値.

派生クラスとして次のクラスがあります:

- ▲ UnicodeEncodeError: エンコード中に Unicode 関連の異常が発生したときに送出されます.
- ▲ UnicodeDecodeError: デコード中に Unicode 関連の異常が発生したときに送出されます.
- ▲ UnicodeTranslateError: コード翻訳に Unicode 関連の異常が発生したときに送出されます.

3.12.4 StopIteration

それ以上要素がないことを知らせるためにイテレータ (iterator) のメソッド next() により送出されます。この状況は通常は異常とみなさないために StandardError ではなく Exception から派生します。

3.12.5 Warning

警告の基底クラスで、その派生クラスとして以下の警告があります:

- UserWarning: 利用者が構築したコードで生成される警告.
- DeprecationWarning: 廃止された機能に関する警告.
- PendingDeprecationWarning: 将来廃止される予定の機能に関する警告.
- SyntaxWarning: 曖昧な構文に関する警告の基底クラス.
- RuntimeWarning: ランタイムの挙動に関する警告.
- FutureWarning: 将来、意味や構成に変更がある文に対する警告.
- ImportWarning: モジュールの読み込みの誤りと思われるものに関する警告.
- UnicodeWarning: Unicode に関する警告.

第4章

数値行列処理について

4.1 数式処理と数値行列処理

数式処理システムの多くがメモリ等のハードウェア等の環境の制約があるにせよ利用者が必要とする精度で数値計算ができますが、その処理は低速です。実際、実数は「**浮動小数点数 (floating point number)**」と呼ばれるある一定の長さの2進数に符号化して処理されますが、任意精度の数値計算では通常用いられる倍精度の浮動小数点数よりも長い2進数で表現するためにデータ量、計算量が共に増大していることに加えて四則演算等の処理がCPUに実装されている倍精度と違って任意精度の数値計算はソフトウェア側で処理しなければならないためです¹。

数値計算のさまざまな処理は数値行列処理で置換えられるために数値計算目的のソフトウェアは数値行列処理の効率化に工夫を凝らしています。このような数値計算を目的としたソフトウェアで著名なものがThe MathWorks, Inc.のMATLABで、現在は商用ソフトウェアですが、本来はFORTRANで記述された数値行列計算ライブラリLINPACKやEISPACK²を学生が容易に扱えることを目的とした教育向けのフリーソフトでした。そのため数値行列操作を行うソフトウェアの多くがMATLABの行列操作を取り入れる大きな要因になり、現在のMATLABはToolboxと呼ばれるライブラリ群を備えた数値行列計算ソフトウェアの標準的存在で、特に制御系の解析ではMATLAB+Simulink³の組合せが本流です⁴。

¹ IntelのCPUに浮動小数点計算ユニット(FPU)が組み込まれる以前の90年代初頭、数値計算を行うアプリケーションでCPUに組込まれた浮動小数点計算ユニットを利用するものとソフトウェア的に浮動小数点数の処理を行うものがありました。FPU版で一瞬で済むことが非FPU版になると利用をためらうほど低速でした。

² これらの数値行列計算ライブラリの後継がLAPACKです。

³ MATLABで制御系のブロック線図の解析を目的にしていましたが、現在はMATLABのGUI環境、特に「**モデルベース開発**」の中核を担っています。

⁴ 構成は「MathWorks 製品一覧 - 製品構成図」(<http://www.mathworks.co.jp/products/pfo>)を参照。

MATLAB の影響を強く受けた「OSS(Open Source Software)」のアプリケーションの代表として「GNU Octave」, INRIA の「Scilab」, C 風の「Yorick」を挙げておきましょう。まず, GNU Octave は GNU の MATLAB クローンと呼ばれ, MATLAB との高い互換性を持ちますが, 単なるクローンではなく, ファイル操作などの独自の改善点も加わっています。Scilab は MATLAB に類似したシステムで, MATLAB の Simulink に類似した「Xcos」を擁する下手な壳物を凌駕するシステムです。GNU Octave や Scilab が機能を満載して重量級のアプリケーションになっている状況に対し, Yorick は逆に軽量言語で, 対話処理が可能な C と言えるほど C に類似し, 扱う数値行列は多次元配列で, そのユニークな配列の添字処理が大きな特徴です [35]。これら GNU Octave 等の MATLAB 風の行列操作が見られる言語をまとめて「MATLAB 系言語」と呼びます。ところで Python 本体の数値関数やリスト処理は MATLAB 系言語と比較して非常に貧弱で, この弱点を強化するパッケージが NumPy です。この NumPy も MATLAB の影響を強く受け, NumPy の多次元配列の添字処理は MATLAB と類似した効率の良い処理ができます。

MATLAB 系言語や NumPy などの数値行列を専門に行うソフトウェアでは数値行列の効率的な処理のために数値行列処理ライブラリのサブルーチンに数値行列データを引き渡して結果を取り込むという手順を採用しています。この手法で行列処理自体は C や FORTRAN と大差ない水準で利用できるものの, 分岐, 反復といった計算以外の処理言語が中心になる処理で時間がかかる, いわゆる, 「**処理言語のオーバーヘッド**」と呼ばれる弱点があります。そのために処理言語で配列のループ処理は極力行わず, 可能な限り数値行列処理ライブラリのサブルーチンを利用する函数に引き渡す工夫が必要です。さらに数値行列処理ライブラリの性能は BLAS と呼ばれる数値行列ライブラリの各種計算機環境への最適化に大きく依存し, 同時に数値計算ライブラリが効果的に動作するようなプログラム上の工夫を行う必要があります。そのために MATLAB 系言語ですべきこととすべきことは NumPy でも同様です。したがって, 実際の処理がどのように行われるかを多少とも知っておくとより良い処理が行えます。この章では浮動小数点数と数値行列処理ライブラリ BLAS について簡単な解説を行い, NumPy での数値行列処理を実例を交えて解説します。

4.2 IEEE 754 による実数の表現

4.2.1 浮動小数点数の概要

実数 \mathbb{R} の濃度は \aleph_0 で, 整数 \mathbb{Z} の濃度 \aleph_0 よりも各段に大きな数です。そして, 実数 \mathbb{R} よりも小さいとはいえる \mathbb{Z} の濃度 \aleph_0 も有限個ではありません。これらの無限個の数を計

算機の有限な資源(たとえばレジスタ)を使って表現するためにはなんらかの工夫が必要です。まず、正整数は2進数表現で、その数列の長さに制約が入るもの、ある範囲の数の表現と処理が可能です。さらに補数表現を利用することで負の整数も表現可能で、これらを併せることである範囲の整数を計算機で扱うことができます。さらに実数の表現では「浮動小数点数」と呼ばれる数の表現が用いられます。これは実数の b 進数への符号化です。その基本的な考え方を簡単に説明しておきましょう。まず、実数 -101.234 の符号化を考えましょう。この数は負の数であるために符号は -1 、残りは絶対値 101.234 です。つぎに絶対値を整数の桁が一桁の「仮数(小数)」と呼ばれる部分と 10 の幂の積で表現し、 $101.234 = 1.01234 \times 10^2$ が得られ、以上から三成分のタプル $(-1, 10^2, 1.01234)$ で -101.234 が表現できます。ところで数の符号は $(-1)^0 = 1, (-1)^1 = -1$ から符号が -1 であれば $1, 1$ であれば 0 を対応させ、 10 の幂は次数だけで十分、仮数はその小数点を除去した 10 進数の列“101234”とすると自然数のタプル $(1, 2, 101234)$ が得られ、これらの成分を繋いで 10 進数の数の列“12101234”で数 -101.234 の符号化が得られます。これが浮動小数点数への符号化の手順です。この符号化では先頭に数の符号情報、基数 10 の幂の次数が配置されていますが、この配置によって符号化した数の大小関係の判定が迅速に行えます。実際、最初に符号、同じ符号なら幂の次数、次数も等しければ仮数で比較して判断するという処理を符号の頭から順番に比較ができます。また、この符号化は 10 進数である必要はありません。正整数 b を使って浮動小数点数を 0 から $b-1$ までの数の列、つまり、 b -進数の数でも構いません^{*5}。この正整数 b を「基数(radix)」と呼びます。浮動小数点数に関してはIEEE 754という国際規格があり、符号小数点数そのものの規格化、浮動小数点数の演算などが規格化されています。

4.2.2 IEEE 754について

IEEE 754は浮動小数点数の書式と演算等の処理を定める国際規格です。最初に2008年に策定されたIEEE 754-2008が定める事柄を挙げておきます：

- 基数が2と10のときの浮動小数点数の書式
- 無限大 $\{-\inf, \inf\}$ 、浮動小数点数例外NaNとして $\{qNaN, sNaN\}$
- 和、差、積、商の四則演算、平方根、および融合積和演算、大小関係の処理
- 整数と浮動小数点数間の変換
- 異なる浮動小数点数間の変換
- 浮動小数点数と文字列としての外部表現との変換

つぎにIEEE 754-2008が想定している実数 \mathbb{R} の符号化の段階を示します：

^{*5} GMPの多倍長浮動小数点数は仮数部の実体を別に持ち、浮動小数点数側には仮数部の番地を指定することで仮数部の長さの制約を外して任意精度浮動小数点数を表現しています。

実数の浮動小数点数への符号化の階層		
第零層:	\mathbb{R}	実数そのもの
	↓	
第一層:	$\mathbb{R} \cup \{-\infty, \infty\}$	無限大の追加で実数拡張
	↓	
第二層	$\{-\infty, \dots, -0\} \cup \{+0, \dots, \infty\} \cup \{\text{NaN}\}$	± 0 と例外 NaN の導入
	↓	
第三層:	$(S, E, T) \cup \{-\infty, \infty\} \cup \{\text{qNaN}, \text{sNaN}\}$	実数を近似、NaN を qNaN, sNaN に分類
	↓	
第四層:	$\underbrace{b_{(0)} b_{(1)} b_{(2)} \dots b_{(n-1)}}_{n\text{-bit}}$	2進数表現として符号化

無限遠 $\{-\infty, \infty\}$ と例外 NaN(Not a Number) の導入による実数 \mathbb{R} の拡張から符号化に至るまでの抽象化の段階を示し、各階層にある矢 (↓) は集合の元との対応関係、特に第二層から第三層の対応関係は多対一になります。なぜなら濃度 \aleph の実数 \mathbb{R} を符号 (S), 指数部 (E) と仮数部 (T) から構成される一定のデータ長のタプルへの射影が丸めで異なる数でも同じ浮動小数点数に符号化されるためです。また、 -0 と $+0$ は 0 の浮動小数点数への符号化で生じ、IEEE 754 で $0 = -0 = +0$ と規定されています。さらに第二層で実数 \mathbb{R} を負の数と正の数に分けられ、負の実数側は負の規格化数、非規格化数、 -0 と $-\infty$ 、正の実数側が正の規格化数、非規格化数、 $+0$ と ∞ で構成されます。ここで NaN は IEEE 754-1985 では NaN のみが IEEE 754-2008 では qNaN(quiet NaN) と sNaN(signaling NaN) の二種類の NaN が規定され、qNaN が $0/0$ や $\infty - \infty$ のような不定値になる演算から出力され、演算を通じて qNaN として処理されます。そして、sNaN は処理で例外が発生した時点での OS の例外として処理されるためにデバッグ向けに用いられます。なお、Python では sNaN と qNaN を区別せずにひとまとめで NaN として扱っています。

4.2.3 浮動小数点数の書式

IEEE 754-2008 では浮動小数点数の基底を 2 あるいは 10 とし、一定の長さの 2進数として符号化します。この符号化の長さを「**符号長**」と呼び、この符号長によって浮動小数点数で表現可能な数の領域、つまり、精度が決定されます。ここで符号長が 32-bit 長の「**単精度 (single)**」、符号長が 64-bit 長の「**倍精度 (double)**」、IEEE 754-2008 では基底が 10 の浮動小数点数に関して倍精度と四倍精度が規格化されています。ここでは基底 2 の話に限定して解説します。

数の符号は $(-1)^0 = 1$ と $(-1)^1 = -1$ から正のときに 0, 負のときに 1 を対応させ, この 0 か 1 のいずれかを指示する部位を「**符号部 (sign)**」と呼んで記号 S で表現し, 符号化した数の先頭に配置します。つぎに与えられた数の絶対値を 2 進数表示で $d_0.d_1d_2\dots d_p$ と 2^e に分解し, この際に $d_0 = 1$ として小数点以下の $d_1d_2\dots d_p$ の符号化を行います。この d_0 に配置される 1 のことを「**隠れ bit**」, あるいは「**暗黙の 1**」, 残りの p 個の数字の列 $d_1\dots d_p$ を「**仮数部 (mantissa, significand)**」と呼んで記号 T と表記します。ところで 2 の幂の次数 e には負の次数もあるために次数が正整数になるように調整します。IEEE 754 では次数 e に「**下駄履き値 (bias)**」, ε を「**下駄履き表現**」と呼ばれる正整数 β を使って, $\varepsilon \stackrel{\text{Def.}}{=} e + \beta$ で定めた ε を用い, この ε を 2 進数で表現した部位を「**指数部 (exponent)**」と呼び, この部位を記号 E と表記します。この符号化の状況を各精度毎にまとめておきます:

基底 2 のときの浮動小数点数の精度と各部の長さ

精度	符号長	仮数部長 (p)	指数部長 (w)	下駄履き値
単精度	32	23	8	$127 (=2^7 - 1)$
倍精度	64	52	11	$1027 (=2^{10} - 1)$
四倍精度	128	112	15	$16337 (=2^{14} - 1)$

この表には載せていませんが, 数の符号を指定する符号部は 1-bit 長です。また, 下駄履き値 (bias) は数の符号化で用いる指数部の上限 (emax) と下限 (emin) の和が 1 になるよう設定するために $2^{\text{指数部長}-1} - 1$ が下駄履き値 β の値で, 同時に指数部で表現できる整数の上限値 emax と一致し, 下限値 emin の値は $1 - \beta$ です。なお, 指数部の表現する数の上限 emax や下限 emin の指数部を持つ浮動小数点数には inf, NaN や 0 が割り当てられます。

4.2.4 浮動小数点数を構成する各部位

IEEE 754 で定められる浮動小数点数は一定の bit 長の符号部 (S), 指数部 (E), 仮数部 (T) で構成される 2 進数の数の列です。ここで符号部 S は 1-bit 長で精度と無関係に固定され, 指数部 E の bit 長を w , 仮数部 T の bit 長を p と表記します。このとき浮動小数点数の符号長は $w + p + 1$ で, この値は各精度に対応する 32, 64, 128 のいずれかの値です。また, 浮動小数点数に符号化した数 a は 2 進数の数の列で, その桁数に対応するように番地を入れます。この番地は a の列の長さが n であれば, 数の列 a の左端を $a_{(n-1)}$, 右端を $a_{(0)}$ とします。

■**符号部 S :** 1-bit 長で 2 進数 a での最大の番地の成分 $a_{(w+p)}$ に対応し, 符号化する数が正であれば 0, 負であれば 1 が設定されます。

■**指数部** E : w -bit 長で符号部に直後に配置されます。2進数 a の w 桁の部分 $a_{(w+p-1)} \dots a_{(p)}$ に情報が格納されます。IEEE 754 で指数部が表現する整数 ε は本来の指数 e にある定数 β を加えた数です。ここで定数 β を「下駄履き値 (bias)」、定数 β を加えた指数部を「下駄履き表示」と呼びます。この下駄履き値 β は指数部の bit 長に依存するために浮動小数点数の精度で異なります。この下駄履き表示に対して $b_{(i)} = a_{(i+p)}$ ($0 \leq i \leq w-1$) のときに指数部で表現される整数 ε は $b_{(0)} \times 2^0 + b_{(1)} \times 2^1 + \dots + b_{(w-1)} \times 2^{w-1}$ で与えられます。

■**仮数部** T : p 桁の2進数 $a_{(p-1)} \dots a_{(0)}$ で表現されます。 w -bit 長の指数部が表現する数 ε で正規化数と非正規化数に分類され、このことが仮想部の表現に影響します：

規格化数と非規格化数の仮数部

種類	指数部が表現する数	指数部の復号化	仮数部の復号化
規格化数	$2^w - 2 \geq \varepsilon \geq 1$	$2^{\varepsilon-\beta}$	$1.d_{(p-1)} \dots d_{(1)}d_{(0)}$
非規格化数	$\varepsilon = 0$	$2^{1-\beta}$	$0.d_{(p-1)} \dots d_{(1)}d_{(0)}$
$\pm \inf, \text{NaN}$	$\varepsilon = 2^w - 1$		

指数部が表現する整数 ε が $2^w - 2 \geq \varepsilon \geq 1$ のときに仮数部が表現する2進数の先頭に「暗黙の1」、あるいは「隠れbit」と呼ばれる1が配置され、 $\varepsilon = 0$ のときに暗黙の1は配置されません。そのために復号化では指数部の幕の次数 e を $-\beta$ ではなく、 $1 - \beta = \text{emin}$ とします。そして、暗黙の1がある浮動小数点数を「規格化浮動小数点数 (normal floatingpoint number)」、略して「規格化数」、暗黙の1がない浮動小数点数を「非規格化浮動小数点数 (subnormal floatingpoint number)」、略して「非規格化数」と呼びます。規格化数は実質的に $p+1$ -bit の情報を持つておらず、隠れbitを加えた桁数 $p+1$ が実際の有効桁になるため、この $p+1$ が「有効桁精度」と呼ばれます。一方の非規格化数は0周囲の微小な数の表現で、非規格化数は最大でも p -bit と有効精度以下の精度になるために規格化数から非規格化数の領域に移行することを「段階的アンダーフロー」、または「漸近アンダーフロー」と呼びます。ここで規格化数と非規格化数の復号化の書式を示しておきます：

規格化数と非規格化数の復号

規格化数 ($\varepsilon = 1$):	$(-1)^s \times 2^{1-\beta} \times (1 + d_{(1)} \cdot 2^{-1} + \dots + d_{(p-1)} \cdot 2^{1-p})$ $= (-1)^s \times 2^{\text{emin}} \times (1 + d_{(1)} \cdot 2^{-1} + \dots + d_{(p-1)} \cdot 2^{1-p})$
非規格化数 ($\varepsilon = 0$):	$(-1)^s \times 2^{\text{emin}} \times (0 + d_{(1)} \cdot 2^{-1} + \dots + d_{(p-1)} \cdot 2^{1-p})$

w -bit 長の指数部が表現する数 ε の取り得る範囲で、規格化数と非規格化数が用いない $\varepsilon = 2^w - 1$ の領域は NaN {qNaN, sNaN} と無限大 $\{-\inf, \inf\}$ 向けに確保されています。

■零, 無限大と NaN の表現: 浮動小数点数の表現の多様性のために用いられる対象です:

特殊な数の定義		
表現される数	ε の値	仮数部の値
0	0	0
$\{-\infty, \infty\}$	$2^w - 1$	0
$\{qNaN, sNaN\}$	$2^w - 1$	0 以外 (符号部は無関係)

IEEE 754 では指数部と仮数部を表現する整数 ε が 0 のとき, すなわち $(-1)^s \times 0$ で数 0 の浮動小数点数への符号化を定義します。このときに $s = 0$ であれば「正の零」と呼んで $+0$ と表記し, $s = 1$ であれば「負の零」と呼んで -0 と表記しますが, $+0 = -0$ であることが IEEE 754 で定められています。この符号付き零の規定は右極限や左極限を考慮する上で重要です。演算では他の浮動小数点数と同様の演算が行え, $1/0$ のような 0 による割算では無限大 ∞ に対応する浮動小数点数の inf を返します。

無限大 $\{-\infty, \infty\}$ は指数部が表現する数 ε が $2^w - 1$, すなわち, w -bit 長の指数部がすべて 1, 仮数部がすべて 0 のときで, その符号部が 0 のときに「正の無限大 (inf)」, 符号部が 1 のときに「負の無限大 (-inf)」になります。IEEE 754 では無限大と 0 を含む通常の浮動小数点数との演算が定義されています。

NaN は仮数部が 0 ではなく指数部の bit がすべて 1 のときです。なお, NaN は 0^0 や $\infty - \infty > 0$ のような不定値が現われる演算を行ったときに生じる例外で, IEEE 754-1985 では例外として NaN のみが策定され, IEEE 754-2008 で qNaN と sNaN の二つの NaN の扱いを規定しています。ここで $0/0$, inf/inf , や $inf - inf$ のように不定値を返す式では例外: qNaN が返され, この qNaN は演算を通して qNaN でありつづけ, OS レベルの例外にはなりませんが, sNaN は不正な処理に対する NaN で, OS レベルの例外として処理されます。

■浮動小数点数で表現できる範囲: 浮動小数点数は有界で有限個であるために表現できない実数があります。まず, $|x| > x_{\max}$ になる数 x を「桁溢れ (overflow)」, 逆に, $x_{\min} > |x| > 0$ になる数 x を「アンダーフロー (underflow)」と呼びます。なお仮数部を w -bit 長, 指数部を p -bit 長とする浮動小数点数で表現可能な数の集合を $\mathcal{F}_{w,p}$, bit 長を固定した状態で問題ないときに \mathcal{F} と表記します。

■計算機イプシロン: 浮動小数点数 1.0 に最も近接する浮動小数点数との差として定義され, IEEE 754 で計算機イプシロンの値は $2^{-52} = 2.220446049250313 \times 10^{-16}$ です。この浮動小数点数は 0 と異なる微小な実数の表現で用いられ, MATLAB では変数 eps,

Scilab なら変数 %eps, NumPy は ‘finfo(float).eps’ に束縛された値から判ります。このことを NumPy を np として読み込んだ Python 2 で確認しておきましょう:

```
>>> 1+2**(-52)==1.
False
>>> 1+2**(-53)==1.
True
>>> np.finfo(float).eps == 2**(-52)
True
```

上の二つの確認は大雑把ですが、計算機イプシロンの定義と矛盾しない結果が得られています。この計算機イプシロンは 0 による割算や対数軸グラフの生成等、0 であることに起因するエラーで落ちて困る処理に利用できます。

4.2.5 浮動小数点の演算について

IEEE 754 は浮動小数点数の書式だけを定める規格ではありません。IEEE 754 は浮動小数点数の書式の他に通常の浮動小数点数に対する「四則演算」: (“+”, “-”, “*”, “/”) と「融合積和演算 (FMA)」, 「平方根」: ($\sqrt{-}$) と「剰余」: (%)、「比較操作」: (>, \geq , =) といった演算、さらには無限大 $-\infty, \infty$ や「NaN(N_ot a N_umber)」に対する処理、さらには「整数と浮動小数点数間の変換」と「異なる浮動小数点書式間の変換」といった処理も規格化しています。

ここで IEEE 754-2008 で新規に追加された「融合積和演算 (fused multiply-add, FMA)」^{*6}は演算 $(a, b, c) \mapsto a + (b \times c)$ のことです。FMA がなければ $b \times c$ を処理して a との和の演算と全体で二度演算を実行するために丸めが 2 度生じる可能性がありますが、FMA が CPU に実装されていれば CPU で一度の処理で済むために丸め誤差も小さくなると同時に高速な処理も期待できます。

このように IEEE 754-2008 では浮動小数点数の四則演算、平方根と融合積和演算を定めているために、IEEE 754-2008 の浮動小数点数を実装した計算機であれば、これらの計算結果は必ず一致します。しかし、三角函数や対数函数等の初等函数が規格化されていないために、これらの演算は利用して数値計算ライブラリやハードウェア上の処理で異なる可能性があります。

■函数 ulp: 閉区間 $[-x_{\max}, x_{\max}]$ に包含される実数でなければ浮動小数点数として表現できませんが、その区間中で連続的な実数とは違って浮動小数点数は離散的であり、実数

^{*6} Intel の x86_64 では Haswell 以降、AMD なら Bulldozer 以降の CPU が対応。

の符号化は基本的に近似値です。ここで `ulp(Unit in the Last Place)`^{*7} という名前の関数を導入しましょう。この関数 `ulp()` は実数 $x \in [-x_{\max}, x_{\max}]$ を数直線上で挟む二つの浮動小数点数間の最小距離 $\text{ulp}(x)$ を与える関数です。 $\text{ulp}(x)$ は x が大きな数値であれば大きくなり、逆に x が小さな数ならば小さな値を返します。たとえば、整数 2^{52} から 2^{53} の間の整数を浮動小数点数で表現するときに仮数部の最小の数が 2^{-52} になるために `ulp` は 1 になります。このことを Python 2 で確認しましょう：

```
>>> print '%16.0f\n' %(2.0**53);
9007199254740992

>>> print '%16.0f\n' %(2.0**53+1);
9007199254740992

>>> print '%16.0f\n' %(2.0**53+2);
9007199254740994
```

このように絶対値が 2^{53} を越える整数は倍精度浮動小数点数で一意に表現できないことを意味しますが、逆に言えば絶対値が 2^{53} 以下の整数は長整数を使うといった工夫をしなくても倍精度浮動小数点数で一意に表現可能で、大量の整数データを処理するときは後述の BLAS 等の数値行列計算ライブラリを使った高速な演算が可能です。

■丸め/切捨：実数 x と $\hat{x} \in \mathcal{F}$ の対応付けを「丸め」、あるいは「切捨」と呼び、次の 4 種類の操作が規定されています：

丸めと切捨

丸めの種類	概要
1. 上向きの丸め	a 以上の浮動小数点数で最小のものを採用: $\triangleright a \stackrel{\text{def}}{=} \min(\{ x \in \mathcal{F} \wedge a \leq x \})$
2. 下向きの丸め	a 以下の浮動小数点数で最大のものを採用: $\triangleleft a \stackrel{\text{def}}{=} \max(\{ x \in \mathcal{F} \wedge a \geq x \})$
3. 最近値への丸め	a に最も近い浮動小数点数を採用: $\odot a$ と表記
4. 切捨	絶対値が $ a $ 以下で a に最も近い浮動小数点数を採用: $\trianglelefteq a$ と表記

ここでの 1. と 2. の丸めによって実数 x に対応する浮動小数点数 \hat{x} との差の絶対値は $\text{ulp}(x)$ 以下、3. と 4. の操作による浮動小数点数からの距離は $\text{ulp}(x)/2$ 以下になります。

*7 「ウルプ」と呼びます

■丸めと切捨の性質: 演算子 $\bigcirc : \mathbb{R} \rightarrow \mathcal{F}$ を演算子 “ \triangleright ”, “ \triangleleft ”, “ \odot ” か “ \trianglelefteq ” のいずれかとします:

丸めの演算子の性質

-
- | | |
|--|-------------------------------|
| 1) $\bigcirc x = x$ | 任意の $x \in \mathcal{F}$ に対して |
| 2) $x \leq y \Rightarrow \bigcirc x \leq \bigcirc y$ | 任意の $x, y \in \mathbb{R}$ に対し |
| 3) $\odot(-x) = -(\odot x)$ | |
| 4) $\triangleleft(-x) = -(\triangleright x)$ | |
| 5) $\triangleright(-x) = -(\triangleleft x)$ | |
| 6) $(\bigcirc x) \bullet (\bigcirc y) = \bigcirc(x \circ y)$ | |
-

性質 2) より、丸めの演算子は「**大小関係の順序を保つ**」ことが判ります。ただし、比較の演算子 “ \leq ” を “ $<$ ” で置換えられません。実際、 $x \in \mathcal{F}$ に対して $0 < |x - y| < \text{ulp}(y)/2$ を充す実数 $y \in \mathbb{R}$ は $\odot y = x$ になって性質 2) を充さないためです。

浮動小数点数は「**近似値**」としての性格を持っていますが、演算結果も同様に近似値でなければなりません。すなわち丸めの演算子 “ $\bigcirc \in \{\triangleright, \triangleleft, \odot, \trianglelefteq\}$ ” に対して実数上での演算子 “ $\circ \in \{+, -, \times, /\}$ ” と、それに対応する浮動小数点上の演算子 “ $\bullet \in \{\oplus, \ominus, \otimes, \oslash\}$ ” との間には条件 6) を満たすことが IEEE 754 で要求されています。

4.3 数値行列ライブラリについて

4.3.1 BLAS の概要

数値計算処理では、その操作を行列演算に置換えられるものが沢山あります。たとえば連立一次方程式の解法で用いられるガウスの消去法は、連立一次方程式の係数を行列に置換えて、行列処理の話にすることができます。そこで主要な行列処理をライブラリ化して必要に応じて利用すれば、その都度、プログラムを作成する手間もかからないために何かと便利です。数値行列処理システム MATLAB も学生に LINPACK 等の数値行列ライブラリを使わせることが目的でしたが、この LINPACK の後継が「**LAPACK(Linear Algebra PACKage)**」で、LAPACK は線形連立方程式の求解などの数値行列処理の効率的な処理を目的とし、「**BLAS(Basic Linear Algebra Subprograms)**」と呼ばれるルーチン群上で構築され、この BLAS の性能が LAPACK の性能に直接影響します。これら LAPACK と BLAS の公式標準実装は netlib^{*8}で公開されていますが、その実体は

^{*8} <http://www.netlib.orgblas/>

FORTRAN で記述された仕様書的なもので、各種計算機環境向けの細かな調整は利用する個人が行う必要があります。そのために各種計算機環境向けに最適化を行った BLAS が存在します。まず、OSS のものに「**OpenBLAS**」と「**ATLAS(Automaticaly Tuned aLgebrA Software)**」^{*9}、商用のものには Intel の「**MKL(Math Kernel Library)**」と AMD の「**ACML(AMD Core Math Library)**」が代表的です。まず、OpenBLAS は高速処理で知られていた GotoBLAS2[13]^{*10}の後継で、SageMath や NumPy では OpenBLAS を利用するように設定されています。MKL は Intel の CPU 向けに最適化された BLAS で、サポートなしであれば無償利用が可能で、Anaconda, Inc. の Anaconda には NumPy 向けに同梱されています^{*11}。これらの BLAS の大雑把な処理速度は MKL が全般的に優れ、その次が OpenBLAS、それから ATLAS、最後に netlib の公式実装の順番になります。

4.3.2 BLAS の構成

BLAS と LAPACK のルーチンは「**精度 + 行列の型 + 処理**」で名付けられており、その名前から精度、行列の性質と処理内容が判ります。そして、BLAS のルーチンは、その処理内容から三つの水準に分類されます：

BLAS サブルーチンの水準

- 第一水準：ベクトル単体やベクトル同士の演算
- 第二水準：行列とベクトルの演算
- 第三水準：行列同士の演算

これらの水準は BLAS 開発の歴史的な経緯に関係します。まず、1979 年にリリースされた最初の BLAS は、1970 年代のベクトル型計算機での効率的動作を目的に構築されたベクトル演算を中心とする第一水準で、それから行列とベクトルの演算を行う第二水準（1987）、行列同士の演算を行う第三水準（1989）を追加と 10 年かけて拡張されます [42][44]。そして、複素数の 1-ノルムと割算を計算するルーチンを第 0 水準が加わります。なお、BLAS は扱う行列が小さい場合は威力を發揮し難く、行列が大きくなるとその威力を發揮します。

BLAS の最適化では計算機内部の動作を考慮します。まず、主メモリは容量が大きいものの実際にデータが転送される時間（レイテンシ）と単位時間でのデータ転送量（バンド

^{*9} ATLAS と OpenBLAS は BSDL で配布されています。

^{*10} 後藤和茂氏が開発・管理されていた BLAS で、後藤氏が Microsoft に移籍されたために開発が中止されています。

^{*11} 2012 年に前述の後藤氏が Intel に移籍され、チューニングに参加されていることです。また、2015 年より利用者登録することでサポートなしで利用目的を問わず無償利用可能（Community licensing for Intel Performance Libraries）になりました。

幅) の特性があまりよくないという傾向があります。また、一度利用したデータや隣り合うデータは再び利用される可能性が高いために主メモリよりも高速なメモリに利用したデータを一時的に保持しますが、これがキャッシュと呼ばれる仕組みです。このキャッシュの特性で重要なことはバンド幅、レイテンシとキャッシュの大きさの順ですが、近年の CPU はコアを増やして並列処理で効率向上を目指すために複数のコアがデータ転送の帯域を奪い合う問題が顕著になり、キャッシュにデータがある間に並列した演算で「**データの再利用**」が重要になります。また、FORTRAN の配列データは、列データがメモリ上に連続して並び、C/C++ の配列は行データがメモリ上に連続して並びます。前者を「**列優先 (Column major)**」、後者を「**行優先 (Row major)**」と呼びますが、FORTRAN は列優先のために、行データへのアクセスはメモリ上を不連続に飛ぶことになって処理速度で致命的な悪化を招きます。この事態は行優先の C では逆になります。これらのこと踏まえて BLAS の最適化では効率的なデータ転送を考慮しなければなりません。ここで各水準の特徴を挙げておきましょう。

■BLAS の第一水準: ベクトル演算が主要であるために計算量が他の水準と比較して少なく、ベクトルの大きさを N とするとデータ量は $O(N)$ 、計算量も $O(N)$ と N に正比例し、その性能は CPU の理論的性能とメモリバンド幅に依存します。また、データの再利用がほとんどできないために、この水準のルーチンの性能向上はさほど期待できません。

■BLAS の第二水準: 行列とベクトルの演算が主要で、計算量とデータ量共に $O(N^2)$ と N^2 の水準で正比例するために第一水準以上に効果が期待できます。その性能は計算機のメモリバンド幅に依存し、ベクトルのみにデータの再利用性があります。

■BLAS の第三水準: 行列同士の処理が主要で、扱うデータ量は $O(N^2)$ 、計算量が $O(N^3)$ と最多であり、その性能も CPU の理論性能値に依存し、 $O(N)$ 回のデータの再利用性があるために、この面での性能向上も望めます。

4.3.3 ベクトルと行列の表記について

BLAS の引数表記には決まりごとがあり、大文字のアルファベット A, B, C, \dots で行列、小文字のアルファベット v, u, w, \dots で(列)ベクトル、小文字ギリシア文字 α, β, \dots でスカラーを表現し、行列 A を格納する配列を a と小文字のアルファベットで、ベクトル v, u, w を格納する配列を x, y, z と表記します。それからベクトルや行列のスカラー積を $\alpha \cdot A$ や $\beta \cdot v$ と表記し、行列 A と行列 B の積は $A B$ 、行列 A とベクトル v の積を $A v$ と表記します。また ${}^t A$ を行列 A の転置、行列 A の転置と複素共役 ${}^t \bar{A}$ を ${}^H A$ と表記することもあります。

4.3.4 配列への格納方法

ベクトルは 1 次元配列で扱いますが、行列は 2 次元配列、あるいは行列の列、あるいは行を繋いで 1 次元配列として格納する方法^{*12}に加え、行列の特性を生かしてメモリを節約しつつ効率的な処理が行える配列への収納の工夫があります。この収納方法に加えて行列の特性に合せた処理で効率向上が狙えます。ここで行列の収納方法には行列を 1 次元的、あるいは 2 次元的に素朴な方法で配列に転記した「一般形式」、三角行列、対称行列やエルミート行列の性質に合せて成分を規則的に取込んで 1 次元配列で表現する「圧縮格納形式」、帯行列向けの「帯格納形式」の三種類があります。ここで BLAS は FORTRAN で記述されているために FORTRAN でメモリ効率が良くなるように行列を配列に格納します。ここで FORTRAN は「列優先 (Column major)」であるため行列の列単位で配列に格納し、「行優先 (Row major)」の C/C++ では行単位で配列に格納します。なお、FORTRAN 以外で列優先のものに MATLAB, GNU Octave が、C 以外で行優先のものには NumPy があります。ここで MATLAB 系言語であれば [1:10] で生成されるベクトルのサイズを調べることでどちらであるかが分かります。ここでの解説では FORTRAN を念頭にしているために列優先で話を進めます。なお、この行優先、列優先は MATLAB 系言語や NumPy の配列データの格納方法と配列の表現にも関係します。

■一般形式: この書式を利用するルーチンは命名規則 3 の “MM” の箇所が “GE” です。一般形式は行列 A の i 行 j 列成分の A_{ij} をそのまま配列 a の成分 $a[i, j]$ に格納するときと行列の列を繋いで 1 次元配列に収納するときがあり、1 次元配列に格納するときは $m \times n$ -行列 A の i, j -成分を 1 次元配列 a の $i + m \cdot (j - 1)$ 成分に格納します。

■圧縮格納形式: この書式を利用するルーチンは命名規則の “MM” の箇所の二文字目が “P” です。「圧縮格納形式 (Packed storage)」と呼ばれる格納方法は上下三角行列、エルミート行列や対角行列の 1 次元配列への格納で用いられます。上三角行列で $i \leq j$ のときに A_{ij} の値を $a[i + j \cdot (j - 1)/2]$ に格納し、下三角行列であれば、 $i \geq j$ のときに A_{ij} の値を $a[i + (2 \cdot n - j) \cdot (j - 1)/2]$ に格納します。具体的に 4×4 の上三角 U と下三角行列 L の格納状況を以下に示します:

^{*12} MATLAB 系言語の多くでは行列を 1 次元配列とみなして処理ができます。また、NumPy の ndarray 型の多次元配列の実データも 1 次元的にメモリ上に連続して配置されています。

行列の配列への格納方法

$$U : \begin{pmatrix} A_{11} & A_{12} & A_{13} & A_{14} \\ & A_{22} & A_{23} & A_{24} \\ & & A_{33} & A_{34} \\ 0 & & & A_{44} \end{pmatrix} \Rightarrow A_{11} \underbrace{A_{12} A_{22}} \underbrace{A_{13} A_{23} A_{33}} \underbrace{A_{14} A_{24} A_{34} A_{44}}$$

$$L : \begin{pmatrix} A_{11} & & & 0 \\ A_{21} & A_{22} & & \\ A_{31} & A_{32} & A_{33} & \\ A_{41} & A_{42} & A_{43} & A_{44} \end{pmatrix} \Rightarrow \underbrace{A_{11} A_{21} A_{31} A_{41}} \underbrace{A_{22} A_{32} A_{42}} \underbrace{A_{33} A_{43}} A_{44}$$

これらの例から上三角、あるいは下三角の列を並べて1次元配列に取り込んで行く様子が分ります^{*13}。この格納方法は対称行列やエルミート行列でも使えます。なぜなら対称行列は $A_{ij} = A_{ji}$ 、エルミート行列は $A_{ij} = \overline{A_{ji}}$ を充し、上下三角行列と同様に行列の上三角領域があれば復元できるためです。行列を圧縮格納形式で配列に格納したときに前述の命名規則“MM”は対称行列であれば“SP”，エルミート行列であれば“HP”的ルーチンを用います。

■帯格納形式： 帯行列の対角成分付近の帯の成分だけを配列に格納する方式です。帯行列 $A \in M(m, n)$ は k_l 行の劣対角成分（対角成分の下側の成分）と k_u 列の優対角成分（対角成分の上側の成分）を持つものとします：

$$k_l + 1 \left\{ \begin{array}{ccccccccc} & & \overbrace{A_{11} \cdots A_{1(k_u+1)}}^{k_u+1} & & & O & & \\ & \vdots & \ddots & & & \ddots & & \\ & A_{(k_l+1)1} & & \ddots & & & \ddots & \\ & & \ddots & & \ddots & & & \ddots \\ O & & & & & & & \ddots \end{array} \right.$$

行列 A の成分 A_{ij} を $(k_u + k_l + 1) \times n$ の大きさの配列 a に格納します。この収納の方法は netlib で公開されているルーチンのコメント中の記述が参考になります：

```
DO 20, J = 1, N
      K = KU + 1 - J
```

^{*13} Row major であれば行になります。

```

DO 10, I = MAX( 1, J - KU ), MIN( M, J + KL )
A( K + I, J ) = matrix( I, J )
10    CONTINUE
20    CONTINUE

```

‘matrix(I,J)’ が行列 A の i,j 成分である A_{ij} に対応し, ‘A’ が配列 a に対応します。配列 a への格納は列単位で行われ、最初に行列の第一列先頭の成分 A_{11} が $a[(k_u + 1), 1]$, 以降の第一列の成分が続いて配列 a の 1 列目に収納されて $i < k_u$ を充す i 列の先頭 A_{1i} が $a[k_u - i, i]$ に収められます。そして $i \geq k_u$ を充す i 列の先頭 A_{1i} が配列 a の i 列の先頭 $a[1, i]$ に格納されます。つまり, $i > k_u$ を充す配列 a の i 列はその先頭の $a[i - k_u, i]$ が一行目に配置されて収納、すなわち対角成分を挟んで上に行 k_u , 下に k_l 行と帶成分が収納されます。ここで, LAPACK マニュアルに記載されている $m = n = 6$, $k_u = 1$, $k_l = 2$ の行列の例を示しておきます:

帯行列の格納方法

本来の行列 A	\Rightarrow	配列 a への格納状態
$\begin{pmatrix} A_{11} & A_{12} & 0 & 0 & 0 & 0 \\ A_{21} & A_{22} & A_{23} & 0 & 0 & 0 \\ A_{31} & A_{32} & A_{33} & A_{34} & 0 & 0 \\ 0 & A_{42} & A_{43} & A_{44} & A_{45} & 0 \\ 0 & 0 & A_{53} & A_{54} & A_{55} & A_{56} \\ 0 & 0 & 0 & A_{64} & A_{65} & A_{66} \end{pmatrix}$		$\begin{bmatrix} * & A_{12} & A_{23} & A_{34} & A_{45} & A_{56} \\ A_{11} & A_{22} & A_{33} & A_{44} & A_{55} & A_{66} \\ A_{21} & A_{32} & A_{43} & A_{54} & A_{65} & * \\ A_{31} & A_{42} & A_{53} & A_{64} & * & * \end{bmatrix}$

ここでの ‘*’ は成分の配置が行われない箇所を示しています。このように帯行列を列単位で対角成分を積み重ねて格納する方法で本来の行列よりも小さな 2 次元配列に格納できます。この帯格納形式は一般形式と同様に列で並べた一次元配列に変換できます。また ‘ $k_l = 0$ ’ であれば上三角行列, ‘ $k_u = 0$ ’ ならば下三角行列になりますが、格納方法は通常形式に一致します。そして、通常形式の場合と同様に行列 A が対称行列やエルミート行列であれば、その行列の上三角成分、あるいは下三角成分を帯格納形式で格納できます。

4.3.5 ルーチンの命名規則

BLAS と LAPACK のルーチンの名前は「**PMMAAA**」の書式に限定されます。この名前は FORTRAN の函数名の制約に由来し、次の意味があります:

$\underbrace{\mathbf{P}}$ $\underbrace{\mathbf{MM}}$ $\underbrace{\mathbf{AAA}}$
 精度 行列の型 処理内容

これら **P**, **MM** と **AAA** を解説しましょう。

■P: 対象の精度、対象が実数、あるいは複素数であるかを指示します:

精度を表現する文字

	単精度 (32bit)	倍精度 (64bit)	拡張倍精度 (128bit)
実数	S	D	Q
複素数	C	Z	X

■MM: ルーチンが扱う行列の型 (かたち) を指示します。行列が対角行列なら「D」、三角行列なら「T」、対称行列なら「S」、エルミート行列なら「H」で、その他の一般の行列は「G」で指示され、これらの行列の性質に統いて行列の格納方式を示す文字が入ります。ここで一般形式なら「G」、圧縮格納形式なら「P」、帯格納形式なら「B」で、MM はこれらを組合せた文字列「**行列の性質 + 格納形式**」です:

行列の型を示す二文字

BD	二重対角行列	DI	対角行列	GB	帯行列
GE	一般行列	GG	一般行列 (一般行列の対)	GT	一般三重対角行列
HB	エルミート帯行列	HE	エルミート行列	HP	エルミート行列 (圧縮格納形式)
HG	上 Hessenberg 行列	HS	上 Hessenberg 行列	OR	直交行列
OP	直交行列 (圧縮格納形式)	PB	正值対称/エルミート帯行列	PO	正值対称/エルミート行列
PP	正值対称/エルミート行列 (圧縮格納形式)	PT	正值対称三重対角行列/エルミート三重対角行列	SB	対称帯行列
SP	対称行列 (圧縮格納形式)	SY	対称行列	TB	三重対角行列/帯行列
TG	三角行列	TP	三角行列 (圧縮格納形式)	TR	三角行列
TZ	台形行列	UN	ユニタリ行列	UP	ユニタリ行列 (圧縮格納形式)

■AAA: ルーチンの処理内容を指示します。文字数は 2 文字、あるいは 3 文字で、BLAS の多くが 2 文字、LAPACK の多くが 3 文字です。BLAS では「M」が行列、「V」がベクトル、「S」が逆行列計算という意味があり、これらを組合せた名前になっています:

処理内容を示す文字列

MV	行列とベクトルの積
SV	逆行列とベクトルの積
MM	行列同士の積
SM	逆行列と行列の積
EV	固有値問題
QRF	QR 分解

4.3.6 ルーチンの引数について

引数の決まりごとにベクトル v, u を格納する配列は x, y と表記します。これらの配列 x, y には引数「INCX」と「INCY」があり、これらの引数は配列 x, y の添字の増分を指示します。この本では ' d_x ', ' d_y ' とも表記し、これらの増分は 0 より大でなければなりません。なお、GotoBLAS2 の解説 [13] によると INCX と INCY が 1 のときのみ最適化され、それ以外では最適化されていないために処理速度が致命的に低下するとあります。

行列 A を格納する配列は小文字の a 、行列の行数を m 、列数を n と表記し、正方行列では行数と列数を区別せずに m を用います。行列 $A \in M(m, n)$ を格納する配列 a は行列 A の行数よりも行数が多くなることがあります。たとえば連立一次方程式の数値的解法で用いられるガウスの消去法がそうで、方程式の係数行列と定数項で構成されるベクトルを合わせた行列で処理を行います。この拡大した配列 a の行数を指示する引数として「Leading Dimension」があり、行列 A の Leading Dimension を「LDA」と略記し、この本では L_A と表記します。一般形式のときは行列 A を列で繋いで格納した 1 次元配列 a で行列 $A(i, j)$ 成分へのアクセスは ‘ $a[i + j*LDA]$ ’ で行なわれ、行列 A を配列 a の部分配列として包含するのでなければ $LDA = \max(1, m)$ 、つまり、LDA の値として「行列 A の行数 m を指定」になります。行列 A が配列 a に「帯行列格納」されたときは、対角成分数 k_u 、劣対角成分数 k_l の $m \times m$ 行列 A の格納先の配列 a の大きさが $(k_u + k_l + 1) \times m$ になるために引数 LDA として $k_u + k_l + 1$ を指定します。

行列の転置や転置共役を指示する行列変換のフラグは、ルーチンの引数として TRANS がありますが、ここは f と表記し、このフラグで指示される行列の作用素 op_f を以下に示しておきます:

作用素 op_f の挙動		
フラグ f の値	$\text{op}_f(A)$	概要
"N", "n"	A	無変換 (要するにそのまま)
"T", "t"	${}^t A$	転置
"C", "c"	${}^t \bar{A}$	転置 + 共役

フラグ f の値は上記の二重引用符 ("") で括った文字列です。

つぎに三角行列が上三角行列か下三角行列を指示するフラグ「**UPL0**」があります。この本では f_U と表記し、この値と意味を以下にまとめておきます：

フラグ $f_U(\text{UPL0})$ の値と意味		
フラグ f_U の値	概要	
"U", "u"	上三角行列の場合	
"L", "l"	下三角行列の場合	

最後に単位三角行列を指示するフラグは **DIAG** です。ここでは f_D と表記し、その値をまとめておきます：

フラグ $f_D(\text{DIAG})$ の値と意味		
フラグ f_D の値	概要	
"U", "u"	単位三角行列のとき	
"N", "n"	単位三角行列でないとき	

4.3.7 BLAS のルーチン

BLAS の第一水準のルーチンが行列の複製や回転、ベクトルの和やノルムの計算、第二水準が行列とベクトルの演算、そして、第三水準が行列同士の演算です。第一水準のルーチンでは直接、値が返却されますが、第二、第三水準のルーチンの多くは計算結果がルーチンの引数末端のベクトルや行列に対応する配列に代入されます。以降の解説では函数名の先頭の精度を示す箇所を表で「**型**」とし、実数单精度、実数倍精度、複素数单精度、複素数倍精度の順で並べ、型以外の語幹に相当する箇所を「**ルーチン名**」と表示します。

4.3.8 BLAS の第 0 水準のルーチン

BLAS の第 0 水準のルーチンは複素数値に対する 1-ノルムと二つの複素数の商を計算するルーチンです：

第0水準のルーチン

型	ルーチン名	概要
s d	cabs1	$ \Re e(x) _1 + \Im m(x) _1$
s d c z	adiv	$x/y \quad x, y \in \mathbb{C}$

4.3.9 BLAS の第一水準のルーチン

行列やベクトルの複製, 二つのベクトル同士の和や内積, ベクトルのノルムを含みます。計算量も少なく, データの再利用性もなく, 性能は CPU の理論性能値とメモリバンド幅に依存します:

第一水準の BLAS ルーチン一覧

型	ルーチン名	概要
s d c z	axpy	ベクトルの和: $\alpha \cdot v + u$
s d c z	copy	ベクトルの複製
s d c z	swap	ベクトルの入替
s d c z	dot[c]	ベクトルの内積: ${}^t \bar{v} u$
c z	dotu	ベクトルの内積: ${}^t v u$
sd d	sdot	ベクトルの内積: ${}^t \bar{v} u$
s d c z	scal	$\alpha \cdot v$ を計算
cs zd	scal	$\alpha \cdot v$ を計算
s d cs zd	rot	Givens 平面回転を計算
s d c z	rotg	平面回転を生成
s d	rotm	平面回転を適用
s d	rotmg	平面回転を生成
s d sc dz	nrm2	ベクトルの 2-norm: $\ x\ _2$
s d sc dz	asum	$\ \operatorname{Re}(x)\ _1 + \ \operatorname{Im}(x)\ _1$
is id ic iz	amax	絶対値が最大の要素の添字を返却

4.3.10 BLAS の第二水準のルーチン

第二水準に行列とベクトルの積, 行列同士の和の処理です。計算量もそれなりにあり, ベクトルに関してデータの再利用性があって性能はメモリバンド幅に依存します。

三角行列とベクトルの積を計算するルーチン

三角形行列 A の $\operatorname{op}_f(A)x$ や $\operatorname{op}_f(A)^{-1}x$ を計算するルーチンを以下に示します:

三角行列とベクトルの積

型	ルーチン	行列の種類
s d c z	trmv	一般の三角行列
s d c z	tvmv	帶行列
s d c z	tpmv	圧縮格納形式の三角行列
s d c z	trsv	一般の三角行列
s d c z	tbsv	帶行列
s d c z	tpsv	圧縮格納形式の三角行列

ルーチン名のうしろ二文字が“mv”のものは行列とベクトルの積, “sv”のものは逆行列とベクトルの積を意味します。三角形行列は対角成分の下側, あるいは上側が全て 0 になる正方行列で, ルーチンの引数は, この行列の形に関連する三種類のフラグ f_U, f, f_D に加え, 行列 A とベクトル v を格納した配列と行列の行数に対応する n があります。また, 圧縮形式以外の配列を利用するときの引数として $L = \max(1, n)$ を充す変数 L を必要とします。それから, ベクトル v に対応する配列の添字の増分も引数に含まれ, 帯行列を処理するルーチンでは帶行列の幅に対応する整数値 k が引数に加わります。以下に形に関するフラグの取り得る値とその意味を纏めておきます:

行列の形に関する三種類のフラグ

フラグ	取り得る値	概要
f_U	{U, u, L, l}	上三角 (U, u) か下三角 (L, l) を指定
f	{N, n, T, t, C, c}	作用素 op_f のフラグ
f_D	{U, u, N, n}	単位三角行列 (U/u) かそれ以外 (N/n) を指定

ここで行列 A の対角成分が全て 1 の三角行列を「**単位三角行列 (unit triangular matrix)**」と呼びます。

$\alpha \cdot \text{op}_f(A)v + \beta \cdot u$ を計算するルーチン

引数は行列の型による違いを除き, 引数として行列の転置, 共役転置を与える作用素 op_f のフラグ f , 行列の行数と列数に対応する m, n , ベクトルに対応する配列 x の添字の増分 d_x とスカラ α, β , さらに配列 a の行数を指定する引数 L があります:

$\alpha \cdot \text{op}_f(A)v + \beta \cdot u$ を処理するルーチン				
型		ルーチン	行列	
s	d	c z	gemv	一般行列
s	d	c z	gbmv	帯行列
		c z	hemv	エルミート行列
		c z	hbmv	エルミート帯行列
		c z	hpmv	圧縮格納形式のエルミート行列
s	d	c z	symv	対称行列
s	d		sbmv	対称帯行列
s	d		spmv	圧縮格納形式の対称行列

行列を生成するルーチン

第二水準では行ベクトルと列ベクトルの積から生成される行列と同じ大きさのベクトルの和を計算します。ここで「**階数 1 の更新 (rank-1 update)**」と「**階数 2 の更新 (rank-2 update)**」と呼ばれる処理があります。階数 1 の更新が $\alpha \cdot v^t \bar{u} + A \mapsto A$ を行う処理、階数 2 の更新が $\alpha \cdot v^t \bar{u} + A \mapsto A$ を $v^H v$ や $v^H v + v^H v$ によって $n \times n$ の正方行列を生成し、階数 2 の更新が二つの n 次元の列ベクトル v, u から $v^H u$ や $v^H u + u^H v$ より $n \times n$ の正方行列を生成します。そして、階数 1 や階数 2 の更新で生成した行列と正方行列 A との和を計算します：

行列を生成するルーチン				
型		ルーチン	処理内容	条件
c	z	her	$\alpha \cdot v^t \bar{v} + A$	$A = {}^t \bar{A}$
c	z	hpr	$\alpha \cdot v^t \bar{v} + A$	$A = {}^t \bar{A}, A:$ 圧縮格納形式
s	d		syr	$\alpha \cdot v^t v + A$
s	d		spr	$\alpha \cdot v^t v + A$
s	d		ger	$\alpha \cdot v^t u + A$
c	z	gerc	$\alpha \cdot v^t \bar{u} + A$	$A \in M(m, n)$
c	z	geru	$\alpha \cdot v^t u + A$	$A \in M(m, n)$
s	d		syr2	$\alpha \cdot v^t u + \alpha \cdot u^t v + A$
s	d		spr2	$\alpha \cdot v^t u + \alpha \cdot u^t v + A$
c	z	her2	$\alpha \cdot v^t \bar{u} + \bar{\alpha} \cdot u^t \bar{v} + A$	$A = {}^t \bar{A}$
c	z	hpr2	$\alpha \cdot v^t \bar{u} + \bar{\alpha} \cdot u^t \bar{v} + A$	$A = {}^t \bar{A}, A:$ 圧縮格納形式

なお、階数 2 のルーチンは名前の末尾に 2 が付いているために容易に判ります。実際、her, hpr, syr, spr, gerc と geru が階数 1 の更新、her2, hpr2, syr2 と spr2 が階数 2 の更新を行

うルーチンです。

第三水準のルーチン

第三水準のルーチンでは行列同士の積を含む計算処理を行います。データの再利用性は大きく、計算量も極めて大きくなります。そして、ここでの性能はCPUの理論性能値に依存します：

第三水準のBLAS ルーチン

型	ルーチン	処理式	フラグ
s d c z	gemm	$\alpha \cdot \text{op}_{f_1}(A) \text{ op}_{f_2}(B) + \beta \cdot C$	
c z	hemm	$\alpha \cdot A B + \beta \cdot C$ $\alpha \cdot B A + \beta \cdot C$	$s \in \{"L", "l"\}$ $s \in \{"R", "r"\}$
s d c z	symm	$\alpha \cdot A B + \beta \cdot C$ $\alpha \cdot B A + \beta \cdot C$	$s \in \{"L", "l"\}$ $s \in \{"R", "r"\}$
s d c z	trmm	$\alpha \cdot \text{op}_f(A) B + \beta \cdot C$ $\alpha \cdot B \text{ op}_f(A) + \beta \cdot C$	$s \in \{"L", "l"\}$ $s \in \{"R", "r"\}$
s d c z	trsm	$\text{op}_{f_1}(A) X = \alpha \cdot B$ $X \text{ op}_{f_1}(A) = \alpha \cdot B$	$s \in \{"L", "l"\}$ $s \in \{"R", "r"\}$
c z	herk	$\alpha \cdot A^t \bar{A} + \beta \cdot C$ $\alpha \cdot {}^t \bar{A} A + \beta \cdot C$	$t \in \{"N", "n"\}$ $t \in \{"C", "c"\}$
s d c z	syrk	$\alpha \cdot A^t A + \beta \cdot C$ $\alpha \cdot {}^t A A + \beta \cdot C$	$t \in \{"N", "n"\}$ $t \in \{"C", "c"\}$
c z	her2k	$\alpha \cdot A^t \bar{B} + \bar{\alpha} \cdot B^t \bar{A} + \beta \cdot C$ $\alpha \cdot {}^t \bar{A} B + \bar{\alpha} \cdot {}^t \bar{B} A + \beta \cdot C$	$t \in \{"N", "n"\}$ $t \in \{"C", "c"\}$
s d c z	syr2k	$\alpha \cdot A^t B + \alpha \cdot B^t A + \beta \cdot C$ $\alpha \cdot {}^t A B + \alpha \cdot {}^t B A + \beta \cdot C$	$t \in \{"N", "n"\}$ $t \in \{"C", "c"\}$

4.3.11 LAPACK の構成

LAPACK は BLAS を基盤に構築され、線形代数の全般的な数値計算ライブラリになっています。ここでは LAPACK について軽く触れることにします。この LAPACK のルーチンはドライバ、計算、補助の三種類に分類されます。

ドライバルーチン

連立一次方程式を解くルーチンなどがあります。

- 線形方程式

- 線形最小二乗法問題 (LLS)
- 一般化線形最小二乗法 (LSE や GLM) 問題
- 標準固有値問題と特異値分解
- 一般化固有値問題と特異値分解

■線形方程式： ルーチン名の末尾が「SV」で終わるものと「SVX」で終るもの二種類があります。「SV」は Simple driver で $AX = B$ の形式の線型方程式を解きます。「SVX」は Expert driver と呼ばれ、次に示す計算に対応します：

————— Expert driver —————

- ${}^t AX = B, A^* X = B$
- 特異点周辺での行列 A の条件数の計算等
- よりよい解の計算、前方/後方誤差の推定値の計算
- 方程式系の均衡化

「方程式系の均衡化/平衡化」とは、行列 A の相似変換で行列成分の絶対値の差を減らす処理です。数値計算では行列成分の最大値と最小値の差が大きければ計算精度を保つ上で不利なため、この差を小さくして誤差の増大を防ぎます。なお、「SVX」ルーチンは通常の「SV」ルーチンよりも特殊な計算であるため、二倍程度の記憶容量が必要になります。

計算ルーチン

ドライバルーチン内部の実際の計算で用いられるルーチンです。ドライバルーチンに必要な機能を持つものがなければ、この計算ルーチンを組合せて問題を解きます。

補助ルーチン

補助的に用いられるルーチンで、BLAS を拡張したものも含まれます。

DGEMM(倍精度汎用行列演算函数)

DGEMM は BLAS の第三水準で規定される倍精度行列積を計算する函数で $\alpha AB + \beta C$ を計算します。行列演算の多くが、この形式に還元可能なために、この函数が BLAS の処理速度を大きく左右し、「Top 500」のランキングで最も重要な(政治的)意味を持つルーチンになります。

4.4 NumPy による数値計算

4.4.1 NumPy の概要

ここまで浮動小数点数と数値行列ライブラリについて概要を述べました。この節ではこれらの応用として NumPy の利用について簡単な例を交えて必要最低限のことについて解説を行います。

本体を簡素にするために円周率 π やネイピア数といった主要な数学的定数、三角函数、指数函数といった初等函数が Python に読み込まれていません。これらの追加はモジュールで行いますが、標準モジュール `math` で導入される初等函数は実数値のみで、基本的な複素数値函数はモジュール `cmath` が必要です。それでも `math` と `cmath` の双方で定義される函数は豊富なものではなく、指数函数を実数と複素数で利用したいときはモジュール `math` とモジュール `cmath` の函数 `exp()` を切替える必要があります。さらに Python のリストは MATLAB 系言語と比較して機能的に貧弱なままであります。パッケージ NumPy は、これら数学的定数、初等函数や多次元配列とその操作を定義するモジュールで、数値計算ライブラリ SciPy, MATLAB と同等のグラフックス機能を付加する Matplotlib などの多次元配列や数値計算を利用するさまざまなパッケージの基礎でもあります。ただし、NumPy が利用している数値行列ライブラリの特性上、小さな配列に対しては標準モジュールの `math/cmath` の方が効率が良いこともあります。この点に関しては扱う対象と目的に応じて使い分けが厳密には必要です。

Python/SageMath でモジュールの読み込みは `import` 文で行います。なお、「`import numpy`」で読み込むと NumPy で定義される定数、函数の呼出は先頭に ‘`numpy.`’ を付ける必要があります。通常、「`import numpy as np`」で NumPy の読み込みを行って ‘`np.`’ を接頭辞とすることが推奨されています^{*14} また、SageMath のように巨大なシステムでは薦められませんが、Python 上の軽い計算で、名前の衝突が生じる恐れがないときに ‘`from numpy import *`’ で読み込むと NumPy で定義される函数や定数が接頭辞なしの名前で呼出が行えます。ちなみにこの本では基本的に ‘`import numpy as np`’ で読み込みを行なったものとし、NumPy のオブジェクトとしては名前だけですが、実例では接頭辞として “`np.`” を付けて説明します。つまり、NumPy で定義された `sin` 函数を指示するときは “`sin()`” と表記し、実例では “`np.sin(x)`” と表記します

^{*14} Import conventions, http://www.scipy-lectures.org/intro/numpy/array_object.html#numpy-arrays

ここでの解説では Python, SageMath と特に区別はしませんが, SageMath の行列の比較では SageMath 上で NumPy を読み込んで処理を行います。また, Python は SageMath と同じに Python 2 を想定していますが, Numba の例で Python 3 も併用します。Python 2 と 3 の差異は print 文か函数 print() であるかの違いが明瞭な違いですが、内部的には NumPy の配列の実データの持たせ方にも違いがあります。ただし、これらの差異がここでの解説に大きく影響するものではありません。

4.4.2 NumPy の配列とその処理

NumPy の配列の基本

NumPy の配列 (ndarray 型) は多次元配列です。この ndarray 型の配列は MATLAB 系言語のベクトル・行列と異なり、その成分は均質的で、单精度、倍精度、四倍精度の実数、あるいは複素数の浮動小数点数、あるいは文字列をその成分として持つことが可能で、タプルやリストからも構築できます。なお、数値を成分に持つ ndarray 型の配列では四則演算が可能ですが、これらの演算は成分単位の演算になります。ベクトル・行列の積演算はメソッド dot() を用いるか、クラス ndarray の派生クラスの matrix 型に変換してベクトル・行列の演算を行うことになります..

クラス ndarray の配列の形やメモリ上の配置に関する属性を以下に示します:

ndarray の属性

属性名	値	概要
ndim	整数	配列の次元
shape	整数のタプル	配列の形
size	整数	配列の成分総数
itemsize	整数	配列の成分の byte 数
strides	整数のタプル	各次元で移動に必要な byte 数
nbytes	整数	配列が占有する総 byte 数
data	buffer/memoryview	配列データのメモリ上の情報
flags	flagobj	配列のフラグ情報
dtype	文字列	配列の成分の型

クラス ndarray のインスタンスの形が属性 shape に反映され、配列の次元は属性 shape のタプルの長さ、あるいは括弧 “[]” の深さに対応します。また、クラス ndarray のインスタンスの成分はすべて属性 dtype で指示される型を持ち、初期化の時点で未指示であれば与えられた配列成分で和演算が閉じるような型（数値塔に対応）に自動的に変換されます。

つまり, ndarray 型の配列はそのデータ型が均質的な成分で構成されます。ここで dtype として指定可能な値は Python 由来のものと NumPy で追加されたものの二種類があります。数値に限定すると、整数には符号付き (int) と符号なし (uint) があり、そのビット長は 8, 16, 32 と 64bit があります。実数の浮動小数点数 (float) については 16, 32(単精度), 64(倍精度) と 128bit(四倍精度)、複素数の浮動小数点数 (cfloat) は実部と虚部の浮動小数点数の対になるために 32, 64bit があります。そして、True, False のみの bool 型 (8bit) もあります。また、Python 由来の数値型には整数の int, Python 2 であればさらに long, 浮動小数点数には float と cfloat がありますが、実質は int は NumPy の int32, float は float64, cfloat が complex128 になります。配列成分の型の指定によって、配列すべての成分がこれらの型に合わされます。このことが属性 itemsize, strides, nbytes に関係します。まず、属性 itemsize は整数值で、配列の各成分の byte 数で表現した大きさに対応します。たとえば、倍精度浮動小数点数の配列であれば倍精度浮動小数点数の符号長が 64 bit = 8 × 8 byte であるために属性 itemsize は 8 になります。そして、属性 strides は各次元の移動に必要な byte 数を示します。それから属性 data が ndarray 型のオブジェクトの配列データのメモリ上の情報で、Python2 では buffer 型で函数 str() で文字列に変換すると 1 次元データとして格納された配列データを 16 進数として見られますが、Python3 ではより汎用性のある memoryview 型です。この属性 data が実際の配列データに対応し、ここでの ndarray 型オブジェクトのデータは属性 itemsize で指示される均質的なデータがメモリ上に連続して並んだ一次元的な構造です。この一次元的なデータを属性 shape で指示されるビューで取り出す仕様になっています。これらの属性に加えて配列のメモリ上の配置に関するフラグもあります:

flags		
C_CONTIGUOUS	Boolean	配列の格納方法の指定
F_CONTIGUOUS	Boolean	配列の格納方法の指定
OWNDATA	Boolean	配列データ保持方法の指定
WRITABLE	Boolean	書込可能性の指定
ALIGNED	Boolean	連続配置の指定
UPDATEIFCOPY	Boolean	指定不可能

これらのフラグで WRITABLE, ALIGNED と UPDATEIFCOPY はメソッド setflags() で変更しますが、C_CONTIGUOUS と F_CONTIGUOUS は配列の生成時に決定されるために配列の生成後は変更できませんが、これらのフラグは配列データの格納方法を指示し、C_CONTIGUOUS が C 流に行優先、F_CONTIGUOUS が FORTRAN 流に列優先です。このフラグ設定は構築子 ndarray() や特定の形の配列生成を行う函数

や reshape() の様なメソッドにオプション order があれば設定できますが、NumPy は CBLAS を用いるために行優先が既定値です。WRITABLE は配列の書込禁止の指定ができるフラグです。残りの ALIGNED と UPDATEIFCOPY は操作することはまずありません。

この他の配列の属性にビューと呼ばれる配列そのものを変更せずに配列の転置や 1 次元配列に変換した配列を返却する属性があります：

ndarray のビュー変更の属性

属性名	概要
T	配列の転置
real	配列の実部
imag	配列の虚部
flat	配列の 1 次元配列化
ctypes	配列の ctypes 型への変換

これらの属性はビューを返すだけで基のインスタンスに影響を及ぼしません。たとえば、`a.T` で配列 `a` の転置を返しますが、配列 `a` の `shape` が `a.T` の影響で変化することはありません。また、‘`b = a.T`’ で変数 `b` に配列 `a` の転置したビューを持つ `ndarray` 型のオブジェクトが生成されて変数 `b` に束縛されますが、配列 `a` にその操作の影響は生じません。ちなみに配列 `a` の `shape` が (n_0, n_1, \dots, n_k) のときに、転置 `a.T` の `shape` は (n_k, \dots, n_1, n_0) で、配列の成分の対応関係は $a[i_0, i_1, \dots, i_k] \mapsto a.T[i_k, \dots, i_1, i_0]$ です。また、配列 `a` の `a.flat` は `flatiter` 型のオブジェクトで、添字で値の参照は `ndarray` 型と同様にできますが、その名前を入力しても返却されるのは名前に束縛されたオブジェクトの情報が返却され、`ndarray` 型のオブジェクトと動作が異なります。そして、`a.ctypes` は外部函数ライブラリ `ctypes` 向けのデータ型になります。

`ndarray` 型のオブジェクトの生成は構築子 `array()` を使います。このときに引数として与える配列データは角括弧 “[]” のリスト型や丸括弧 “()” で括ったタプル型のオブジェクトです。まず、数値、あるいは文字の列を一重に括弧で括ったリストやタプルからは 1 次元配列が生成されます。この一重に括弧で括られたオブジェクトの列を「行」と呼びます。NumPy では行優先であるために行の成分はメモリ上に連続して並びます。そして、2 次元配列は 1 次元配列のリストやタプル、つまり、行の列から構築されます。そして、`ndarray` 型のオブジェクトの次元は括弧の深さ (=`shape` のタプルの長さ) に対応し、属性 `ndim` の値もあります：

```
>>> import numpy as np
```

```

>>> a = np.array([1,2,3,4,5])
>>> a
array([1, 2, 3, 4, 5])
>>> a.shape
(5,)
>>> a.size
5
>>> b = np.array([[1], [2], [3], [4], [5]])
>>> b
array([[1],
       [2],
       [3],
       [4],
       [5]])
>>> b.shape
(5, 1)
>>> b.size
5
>>> c = np.transpose(b)
>>> c
array([1, 2, 3, 4, 5])
>>> c.shape
(1, 5)
>>> d = np.array([[1, 2, 3], [5, 4, 3]])
array([[1, 2, 3],
       [5, 4, 3]])
>>> d.ndim
2

```

ndarray 型の配列の生成では括弧 “[]” で括ったリストや括弧 “()” で括ったタプルが使えます。ただし、タプルであるとは言え、「1,2,3」のような「式の列」、すなわち、「括弧のない式のタプル」はエラーになります。なお、多次元配列の定義でタプルとリストの混在は問題ありません。そして、これらの括弧の深さが配列の次元に対応します。たとえば、行列は括弧 “[]” の深さが 2 層になるために 2 次元です。ちなみに NumPy の配列の次元は属性 ndim に割り当てられています。そして、配列を 1 次元配列とみなしたときの長さは属性 size、多次元配列の形は属性 shape で調べられます。

構築子 array() で属性 dtype を設定することで配列のデータ型を指示できます。ただし、属性 dtype が未指定のときは、成分として与えられたデータが全て整数であれば整数、実数表記があれば浮動小数点数と、数値塔の考え方でその型が各成分に与えられます：

```
>>> a = np.array([1], dtype=np.complex128)
```

```
>>> a
array([ 1.+0.j])
>>> type(a[0])
<class 'numpy.complex128'>
>>> b = np.array(range(24),dtype=np.double).reshape(4,3,2)
>>> b
array([[[ 0.,  1.],
       [ 2.,  3.],
       [ 4.,  5.]],

      [[ 6.,  7.],
       [ 8.,  9.],
       [10., 11.]],

      [[12., 13.],
       [14., 15.],
       [16., 17.]],

      [[18., 19.],
       [20., 21.],
       [22., 23.]])
```

```
>>> b.ndim
3
>>> b.itemsize
8
>>> b.strides
(48, 16, 8)
>>> b[0,...]
array([[ 0.,  1.],
       [ 2.,  3.],
       [ 4.,  5.]])
>>> b[0,0,...]
array([ 0.,  1.])
```

この例では NumPy の complex128 型 (四倍精度) と double 型 (倍精度) の配列を生成しています。最初の配列 a は一成分のみの配列です。二番目の配列 b は $4 \times 3 \times 2$ の多次元配列です。この配列の型は倍精度であるために成分の大きさは属性 itemsize から 8byte, 次元は属性 ndim から 3 次元であることが判ります。属性 strides は各次元の移動に必要な byte 数のタプルが収納されています。この例の配列 a では $a[0,...]$ の長さが 6×8 より 48 byte, $a[0,0,...]$ の長さが 2×8 より 16 byte, そして, $a[0, 0, 0]$ と $a[0, 0, 1]$ の長さが 1×8 より 8 byte になります。属性 data は Python2 と Python3 で型が異なります。これは NumPy で属性 data として buffer 型を使っていましたが、これが memoryview 型

に切替わったためです。memoryview型はbuffer型よりもPython言語のオーバヘッドを伴わないためにより効率的なアクセスが可能で、より高い汎用性を持っています。

構築子array()によるオブジェクトの生成ではMATLAB風の記号“:”を利用した数列の生成や記号“;”を利用した行列の行の指示ができません。構築子array()が配列のデータとして取り込める書式はリストかタプルに限定されます。また、ndarray型のオブジェクトの演算は基本的に成分単位の演算で、行列やベクトルの積演算はそのままでは行えません。ベクトルと行列の演算を円滑に行うためにNumPyにはmatrix型もあります。このmatrix型はndarray型の派生クラスでベクトルと行列の表現に用いられ、演算もndarray型の成分単位の演算ではなくベクトル・行列演算に対応するように積演算“*”と冪演算“**”が上書きされています。このmatrix型のオブジェクトの生成は構築子matrix()を用い、その際にMATLAB風の行列表記も可能です：

```
>>> import numpy as np
>>> a = np.matrix([[1,2,3],[5,4,3]])
>>> a
matrix([[1, 2, 3],
        [5, 4, 3]])
>>> type(a)
<class 'numpy.matrixlib.defmatrix.matrix'>
>>> b = np.array([[1,2,3],[5,4,3]])
>>> b
array([[1, 2, 3],
       [5, 4, 3]])
>>> type(b)
<type 'numpy.ndarray'>
>>> c = np.matrix(b)
>>> type(c)
<class 'numpy.matrixlib.defmatrix.matrix'>
>>> c
array([[1, 2, 3],
       [5, 4, 3]])
>>> b is c
False
>>> d = np.array(a)
>>> d
array([[1, 2, 3],
       [5, 4, 3]])
>>> type(d)
<type 'numpy.ndarray'>
>>> a.T
matrix([[1, 5],
```

```
[2, 4],  
[3, 3])
```

matrix 型のオブジェクトと ndarray 型のオブジェクトはそれぞれの構築子を使って互いの型に変換できます。また、構築子 matrix() による行列の定義を記号“;”を使って MATLAB 風に行えますが、Python のオブジェクトに記号“;”を使う表記がないために全体を文字列にする必要があります：

```
>>> m1 = np.matrix('[1, 2, 3; 4, 5, 6]')  
>>> m1  
matrix([[1, 2, 3],  
       [4, 5, 6]])  
>>> m2 = np.matrix('1, 2, 3; 4, 5, 6')  
>>> m2  
matrix([[1, 2, 3],  
       [4, 5, 6]])  
>>> m3 = np.matrix('1 2 3; 4 5 6')  
>>> m3  
matrix([[1, 2, 3],  
       [4, 5, 6]])  
>>> m4 = np.matrix('1:3;4:6')  
>>> m4  
matrix([[13],  
       [46]])
```

最初の 3 種類が MATLAB 風の行列定義の方法ですが問題なく生成できていますが最後の記号“;”を用いる例だけが本来意図した行列の生成に失敗しています。このようにスラッシュ書式を使った行列の生成は意図した通りにできないこともあります。

NumPy の多次元配列の生成では、最初に 1 次元配列として生成し、それから多次元配列としての形式をメソッド reshape() で変更するという方法もあります：

```
>>> a = np.random.random((4,3,2))  
>>> a  
array([[[ 0.89448666,   0.12482228],  
       [ 0.15401515,   0.17503209],  
       [ 0.20679215,   0.14524648]],  
  
      [[ 0.89211856,   0.83433077],  
       [ 0.32175697,   0.34052149],  
       [ 0.31405685,   0.99350275]],  
  
      [[ 0.10147491,   0.79617256],
```

```

[ 0.81234934,  0.2516962 ],
[ 0.28776398,  0.66480801]],

[[ 0.99904254,  0.97522365],
 [ 0.68656026,  0.53876777],
 [ 0.67829143,  0.40291106]]])

>>> id(a)
80768496L
>>> a = a.reshape((4,6))
>>> a
array([[ 0.89448666,  0.12482228,  0.15401515,  0.17503209,  0.20679215,
       0.14524648],
       [ 0.89211856,  0.83433077,  0.32175697,  0.34052149,  0.31405685,
       0.99350275],
       [ 0.10147491,  0.79617256,  0.81234934,  0.2516962 ,  0.28776398,
       0.66480801],
       [ 0.99904254,  0.97522365,  0.68656026,  0.53876777,  0.67829143,
       0.40291106]])

>>> id(a)
80525952L
>>>a.reshape((4,3,2))
array([[[ 0.89448666,  0.12482228],
        [ 0.15401515,  0.17503209],
        [ 0.20679215,  0.14524648]],

       [[ 0.89211856,  0.83433077],
        [ 0.32175697,  0.34052149],
        [ 0.31405685,  0.99350275]],

       [[ 0.10147491,  0.79617256],
        [ 0.81234934,  0.2516962 ],
        [ 0.28776398,  0.66480801]],

       [[ 0.99904254,  0.97522365],
        [ 0.68656026,  0.53876777],
        [ 0.67829143,  0.40291106]]])

>>> a.shape
(4L, 6L)

```

この例ではモジュール random の函数 random() を使って $4 \times 3 \times 2$ の乱数配列を生成し、メソッド reshape() で 4×6 の配列に変換していますが、このメソッド reshape() は配列そのものを弄るメソッドではなく、その見栄え (view) を変更するソッドです。ただし、その出力値を变数 a に束縛してしまえば、メソッド reshape() で指示した配列の形への変更

ができます。ndarray型のオブジェクトの配列データは前述のように1次元的にメモリ上に行優先で連続して配置されます。既定値としては行優先で配置されますが、生成時に指定することで列優先で配置することもできます。ただし、列優先の配列を利用して生成する配列は、列優先であることの指定を行わなければ行優先の配列として生成されます。

```
>>> c1 = np.array(range(6)).reshape((2,3),order='C')
>>> f1 = np.array(range(6)).reshape((2,3),order='F')
>>> c1
array([[0, 1, 2],
       [3, 4, 5]])
>>> f1
array([[0, 2, 4],
       [1, 3, 5]])
>>> c1.flags
C_CONTIGUOUS : True
F_CONTIGUOUS : False
OWNDATA : False
WRITEABLE : True
ALIGNED : True
UPDATEIFCOPY : False
>>> f1.flags
C_CONTIGUOUS : False
F_CONTIGUOUS : True
OWNDATA : False
WRITEABLE : True
ALIGNED : True
UPDATEIFCOPY : False
>>> mcl = np.matrix(c1)
>>> mf1 = np.matrix(f1)
>>> mcl
matrix([[0, 1, 2],
       [3, 4, 5]])
>>> mf1
matrix([[0, 2, 4],
       [1, 3, 5]])
>>> mcl.flags
C_CONTIGUOUS : True
F_CONTIGUOUS : False
OWNDATA : True
WRITEABLE : True
ALIGNED : True
UPDATEIFCOPY : False
>>> mf1.flags
C_CONTIGUOUS : True
```

```
F_CONTIGUOUS : False
OWNDATA : True
WRITEABLE : True
ALIGNED : True
UPDATEIFCOPY : False
```

この例ではメソッド reshape() で配列の shape を変更していますが、その際にオプション order に'C' で行優先にしたときと'F' を指定することで列優先にしたことで得られる配列が同じ shape であっても異なることに注意して下さい。なお、行優先のときは行で、列優先のときは列で内部的には連続しています。また、配列の flags からも行優先か列優先かが判ります。なお、配列の生成で order の指定を行わなければ配列は行優先として生成されます。特に NumPy の行列型 matrix の構築子 matrix() にはこの並びの指定ができないため、matrix 型のデータは行優先になりますが、このときは基の配列の内部データではなく、その配列のビューが用いられることに注意が必要です。

特定の配列の生成

NumPy ではよく使われる形式の配列を生成する函数があります。

特殊な形の配列を生成する函数

```
zeros( shape, dtype, order)
ones( shape, dtype, order)
diag( shape, dtype)
eye( 整数, dtype)
tri( 整数, 整数, 整数 , dtype )
shpe ::= "(" | "[" 整数列 "]" | ")"
整数列          ::=      整数 | 整数列 ","
dtype           ::=      "dtype=" keyword
order           ::=      "FORTRAN" | "fortran" | "F" |
                      "C" | "c"
```

■zeros(): 全ての成分が 1 の配列を生成します。dtype を未指定のときは倍精度浮動小数点数 (float64) の型の配列になります。

```
>>> np.ones([2,3])
```

```
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
```

■ones(): 全ての成分が 0 の配列を生成します。dtype が未指定のときは倍精度浮動小数点数 (float64) の型の配列になります。

```
>>> np.zeros([2,3])  
  
array([[ 0.,  0.,  0.],  
       [ 0.,  0.,  0.]])
```

■`diag()`: 第1引数が1次元配列のときは、引数の配列成分を対角成分を持つ2次元配列を生成し、第1引数が2次元配列であれば、その配列の対角成分に対応する1次元配列を構築します。返却される配列の型は対角成分として与えた行(タプル、あるいはリスト)の成分で自動的に定まります。この函数 `diag()` は配列の行優先か列優先かの並びの指定ができません。

```
>>> np.diag([1,2,3])  
  
array([[1, 0, 0],  
       [0, 2, 0],  
       [0, 0, 3]])  
  
>>> np.diag([1,2,3],3)  
  
array([[0, 0, 0, 1, 0, 0],  
       [0, 0, 0, 0, 2, 0],  
       [0, 0, 0, 0, 0, 3],  
       [0, 0, 0, 0, 0, 0],  
       [0, 0, 0, 0, 0, 0],  
       [0, 0, 0, 0, 0, 0]])  
  
>>> np.diag([1,2,3],-2)  
  
array([[0, 0, 0, 0, 0],  
       [0, 0, 0, 0, 0],  
       [1, 0, 0, 0, 0],  
       [0, 2, 0, 0, 0],  
       [0, 0, 3, 0, 0]])  
  
>>> A=np.random.randn(3,3)  
>>> A  
  
array([-0.37655391, -0.80374079,  0.12192416],  
      [-0.50526631,  1.400896   , -0.01749459],  
      [-0.64377851, -1.88182683, -2.361904   ])]  
  
>>> np.diag(A)  
array([-0.37655391,  1.400896   , -2.361904   ])
```

■eye() 指定した大きさで対角成分のみが 1 の 2 次元配列を生成します。dtype が未指定のときは倍精度浮動小数点数 (float64) の型の 2 次元配列になります。この函数 eye() も行優先か列優先かの並びの指定ができません。

■tri(), triu(), tril(): 2 次元配列で、三角行列に対応する配列を生成。これらの函数も行優先か列優先かの成分の並びの指定ができません。

その他の配列の生成方法

CSV 形式のファイルを読み込んで配列を生成する函数 loadtxt() や既存の配列の複製で新たに配列を生成する函数 copy() やバッファから配列を生成する函数 frombuffer() 等が用意されています。ここでは函数 loadtxt() と函数 copy() について簡単に解説しておきます。

■loadtxt(): CSV 形式のファイルを読み込んで配列を生成する函数です。一般的にこのようなファイルにはヘッダの部分があり、そのヘッダの下に CSV 形式のデータが配置されますが、函数 loadtxt() では、読み飛ばすべきヘッダ行の数をオプションの skiprows、実データの区切文字の指定をオプションの delimiter、読み込むべき列の指定はオプションの usecols で行えます。さらに一つの配列ではなく、列単位で出力させることも可能で、この場合はオプションの unpack を既定値の True から False に変更します。なお、函数 loadtxt() の逆操作を行う函数に savetxt() があります。

■copy(): 配列の複製を生成する函数です。Python では変数はオブジェクトを指示する札でしかないとため、「 $a = b$ 」は変数 b のオブジェクトを変数 a に複製して代入するという意味ではなく、変数 b が指示するオブジェクトを変数 a も指示するという意味でしかありません。そのために変数 a でメソッドや函数を使って変更可能なオブジェクトに変更を加えると当然、同じオブジェクトを変数 b が参照しているために、変数 b から見てもオブジェクトが変化していることになります。この状況を避ける必要があるときは函数 copy() で配列の複製を生成します。

スライス表記による配列の生成と成分の取り出し

Python の配列は MATLAB 系の言語と異なり 0 から開始することと、スライス表記の刻幅が始点、終点のスライス表記の末尾に記載する点が異なります。また、MATLAB 系言語と異なりスライス表示が生成的な性格を持たないためにスライス表示は配列の生成では関係しません。つぎに NumPy の構築子 array() を使った ndarray 型の配列の生成と成分の取出の例を示しておきましょう：

```
>>> a = np.array([i*0.2 for i in range(0,11)])
>>> a
```

```
array([ 0. ,  0.2,  0.4,  0.6,  0.8,  1. ,  1.2,  1.4,  1.6,  1.8,
2. ])
>>> a[0]
0.0
>>> a[-1]
2.0
>>> a[0:4]
array([ 0. ,  0.2,  0.4,  0.6])
>>> a[4:]
array([ 0.8,  1. ,  1.2,  1.4,  1.6,  1.8,  2. ])
>>> a[:]
array([ 0. ,  0.2,  0.4,  0.6,  0.8,  1. ,  1.2,  1.4,  1.6,  1.8,
2. ])
>>> a[0:11:2]
array([ 0. ,  0.4,  0.8,  1.2,  1.6,  2. ])
>>> a[:11:2]
array([ 0. ,  0.4,  0.8,  1.2,  1.6,  2. ])
>>> a[0::2]
array([ 0. ,  0.4,  0.8,  1.2,  1.6,  2. ])
>>> a[::2]
array([ 0. ,  0.4,  0.8,  1.2,  1.6,  2. ])
```

ここで示すように Python で配列の添字は 0 から開始します。そして、配列の末尾の成分は -1 で指示できます。以降、-2, -3, ... と負数を指定すると、配列の末尾から要素を取り出すことができます。また、スライス表記を使って配列の一部分を取り出すことも可能で、MATLAB 系言語と同様の添字処理ができます。ただし、注意すべきことは配列が 0 から開始するために、その分、添字が MATLAB 系言語の行列の添字と違いが生じます。それと配列の増分の指定の仕方が MATLAB 系と異なり、 $[n_0 : n_x : dn]$ と添字の増分 dn を末尾に記載します。次に行列のスライス表記の例を示します：

```
>>> a = np.array(range(0,10)).reshape(2,5)
>>> a
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])
>>> a.shape
(2,5)
>>> a[:,1]
array([1, 6])
>>> a[:,2:4]
array([[2, 3],
       [7, 8]])
>>> a[0,:]
array([0, 1, 2, 3, 4])
```

```
>>> a[0:2,2:5]
array([[2, 3, 4],
       [7, 8, 9]])
>>> a[...]
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])
>>> a[...,1]
array([1, 6])
>>> a[1,...]
array([5, 6, 7, 8, 9])
```

と、これも MATLAB と同様です。ただし、ここで Ellipsis“...”を入れています。これは記号“:”で始点と終点を入れなかったときと同様の「省略」の意味もありますが、この Ellipsis にはそれ以上に「次元の省略」の意味があります。たとえば、「a[1,...]」と「a[...,1]」は「a[1,:]」と「a[:,1]」と同じ結果ですが、「a[...]」は配列 a 全てを指示しています。これは 3 次元以上の配列を操作すると明瞭になります:

```
>>> a = np.array(range(0, 3*2*4)).reshape(3,2,4)
>>> a
array([[[ 0,  1,  2,  3],
       [ 4,  5,  6,  7]],
      [[ 8,  9, 10, 11],
       [12, 13, 14, 15]],
      [[16, 17, 18, 19],
       [20, 21, 22, 23]]])
>>> a.shape
(3, 2, 1)
>>> a.ndim
3
>>> a[:, :, 1]
array([[ 1,  5],
       [ 9, 13],
       [17, 21]])
>>> a[..., 1]
array([[ 1,  5],
       [ 9, 13],
       [17, 21]])
>>> a[1, ...]
array([[ 8,  9, 10, 11],
       [12, 13, 14, 15]])
>>> a[1, :, :, ]
```

```
array([[ 8,  9, 10, 11],
       [12, 13, 14, 15]])
>>> a[:,1,:]
array([[ 4,  5,  6,  7],
       [12, 13, 14, 15],
       [20, 21, 22, 23]])
```

ここでは配列 a として 3 次元の配列を生成しています。この配列の大きさは $3 \times 2 \times 4$ とされています。ここで $a[:, :, 1]$ と $a[..., 1]$, $a[1,:,:]$ と $a[1, ...]$ が同じ意味であることが理解されるかと思います。このように Ellipsis“...”は次元を含めた省略記号ですが、添字としての“:”は始点と終点を省略した、その添字が置かれた箇所(次元)のみの省略記号です。このように NumPy では多次元向けの機能が拡張されています。なお、この Ellipsis は Maple では数列を生成する演算子としての意味を持ちます。この性質は NumPy にはありませんが、SageMath ではそのような使い方が可能になっています：

```
sage: [1..10]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
sage: type([1..10])
<type 'list'>
sage: type((1..10))
<type 'generator'>
```

このように SageMath では Ellipsis に generator としての機能を持たせていますが、通常の Python や NumPy にはその機能はありません。あくまでもスライス処理での次元を含めた省略の意味以上はありません。

配列のパターンマッチング処理

NumPy の ndarray 型配列でも MATLAB 系言語と同様の処理が可能です。ただし、matrix 型配列では積演算が行列演算になるために、類似の処理を行うためにあらかじめ構築子 array() で ndarray 型に変換しておく必要があります。また、MATLAB 系言語で条件に合致する添字を出力する函数 find() に対応する函数として NumPy には where 函数があります：

```
sage: a = np.random.randn(5,5)
sage: a

array([[ 0.68168446,  0.28548449,  0.25641134, -0.87469883,  0.21570884],
       [ 0.2518589 ,  1.6882941 , -0.84768403, -0.39994859, -0.58862085],
       [ 1.00923789, -1.48754533, -1.22929953,  0.04197003, -1.27458586],
       [ 0.44328701, -1.33041769,  0.93178793,  1.01597053,  0.43260337],
       [ 0.37590752,  0.37131394,  0.80295911,  1.38822051, -0.30192851]])

sage: x=np.where(a>1)
```

```
sage: a[x]
array([ 1.6882941 ,  1.00923789,  1.01597053,  1.38822051])
sage: x
(array([1, 2, 3, 4]), array([1, 0, 3, 3]))
sage: 10*a*(a>1)-3*(a<0)

array([[ 0.          ,  0.          ,  0.          , -3.          ,  0.        ],
       [ 0.          , 16.88294096, -3.          , -3.          , -3.        ],
       [ 10.09237886, -3.          , -3.          ,  0.          , -3.        ],
       [ 0.          , -3.          ,  0.          , 10.15970527,  0.        ],
       [ 0.          ,  0.          ,  0.          , 13.88220513, -3.        ]])
```

この例では函数 `randn()` で 5×5 の乱数配列を生成し, 函数 `where()` を使って, この配列で 1 より大になる成分の添え字を検出させています. また, ‘ $10*a*(a>1)-3*(a<0)$ ’ がパターンマッチングを使った式で, 配列 `a` の成分で 1 より大の成分を 10 倍, 0 より商の成分を-3 倍し, それ以外の成分を 0 にするという意味になります. MATLAB 系言語や NumPy ではこのような配列/行列に対するパターンマッチングと配列演算を上手く使うことで, if 文や for 文を利用することで生じる言語のオーバヘッドを回避できます. とは言え, どうしても大きな配列に対して for 文を利用しなければならないことがあります. その場合は Numba の利用も検討すべきです.

4.4.3 NumPy の配列演算

MATLAB 系言語では, 加減算, 積, 商, 幂といった基本的なベクトルや行列演算は数式に近い表記です. NumPy を取り込んだ Python もこの点は同様ですが, 演算子の機能に関して微妙な違いがあります. これは NumPy の `ndarray` 型の多次元配列の演算が同じ `shape` の対象に対する成分単位の演算で, 幂乗の演算子が MATLAB 系の言語で用いられる演算子 “`^`” ではなく演算子 “`**`” を使う点です. ちなみに MATLAB 系言語で成分単位の演算は俗に `dot` 演算子と呼ばれる積 “`*`”, 商 “`.`” と幂 “`.^`” を用います. また NumPy でベクトルや行列の積を計算するメソッドとして `dot()` がありますが, こちらは演算子でないために使い勝手の良いものではありません. ところで, NumPy にはクラス `ndarray` を継承したクラス `matrix` があり, このクラス `matrix` の積 “`*`” と幂 “`**`” はベクトル・行列演算に対応した演算子として上書きされています. そのため, 配列の成分単位の演算が中心であれば `ndarray` 型, 行列・ベクトル演算が中心であれば `matrix` 型と使い分けられます.

なお, SageMath でベクトル・行列に対する積 “`*`” と幂 “`^”, “**”” は通常のベクトル・行列の演算であって, ndarray 型の各演算のように成分単位の演算ではありません. 逆に言えば, 成分単位の演算を行うためにはメソッド apply_map() の利用や NumPy の ndarray`

型として計算を行うといった工夫が必要です。

ところで、MATLAB 系言語で成分単位の演算速度を比較すると積演算が最も速く、幂演算は比較的重く、函数演算は幂の指数の影響を受けないために、ある程度大きな演算になると幂よりも函数演算の方が逆転する現象があります。この傾向を SageMath でも確認してみましょう。この検証では配列の 2 乗、3 乗と 12 乗の計算を行い、その処理速度を計測します。計算する配列は 1000 成分の 1 次元の乱数配列とします。ここで乱数配列は NumPy の函数 random.randn() を基にします。SageMath の行列は、この random.randn() の結果を利用して構築子 matrix() で SageMath の matrix 型に変換します。ここで SageMath の浮動小数点数には任意精度のものと倍精度があるために双方の比較もしてみましょう。まず、 ndarray 型の配列計算で処理する内容です：

```
b[1]
M=[]
t1=time.clock();b*b;t2=time.clock();
M.append(t2-t1);
t1=time.clock();b**2;t2=time.clock();
M.append(t2-t1);
t1=time.clock();np.exp(2*np.log(b));t2=time.clock();
M.append(t2-t1);
t1=time.clock();b*b*b;t2=time.clock();
M.append(t2-t1);
t1=time.clock();b**3;t2=time.clock();
M.append(t2-t1);
t1=time.clock();np.exp(3*np.log(b));t2=time.clock();
M.append(t2-t1);
t1=time.clock();b*b*b*b*b*b*b*b*b*b;t2=time.clock();
M.append(t2-t1);
t1=time.clock();b**12;t2=time.clock();
M.append(t2-t1);
t1=time.clock();np.exp(12*np.log(b));t2=time.clock();
M.append(t2-t1);
```

これらの式は ndarray 型の配列 b に対して行う積、幂、函数式で、内容は 2 乗、3 乗と 12 乗の三種類です。また、比較のために SageMath の行列を使って同様の計算をさせます：

```
A[0,0]
M=[]
```

```

t1=time.clock();A.elementwise_product(A);t2=time.clock();
t1=time.clock();A.apply_map(lambda x:x*x);t2=time.clock();
M.append(t2-t1)
t1=time.clock();A.apply_map(lambda x:x^2);t2=time.clock();
M.append(t2-t1)
t1=time.clock();A.apply_map(lambda x:exp(2*log(x)));
t2=time.clock();M.append(t2-t1)
t1=time.clock();A.apply_map(lambda x:x*x*x);t2=time.clock();
M.append(t2-t1)
t1=time.clock();A.apply_map(lambda x:x^3);t2=time.clock();
M.append(t2-t1)
t1=time.clock();A.apply_map(lambda x:exp(3*log(x)));
t2=time.clock();M.append(t2-t1)
t1=time.clock();A.apply_map(lambda x:x*x*x*x*x*x*x*x*x*x*x*x*x);
t2=time.clock();M.append(t2-t1)
t1=time.clock();A.apply_map(lambda x:x^12);t2=time.clock();
M.append(t2-t1)
t1=time.clock();A.apply_map(lambda x:exp(12*log(x)));
t2=time.clock();M.append(t2-t1)

```

これらの例では SageMath の matrix 型のオブジェクトに対して成分単位の演算をメソッド `apply_map()` で実現させています。このメソッドの引数として無名函数を記述すれば、各成分に対してその処理が実行されます。ただし、積のみは成分単位の積演算を行うメソッド `elementwise_product()` があります。そのため、最初にメソッド `elementwise_product()` を用いた 2 乗の計算を追加しています。それから、matrix 型のオブジェクトの生成で、成分を実数とするときに何も指定しなければ任意精度の浮動小数点数のクラス RR のインスタンスとして生成されます。ndarray 型の配列は倍精度浮動小数点数 float64 を成分とする配列として生成しているために、倍精度浮動小数点数のクラス RDF の行列の計算も追加しておきましょう：

配列/行列データの詳細

ndarray 型	<code>10*np.abs(np.random.randn(10000))+eps</code>
ndarray 型	<code>10*np.abs(np.random.randn(100,100))+eps</code>
matrix 型	<code>random_matrix(RR,10000,1).apply_map(lambda x:abs(x)+eps)</code>
matrix 型	<code>random_matrix(RDF,10000,1).apply_map(lambda x:abs(x)+eps)</code>
matrix 型	<code>random_matrix(RR,100,100).apply_map(lambda x:abs(x)+eps)</code>
matrix 型	<code>random_matrix(RDF,100,100).apply_map(lambda x:abs(x)+eps)</code>

ここで、`eps = np.finfo(float).eps` としますが、この `eps` は機械イプシロンで、函数式で `log` フィルクスを用いているため、0 で処理が止まることを避けるために用いています。最初に ndarray 型の配列の `shape` が 10000 と 100×100 で比較した結果です：

NumPy: float64

式	10000-成分	100 × 100-成分
$x * x$	0.00011299999999891952	9.1999999995096e-05
$x * 2$	0.00044399999998745443	0.00010700000001406806
$\exp(2 \log x)$	0.0005280000000027485	0.0003630000000214295
$x * x * x$	0.00010899999998059684	9.599999998499698e-05
$x * 3$	0.0004639999999938027	0.0005740000000002965
$\exp(3 \log x)$	0.0005130000000121981	0.0003849999999430656
$\underbrace{x * \dots * x}_{12}$	0.0004020000000082291	0.0002199999998456587
$x * 12$	0.0005160000000046239	0.0005890000000192686
$\exp(12 \log x)$	0.0004399999999755346	0.000389999999816828

あまり厳密なベンチマークではありませんが、おおよその傾向が判ります。配列の形に関するでは単純に成分が 1 次元的に連続で並ぶ `shape` が 10000 の配列よりも `shape` が 100×100 の配列の方が高速に処理ができます。演算に関するでは積や函数式よりも幂が比較的重い処理で、積は計算量の増加に比例して処理時間がかかるものの処理速度の面では速く、函数は指数の大きさに比較的左右されずに処理ができるという常識的な結果が得られています。同様の内容を SageMath の行列で処理した結果を示しておきましょう：

SageMath: 10000×1		
式	RR	RDF
<code>elementwise_product()</code>	0.003810999999846877	0.00126299999994463
$x * x$	0.5370900000000063	0.07301000000001068
$x * * 2$	0.4662849999999992	0.0535849999999822
$\exp(2 \log x)$	0.6000589999999875	0.05980900000000133
$x * x * x$	0.9020840000000021	0.04311000000001286
$x * * 3$	0.6552650000000142	0.051678000000009663
$\exp(3 \log x)$	0.7265020000000106	0.0597509999999724
$\underbrace{x * \cdots * x}_{12}$	0.4797990000000141	0.0429070000000137954
$x * * 12$	0.46864400000001183	0.04351299999990144
$\exp(12 \log x)$	0.6582860000000039	0.0640849999999773

ここでの処理は `shape` が 10000 の配列に対応する 10000×1 の行列に対して `ndarray` と類似の計算を行っています。この結果から数値計算では任意精度の RR よりも倍精度浮動小数点数の RDF の方が 9 倍程度は高速ですが、`ndarray` 型の演算と比べると高々 $1/100$ 程度の処理速度でしかなく、メソッド `apply_map(lambda x:x*x)` よりもメソッド `elementwise_product()` の方が 60 倍程度高速ですが、このメソッド `element_product()` の処理も `ndarray` 型の積演算と比較して $1/50$ 程度でしかありません。つぎに 100×100 行列で同じことをさせてみましょう：

SageMath: 100×100		
式	RR	RDF
<code>elementwise_product()</code>	0.004983000000009952	0.00651899999997277
$x * x$	0.464446000000009463	0.4907630000000154
$x * * 2$	0.5991640000000018	0.558353000000011
$\exp(2 \log x)$	0.6539089999999987	0.649574999999987
$x * x * x$	0.657161999999996	0.4653920000000847
$x * * 3$	0.5640470000000164	0.513542000000001
$\exp(3 \log x)$	0.619923	0.609899999999819
$\underbrace{x * \cdots * x}_{12}$	0.47851600000001326	0.4766840000000059
$x * * 12$	0.48216100000001916	0.468801999999824
$\exp(12 \log x)$	0.6141540000000134	0.623852999999969

興味深いことは RR と RDF の演算の違いが 100×100 の行列のメソッド `apply_map()`

による演算でほとんどなくなることです。実際、メソッド `apply_map()` の実体は Python の内包表現^{*15}を使ったマッピング処理のために RR と RDF の違い以上に Python 言語のオーバーヘッドが問題になります。行列が 10000 のときは配列が行優先であることからメモリ・アクセスの問題がなかったために `ndarray` 型と比べて 1/100 で済んだところが、行列が 100×100 になると各成分へのメモリ・アクセス等の問題も加わってさらに遅延したと考えられます。したがって、配列成分の個別の演算であれば SageMath の `matrix` 型ではなく、NumPy の `ndarray` 型を積極的に利用すべきという結論になります。

4.5 Numbaについて

ここで Python を使った数値計算で注目されている手法が Numba を利用する方法です。この Numba^{*16}は Python の数値配列と関連する函数のコンパイラで、NumPy の函数が Numba に対応していれば、C/C++ や FORTRAN と同程度の処理が可能になります。Numba Project は Anaconda の開発元である Anaconda, Inc. と The Gordon and Betty Moore Foundation が支援しており、Numba は Anaconda に包含されていますが、SageMath のパッケージには含まれてはいません。

ここでは Numba のホームページにある例を利用して簡単な確認をしておきましょう。まず、Numba の函数の記述です：

```
import numba
import numpy as np
import time
from matplotlib import pyplot as plt

@numba.jit
def sum2d(arr):
    M, N = arr.shape
    result = 0.0
    for i in range(M):
        for j in range(N):
            result += arr[i, j]
    return result
```

*15 実質的に for 文。

*16 <https://numba.pydata.org>

この例ではデコレータ@numba.jit を使って何でもない Python の関数を Numba の JIT(Just In Time Compilator, 実行時コンパイラ) でコンパイルすることを指示しています。当然、Numba の JIT に対応したオブジェクトでなければ Numba のご利益はありません。この動作確認を次の例で行ってみましょう。この動作確認は Anaconda3 で行うことにします：

```
N0 = 10
rslt = np.zeros((N0,N0,5))

for i in range(N0):
    M = 3**i
    for j in range(N0):
        N = 3**j
        a = np.random.rand(M, N)
        sum2d(a)

        start = time.clock()
        sum2d(a)
        rslt[i, j, 0] = time.clock() - start

        start = time.clock()
        np.sum(a)
        rslt[i, j, 1] = time.clock() - start

        start = time.clock()
        a.sum()
        rslt[i, j, 2] = time.clock() - start

        start = time.clock()
        sum(sum(a))
        rslt[i, j, 3] = time.clock() - start

s = 0
start = time.clock()
for m in range(M):
    for n in range(N):
```

```
s += a[m, n]
rslt[i, j, 4] = time.clock() - start

from mpl_toolkits.mplot3d import Axes3D
T = ['Numba, for', 'Numpy, func:sum', 'Numpy, method:sum',
      'Python, sum', 'Python, for']
x = np.arange(10)
y = np.arange(10)
for i in range(5):
    fig = plt.figure()
    ax = Axes3D(fig)
    X, Y = np.meshgrid(x, y)
    Z = np.log(rslt[..., i])
    ax.contourf3D(X, Y, Z)
    plt.title(T[i])
    plt.ylabel('Column (3**x)')
    plt.xlabel('Row (3**y)')
    plt.show()
```

ここでは倍精度浮動小数点数の2次元配列の総和を、Numba, NumPyの関数sum(), メソッドsum(), Python本体の関数sum()とfor文で計算します。2次元配列は 1×1 から $3^9 \times 3^9$ の範囲で計算し、縦軸を列数、横軸を行数、高さを処理速度の対数値として結果を3次元的に可視化してみましょう：

for文以外は2次元配列のshapeの影響が現われています。まず、for文は等高線の間隔が均等になっていますが、ここで等高線図は計算時間を対数関数で処理しているために、for文は単純に計算量でその処理時間が決定付けられることが判ります。このことはNumbaとNumPyに関しても配列が大きくなるにしたがって、その傾向が見られますが、NumbaとNumPyの関数sum()では配列が2000要素以下で2次元配列の行数や列数の影響も強く出てきます。この傾向が顕著なものがPythonの組込の関数sum()です。このPythonの関数sum()はndarray型の配列に対しては2次元配列の各列の総和リストを返却する関数として定義されていますが、NumPyの配列はCの行優先でデータが保存されるために行方向がメモリ上連續し、列方向はメモリ上不連續となり、その結果、列間の移動はメモリ上飛び飛びに移動します。そのために列が増加することは関数sum()にとってメモリ上のアクセスの手間が増えて処理時間の増大に直接関係します。この傾向はNumbaやNumPyでもその傾向が見られます。また、 $3^9 \times 3^9$ でのNumba, NumPy,

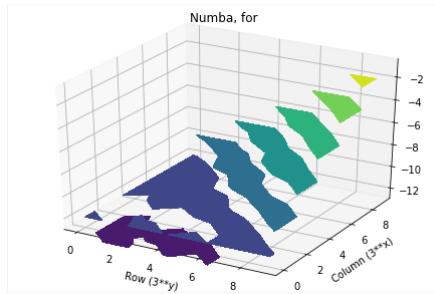


図 4.1 Numba, for

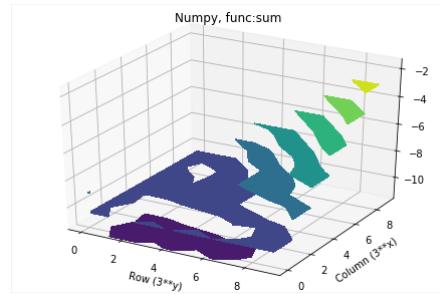


図 4.2 NumPy, Function: sum

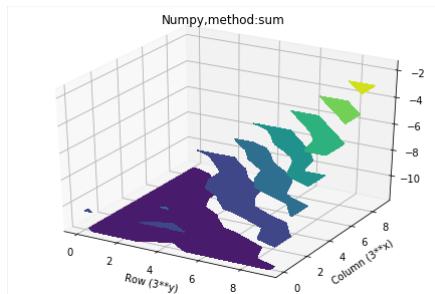


図 4.3 NumPy, Method: sum

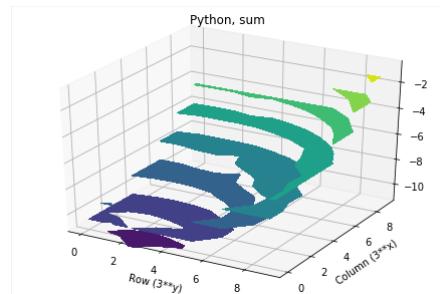


図 4.4 Python, sum

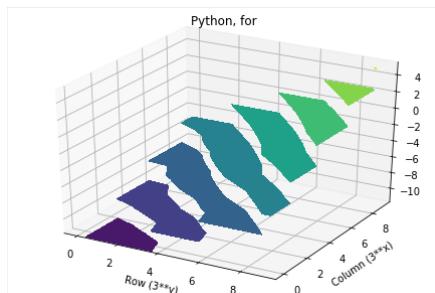


図 4.5 Python, for

Sum と For の比は $0.0038 : 0.0012 : 0.0030 : 1$ と for 文は NumPy の 816 倍も処理時間を必要で、Numba は行列が大きくなると NumPy に劣るもの、要素数が 2000 以下であれば NumPy よりも高速な場合もあって、その威力は十分でしょう。ちなみに Anaconda3 の NumPy では BLAS として MKL が搭載されており、BLAS として OpenBLAS を用いている SageMath で同様の処理を行うことで、OpenBLAS の評価もえることになります。これらの結果から言えることは、数値行列の処理では for 文は Numba のような工夫をしない限り御法度であり、Numba が高速であるとはいえ、行列計算で表現できるものはやはり行列計算として表現すべきであると言えるでしょう。

4.6 SageMath と NumPy の計算比較

では、上記と類似の処理を行ったときに SageMath はどのような傾向を示すでしょうか？ 残念ながら、SageMath には Numba が組込まれていないために Numba 抜きで先程の計算を行ってみます。ここで計算では SageMath の行列 (RDF) で処理した場合も含めています。

```
M0 = 8
rsltm = np.zeros((M0,M0,4))

for i in range(M0):
    M = 3**i
    for j in range(M0):
        N = 3**j
        a = random_matrix(RDF, M, N)
        b = np.array(a)
        a[0,0];
        b[0,0];

        start_t = time.clock()
        sum(sum(a))
        end_t = time.clock()
        rsltm[i, j, 0] = end_t - start_t

        start_t = time.clock()
        np.sum(b)
        end_t = time.clock()
        rsltm[i, j, 1] = end_t - start_t

        start_t = time.clock()
        b.sum()
        end_t = time.clock()
        rsltm[i, j, 2] = end_t - start_t

s = 0
```

```

start_t = time.clock()
for m in range(M):
    for n in range(N):
        s += a[m, n]
end_t = time.clock()
rsitm[i, j, 3] = end_t - start_t

from matplotlib import pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
x = y = np.arange(M0)
T = [ 'Matrix: sum' , 'NumPy: func.sum' ,
      'NumPy: method.sum' , 'Matrix: for' ]
for i in range(4):
    fig = plt.figure()
    ax = Axes3D(fig)
    X, Y = np.meshgrid(x, y)
    Z = np.log(rsitm[..., i])
    ax.contourf3D(X,Y,Z)
    plt.title(T[i])
    plt.ylabel('Column (3**x)')
    plt.xlabel('Row (3**y)')
    plt.show()

```

ここで乱数行列の生成を SageMath の函数 random_matrix() を用いています。ただし、考察すべき世界は倍精度の浮動小数点数に限定されるために RDF 上の行列とします。また、ここでの比較では NumPy での計算とするために SageMath 上の乱数行列を ndarray 型に変換しておく必要がありますが、この変換は構築子 array() で行えます。また、Python では函数で生成された配列データは必要になった時点で配列にコピーされるため、計算時間の計測ではこのことを考慮しておく必要があります、sum(b) と無駄に配列の計算を行なっている理由はそのためです。

配列の大きさが $3^7 \times 3^7$ のときに SageMath では、SageMath の函数 sum(), NumPy の函数 sum() と for 文の比が $0.4533 : 0.0001 : 0.0001, 1$ と for 文が無駄に遅い点は Python の場合と同様ですが、SageMath の函数 sum() は for 文よりもマシな程度で、NumPy の計算には敵いません。興味深いことは SageMath の函数 sum() は ndarray 型の配列に対しては NumPy の函数 sum() と同程度の結果になることです。つまり、SageMath の函数 sum() はなんらかの形で BLAS を利用していることが判ります。そして、メソッド sum()

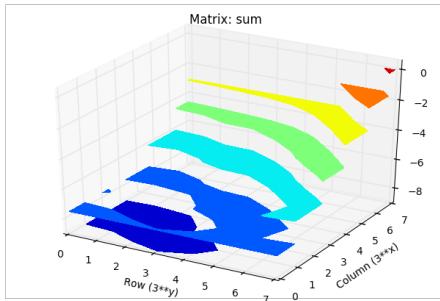


図 4.6 Matrix: sum

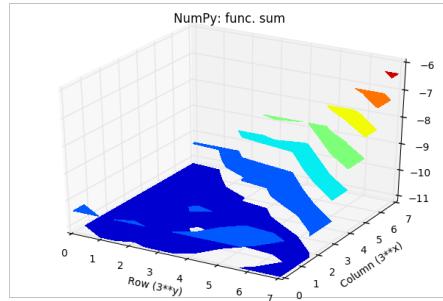


図 4.7 NumPy, Function: sum

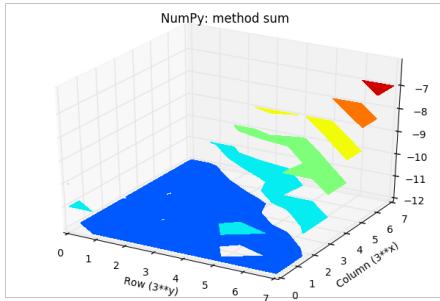


図 4.8 NumPy, Method: sum

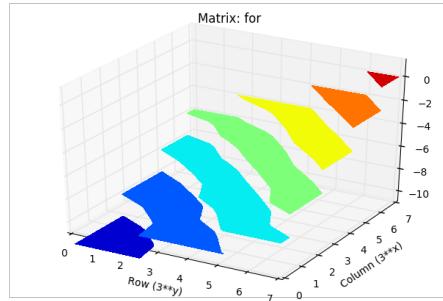


図 4.9 Python, sum

は函数 `sum()` よりも格段に高速であることも判ります。つまり、オブジェクトに近接したメソッドの処理が函数よりも高速であると判断されます。これらの結果から、大きな数値行列の計算は可能な限り `ndarray` 型の配列として数値計算を実行することが有利であると結論付けることができます。

このように SageMath は数式処理システムとして比較的高速な数値計算が可能であるとはいえる、NumPy と比べて著しく高速ではありませんが、SageMath が NumPy を包含しているために「繋目なし」に両者の都合の良い機能を利用することができます。また、数式処理システムはその数式処理の機能の強力さの一方で数値計算機能の貧弱さから、プロトタイプの検討に向いているとされていましたが、SageMath はそのプロトタイプの検討だけではなく NumPy で可能な比較的大規模な規模の計算までをカバーすることができます。そして、OSS であることから函数やメソッドの処理に疑問があれば確認ができるだけではなく、改良することさえもできます。このことが既存の数式処理や MATLAB 系言語と比べて格段に優れている点であり、この理由だけでも SageMath を使いこなすべきであると言えます。

第5章

数学的対象の表現

5.1 はじめに

SageMath は仮想端末上で利用していると、そのプロンプトが ‘sage:’ であることを除いて IPython を Python のシェルとして使っている状況と大差がなく、Jupyter 上で使っていれば kernel を Python にしている状況と違いがほとんどありません。だからといって SageMath は各種アプリケーションやライブラリの入出力を整えただけの Python の拡張ではありません。実際、SageMath は数学的対象を的確に、より効率的に処理するための工夫があり、そのために Python と SageMath の両者に違いが生じます。その具体的な違いが演算子 “ \wedge ” に現われています。Python とその上で動作するパッケージの SymPy で演算子 “ \wedge ” は排他論理和の演算子として扱われ、演算子 “ $\wedge\wedge$ ” が冪乗の演算子として用いられていますが、式を扱う分野で幅広く利用されている組版ソフト TeX で演算子 “ \wedge ” が冪演算子として用いられており、このこともあって多くの式処理で冪乗の演算子として記号 “ \wedge ” が用いられています。そのこともあって SageMath でも式の冪乗の演算子として演算子 “ \wedge ” が用いられていますが、このことが演算子 “ \wedge ” が冪乗の演算子として全面的に置き換えられていることを意味しません。実際、SageMath 上で int 型や float 型の数に演算子 “ \wedge ” を使うと本来の論理排他和になります。このことは SageMath が表で扱っている整数や実数といった数自体が本来の Python のそれと別物であることを示唆しています。

実際にこのことを確認してみましょう。まず、SageMath 上で組込函数 type() を使って ‘1’ と ‘1.0’ という表記がどのような意味付けをされた対象であるか調べてみましょう^{*1};

```
sage: type(1)
<type 'sage.rings.integer.Integer'>
sage: type(1.0)
<type 'sage.rings.real_mpfr.RealLiteral'>
```

このように SageMath で ‘1’ という表記から生成される対象は sage.rings.integer.Integer, ‘1.0’ で生成される対象は sage.rings.real_mpfr.RealLiteral というクラスのインスタンスであることが判ります。同様に Python 2.x の結果を次に示しておきましょう;

```
>>> type(1)
<type 'int'>
>>> type(1.0)
<type 'float'>
```

^{*1} 組込函数 type() はオブジェクトの型を調べることができるだけではなく、指定した型を持つオブジェクトを生成する能力を持つ函数でもあります。

Python 2.x で ‘1’ は int 型, ‘1.0’ は float 型であると返しており^{*2}, 先程の SageMath の結果と違います. こちらは通常の計算機言語で見られるように整数と浮動小数点数であることを示しています. この観察から SageMath で数を表現するオブジェクトが SageMath での数学的構造に対応するオブジェクトで置き換えられていることが判ります. ところで, リテラルが一致する二つのオブジェクトであっても属するクラスが異なれば同一の処理が行えない可能性があります. オブジェクト指向プログラミングではオブジェクトがどのクラスに属するものであるかを常に意識しなければなりません. この点を不明瞭にしていると, 「使える筈のメソッドが使えない」という一見, 意味不明な現象に陥ります.

そのような例を示しておきます;

```
sage: 8.factor()
2^3
sage: for i in range(4,10):
        print i.factor()

AttributeError                               Traceback (most recent call last)
<ipython-input-117-fbb8ce7643ba> in <module>()
      1 for i in range(Integer(4),Integer(10)):
----> 2     print i.factor()

AttributeError: 'int' object has no attribute 'factor'
sage: for i in range(4,10):
        print Integer(i).factor()

2^2
5
2 * 3
7
2^3
3^2
```

この例では最初に整数 8 の因数分解をメソッド factor() で行い, それから最初に 4 から開始して 10 を超えない整数の因数分解を for 文を使って実行させようとしています. ところが, 変数 i に割り当てられたオブジェクトが int 型であるというエラーが出ています. そこで, for 文で生成した変数 i に割り当てられた数を SageMath の整数型の構築子 Integer() で初期化してやると今度は希望通りの処理ができます. すなわち, for 文で生成した数オブジェクトが Python の int 型であって SageMath の Integer 型ではなかつたということですが, 表示も同じものであるだけに混同し易く, 一旦, 混同してしまえば, 「整数」と思い込んでいる分, 分かり難い間違いです. このように処理すべきオブジェクト

^{*2} Python 3 では “type” ではなく “class” になります.

が「何であるか」と「どのようなものか」を常に意識していなければなりません。

この例の意味することはそれだけではなく、整数や実数の表現が SageMath と Python で異なることです。ところで、気になることは Python の数 ‘1’ と SageMath の数 ‘1’, つまり Python の int 型と SageMath の sage.rings.integer.Integer 型にある間隙です。この隙間にどのような定義や設定があるのでしょうか？また、どのような処理が行われているのでしょうか？この章では、Python における数の構成を含めて SageMath でどのような実装が行なわれているかを観察することを目的にします。

5.2 Python の数の構成

5.2.1 Python における数オブジェクトの扱い

おおよその計算機の CPU には整数、実数といった数の表現とそれらの主要な演算は最初から実装されています。実際、自然数や整数は 2 進数で表現され、実数は浮動小数点数と呼ばれる 2 進数表現への符号化として近似され、さらに浮動小数点数についてはその四則演算、大小関係やいくつかの演算、丸めの処理も含めて IEEE 754 で規格化されています。そのために計算機言語で、学習目的以外で数そのものと演算をフレーベの「算術の基礎」や集合論のように最初から全てを構成する必要がなく、ハードウェアに実装されている機能をどのようにして効率良く使うかということが問題になります。ただ、オブジェクト指向言語では扱う対象が全てオブジェクトで、それらの関係を利用して処理を行うという性格上、数についても相互の関係を明確にしておく必要があります。たとえば、整数、有理数と実数には集合として包含関係があり、これらの算術では共通の演算子を用いています。そして個々の数はそれらの集合のメンバー、つまり、インスタンスであり、演算はインスタンスのメソッドとして捉えられます。そして、そのメソッドがどのクラスから派生したものであるかを捉えることで、それらが保持するメソッドの体系的な捉え方が可能になります。このような抽象的な関係を入れるために Python には「**抽象基底クラス (abstract base class, ABC)**」と呼ばれるメタクラスが用意されています。

ここで Python の数のクラス (Numeric Class) の定義は PEP-3141^{*3}に沿って行われています。この PEP-3141 は PEP-3119^{*4}で導入された「**抽象基底クラス (ABC, Abstract Base Class)**」を Python の数の構成に適用しています。このことは「Python 言語リファレンス」の数の説明にて、「numbers.Number’, ‘numbers.Real’, ‘numbers.Complex’

^{*3} 原文:<http://www.python.org/dev/peps/pep-3141/>

^{*4} 原文: <http://www.python.org/dev/peps/pep-3119/>

日本語訳：<http://mft.la.coocan.jp/script/python/pep-3119.html>

といった型の表記からも伺え、実際、これらのクラスは数の抽象基底クラスです。これらのクラスに対応する Python の int, long, float, complex といった型が抽象クラスを現実化した具象化クラス (concrete class) と呼ばれるオブジェクトに該当します。とは言え、int 等の型は PEP-3141 よりも当然、古くから存在する型で、モジュール numbers も通常の Python で最初に読み込まれるモジュールではありません。モジュール numbers は実質的に「**あとづけ**」のモジュールで、数オブジェクトの階層付けで用いられています。そして、その目的は既存の数のクラスを PEP-3141 で提示された数の体系に統合するためです。では、このモジュール numbers にはどのようなことが記載されているのでしょうか？そこでこのモジュールの内容を眺めてみましょう。

5.2.2 numbers モジュールを眺めてみよう

こののはじめにモジュール numbers がどのようなものであるかを知るために、Python の関数 help() を利用してみましょう。ここで Python のインタプリタを利用するのであればあらかじめ ‘import numbers’ で numbers モジュールを読み込んでから、SageMath ならそのまま ‘help(numbers)’ と入力してみましょう；

Help on module numbers:

NAME

numbers – Abstract Base Classes (ABCs) for numbers, according to PEP 3141.

FILE

/usr/local/Cellar/python/2.7.9/Frameworks/Python.framework/Versions/2.7/lib/python2.7/numbers.py

MODULE DOCS

<http://docs.python.org/library/numbers>

DESCRIPTION

TODO: Fill out more detailed documentation on the operators.

CLASSES

__builtin__.object
Number
Complex
Real
Rational
Integral

ここでは macOS 上の SageMath での関数 help() の結果を示していますが、モジュール

ル numbers が PEP-3141 に基づく抽象基底クラス (ABC) であることが明記され、CLASSES の項目でモジュール numbers で定義されるクラスの親子関係 (階層構造) を字下げで示しています。ここで定義されるクラスは object を頂点に Number, Complex, Real, Rational, Integral の階層があり、この階層がクラス間の継承関係 (親-子) を示しています。この階層構造は「**数値塔 (Numerical tower)**」と呼ばれる Python の数オブジェクトの階層構造に対応します。ただし、数値塔では整数、有理数、実数、複素数と派生クラスを塔の上側に配置するために、この CLASSES での表記と逆になります。この数値塔で重要なことは塔の上側の型 (継承する側) とその土台の型 (継承される側) との間の演算で土台の型に自動変換 (暗黙の型変換, implicit conversion) が行われ、その逆は明示的に型変換を行わない限り生じません。たとえば、整数の和、積といった演算は整数内部で閉じていて、実数を表現する浮動小数点数へは型の変換が必要です。しかし、整数と浮動小数点数の演算では自動的に浮動小数点数の演算として型の変更が行われ、結果も浮動小数点数になるために型変換は不要です。この自動変換が暗黙の型変換です。

また、このオンラインヘルプの FILE の項目にモジュール numbers.py の在処が掲載されています。そこで、ファイル numbers.py からクラス Number の定義を抜き出しておきましょう；

```
# Copyright 2007 Google, Inc. All Rights Reserved.
# Licensed to PSF under a Contributor Agreement.

"""Abstract Base Classes (ABCs) for numbers, according to PEP 3141.

TODO: Fill out more detailed documentation on the operators."""

from __future__ import division
from abc import ABCMeta, abstractmethod, abstractproperty

__all__ = ["Number", "Complex", "Real", "Rational", "Integral"]

class Number(object):
    """All numbers inherit from this class.

    If you just want to check if an argument x is a number, without
    caring what kind, use isinstance(x, Number).
    """

    __metaclass__ = ABCMeta
    __slots__ = ()

    # Concrete numeric types must provide their own hash implementation
    __hash__ = None
```

クラス Number の定義では import 文を使ってモジュール abs からクラス ABCMeta 等の読み込みとクラス属性 __metaclass__ に ‘ABCMeta’ を割り当てています。これらの設定

は Python の抽象基底クラスの定義で必須です。ところで、このクラス Number にはメソッドの設定がなく、クラス属性の設定だけです。すなわち、このクラス Number の役割は数という枠組を提供することが目的で、数が何であり、それがどのような性質であるかを指定することではありません。数の本質的な属性やメソッドは、クラス Number を継承するクラス Complex 等のサブクラスに記載されます。そこで今度は複素数を表現する抽象基底クラス Complex の numbers.py からの抜粋を載せておきましょう；

```
class Complex(Number):
    """Complex defines the operations that work on the builtin complex type.

    In short, those are: a conversion to complex, .real, .imag, +, -,
    *, /, abs(), .conjugate, ==, and !=.

    If it is given heterogenous arguments, and doesn't have special
    knowledge about them, it should fall back to the builtin complex
    type as described below.
    """

    __slots__ = ()

    @abstractmethod
    def __complex__(self):
        """Return a builtin complex instance. Called for complex(self)."""

    # Will be __bool__ in 3.0.
    def __nonzero__(self):
        """True if self != 0. Called for bool(self)."""
        return self != 0

    @abstractproperty
    def real(self):
        """Retrieve the real component of this number.

        This should subclass Real.
        """
        raise NotImplementedError

-- 略 --

    @abstractmethod
    def __eq__(self, other):
        """self == other"""
        raise NotImplementedError

    def __ne__(self, other):
        """self != other"""
        # The default __ne__ doesn't negate __eq__ until 3.0.
        return not (self == other)

Complex.register(complex)
```

抽象基底クラス Complex の定義からな特徴的なメソッドの抜粋をここに掲載していますが、このクラス Complex では複素数の四則演算を含む演算に関連するメソッドと同値性に関連するメソッド等がデコレータ付きで定義されています。ただ、それらの定義内容は本来メソッドで定義されるべき処理内容が長文書文字列で記載されているだけで、実際に行われる処理文は NotImplementedError エラーを送出する raise 文だけです。また、ここで定義されているメソッドは '@abstractmethod' や '@abstractproperty' といったデコレータを伴い、クラスの定義のあとでメソッド register() で Python の組込のクラス complex を引数として与えられた文が置かれています。この構成はモジュール numbers で定義されるクラスに共通する特徴で、ここで示すように抽象基底クラスは抽象メソッドと呼ばれる形式的なメソッドを定義し、これらのメソッドの具体的な処理は、この抽象基底クラスを継承する具象化クラスを使って記述されます。

このモジュール numbers で定義されているクラスは Complex, Real, Rational, Integral で、これらの構成方法は実利的な側面があります。ここで図式的に示すためにクラス A がクラス B のサブクラスであることを $A \rightarrow B$ と表記します。集合論の数の構成に従うのであれば $\mathbf{N} \rightarrow \mathbf{Z} \rightarrow \mathbf{Q} \rightarrow \mathbf{R} \rightarrow \mathbf{C}$ と「塔」の上側から順に数を定義し、演算も順次拡張することで数値塔の継承順と逆の構成順になります。しかし、現実にはこれらのオブジェクトとその中で閉じた演算がメソッドとして実装されています。たとえば、複素数では複素数同士の演算はもちろん、複素数と実数、複素数と有理数、そして複素数と整数の演算は、実数、有理数と整数が複素数のサブクラスであることから問題なく複素数の領域で行えますが、たとえば、二つの整数 2, 3 の積 2×3 の結果を $6 + 0i$ や $6.0e - 0$ と複素数や実数（浮動小数点数）と整数よりも上の階のインスタンスとして返却する必要はありません。つまり、上階のインスタンスとの演算では上階のメソッドをそのまま継承、同じ階のインスタンス同士の演算では同じ階で閉じるようメソッドを再定義、下の階のインスタンスとの演算では、結果がその階のインスタンスになるようにメソッドを再定義したものとなり、結果に対する制約がそのままメソッドへの制約になる形で再定義されることになります。つまり、この塔構成はメソッドへの制約の入り方と密接に関連しますが、数の成り立ちとは別のものです。

そして、Python ではこれらの数と演算は、組込のオブジェクトとして個別に用意されており、抽象基底クラスを導入することで、後付的に数オブジェクトとして継承関係も含めて再定義しています。もちろん、この構成は計算処理だけが目的であればあってもなくても良いものかもしれません。しかし、既存のオブジェクトに構造を導入することで抽象化した立場で考察が可能になること、さらには多項式等の数の拡張で、この抽象化が威力を

發揮します。

5.3 SageMath の数の構成

5.3.1 SageMath の数

最初に見たように SageMath では、整数、有理数、実数と複素数といった数オブジェクトを Python の数オブジェクトからそのまま継承していません。この理由としては

- 整数が 32bit 整数と長整数の異なるリテラルを持つオブジェクトに分かれている。
- 有理数が実装されていない。
- 浮動小数点数が倍精度に固定されていて任意精度ではない。
- 算術演算がそもそも効率の良いものではない。

といった点が挙げられます。最初の整数のリテラルの問題は Python 2 の問題で、Python 3 ではありませんが、このような現実的な問題が根底にあり、さらに SageMath が Python で数学を表現する性格上、Python の数が妥当なものであるとは言い難いものです。SageMath ではこれらの問題に対処するために、GMP(GNU MP, [GNU Multi Precision Arithmetic Library](#)) と GMPFR([GNU Multiple Precision Floating-point Reliability](#)) を使って整数、有理数、実数と複素数とそれらの基本演算を定義し直しています^{*5}。そのため SageMath で数式を構成する数が Python に既存の数ではなく SageMath 独自の数オブジェクトである理由です。特に整数と有理数に関しては GMP 以外に数論専用の数式処理 PARI が組み込まれており、この PARI が出力した数オブジェクトを SageMath の数として変換する機能も追加されています。なお、前述のように SageMath 上のプログラムの 32bit 整数や実数のリテラルは型変換を行わない限り、それらのリテラルに対応する Python の数の型になります。このことは最初に挙げた例のように for 文や while 文でカウンターとして使っている整数が Python の数オブジェクトで、SageMath の数オブジェクトではないことを意味します。このことから、扱う数が SageMath の数オブジェクトかそうでないかを明瞭に意識しておく必要があります。

5.3.2 GMP の概要

GMP は C で記述された整数、有理数と浮動小数点数の任意精度で演算を行うための GPL のライブラリです。なお、ここでの任意精度とは計算機のメモリ等のハードウェアに由来する最大桁数を超えない任意の桁数で数の処理が行えることを指します。もちろん、大きな桁数の数を扱うことはそれに応じてメモリの消費や処理時間の増大を招きますが、

^{*5} GMP を利用している主要なアプリケーションに商用の数式処理 *Mathematica* があります。

計算精度が向上することで収束計算の反復回数が減るために却って全体の処理時間の増大が抑制されたり、精度が足りずに解析できなかつた事象の解析が可能になることもあります。この GMP は単に任意精度の計算が可能なだけではなく、通常の C が提供する以上の精度で可能な限り最速の数値計算を実現することを目的にしています。そのために GMP の計算最適化には Intel や AMD 等の主要な CPU 向けに最適化されたアセンブリコードが含まれています。ただし、数値行列を扱い、そこそこの精度で十分で、むしろ、処理速度が要求されるのであれば任意精度ではなく倍精度の浮動小数点数による数値計算を検討すべきです。その場合は Python のパッケージの NumPy を直接利用したり、SageMath で倍精度浮動小数点数に限定して数値計算を行うことになります。

ここで GMP の定める数を以下に示します：

SageMath の数の構成に必要な GMP のクラス

<code>mpz_t</code>	符号付整数とその演算、および名前が <code>mpz_</code> で始まる約 150 の函数
<code>mpq_t</code>	<code>mpz_t</code> 型の整数から構成される有理数とその演算、および名前が <code>mpq_</code> で始まる約 35 の函数
<code>mpf_t</code>	浮動小数点数とその演算、および名前が <code>mpf_</code> で始まる約 70 の函数

ここで GMP の有理数は `mpz_t` 型の整数対として表現されるために `mpz_t` 型の整数演算の函数も必要になります。ところで、SageMath は多倍長精度の浮動小数点数の表現のために GMP ではなく GMPFR を用います。この GMPFR は GMP 上に構築されたライブラリで、より多くの函数、符号付き零、無限大 inf と NaN を提供しています。この GMPFR の提供する浮動小数点数は、IEEE 754 による浮動小数点数が固定長の 2 進数として表現されているのに対して GMPFR のそれは仮数部が分離したデータとして表現されます；

GMP の浮動小数点数の表現

<code>_mpfr_prec</code>	仮数部の bit 長
<code>_mpfr_sign</code>	符号と仮数部における非零要素数
<code>_mpfr_exp</code>	指数部
<code>_mpfr_d</code>	仮数部へのポインタ

‘`type(1.0)`’ で調べたときに ‘`sage.rings.real_mpfr.RealLiteral`’ と型が返却されていましたが、ここでの ‘`mpfr`’ で分かるように GMPFR が定める浮動小数点数が SageMath の実数の正体です。なお、浮動小数点数は本質的に近似の数で、GMP や GMPFR では浮動小数点数への丸めを行うための函数が用意されています。そのために自然数 \mathbb{N} 、整数 \mathbb{Z} や有理数 \mathbb{Q} 、さらには整数係数の多項式の解になる数の集合である代数的数と別の扱いになり

ます。

ここで Python に標準で実装された長整数による処理と GMP を使った処理ではどの程度違うものでしょうか? Python での長整数の演算処理は、たとえば乗算でカラツバ法 (Karatsuba algorithm) が用いられていますが、各種の計算機環境に最適化されたものではありません。次の例^{*6}で SageMath の整数が Python 標準の長整数よりも高速に処理されることを確認しましょう。そのために次の函数をあらかじめ定義しておきます;

```
def factorial(n, stop=0):
    o = 1
    while n > stop:
        o *= n
        n -= 1
    return o

def choose(n, k):
    return factorial(n, stop=k) / factorial(n - k)
```

ここで定義した函数 factorial() と choose() は階乗と組合せの値を計算する整数値函数で、演算も積と差が中心で、choose() で商がある程度の、ある意味平凡な函数です。これらの函数定義を Python と SageMath にそのまま評価させます。SageMath に関してはその引数の整数が自動的に整数クラスの構築子 Integer() で初期化されるために、内部の計算が Python の整数型ではなく、SageMath の整数で処理するために効果が判り易くなっています。そして、Python と SageMath での処理速度の計測は以下で行います;

```
import time
start = time.clock()
a = choose(50000, 50)
elapsed = (time.clock() - start)
print elapsed
```

この函数 time.clock() はプロセッサ処理時間を秒数で返す函数で、モジュール time に含まれています。そのためにあらかじめ ‘import time’ でモジュール time を読み込んでおく必要があります。それから処理時間は変数 elapsed に束縛されるために、この変数 elapse の値で小さな方がより高速な処理が行われていると判断できます。さて、この式の比較は Jupyter を使っていれば kernel を SageMath や Python 2 に変更して函数定義と計測を行えば容易に行えます。この式の評価を CoCalc で確認すると、Python 2 で 1.943、SageMath で 0.594 という結果を得て圧倒的に SageMath の整数処理が 4 倍近く高速で

^{*6} <http://jasonstitt.com/c-extension-n-choose-k> の記事: 「GMP bignums vs. Python bignums: performance and code examples」の例です

あることが判ります。つまり、この程度の計算処理でも GMP と MPFR で数を定義した SageMath が圧倒的に高速であることから、Python による数値計算で処理速度を要求するのであれば、数値行列演算ライブラリのことだけでなく、GMP の利用も考慮すべきです。また、整数演算で倍精度の浮動小数点数の範囲内で済む演算であれば最適化された BLAS+Python で処理するのも一手でしょう。ただし、数式の検証やアルゴリズムの検討といった段階で黙って SageMath を使うことが最も安全で効率が高い方法です。

SageMath はこの GMP のような C のライブラリを取り込むために Cython を用います。この Cython は C と Python を継ぎ目なしに融合することができる Python で記述された処理系で、この Cython を用いることで C のライブラリが利用できるだけではなく、全体の処理の高速化を図ることができます。

5.3.3 Cython の概要

Cython は修飾子が pyx と pxd の二種類のファイルを必要とし、pyx ファイルが函数等の定義を行う本体で、C の函数も Python の函数等の定義を継ぎ目なしに Python 風に記述することができます。もう一方の pxd ファイル C の定義ファイルに相当し、外部公開の C の宣言の共有、C コンパイラにインライン化させたい函数の定義、それから pyx ファイルが存在するときに Cython モジュールに Cython 用のインターフェイスを構築するため用いられます。これらのファイルを基に Cython コンパイラが C のコードに変換^{*7}し、それをコンパイルして Python の拡張モジュールが得られます。その結果、C で記述したものと大差ない程度に高速化できます。

Cython の構文は Python 言語の一部に C の函数呼出と C の变数やクラスの型宣言が追加され、Python の文に交じって、C のライブラリに関連する变数や函数の宣言、ライブラリの読み込みに関する文が混在しています。そして、これらの文で Python の命令文の先頭に文字 “c” が付いた命令文 (cdef, cimport 等) が C のライブラリ操作に関連する文になります。たとえば、C の变数と型定義では cdef 文が用いられます。cdef 文は C の宣言文の頭に “cdef” をそのまま載せた構文で、int 型の变数 i, j, k と float 型の变数 f, 配列 g, ポインタ g を宣言するときは

```
cdef int i, j, k
cdef f, g[42], *h
```

と記述します。また、cdef をブロックとして記述するときは

```
cdef:
```

^{*7} ソースの構文解析は Pyrex で行います。

```
int i, j, k
float f, g[42], *h
```

とグループ化して記述することもできます。また, `cdef` は C の函数呼出でも同様の書式ですが、函数が output する型や引数の型を C 風に記述しなければなりません。と、このように幾つか注意すべき点がありますが、Python のプログラムと混在して記述できます。なお、ここでは SageMath のソースファイルでどのようなことを行っているか分かる程度の解説しか行いません。

5.3.4 整数の実装

SageMath の整数の実装 GMP の `mpz_t` 型のオブジェクトを Cython で SageMath にクラス `Integer` として組込んでいます。クラス `Integer` は GMP の整数型クラス `mpz_t` のラッパー、すなわち、具象化クラスとして定義されています。この整数型の機能の一つに Python の `int` 型や `long` 型、numpy の整数オブジェクト、それと整数論向けの数式処理 PARI の整数を SageMath の整数 `Integer` 型に変換する機能もあります。整数の定義は `src/sage/rings` にある `integer.pxd` と `integer.pyx` で行われていますが、`integer.pxd` が函数やクラスの形式的な定義であるのに対し、`integer.pyx` がクラス `Integer` の実質的な定義になります。そして、クラス `Integer` は `sage.structure.element.EuclideanDomainElement` のサブクラスとして定義されています。

5.3.5 有理数の実装

有理数は GMP の `mpq_t` 型を Cython でクラス `Rational` として取り込んで実装しており、クラス `Rational` は PARI や numpy からの数オブジェクトの変換機能を持っています。実質的な定義は `src/sage/ring/rational.pyx` にクラス `Rational` が `sage.structure.element.FieldElement` のサブクラスとして定義されています。

5.3.6 実数の実装

実数は有限な資源しかない計算機では近似される数です。この近似では浮動小数点数と呼ばれる 2 進数に符号化された数が用いられます。通常の計算機言語では单精度と倍精度の 2 種類の浮動小数点数が用意されていますが、Python 本体の浮動小数点数は倍精度のみ、NumPy で单精度、倍精度と四倍精度の浮動小数点数が利用可能となり、SageMath では NumPy に加えて GMPFR を利用することで倍精度以上の精度を持つ任意精度の浮動小数点数で実数を近似できます。倍精度以上の高精度の演算を行えば、誤差に左右され易い対象の解析に有利な一方で、倍精度浮動小数点数であれば CPU で直接処理が可能なために計算速度では圧倒的に有利です。そのために任意精度で浮動小数点数を扱う際

に、対象の特性を考慮した上で判断する必要があります。SageMath の倍精度浮動小数点数クラスは RealDoubleField_class, 略して RDF で、任意精度の浮動小数点数クラスが RealField_class, 略して RealField です。これらは共にクラス Field のサブクラスです。また、線形代数、特にベクトルのクラスである Vector は倍精度浮動小数点数から構成されます。これは SageMath は数値行列の処理のために倍精度浮動小数点数向けの BLAS が用意されているためです。このように SageMath ではどのような数を相手にすべきかで、相手にすべき数のクラスが異なることに注意が必要です。とはいっても、深く考えなくても十分に良好な結果が得られるように工夫はされています。

5.3.7 複素数の実装

複素数 \mathbb{C} は実部と虚部の実数対 $\mathbb{R} \times \mathbb{R}$ として表現できます。また、実数は浮動小数点数で近似されるために複素数の実装も実数と同様に倍精度と任意精度の浮動小数点数の二種類の対象で近似されます。ここで倍精度浮動小数点数で近似した複素数が ComplexDoubleField_class, 任意精度の浮動小数点数で近似した複素数がクラス ComplexNumber になります。

5.4 SageMath のオブジェクト

5.4.1 SageMath のオブジェクトについて

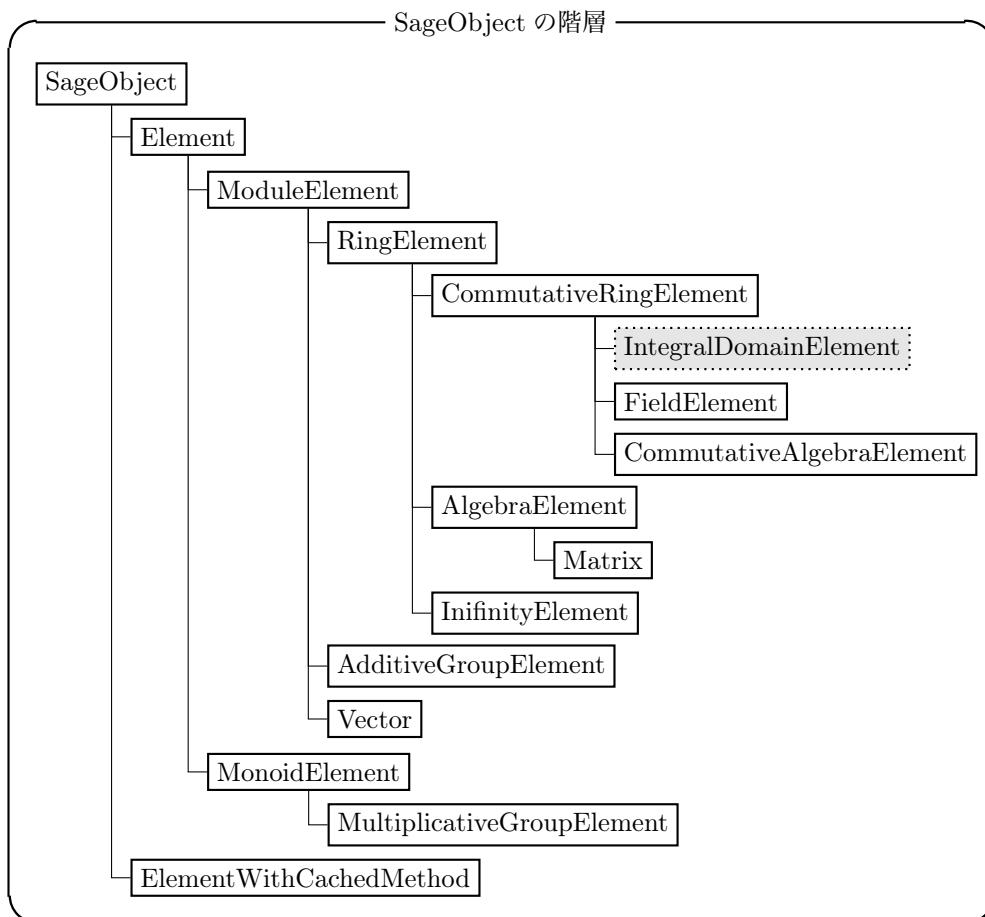
SageMath のオブジェクトは数だけではありません。自然数、整数、有理数、実数と複素数に加えて、それらを係数とする多項式環…と多岐にわたります。これらの数学的対象をどのように纏め、階層化しているかをこの節で述べることにします。

5.4.2 SageMath の抽象基底クラス

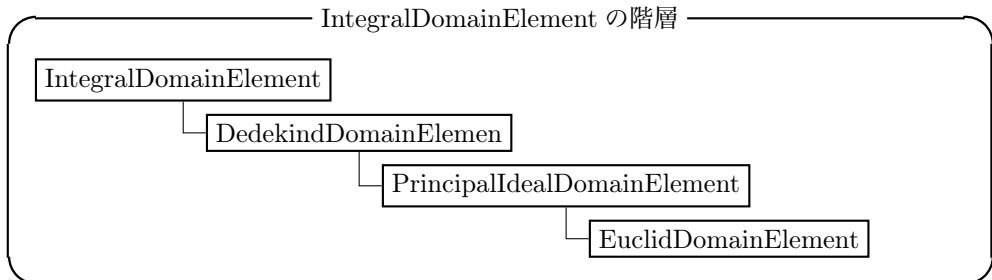
SageMath で利用者が認識できるオブジェクトで最上位のオブジェクトがクラス SageObject です。SageMath のオブジェクトで利用者が直接扱ったり返却されるオブジェクトが含まれるクラスはクラス SageObject と継承関係になければなりません。このクラス SageObject は注釈や文書で Python の抽象基底クラスと同様に “abstract base class” と呼ばれ、実際に Python の数オブジェクトで見られるような後付の階層構造が導入されていますが、抽象メソッドのデコレータが @abstract_method と Python の抽象メソッドのデコレータが @abstractmethod と双方で異なり、@abstract_method が SageMath で新たに定義されたデコレータであることから、SageMath の “abstract base class” は Python のそれと異なる SageMath 独自のものであることが判ります。しかし、SageMath 上の整数、有理数などの数オブジェクトが GMP 由来で、該当するオブジェク

トの具象化クラスとして現れており、Python の抽象基底クラスと類似の働きをしています。

この SageObject の継承関係図は src/sage/structure ある element.pyx から抜粋したもので最初に SageObject から六層までの階層を示します。それから六層目の灰色で塗り潰している IntegralDomainElement にはさらに三階層ありますが、継承関係による階層構造が分かり難くならないように二つに分けて示しています；



次にクラス IntegralDomainElement の階層図を示します；



なお, SageObject がこれらの抽象基底クラスの何れかに分類されるという意味ではありません. あくまでも抽象基底クラスの継承関係を図示したものです. そしてクラス Element の継承関係は数学的な構造が反映されたものになっていますが, 数学的な厳密さ以上に実装が容易になるように継承関係が導入されています.

5.5 SageObject の各階層

5.5.1 はじめに

ここでは SageObject の抽象基底クラスを階層ごとに解説します. この抽象基底クラスの最上位のクラス SageObject の階層を 0 とします. ここで解説するクラスは全て SageMath の抽象基底クラスで, 利用者がこれらのクラスの具象化クラスを構築するときは継承によって定義することができます. また, 通常の Python のクラスで演算や比較を実行するためには、SageObject の抽象基底クラスに用意された特殊メソッドを用います. ここではその用意されたメソッドについても軽く触れます. ただし、上書きのために用意されたメソッドは、特殊メソッドと比べて先頭の文字 “_” が一つ少ないメソッド名です. このメソッドの書換によって演算を新たに定義しなおして、演算の持つ分配律や結合律といった性質をそのまま引き継ぐことが可能になります. ちなみに 0 層から下降していく様は何かの地獄旅行のようですが、ダンテの地獄旅行とは異なり、下降するにしたがって逆に多様性を増し、花園を愛するような気分になることでしょう.

5.5.2 第 0 層

■SageObject: SageMath で利用者が直接扱い、見ることのできる継承関係で最上位のオブジェクトのクラスです. このクラスのメソッドに計算結果をアスキーアートとして出力するメソッド、結果をファイルに保存するメソッド、SQLite 等の DB を操作するメソッドや SageMath が利用する数学アプリケーションとのインターフェイスが含まれます. その意味ではありません数学的ではないソフトウェアに関連する階層です.

第1層

■Element: SageObject を親とする抽象基底クラスで, Element を継承するクラスは加群 (Module) と単系 (モノイド, Monoid) に対応する抽象基底クラスです.

■ElementByCachedMethod: キャッシュを有するメソッドの抽象基底クラスです. 計算結果を一時的に蓄えるメソッドといったものが対応します.

5.5.3 第2層

■ModuleElement: Element を継承する (R-) 加群 (Module) の抽象基底クラスです. R-加群の演算を表現するメソッドとしては, 係数環 R と可換群 A の和 “+” については `_add_()`, 左作用のときは `_lmul_()`, 右作用のときは `_rmul_()`, 両側であれば `_mul_()` があります. また, $a \in A$ の逆元 $-a$ をメソッド `_neg_()` で定めます. 利用者が (R-) 加群を定すときは必要に応じてこれらのメソッドの再定義を行います.

■MonoidElement: Element を継承する単系 (モノイド, Monoid) の抽象基底クラスです. ここで単系は可換であるとは限らない二項演算 “*” とその演算の単位元を持つ対象です.

このクラスの具象化クラスは積 “*” のメソッド `_mul_()` の書換えを必要とします.

5.5.4 第3層

■RingElement: ModuleElement を継承する環 (Ring) の抽象基底クラスです. 環は加法 “+” と乗法 “*” の二つの二項演算子を持ち, 加法に関しては群になり, 乗法に関しては閉じている数学的対象です.

■AdditiveGroupElement: ModuleElement を継承する加法群 (Additive group) の抽象基底クラスです. 可換な二項演算である和 “+” のみを演算として持つ群です.

■Vector: ModuleElement を継承するベクトル (Vector) の抽象基底クラスです. 可換な二項演算である和の他に実数あるいは複素数との積 (スカラー積) を有します. なお, SageMath のベクトルは近似の数である浮動小数点数で表現され, 代数的な数ではありません. そのために代数から独立して置かれています.

■MultiplicativeGroupElement: MonoidElement を継承する乗法群 (Multiplicative Group) の抽象基底クラスです. 可換であるとは限らない二項演算子である積 “*” のみを持つ群を表現します. 加法群と分けたのも二項演算の実装に由来し, Python では和に対応するメソッドが `__add__()`, 積 “*” に対応するメソッドが `__mul__()` に対応するた

めです。

5.5.5 第4層

- **CommutativeRingElement**: `RingElement` を継承する可換環 (Commutative Ring) の抽象基底クラスです。可換環では乗法 “ $*$ ” の可換性が通常の環に加わった数学的対象です。
- **AlgebraElement**: `RingElement` を継承する代数 (Algebra) を表現する抽象基底クラスです。代数はベクトルを一般化した代数的構造です。この抽象基底クラスは数学的構造が加群に由来することではなく、むしろ、和と積の二項演算を持つというメソッドの実装上の類似によるものです。
- **InfinityElement**: `RingElement` を継承し、無限遠 ∞ に対応する抽象基底クラスです。

5.5.6 第5層

この第5層には `CommutativeRingElement` を継承する抽象基底クラスのみです。ここに含まれるオブジェクトの積演算は全て可換です。

- **DedekindDomainElement**: `CommutativeRingElement` を継承する抽象基底クラスで、デデキント整域を表現します。ここでデデキント整域とは 0 生成されたイデアル (0) と異なるイデアルが有限個の素イデアルの積として表わせる整域です。
- **IntegralDomainElement**: 整域 (Integral Domain) に対応する抽象基底クラスです。ここで整域は環 R で $ab = 0$ となる $a, b \in R$ が存在するときに $a = 0$ か $b = 0$ の何れかが必ず成立するときで、環 R が零因子を持たないときと言い換えられます。
- **FieldElement**: 体 (Field) に対応する抽象基底クラスです。 R が体であれば、 R はまず和演算 “ $+$ ” と積演算 “ $*$ ” を持つ可換環で、さらに $(R - \{0\}, *)$ が可換群になる環です。たとえば、整数 \mathbb{Z} は和と積の二つの演算を持ちますが、積に関しては可換で単位元 1 を持っていても、 $\mathbb{Z} - \{0\}$ の任意の元が逆元を持たないために体にはなりません。有理数 \mathbb{Q} は $(\mathbb{Q}, +)$ が可換群、そして、 $(\mathbb{Q} - \{0\}, *)$ も可換群であることから体になります。
- **CommutativeAlgebraElement**: 可換代数 (Commutative Algebra) に対応する抽象基底クラスです。

第5層以下

整域を継承し、デデキント整域 (Dedekind Domain) を表現する抽象基底クラスです。

■PrincipalIdealDomainElement: デデキント整域 (DedekindDomainElement) を継承し, 単項イデアル整域 (PID, Principal Ideal Domain) を表現する抽象基底クラスです. ここで PID は任意のイデアルが単項生成になる整域です.

■EuclidDomainElement: PrincipalIdealDomainElement を継承し, ユークリッド整域 (Euclid Domain) を表現する抽象基底クラスです. 整域 R がユークリッド整域と呼ばれるためには, ユークリッド函数と呼ばれる写像 $f : R - \{0\} \rightarrow N$ が存在し, 任意の $a, b \in R$ に対して $f(a) < f(b)$ であるときに $r = 0$ かつ $f(r) < f(a)$ かつ $b = aq + r$ を充たす $q, r \in R$ が存在しなければなりません.

第 6 章

結び目理論への適用

6.1 概要

この章では SageMath を使って 3 次元空間内の結び目や絡み目に関連した計算に注目したいと思います。ここで SageMath には群論専用の数式処理システム GAP に由来する有限群のライブラリがあり、その中に自由群や組紐群が含まれています。そのために結び目/絡み目を組紐で表現して組紐群として処理できますが、ここではガウス・コードと呼ばれるリストで結び目/絡み目を表現します。それから結び目/絡み目に連結和と呼ばれる操作からモノイドとしての代数的構造を導入し、スケイン多項式と呼ばれる結び目/絡み目の不変量を計算する函数も構築します。ところで、このスケイン多項式の計算過程で膨大なデータが発生します。安易な方法はこれらの中間データをリストでメモリ上に蓄え込むことですが、交差点の増加と指数函数的にデータが増加し、そうなると状況の把握も困難になって函数自体が事実上のブラックボックスになり兼ねません。それを避けるためには、これらの中間データを RDB で保管することです。ここでは RBD として SQLite3 を用い、中間データの可視化も行えるようにすることを目標とします。

6.2 結び目/絡み目とは

最初に結び目と絡み目が何であるか明瞭にしておきましょう。まず、絡み目は 1 個以上の互いに交わらない結び目で構成された図形です。したがって、絡み目が何であるかを説明するためには結び目が何であるかを最初に説明しておく必要があります。さて、現実の結び目には「蝶々結び」等の色々な紐の結び方がありますが、これらの結び目を構成する紐が互いに交わりません。そして、どんなにもつれた状況でも適当な大きさの箱に納められるような大きさの 3 次元空間内の図形です。このことから結び目は適当な大きさのボール、つまり、3 次元球 B^3 の中にすっぽりと入っていると考えられます。さらに結び目は紐の両端を繋いだ円、すなわち 1 次元球面 S^1 として考えます。これは実用上の問題で、もし

も、結び目の両端が切れたままだと一方の端点から別のもう一方の端点に向けて紐を縮め、それからまっすぐに延ばせば結び目を解消、すなわち、紐を真っ直ぐな棒状にすることができますが、紐の両端を繋いで輪にしてしまえば解けるものと解けないものが出てくるでしょう。だから端点を繋いで1次元球面 S^1 で考察します。

ここで結び目が3次元球 B^3 にすっぽり入っていると言いましたが、では実際にどのように3次元球面 B^3 の中にあるのでしょうか？これは自分自身が交わることのないようにきれいな形で入っているべきです。実際、自分自身が交わった結果、複数個の穴の開いたドーナツみたいな結び目を解けるかどうか考える人はいないでしょう。それにこの場合は「ブーケ (bouquet)」と呼ばれる图形で、その輪の数で一意に分類できてしまいます。このように自分自身が交差することがない状態で1次元球面 S^1 が3次元空間に包含された状態を3次元空間への1次元球面の「埋め込み」と呼びます。以上から、我々が扱う「結び目 (Knot)」は「一つの1次元球面 S^1 の3次元球への埋め込み」です。これで結び目がどのようなものであるかが明瞭になりました。ところで、絡み目は複数の互いに交わらない結び目で構成された空間图形です。したがって、「絡み目 (Link)」は「互いに交差しない複数の1次元球 $\bigcup_{i=0}^n S_i^1$ の3次元球への埋め込み」です。また、結び目や絡み目には「向き (orientation)」を入れることができます。これは絡み目を構成する各成分 (=結び目) が円であり、それらに反時計回りや時計回りの二種類の向きが入れられるためです。各成分に向きが入った絡み目/結び目を「向き付けられた絡み目/結び目」と呼びます。しかし、この結び目/絡み目の定義は実用的であっても美的ではありません。このことは「結び目/絡み目の補空間」と呼ばれる空間を考えると明瞭になります。この補空間は結び目/絡み目が埋め込まれた空間から除去すべき結び目/絡み目を自分自身で交わらない程度に太らせた「管状近傍 (tubular neighborhood)」と呼ばれる空間 $N(K)$ を抜き出した空間で、絡み目に沿って虫が果実を食べてしまった状態に似ています。この補空間の境界には結び目/絡み目の環状近傍の境界であるチューブ状の境界と3次元球 B^3 の表面である2次元球面 S^2 の2つの曲面が現れます、環状近傍の境界に意味があっても2次元球面は無駄です。そこで話を簡単にするために結び目/絡み目が入っている球面に別の3次元球を貼って境界の一つである2次元球面を除去してしまいます。この2つの3次元球の張り合わせの結果、3次元球面 S^3 が得られます。このことをもう少し詳しく説明すると、4次元空間内の一定の距離の点の集合 $\{(x, y, z, w) | x^2 + y^2 + z^2 + w^2 = 1\}$ で3次元球面 S^3 が表現できます。この3次元球面 S^3 は $B_+^3 = \{(x, y, z, w) | x^2 + y^2 + z^2 \leq 1, w \geq 0\}$ と $B_-^3 = \{(x, y, z, w) | x^2 + y^2 + z^2 \leq 1, w \leq 0\}$ の二つの3次元球に均等に分けられます。この様子は2次元球面をZ軸正方向の半球とZ軸負方向の半球に分割する方法と同じものです。このときに B_+^3 と B_-^3 の境界が半径1の2次元球面 $\{(x, y, z, 0) | x^2 + y^2 + z^2 = 1\}$ であることが判るでしょう。ここで絡み目が3次元球 B_+^3 に入っていると考えれば良い

訳です。また、ここでの3次元球の貼り合わせには別の考え方があります。つまり、境界の球面を1点に潰すという考え方です。そうすると3次元球面 S^3 は通常の3次元空間 \mathbb{R}^3 に無限遠点 ∞ を追加した空間として考えられ、この貼り合わせを「**1点コンパクト化**」とも呼びます。このように結び目/絡み目は3次元球面 S^3 への円周の埋め込みとして考えます。

6.3 正則射影図

結び目/絡み目は3次元球面 S^3 に埋め込まれた円周ですが、3次元の対象として考察することは流石に難しいために平面に射影して考察します。これが結び目/絡み目の「**射影図**」と呼ばれる射影図です。この考え方をSageMathを使って説明しておきましょう。ここで示す結び目は「**三葉結び目 (trefoil)**」と呼ばれる結び目です。この結び目はトーラス(=ドーナツの表面)上の曲線として描くことができます。SageMathのスクリプトを以下に示します：

```
var('t,u,v')
a, b = 3, 1
x = (a + b*cos(u))*cos(v)
y = (a + b*cos(u))*sin(v)
z = b*sin(u)
T = parametric_plot3d([x,y,z],(u,0,2*pi),(v,0,2*pi),
    opacity=0.3, aspect_ratio=1)
c = (3*t,2*t)
s =[_.subs(dict(zip((u, v), c))) for _ in (x, y, z) ]
K = parametric_plot(s, (t,0,2*pi), color='red',
    thickness=20,plot_points=200 )
sb =[_.subs(dict(zip((u, v), c))) for _ in (x, y, -7) ]
Kb = parametric_plot(sb,(t,0,2*pi), color='blue',
    thickness=20,plot_points=200)
(K + Kb + T).show()
```

このSageMathのスクリプトでは助変数として t, u, v を持つ式を用いるためにあらかじめ $\text{var}('t, u, v')$ でこれらの変数の宣言を行います。それから変数 x, y, z への代入式の右辺が結び目が巻きつくトーラスの点の座標式で、変数 a がこのトーラスのXY平面上の半径、変数 b がトーラスのZX平面での断面の半径で、これらの値を基に函数 $\text{parametric_plot3d}()$ で描きます。なお、 $\text{opacity}=0.3$ とすることで曲面が半透明になるように設定しています。SageMathの3次元オブジェクト表示では視点の変更やオブジェ

クトからの距離をマウスを使って自由に動かせます。ここでトーラスには穴の周りを一周する「緯線 (longitude)」とそれに直交しトーラスの腕を一周する「経線 (meridian)」の二つの座標があります。ちなみにトーラスは二つの円周の直積: $S^1 \times S^1$ と同相で、先程の変数 a が緯線 (longitude) 側の円の半径、変数 b が経線 (メリディアン) 側の円の半径に対応します。そしてトーラス上の結び目は互に素な整数対 (p, q) で、経線を p 回ほど回る間に緯線を q 回ほどまわることで表現されます。この整数対 (p, q) で表記可能な結び目を「 (p, q) 型のトーラス結び目」と呼びます。ここで描画しようとしている三葉結び目は「(2, 3) 型のトーラス結び目」で、変数 c に設定したタプル $(3*t, 2*t)$ がこの状況に対応します。それから結び目の3次元空間内部のX, Y, Z 座標はトーラスの座標に緯線と経線の角度情報を入れます。この設定は変数 c に設定した角度情報の割当てを行っています。そして、変数 s に割り当てた結び目の座標情報から函数 `parametric_plot()` で結び目を描画し、同様に $Z = -7$ にある XY 平面への結び目の射影も描画します。これらのグラフィックス・オブジェクトを重ね合わせたオブジェクトを生成し、メソッド `show()` で表示したものが図 6.1 に示すグラフです:

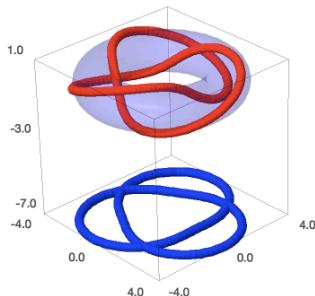


図 6.1 3 次元空間内の三葉結び目

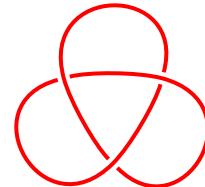


図 6.2 結び目の射影図

ところで図 6.1 のように絡み目や結び目を3次元空間内の対象としてそのまま考察することは困難です。そこで図の下側に示す平面への射影図にすると、平面上の図形として扱えるためにより簡便になりますが、この射影図にどの紐が上か下かという交差点の情報が欠落しています。そこで地図で道路の立体交差を表現する要領で上を通る紐で下を通る紐が寸断されたように描きます。また、交差点で二つの紐だけが交差して紐の向こう側にもう一方の紐が渡りきる性質、つまり、「横断的」と呼ばれる性質が射影図のすべての交差点にあるとし、逆にあってはならない状況に、2本以上の紐が長々と重なること、3本以上の紐が1点で交差することと紐がどちらかの接線になることとしますが、これらの状況は紐を局所的に動かせば容易に解消できます。そして、横断的な二重点のみで構成される射影図を「正則射影図」と呼びます。一例として図 6.2 に三葉結び目の正則射影図を示しておき

ます。それから正則射影図の交差点で上側の紐を「**上道**」、下側の紐を「**下道**」と呼びます。

正則射影図が出たところで結び目/絡み目に現実的な制約をもう一つ入れましょう。すなわち、ここで扱う結び目/絡み目はその正則な射影図が有限個の折線で近似できるものに限定します。この有限個の折線で近似される性質を「**順 (tame)**」と呼びます。逆に無限個の折線がどうしても結び目/絡み目の近似で必要なときに「**野性的 (wild)**」と呼びます。順な結び目/絡み目は交差点は有限個、野性的なものは(可算)無限個の交差点を持ちます。

6.4 結び目/絡み目の同値性

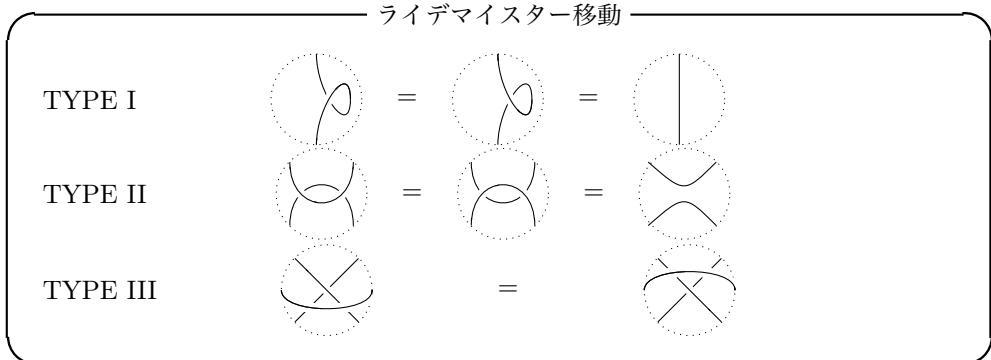
では、与えられた二つの結び目/絡み目が「**同じである**」と言えるのはどのような状態でしょうか？三角形であればそれが重なり合うことを示せれば十分ですが、結び目/絡み目はそんなに簡単ではありません。なお、結び目は1成分の絡み目であるために絡み目で話を進めることにします。まず第一に、一方の絡み目の成分の総数が n であればもう一方の成分の総数も n でなければなりません。それから絡み目をゴム紐でできたものと考えて3次元球面 S^3 内部で紐を切ったり、紐を交差させずに変形することで互いの絡み目に移り合えるときに同じ絡み目と呼ぶことにしましょう。このことは二つの同じ成分数 n の絡み目 L_1 と L_2 の間に「**アンビエント・イソトピー (ambient isotopy)**」と呼ばれる連続写像 $F : S^3 \times [0, 1] \rightarrow S^3 \times [0, 1]$ が存在することに対応します：

——アンビエント・イソトピーの性質——

- $t \in [0, 1]$ に対して $F_t(x) (\stackrel{\text{Def}}{=} F(x, t))$ は3次元球面 S^3 の同相写像である
- $F_0 : S^3 \rightarrow S^3$ は恒等写像である
- $t \in [0, 1]$ に対して $F_t(L_1)$ は共通な部分集合を持たない n 個の1次元球面 S^1 の和集合と同相である
- $F_0(L_1) = L_1$ かつ $F_1(L_1) = L_2$

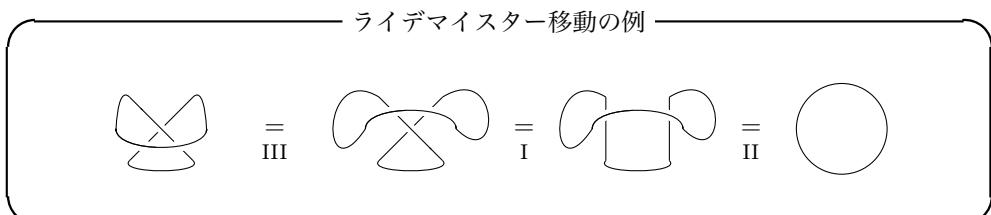
二つの同じ成分数 n の絡み目 L_1 と L_2 の間にアンビエント・イソトピーが存在するときに、これらの絡み目が「**同値**」と呼び $L_1 \Leftrightarrow L_2$ と表記します。このアンビエント・イソトピーが存在するということは閉区間 $[0, 1]$ を時間、時間 0 を現在、時間 1 を未来とするときに、現在 $t = 0$ で絡み目 L_1 、その絡み目 L_1 を3次元球面 S^3 内部で連続的に変形して未来 $t = 1$ で絡み目 L_2 にする映画が存在することと言い換えられます。言ったものの、このアンビエント・イソトピーは式をいじくり倒すことで簡単に構築できるような代物ではありません。そもそも、3次元空間での変形を延々と考えなければなりません。ここを1次元でも減らして2次元空間で、それも局所的な変形で考えたいものです。

そこで正則射影図の局所的な操作を有限回繰り返すことで互いに移りあえるという操作はどうでしょうか? このような都合の良い正則射影図の変形操作に「ライデマイスター移動 (Reidemeister Move)」と呼ばれる正則射影図に対する操作があります:



ライデマイスター移動はこれら 3 種類の正則射影図の局所的な変形操作の組み合わせから構成されます。最初の TYPE I は紐に捩れがあったときに、それをまっすぐにしても同じ結び目/絡み目になるという操作、TYPE II は絡んでいない紐の交差点を解消する操作です。最後の TYPE III は交差点付近で絡まっていない紐を並行移動させる操作です。これら 3 種類の基本操作の組み合わせで互いに正則射影図に移りあえるときに正則射影図の元になった結び目/絡み目の間にアンビエント・アイソトピーが存在することが知られています。このライデマイスター移動を使って結び目/絡み目の同値性が判断できます。

ここで、3 次元球面 S^3 内部の結び目 K の XY 平面への射影図が $\{(x, y) \in S^3 | x^2 + y^2 = 1\}$ で与えられる結び目の正則射影図にライデマイスター移動で変形できるときに結び目 K を「**自明な結び目 (trivial knot)**」と呼びましょう。次にライデマイスター移動で自明な結び目になる例を挙げておきましょう:



この例では最初に TYPE III で横紐を上に動かし、下側の捩れを TYPE I で解消し、それから最後は TYPE II で上側の横紐を下に動かして自明な結び目が得られます。なお、最後の TYPE II は TYPE I を左右の捩れに施す操作に置き換えられます。

このように二つの結び目/絡み目が与えられたときに、その同値性を確認するために知恵の輪よろしく射影図間のライデマイスター移動を試行錯誤して、見つかれば同値である

と結論付けられることが判りました。では、二つの結び目/絡み目が同値でないことをどう示せば良いでしょうか？「**同値であればライデマイスター移動が必ず存在する**」の対偶の「**ライデマイスター移動が存在しなければ同値でない**」からライデマイスター移動が二つの同じ成分数の絡み目に存在しないことを示すためにどうすれば良いのでしょうか？試行錯誤の結果、「**ライデマイスター移動が見つからない**」ことと「**ライデマイスター移動が存在しない**」ことは本質的に違います。そこで結び目/絡み目の固有の値を求め、それらの値の比較に落とし込むことができないものでしょうか？当然、その固有の値はライデマイスター移動で不变な値であり、可能なら機械的な操作で求まるものであるべきです。この特徴付けを行うために「**群**」という概念を結び目/絡み目に導入します。

6.5 群について

「**群**」は天下り的に「**性質の良い二項演算を持った集合**」です。実際、群 $(A, *)$ の A は集合で、この集合 A には「**演算**」と呼ばれる集合 A の二つの元 a_1 と a_2 から新しい元 “ $a_1 * a_2$ ” を生成する能力があります。この演算の演算記号 “ $*$ ” を「**演算子**」と呼びます。そして、群を集合と演算の対 ‘ $(A, *)$ ’ で表記し、演算を省略しても問題がないときは群 A と簡単に記述します。同様に ‘ $a * b$ ’ という式の表記で演算子を省略して単に ‘ $a b$ ’ と表記することができます。

以下に群の条件を纏めておきましょう：

群の条件

- 演算 $*$ に対して閉じている:

$$a, b \in A \rightarrow a * b \in A$$

- 結合律が成立:

$$(a * b) * c = a * (b * c) \text{ を充す.}$$

- 単位元 1 の存在:

$$a * 1 = 1 * a = a \text{ となる } 1 \in A \text{ が存在する.}$$

- 逆元の存在:

$$\text{任意の } a \in A \text{ に対し, } a * b = b * a = 1 \text{ を充す } b \in A \text{ が存在する.}$$

ここで最初の二つの条件が半群になるための条件です。さらに単位元を有する半群を「**单系 (モノイド, monoid)**」と呼びます。また、演算 “ $*$ ” が常に ‘ $a * b = b * a$ ’ を充すときに演算 “ $*$ ” を「**可換 (commutative)**」と呼び、可換な演算子を持つ群 $(A, *)$ を「**可換群 (commutative group)**」と呼びます。また、演算が可換でなければ「**非可換 (non-commutative)**」、非可換な演算を持つ群を「**非可換群 (non-commutative group)**」と呼びます。ここで可換群の例としては整数 $(\mathbb{Z}, +)$ や有理数 (\mathbb{Q}, \times) 、非可換群

の例としては n を 2 以上の自然数とするときの n 次正方行列 $(M(n), \cdot)$ を挙げておきます。

ここで半群や群といった数学的構造を持つ集合は整数や行列といった数や多項式で構成された数学的対象から構成された集合であるとは限りません。絡み目や結び目といった幾何学的対象でも、その性質を吟味することで数学的構造を引き出すことができます。たとえば、絡み目/結び目の集合に対しては「**連結和**」と呼ばれる新しい絡み目/結び目を作り出す操作があります。この操作は、向き付けられた二つの結び目 K_1, K_2 の紐がまっすぐな箇所を選び、その箇所を取り外して双方の向きを保つように繋ぎ合わせて新しい結び目 $K_1 \# K_2$ を作ります。絡み目に対しては、その第一成分に対して結び目のときと同様に行うものとします。

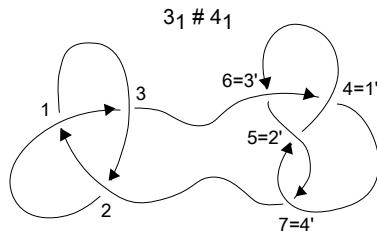


図 6.3 連結和の例

図 6.3 で示している結び目は、向き付けられた三葉結び目（左側の 3_1 ）と八の字結び目（右側の 4_1 ）の連結和 $3_1 \# 4_1$ です。連結和は向き付けられた絡み目/結び目に対して一意に定まり、しかも、左右の被演算子の入れ替えができます。実際、結び目 K_1, K_2 の連結和 $K_1 \# K_2$ にて K_1 を小さく潰して K_2 上の好きな位置に動かして $K_2 \# K_1$ に変形できるためです。そこで演算子 “#” を和 “+” のようなものと見做しましょう。このときに自明な結び目 O と任意の結び目 K との連結和を考えてみましょう。すると、 $K \# O = O \# K = K$ であることは連結和の定義から問題がないでしょう。したがって、(向き付けられた絡み目/結び目, #) は自明な結び目を単位元 0 とする半群になることが判ります。ところで、この連結和は延々と結び目を生成するだけで自明な結び目同士でなければ自明な結び目が得られません。実際、 $K_1 \# K_2 = 0$ とするときに K_1 を小さく潰して動かしたところで結び目が解けるのは K_2 が自明な結び目のときだけです。これは K_1 についても同様であることから $K_1 \# K_2 = 0$ になるのは双方が自明な結び目 0 のときに限られることが判ります。さらに、「**二つの結び目の連結和として表現できない結び目**」のことを「**素な結び目 (prime knot)**」と呼びます。なお、連結和から構成された結び目はスケイン多項式と呼ばれる結び目/絡み目の不变量が連結和を構成する結び目の積になり、自明な結び目がスケイン多項式で 1 になることから素な結び目は結び目のスケイン多項式の既約性に関係します。そして多項式不变量が 1 になる結び目が自明な結び目に限定されるかという重要な問題があります。

図 6.3 で示している結び目は、向き付けられた三葉結び目（左側の 3_1 ）と八の字結び目（右側の 4_1 ）の連結和 $3_1 \# 4_1$ です。連結和は向き付けられた絡み目/結び目に対して一意に定まり、しかも、左右の被演算子の入れ替えができます。実際、結び目 K_1, K_2 の連結和 $K_1 \# K_2$ にて K_1 を小さく潰して K_2 上の好きな位置に動かして $K_2 \# K_1$ に変形できるためです。そこで演算子 “#” を和 “+” のようなものと見做しましょう。このときに自明な結び目 O と任意の結び目 K との連結和を考えてみましょう。すると、 $K \# O = O \# K = K$ であることは連結和の定義から問題がないでしょう。したがって、(向き付けられた絡み目/結び目, #) は自明な結び目を単位元 0 とする半群になることが判ります。ところで、この連結和は延々と結び目を生成するだけで自明な結び目同士でなければ自明な結び目が得られません。実際、 $K_1 \# K_2 = 0$ とするときに K_1 を小さく潰して動かしたところで結び目が解けるのは K_2 が自明な結び目のときだけです。これは K_1 についても同様であることから $K_1 \# K_2 = 0$ になるのは双方が自明な結び目 0 のときに限られることが判ります。さらに、「**二つの結び目の連結和として表現できない結び目**」のことを「**素な結び目 (prime knot)**」と呼びます。なお、連結和から構成された結び目はスケイン多項式と呼ばれる結び目/絡み目の不变量が連結和を構成する結び目の積になり、自明な結び目がスケイン多項式で 1 になることから素な結び目は結び目のスケイン多項式の既約性に関係します。そして多項式不变量が 1 になる結び目が自明な結び目に限定されるかという重要な問題があります。

では半群や群といった数学的対象の定義はできました。では、それらが具体的にどのようなものであるかを表記する方法はないでしょうか？この一つの方法として「**群の表示**」あります。ここで群の表示の説明のために語の集合の例を挙げて説明しましょう。まず集合 A を a から z までのローマ小文字を並べた文字列（「**語**」と呼びます）と空白 “”（ここでは ϵ と表記します）の集合とします。それから演算 “*” を単純に二つの語を繋ぐ操作とします。たとえば $mike * neko$ の結果は $mikeneko$ と演算子 “*” の左右の語を繋いだ文字列です。また $WWWWW\cdots$ のように同じ語 W が n 回続くときに W^n と表記します。ここで空白 “” を文字として考えると、この空白を語の左右に繋いでも元の語になるので空白 “” が演算 * の単位元になることが判ります。そして、語 $W_1W_2W_3$ は $(W_1 * W_2) * W_3$ と $W_1 * (W_2 * W_3)$ の双方から構成されること^{*1}から演算子 “*” は結合律を充します。このように単位元が存在して結合律を充すために集合 A と演算子 * の対 $(A, *)$ は半群になります。さらに語 W に対して W^{-1} を $W * W^{-1}$ と $W^{-1} * W$ を空白で置換する操作とします。具体的には語 w が 2 個以上のアルファベット小文字で構成された語であれば、その文字の並びを逆にして各文字を対応する文字の除去操作で置き換えます。たとえば語 W が $neko$ であれば W^{-1} は $o^{-1}k^{-1}e^{-1}n^{-1}$ になります。こうすることで語 W^{-1} は語 W の逆元になり、以上から $(A, *)$ が群になることが判ります。ここで群 A の元は a から z までのローマ小文字で構成されるために、そのことが判るよう $\langle a, \dots, z \rangle$ と記述します。ここで用いた記号 “...” は Python の拡張スライス構文で省略を意味する Ellipsis というオブジェクトになりますが、ここで表記も同様の省略を意味する記号です。このアルファベット小文字の例と同様に n 個の対象 a_1, \dots, a_n を連結することで生成されるものも群になるため、これも $\langle a_1, \dots, a_n \rangle$ と表記し、この群を「**自由群**」と呼びます。そして $\langle a_1, \dots, a_n \rangle$ の中の対象 a_1, \dots, a_n を「**群の生成元**」と呼び、生成元が n 個の自由群を F_n と表記します。このように自由群は単純にその群の元を繋ぎ合せる処理と削除を持つ集合です。また、積の順序を入れ替えることは語順を入れ替えることと同値で、生成元が 2 個以上の自由群であれば、もとの語と別の語ができてしまいます、実際、最初の語の例で ‘inu’ と ‘uni’ は別物であるために、生成元が 2 個以上の自由群は非可換群です。しかし、生成元が一つだけのときのみ可換群になります。

ところで、全ての群が自由群ではありません。たとえば、スイッチボタンによる ON/OFF 操作も群としての構造を持ちます。実際、スイッチボタンを押すという操作を a とすると生成元はこの a だけになりますが、状態は ON と OFF の二つだけで、さらにボタンの二度押しで元に戻ることから $a^2 = 1$ という「**規則**」がありますが、自由群は延々と新しい元を演算子 “*” を使って生成するだけで、このような規則がありません。このように群の生成元を使って、その群が持つ規則を表現した式を「**関係式**」と呼び

^{*1} 「太った猫と犬」の例のようにその意味を考えることはなく、単に文字の羅列としてです。

ます。スイッチの例では $a^2 = 1$ がその関係式になります。だから集合の表記にならって $\langle a | a^2 = 1 \rangle$ とこの群を表示します。この方法に倣って、群の生成元が a_1, \dots, a_n で関係式が r_1, \dots, r_m であれば群を次で表示します：

群の表示

$$\langle a_1, \dots, a_n | r_1, \dots, r_m \rangle$$

ここで関係式を変形して単位元 1 に等しい式で置き換え、さらに ‘= 1’ を自明なものとして省略することができます。スイッチの例で関係式が $a^2 = 1$ のために $\langle a | a^2 \rangle$ という群の表示が得られます。そのような 1 に等しくなる式 r_1, \dots, r_m を「**関係子 (relator)**」と呼びます。この関係子で群 G の表現を与えると群 G の生成元で生成される自由群 F_n から群 G への自然な写像 f が考えられます：

$$\begin{array}{ccc} f & : & \langle a_1, \dots, a_m \rangle \rightarrow \langle a_1, \dots, a_m | r_1, \dots, r_n \rangle \\ & \Downarrow & \Downarrow \\ a_i & \mapsto & a_i \end{array} \quad i \in \{1, \dots, m\}$$

この写像 f は自由群 F_n の生成元 a_i をそのまま群 G の生成元 a_i に写すだけの写像で自由群での積 ab はそのまま群 G での積 ab に対応させ、自由群 F_n の単位元 1 もそのまま群 G の単位元 1 に写します。つまり、 $f(ab) = f(a)f(b)$ と自由群側の演算を写した側の群でも保ち、 $f(1) = 1$ と単位元を単位元に写す性質があります。この積と単位元を保つ性質を持つ群から群への写像を「**(群) 準同型写像**」と呼びます。この写像 f によって自由群 F_n の項である関係子 $r_{i,i \in \{1, \dots, m\}}$ は全て 1 に写され、さらに、それらの逆元も積も群 G の単位元 1 に写されます。また写像 f が準同型であることから、自由群 F_n の単位元 1 も写像 f で群 G の単位元 1 に写されます。このことから関係子からも群が構成されることが分かります。この関係子の集合のように群の部分集合で群の性質を充たすものを「**部分群**」と呼びます。そして、関係子が構成する群のことを特に「**帰結群**」と呼びます。

一般的に群 G_1 から群 G_2 への準同型写像 $f: G_1 \rightarrow G_2$ で群 G_2 の単位元 1 に写される群 G_1 の元の集合を $\text{Ker}(f)$ と表記して準同型写像 f の「**核 (kernel)**」と呼びます。この核 $\text{Ker}(f)$ は帰結群の議論で見たように群 G_1 の部分群になります。ここで見たように関係子を記述することは自由群からの自然な準同型写像の核を記載することに対応します。このように群の表示を与えることで群の具体的な形が見えてきます。これで絡み目/結び目を指示する群を表現する手段が得られました。つぎは絡み目/結び目の群の表示の具体的な話に移りましょう。

6.6 結び目/絡み目を表現する群

6.6.1 基本群と組紐群

結び目/絡み目に固有の群に、それらの補空間の「**基本群 (fundamental group)**」と呼ばれる群があり、これらの群は単に「**結び目/絡み目群**」と呼ばれます。また、結び目/絡み目を組紐と呼ばれるものに変形することで「**組紐群**」と呼ばれる群が構築できます。これらの群の構成は正則射影図から容易に行うことができます。ここでは基本群と組紐群について概要を述べましょう。

6.6.2 基本群について

ここでは結び目/絡み目の基本群の構成手順を述べます。結び目/絡み目の基本群の構成では向き付けをした正則射影図を用います。

この正則射影図の向き付けは結び目/絡み目を構成する各1次元球面に向きを入れ、その向きを正則射影図で矢印で表現しています。ここで三葉結び目に向きを入れた正則射影図を図6.4に示しておきますが、絡み目/結び目の穂空間の基本群の構成方法には「**Wirtinger表示**」と「**Dehn表示**」の二種類の構成手順があり、ここでは Wirtinger 表記を採用します。この Wirtinger 表示による結び目/絡み目群の表示は次の形になります：

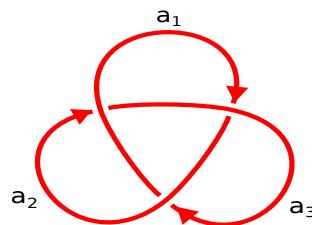
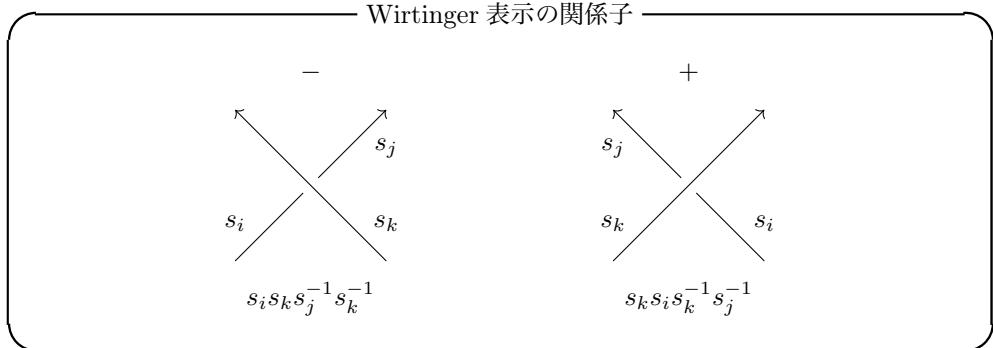


図 6.4 向き付けられた正則射影図

結び目/絡み目群の Wirtinger 表現

$$\text{結び目/絡み目群} = \langle \text{上道}_1, \dots, \text{上道}_n | \text{関係子}_1, \dots, \text{関係子}_n \rangle$$

ここで n は正則射影図の交点の数、すなわち道の総数で、関係子は各交差点で定まります。ただし n 番目の関係子は他の $n - 1$ 個の関係子から生成できるために不要です。これらの関係子は次で与えられます：



各交差点の上にある $+1$ と -1 は「**交差点の符号**」と呼ばれ、正則射影図 \mathcal{D} の交差点の符号の総和を「**捻れ**」と呼び、 $w(\mathcal{D})$ と表記します。なお、基本群の元は空間内部の一定の基点から出て戻る向き付けられた閉曲線として表現することができます。特に単位元 1 になる元は、それに対応する基点付きの閉曲線に滑らかな 2 次元の円盤が空間内部で張れるという幾何学的な性質（デーン (Dehn) の補題）が対応します。このように結び目/絡み目の基本群の表示の構成自体は非常に機械的に行えますが、群の表示が得られたからといって、その群が同じものであるかを確認することは簡単ではありません。また、基本群は幾何学的な構造を反映する群であっても、誰もが直感的・視覚的に判り易い群ではありません。そこでより視覚的に分かりやすいもう一つの結び目/絡み目を表現する群である組紐群 (Braid group) について解説しましょう。

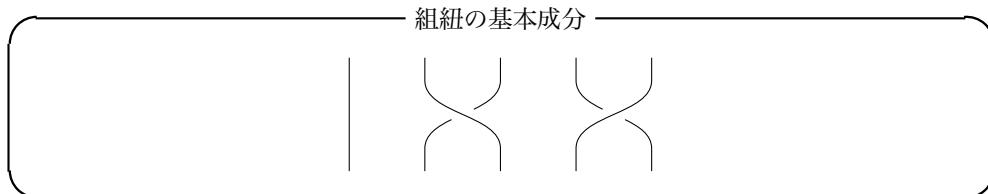
6.6.3 組紐について

組紐群は組紐と呼ばれる図形に関する群で、基本群よりも群としての構造が判り易くなります。まず、幾何学上の対象としての「**組紐 (Braid)**」は n 本の紐 (閉区間 $[0, 1]$ と同相) の 3 次元球 B^3 への埋め込みです：

組紐の定義

1. n 本の紐の 3 次元球 $B^2 \times [0, 1]$ への埋め込みである。
2. 紐と 3 次元球の断面 $B^2 \times t, t \in [0, 1]$ の交差点は 1 点のみである。
3. 蓋 $B^2 \times 1$ と底 $B^2 \times 0$ での紐の交差点は等間隔に並ぶ。

組紐は酒樽状に整形した 3 次元球 $B^2 \times [0, 1]$ の上下の蓋 ($B^2 \times 1$ と $B^2 \times 0$) に取り付けられた紐で、その紐は互いに交差することなく上から下へと垂れた状態、つまり、紐が縦軸に対して右回りや左回りで互いに絡んでいます。このことから組紐は以下の基本的な 3 成分で構成されることが分かります：



組紐の一例として図 6.5 に SageMath の BraidGroup モジュールを使って描いた組紐を示しておきます。この例は 3 本の紐で構成された組紐です。このように組紐は非常に具体的で、その構成も基本成分をどのように積み上げるかで決まります。例では 3 本の紐でしたが、組紐を構成する紐の数が n 本のときに「 n 糸の組紐」と呼びます。組紐の定義にはこのような具体的な定義に加えて配置空間 (configuration space) と呼ばれる空間を使った定義もあります [39]。なお、組紐の充たすべき性質で 2. を充たさない紐の埋め込みのことを「タングル (tangle)」呼びます。タングルは組紐のように紐が上蓋から出発して下蓋に向かって下がる一方でなく、紐に結び目を作ることも可能になるために組紐よりも複雑な図形になりますが、基本成分は新たに「最小 (消滅)」と「最大 (生成)」の二つが加わるだけです：

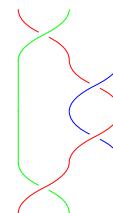


図 6.5 組紐の例

タングルの基本成分



ここでは紐に向きを入れていないために 5 成分のみですが、向きを入れたタングルでは組紐のように上から下への一方的な向きにならないため、この 5 成分から二つの交差を除いた成分を基に 8 成分になります。それに応じて正則射影図上の変形操作も組紐よりも操作が増えます。

組紐の同値性は結び目/絡み目のときと同様に、直感的には紐を切らずに紐を延したり縮めたり、局所的に平行移動させることで相互に移りあえるときです。つまり、二つの組紐に「アンビエント・イソトピー (ambient isotopy)」が存在するときと結び目/絡み目のときと同様に言い換えられます。ただし、組紐は結び目/絡み目よりも視覚的にも明瞭な代数的な表現に訴れます。それが組紐群と呼ばれる群です。

6.6.4 組紐群

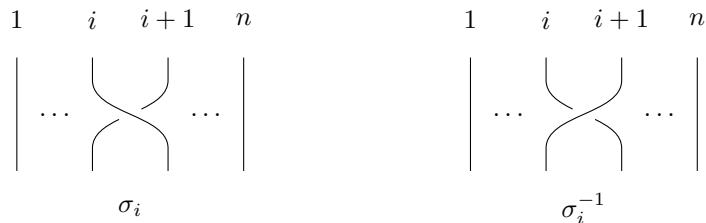
組紐からは「**組紐群 (Braid group)**」と呼ばれる群が構成されます。この群は紐が n 本の組紐、すなわち n 糸の組紐であれば次の群の表示になります：

n 糸の組紐群

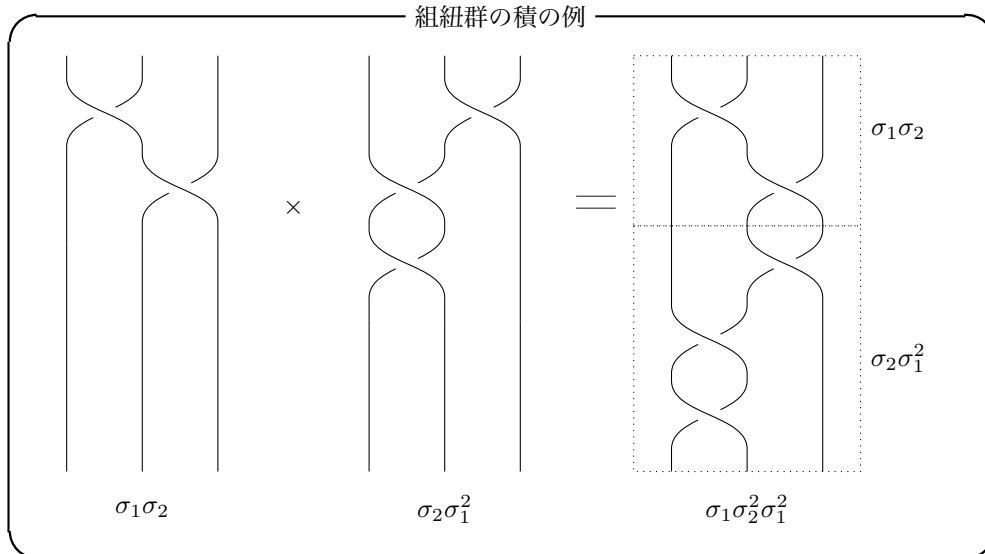
$$\left\langle \sigma_1, \dots, \sigma_{n-1} \mid \begin{array}{l} \sigma_i \sigma_k = \sigma_k \sigma_i, (|i - k| \geq 2, i, k \in [1, n-1]), \\ \sigma_i \sigma_{i+1} \sigma_i = \sigma_{i+1} \sigma_i \sigma_{i+1}, (i \in [1, n-2]) \end{array} \right\rangle$$

では生成元 σ_i は具体的にどのようなものでしょうか？ つぎに σ_i と σ_i^{-1} と対応する組紐を示しておきましょう：

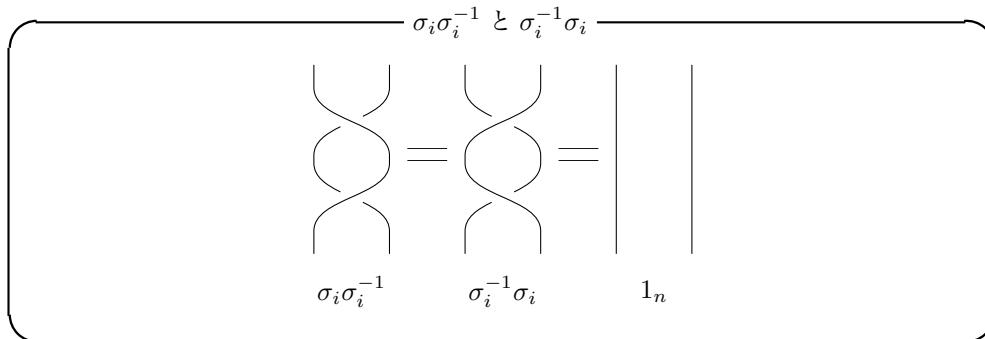
n 糸の組紐群の生成元



ここで図示したように σ_i は左から i 番目と $i+1$ 番目の紐を縦方向を Z 軸として反時計回りに 180 度回すという操作、 σ_i^{-1} は紐を反対の時計回りに 180 度回す操作です。ここで σ_i^{-1} の右肩の -1 の意味は組紐群の積に対応する操作の結果から判ります。それから組紐群の積演算は二つの組紐 a, b が与えられたときに組紐 a を上、組紐 b を下にしてこれらの組紐を縦に繋ぐ操作に対応します。具体的な例を以下に示しておきましょう：



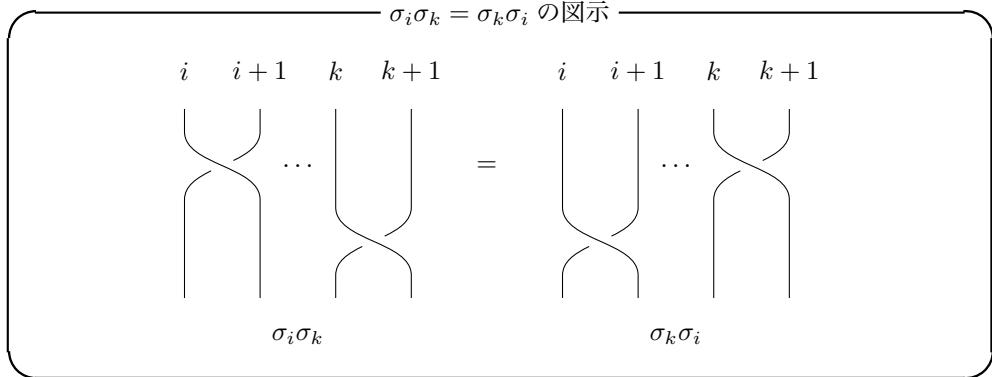
この例では $\sigma_1\sigma_2$ と $\sigma_2\sigma_1^2$ の積 $\sigma_1\sigma_2^2\sigma_1^2$ を示しており、右に示すように $\sigma_1\sigma_2$ を $\sigma_2\sigma_1^2$ の上に積み上げる形になります。また積の順番から言えば、左側から右にかけて対応する組紐を上から下に繋ぐ恰好になります。さて、ここで n 本の紐がまっすぐに垂れた組紐を n 糸の組紐の上に付けても下に付けても各紐の長さを調整してしまえば元の n 糸の組紐に戻せます。つまり、このことはまっすぐに垂れた n 本の組紐が n 糸の組紐群の単位元 1_n であることを示しています。また σ_i に対する σ_i^{-1} の意味ですが $\sigma_i\sigma_i^{-1}$ と $\sigma_i^{-1}\sigma_i$ を描いてみると共に単位元 1_n と同じ組紐であることが判ります：



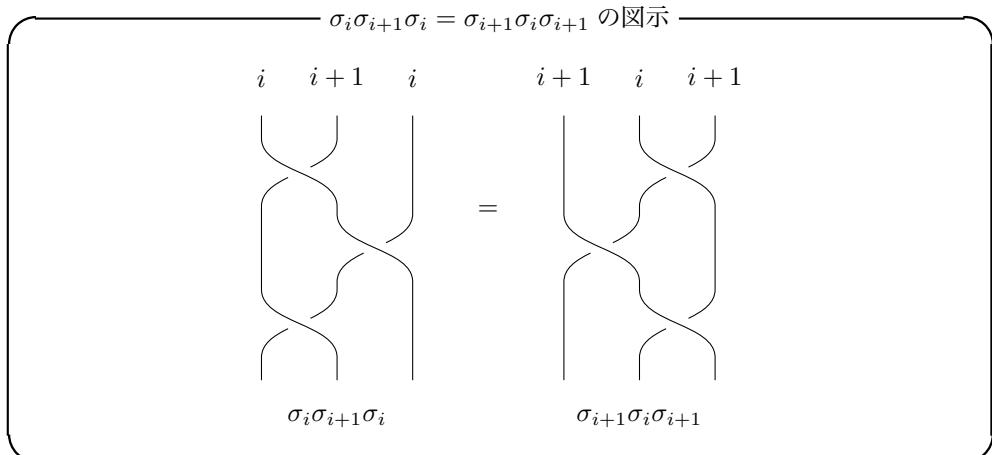
実際、左側の状態の紐を引っ張って長さを調整すれば 2 本のまっすぐな紐になります。このことから σ_i^{-1} は表記どおりに σ_i の逆元です。また $a_1a_2 \cdots a_n$ が与えられたとき、その逆元は $a_n^{-1} \cdots a_2^{-1}a_1^{-1}$ と語の順序を逆にして幕の正負を逆にしたもので与えられることも判ります。

ここまで群の表示の生成元は理解できたでしょう。要するに組紐を重ねるという操作がそのまま組紐を表現する項同士の積なのです。それから群の表示には生成元の右側に二

つの関係式: $\sigma_i\sigma_k = \sigma_k\sigma_i$ と $\sigma_i\sigma_{i+1}\sigma_i = \sigma_{i+1}\sigma_i\sigma_{i+1}$ があります。この関係式は図示するとその意味が明瞭になります。そこで最初に $\sigma_i\sigma_k = \sigma_k\sigma_i, |i - k| \geq 2$ を図示しましょう：



この図から $|i - k| \geq 2$ のときに $\sigma_i\sigma_k$ と $\sigma_k\sigma_i$ は互いの操作が影響する範囲にないために交差点を上下を動かせることに対応します。この上下に紐を移動させるという操作で互いの組紐が得られることから、これらの組紐を同じものとみなすことに問題はないでしょう。次に $\sigma_i\sigma_{i+1}\sigma_i$ と $\sigma_{i+1}\sigma_i\sigma_{i+1}$ で表現される組紐を図示してみましょう：

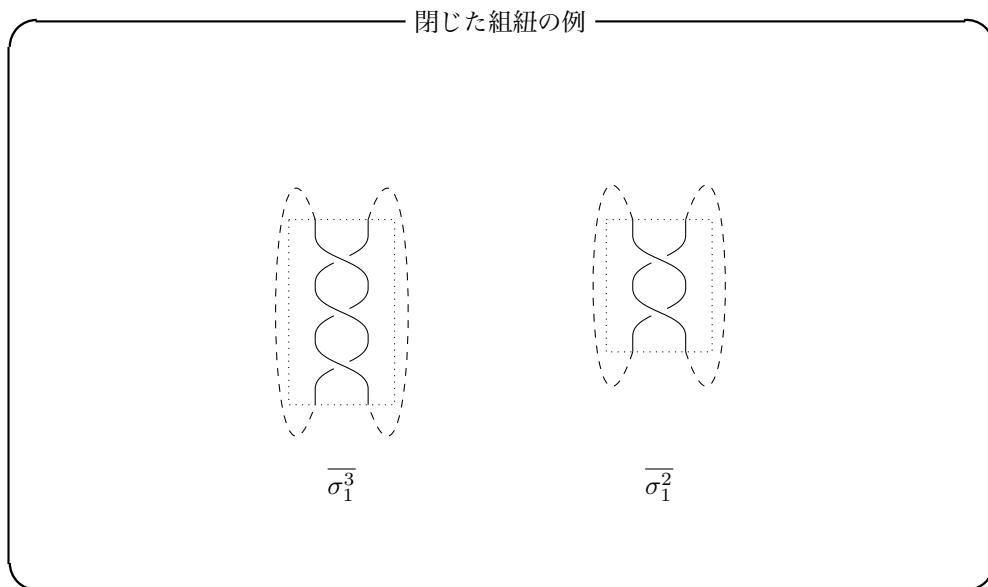


この関係式も図示してしまえば明確です。実際、この関係は i 番目の紐を上下に動かして相互の組紐に変形できることを意味します。この操作は同値な結び目/絡み目が得られるライデマイスター移動の TYPE III に対応する操作です。このように関係式は項の同値性を与える式で、この式を利用して項をより簡単なものにまとめられますが、この関係式をこのように可視化することで、その図形的な解釈が明瞭になります。

6.6.5 組紐と結び目

組紐の上下の端点を交互に繋ぐ操作で得られる図形を「閉じた組紐 (closed braid)」と呼びます。そして閉じた組紐で幾つかの円が得られます。円が一つだけのときは結び目、円が二つ以上のときは絡み目になります。ここでは縦に閉じた n 糸の絡み目 b を \bar{b} と表記します。

ここで実際に例をみておきましょう：



この例は 2 糸の組紐を縦に閉じたもので、他に頂部と下部で横に繋ぐ方法もあります。なお、横に繋ぐ場合は紐の本数が偶数個必要になりますが、ここで採用した縦に閉じる方法は紐の本数と無関係に閉じることができます。例の左側の結び目は σ_1^3 から得られる「三葉結び目」と呼ばれる結び目で、右側が σ_1^2 から得られる「ホップ絡み目 (Hopf Link)」と呼ばれる絡み目です。

このように組紐を閉じることで結び目/絡み目が得られますが、逆に「結び目/組紐は閉じた組紐として表現可能である」ことが知られています [12]。この閉じた組紐としての結び目の記述を「結び目の組紐表現」と呼びます。このときに結び目の組紐表現で最小の組紐の本数を「組紐指数 (braid index)」と呼び、この組紐指数は結び目の不変量の一つであることが知られています。

なお、二つの閉じた n 糸の組紐がアンビアント・イソトピーで移りあえるのは次のマル

コフ移動 M I, M II を許容するときに限ることが知られています:

マルコフ移動

$$\begin{array}{lcl} \text{M I. } \overline{ab} & = & \overline{ba} \\ \text{M II. } \overline{a\sigma_n} & = & \overline{a\sigma_n^{-1}} = \overline{a} \end{array}$$

これも図示すると明快になります。移動 M I は a を構成する生成元の閉じた組紐であることから a の上から順にまわして b の下に移すことができることに対応します。また移動 M II は最も右側の紐がライデマイスター移動の I と同じ状況に対応しますが、ただ、この場合は紐の数に変動が生じます。

6.7 組紐群と置換群

ここで $n+1$ 糸の組紐の上蓋で紐の端点に左側から番号を $1, 2, \dots, n$ と振ります。それから紐をたどって下蓋に到着したところで紐に対応する番号を配置します。すると上蓋の $1, 2, \dots, n$ は下蓋では並び替えられています。この並び替える操作を σ と表記すると σ は $\{1, 2, \dots, n\}$ から $\{1, 2, \dots, n\}$ への全単射写像になります。また上蓋の $1, \dots, n$ が (x_1, \dots, x_n) に対応するときに

$$\sigma = \begin{pmatrix} 1 & \dots & n \\ x_1 & \dots & x_n \end{pmatrix}$$

と表記します。このような $\{1, 2, \dots, n\}$ から $\{1, 2, \dots, n\}$ への全単射写像の集合に対し、その演算を函数の合成○とすると、この集合は群になります。この群を「置換群 \mathfrak{S}_n 」と呼びます。組紐群 B_{n+1} との関係は、ここで述べた対応関係によって置換群 \mathfrak{S}_n への自然な写像が得られ、しかも、この写像は積を保つので準同型写像になります。ただし、この準同型写像は「情報の欠落」が生じます。たとえば σ_i と σ_i^{-1} は置換群としては i と $i+1$ を入れ替える操作のために一致しますが、組紐群としては回転の方向が反対で別物です。このように紐の回転の向きの情報に欠落が生じています。

この置換群を使うと組紐を閉じたときに得られた絡み目の成分数が置換群の分析から判ります。この分析では置換群を巡回置換の積として表現しなければなりませんが、ここで置換 σ と $k \in [1, \dots, n]$ に対して $k \xrightarrow{\sigma} \sigma(k), \sigma(k) \xrightarrow{\sigma} \sigma^2(k), \dots, \sigma^i(k) \xrightarrow{\sigma} \sigma^{i+1}(k) = k$ と k の置換 σ による像を追いかけることで次の置換:

$$\begin{pmatrix} k & \dots & \sigma^{i-1}(k) \\ \sigma(k) & \dots & \sigma^i(k) \end{pmatrix}$$

が得られます。これが「巡回置換」と呼ばれる置換で、より簡潔に $(k, \sigma(k), \dots, \sigma^i(k))$ と表記され、任意の置換 $\sigma \in \mathfrak{S}_n$ は巡回置換の積として表現できます。実際、 $\sigma \in \mathfrak{S}_n$ に対

し, 1 から 1 に戻るまで置換 σ で写して巡回置換を求め, この巡回置換で現れなかった数に対して 1 のときと同様に置換 σ で写して巡回置換を求めます. この作業で全ての数が出た時点で置換 σ を構成する巡回置換が全て求められ, これらの巡回置換の積として置換 σ が得られます. さて, ここで組紐を組紐群で表現し, それを置換群への自然な写像で置換 σ に写されたとします. ここで巡回置換は k 番目の始点から出発して $\sigma^{i+1}(k)$ で出発点に戻るために, その軌跡が一つの円周を構成していることが判ります. そして, 置換 σ が巡回置換の積として表現されることから, その巡回置換の数だけ円周が, すなわち, 絡み目の成分が現われます.

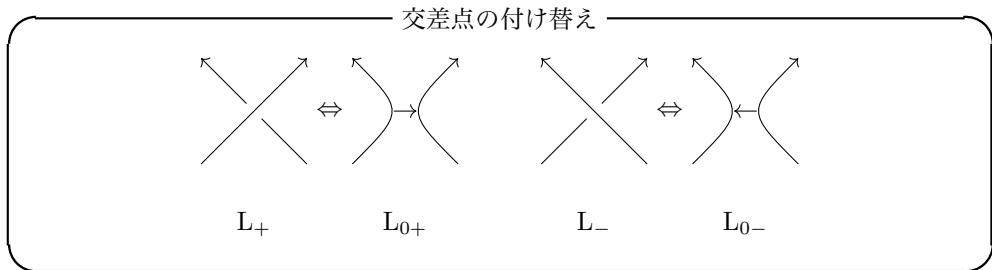
6.7.1 ザイフェルト曲面

結び目/絡み目には組紐への変換処理がありますが, この処理には結び目/絡み目の不变量の計算で用いるさまざまな操作が含まれ, 特に「ザイフェルト曲面 (Seifert surface)」と呼ばれる曲面の作成手順にも重なるために, この曲面の構成方法も含めて解説します. なお, ザイフェルト曲面は, その境界が結び目/絡み目になる向き付け可能な曲面ですが, ここでの「向き付け可能」という意味は曲面に「裏と表がある」ということです^{*2}. そして, 向き付け可能な閉曲面は「種数 (genus)」と呼ばれる自然数で完全に分類されることが知られています. この種数は「ドーナツの穴の数」に対応し, 種数 0 は 2 次元球面 S^2 , 種数 1 がトーラス T^2 , すなわち穴一つのドーナツの表面になります. このことからザイフェルト曲面は絡み目の本数が n であれば, n 個の互いに離れた円盤をある種数 m の閉曲面から取り除いた曲面に分類できます. そして, 結び目/絡み目のザイフェルト曲面は以下に述べる手順で機械的に構築できます.

■正則射影図の準備: 結び目/絡み目にはあらかじめ向きを入れ, 向き付けられた結び目/絡み目の正則射影図を構築します. 正則射影図に交差点が一つも存在しなければ絡み目の全ての成分は自明な結び目で, 円盤を貼りつけることでザイフェルト曲面が得られます. しかし, 通常は有限個の交差点が存在するために各交点に対して次に述べる交点の解消処理を行います.

■交差点の解消: 絡み目の向き付られた正則射影図の各交差点を次の手順で書き直します:

^{*2} 向き付られない曲面の代表格が帶に 180 度の捻りを入れて繋いでできるメービウスの帯 (Möbius band) です.



ここでこの処理は、交差点に入る下道は交差点から出る上道に繋ぎ、交差点に入る上道と交差点から出る下道に繋ぎ替えるという処理です。この処理を結び目の正則射影図で行うと成分の分離が生じますが、上道と下道が絡み目の別成分であれば二つの絡み目の成分が一つに融合します。この処理を全ての交差点に対して行うと正則射影図から交差点が消えて幾つかの孤立した円周とそれらを繋ぐ矢印だけが残ります。この最終的に得られる正則射影図の円周を「ザイフェルト円周」、ザイフェルト円周とそれらを繋ぐ矢印の全体を「ザイフェルト系」と呼びます。なお、ザイフェルト系の各円周を繋ぐ矢印はそれらを繋ぐ橋として各交差点の符号に対応した捩れた帯の表現になります。では、実際に結び目を使って変形操作を説明しましょう。最初に結び目の正則射影図に向きを入れ、交差点を線分で置換えたものを以下の図6.6に示します：

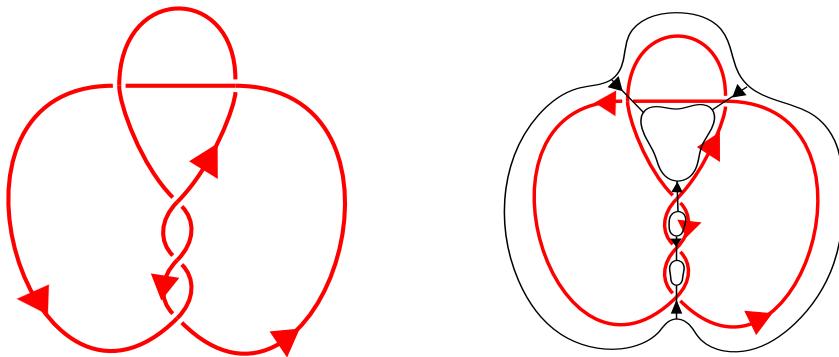


図 6.6 結び目と交差点の変換

それから左側の向きを入れた結び目に対して交差点での変形操作を行うことで右図の4個のザイフェルト円周と交差点に対応する5個の矢印で構成されたザイフェルト系が得られます。

■ザイフェルト曲面の構成: ザイフェルト系に対してそのザイフェルト円周に円盤を貼り付け、矢印に本来の帯の正負に対応する捻りを入れた帯で置き換えて得られる曲面がザイフェルト曲面です。この曲面は構築手順から向き付られた曲面で、その境界は結び目/絡

み目になります。なお、曲面の構築を次の円の向きを揃えた時点で行っても構いませんが、円の向きを揃える操作で後述のように線分が増え、種数が増加した曲面になります。それから円盤の貼り方にも二通りの方法があります。これは結び目/絡み目を一点コンパクト化で3次元球面 S^3 内の対象として捉えたことから、円盤を無限遠点を含まないように貼る通常の貼り方と無限遠点を含むように貼る貼り方です。無限遠点を含まないように貼るとでき上がった曲面はホットケーキを数段重ねたような形になりますが、無限遠点を含むように貼ると幾らか平面的な曲面ができ上がります。図6.7にこれら二通りの円盤を貼り付けたものを示しておきます：

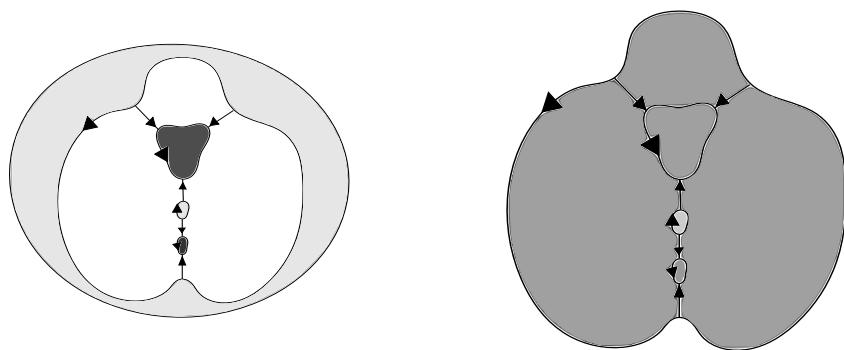


図 6.7 結び目の円盤

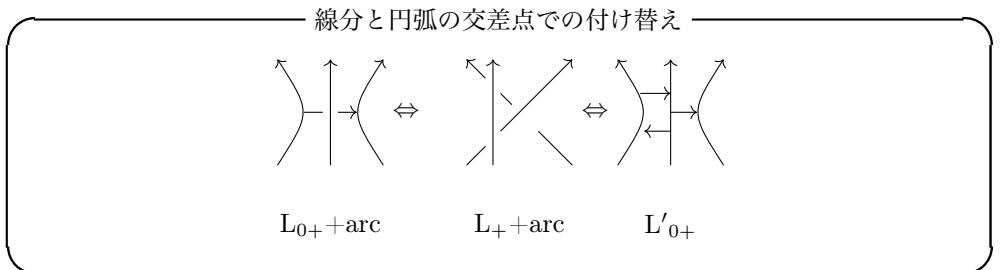
左図が無限遠点を含む円盤も貼ったもの、右が無限遠点を含まない円盤のみを貼り付けたものです。左側の方法では平面上に配置されて孤立した円盤を帶で結び付けるかたちになりますが、右側では有界な円盤を貼るために最も外側の円に貼る円盤が、それまで貼った円盤の下になるように貼り、円盤の配置が同一平面上にありません。ザイフェルト曲面はこれらの円盤を繋ぎ合わせている矢印を、その矢印に対応する捩りを入れた帶で置換えることで構築できます。これでザイフェルト曲面の構築は終わりですが、さらに手を加えると組紐表現が得られます。

■円の向きを揃える：ここで円盤を貼る前のザイフェルト系に対して操作を行います。この時点ですべての円と矢が交差することのないように平行移動が可能で、円の向きが一致していれば線分を円周上で移動させて円の左側に配置させます。円の中心と向きが揃っていないければ各円の中心を一致させて向きも揃えます。そのための手順を次に示します：

1. 基準円を一つ定める。選び方は、複数の円の最も内側になっているもの、あるいは線分が一番多く出ているもののいずれか。

2. 基準円に線分でつながった円で、その向きが一致した円は、他の円周や線分と交差しないように中心点を揃える。逆向きの円があれば構築手順から基準円をその内部に包含しないため、その円から出ている線分を適宜、演習に沿って移動させ、線分と交差点を持たない側の円弧を動かして基準円を包含するようにできる。このときに基準円から出ている線分と交差が生じるが、この線分は捻りの入った帶であるために交差点が新たに二つ生じる。そのため、図式では外側に向かう一つの線分を二つの線分で置き換える形になる。これらの交差点は有限個のために有限階の操作で済ますことができる。基準円と線分で繋がっていない円があるときは基準円と向きと中心を合わせ、それからこの円を基準に線分で繋がった円に対して向きと中心を合わせる操作を行う。

2. の操作で生成された交差点の線分への置き換え操作を行います。たとえば L_+ の線分は次の置き換えが生じます。これは L_- の交差点でも本来の線分の向きを逆向きにするだけで、新たに加わる交差点の置き換えになる線分の向きは一緒です:



左図が L_{0+} に他の円が線分に交差した状況で、本来は真ん中の絵の交差です。この交差を解消したものが右図の状態で、線分が二つに割れた状況になります。これらの処理によって複数の円は共通の中心点を持つ同じ向きの円として再構成できます。この操作を先程の結び目を使って説明しましょう。まず、ザイフェルト系から基準円を選択した様子を以下に示します:

この図で左側が結び目のザイフェルト系で、そこから基準円を選択し、他の円を動かした結果が右図です。星印を包含する最小の円が基準円で、この円を包含する円は最も外側の円のみで他の二つは違います。だから、これらの二つの円に変形操作を行わなければなりません:

左図では基準円に線分で直接繋がっている円の向きを合せて中心を揃えています。このときに線分で繋がっている側の円弧はそのままで線分で繋がっていない円弧を引いて基準円を包含するように移動させますが、他の円と基準円を繋ぐ線分との交差にも置換を行うことで図では新たな二つの線分が外側に生じます。右図は残った円に対する操作で、この円は向きは一致しており、中心を合致させるように平行移動させます。ここでも新たな線分との交差が生じるために、その交差を二つの線分で外側に置換します。

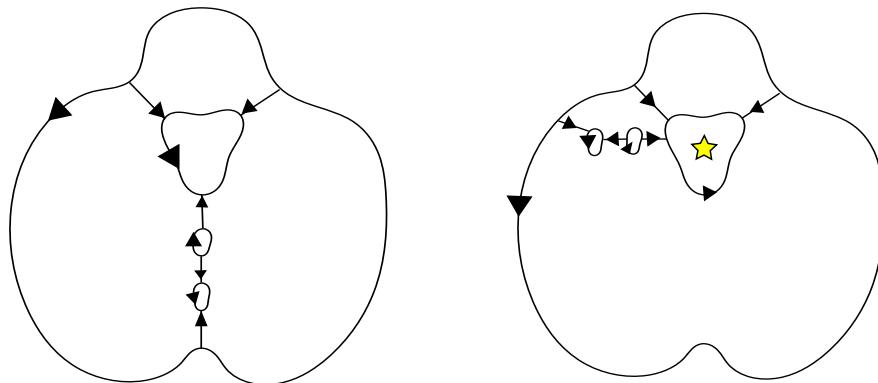


図 6.8 結び目のザイフェルト系と基準円の選定

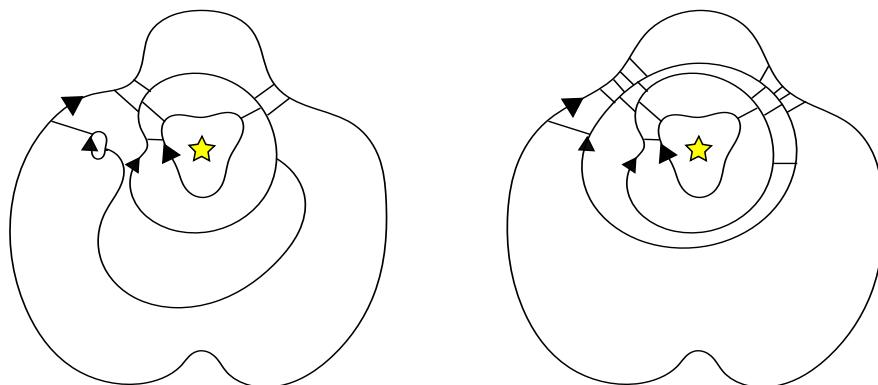


図 6.9 結び目のザイフェルト円周の変形操作

■閉じた組紐表現: 全ての円が同じ向きで共通の中心点を持っていれば、これらの円を繋ぐ線分を円周に沿って中心の左側に移動させて纏め、それから線分を本来の交差に戻します。これらの処理で閉じた組紐の表現が得られるが、この変換は交差数が最小になる「綺麗な組紐」を与えるものではありません。

この様子を先程の例に対して行ったものが図 6.10 です:
閉じた組紐表現が得られた時点での各円に円盤を貼り、線分を帶で置き換えてザイフェルト曲面が得られます。ただし、閉じた組紐表現から構成された曲面はホットケーキを重ねたような曲面になります。実際、閉じた n 級の組紐が得られたのであれば n 段重ねのホット

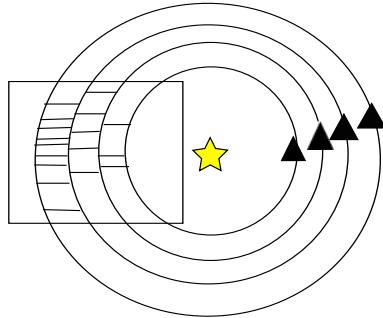


図 6.10 結び目の閉じた組紐としての表現

ケーキ状の曲面が構築され、この曲面が最小種数を持つ可能性はまずないでしょう。しかし、この手法では有界な円盤と 180 度の捻りを入れた帯だけでザイフェルト曲面が機械的に構築できます。

6.8 ガウス・コード

絡み目/結び目の正則射影図をリストを使って効率的に表現する方法に「ガウス・コード」があります。このガウス・コードは絡み目の各成分の向きを入れ、その正則射影図の各交差点に重複がないように番号を割り当てた状態から構築されるリストで、交差点番号の指定や基点の取り方次第で異なったリストが得られます。このガウス・コードの構成手順を順番に説明しましょう。まず、最初に向き付けられた絡み目/結び目の正則射影図を構築します。つぎに各交差点に番号を重複がないように配置し、絡み目上の各成分に基点を定めて結び目の向きに沿って結び目上を移動します。ここで交差点を通過する際に下道を通過するときは交差点番号に ‘-’ を付け、上道を通過するときは交差点番号だけにします。この操作を基点に戻るまで行います。この手順を三葉結び目で確認しましょう。

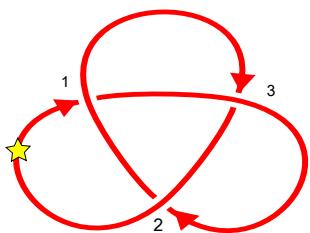


図 6.11 三葉結び目でのガウス・コードの生成

この三葉結び目の基点を \star とします。さて、この基点 \star から開始するときに、基点の乗った道が早速、交差点 1 では下道になるために ‘-1’、交差点 3 では上道になるために ‘3’、それから交差点 2 は下道になるために ‘-2’、それから二度目に通過する交差点 1 では上道のために ‘1’、同様に交差点 3 では下道になるために ‘-3’、次の交差点 2 では上道であることから ‘2’ になって出発点に戻ります。この数列を最初から並べると $-1, 3, -2, 1, -3, 2$ で、この数列を「ガウス・コード (Gauss code)」と呼

びます。また、この三葉結び目の例のようにガウス・コードの成分の符号が正負を交互に変化する結び目を「**交代結び目 (alternating knot)**」と呼びます。絡み目の場合も同様ですが、絡み目の成分に順序を入れ、その順序に従って上記の方法で数列を構成します。問題になるのは他の成分との交差で、二度目に交差点番号が現われる成分で、その交差点番号の符号を付与した数にします。また、成分が一つだけの絡み目が結び目であるために、計算機上のガウス・コードの表現をリストのリストとします。たとえば、三葉結び目は $[-1, 3, -2, 1, -3, 2]$ と表記します。そして、ガウス・コードの成分 $[-1, 3, -2, 1, -3, -2]$ を絡み目の成分に対応するガウス・コードの成分リスト、あるいは単に成分リストと呼びます。

このガウス・コードには交差点の符号の情報は含まれておらず、道の繋がり具合から交差点の符号が判るという代物です。しかし、この数列の構成では交差点の分析を行っているためにその情報も追加しておきたいものです。そこで、この交差点の符号の情報を追加した「**拡張ガウス・コード**」を構築しましょう。この拡張ガウス・コードは二種類あり、一つは交差点番号の前に ‘p’, ‘m’ 等の正負を表現する記号を付与する表記方法です。もう一つの方法は二度目に通過した時に交差点の符号を付ける表記です。最初の方法は文字式のリストになりますが、二番目の方法は整数リストになるために計算機での処理は俄然、後者が有利です。ただし、後者の方法は結び目であれば最初の符号によって二度目の符号が決定されているために交差点に関する十分な情報が得られますが、絡み目であれば他の成分の参照が必要になります。実際、絡み目 L の成分 K_i, K_j が交差しているときに、これらの成分が横断的に交差するために交差点は必ず偶数個になり、絡み目 L の成分に順序が入っていて $K_i > K_j$ となっているときに K_i の交差点では通常の上下関係による符号、 K_j の拡張ガウス・コードに交差点の符号に対応する符号が添付されます。ただし、交差点番号 p が一つだけ成分 K_i に現れなかつたときに p が含まれる成分 K_j を探し、絡み目の成分の順序から符号の意味を判断するという操作が必要になります。ここで図 6.11 の三葉結び目に対してこれらの拡張ガウス・コードを示しておきましょう。まず符号に対応する記号を付与する方法であれば $[-m1, m3, -m2, m1, -m3, m2]$ 、後者の二度目に符号を付与する方法であれば $[-1, 3, -2, -1, -3, -2]$ になります。なお、後者の拡張ガウス・コードは SageMath 7.2 から結び目理論パッケージに「向き付けられたガウス・コード (oriented gauss code)」として提供されています。

この交差点の符号が追加されている拡張ガウス・コードであれ絡み目/結び目の交差点を復元可能で、さらに Wirtinger 表現で基本群の表現が計算できます。まず、基本群の生成元は交差点番号が道に対応するために簡単に構築できます。残りの基本群の関係子は n 個の交差点を有する正則射影図であれば、独立した関係子は $n - 1$ 個になるために $n - 1$ 個の交差点の関係子を求めればよいことになります。ここで用いる拡張ガウス・コードは

二度目に交差点番号が現れた時点で交差点の符号を付与する方法にします。その理由は処理が多少煩雑になっても、データ処理が整数リストの処理になって汎用性があるためです。では、手順を次にまとめておきましょう：

拡張ガウス・コードの処理手順

1. 交差点の抽出
2. 交差点の符号の抽出
3. 正規ガウス・コードへの変換
4. 交差点情報の復元

ここに挙げた手順について考察します。ここで絡み目のガウス・コードは絡み目の各成分に対応する成分リストで構成されたリストで、絡み目の向きに由来する向きは成分リストの順序として入っています。この向きは成分リストでリストの先頭から末尾へ向う方向を順方向、末端から先頭に向う方向を逆方向になります。また、道の探索では成分リストをそのリストの両端を繋いだ円環として考え、その円環の順方向は直線のときの順方向に一致させます。なお、ガウス・コードの系全体の交差点の出現順序が重要であるときはガウス・コードの成分を成分リストとして区分する括弧“[]”を外した平坦なリストを用います。このリストを平坦化したガウス・コードと呼び L_f と表記します。このリスト L_f の絶対値を取り、集合に変換すると交差点集合 S_c が得られ、その成分数（濃度） n_c が交差点数になります。

■交差点の抽出：ここでは交差点リスト L_c を構成します。平坦化したガウス・コード L_f の先頭から順番に数 i を抽出し、その絶対値 $|i|$ が L_c に包含されていなければ $|i|$ を L_c に追加します。ここで L_c のリストの長さが交差点数 n_c に一致した時点でこの操作を終えます。この方法はガウス・コードに交差点番号が出現する順番を考慮した方法です。ただし、より簡単な SageMath で実装可能な方法は、 L_f の絶対値を取り、それを集合型からリスト型に変換する方法です。この方法は平坦化したガウス・コード内の出現順を考慮しない、番号順に自動的に並び替えられた交差点リストの取得になります。

■交差点の符号の抽出：ここでの処理では交差点番号リスト L_c に対応する交差点符号リストを L_s を構築します。まず、交差点番号リスト L_c から順番に番号 i を取り出し、 L_f で二度目に番号 i に対応する数が出たときに、その符号を L_s に追加します。これで交差点番号リスト L_c に対応する交差点符号リスト L_s が得られます。ここで交差点リスト L_c と交差点符号リスト L_s は対で処理しなければならないことに注意します。

■正規のガウス・コードへの変換：この処理では交差点の符号情報を除いた正規のガウス・コードを生成します。交差点リスト L_c の先端から順番に i を取り出し、 i が L_f で最初に現われたときの符号を i_s とし、 L_f の先頭から二度目に i_f として現われたときに

$-i_s \times i_f$ で i_f に対応するガウス・コードの元と入れ替えます。この操作で正規のガウス・コード G_c に変換できます。

■自明な成分の除去: 正規のガウス・コードに交差点番号が正のものだけ、負のものだけの成分が存在することがあります。このような成分は自明な成分で他の絡み目と本質的に絡み合っておらず、基本群にこれらの成分が生成元として現れないこと、それと次に述べる交差点情報の復元が煩雑になる原因になるためにガウス・コードからこれらの成分を除去し、同時に自明な成分の除去後に不要になった交差点の削除も行います。なお、後述のスケイン多項式の計算では自明な成分の個数だけが必要になりますが、除去した成分数は拡張ガウス・コードと正規のガウス・コードの成分数の差から得られます。

■交差点情報の復元: 正規のガウス・コード G_c を基に、交差点リスト L_c の順に交差点情報を復元します。ここで復元すべき情報は基本群の関係子の構成に必要な上道と下道の配置状況です。交差点リスト L_c の番号 i が平坦化したガウス・コード L_f に最初に現わされたときの符号が交差の状況に対応します。したがって、ここでの処理で考察すべきことは、交差点番号 i で上道が何で、下道が何と何であるかを調べる方法です。ここで下道の定義から交差点番号 i に下道 a_i が必ず入り込むために、交差点 i から出る下道と交差点 i を通過する上道をどのように探すかということが問題になります。そこために $i \in L_c$ が正規ガウス・コード G_c でどのようななかたちで現われるかで分類して考察しましょう：

- $-i$ の場合:

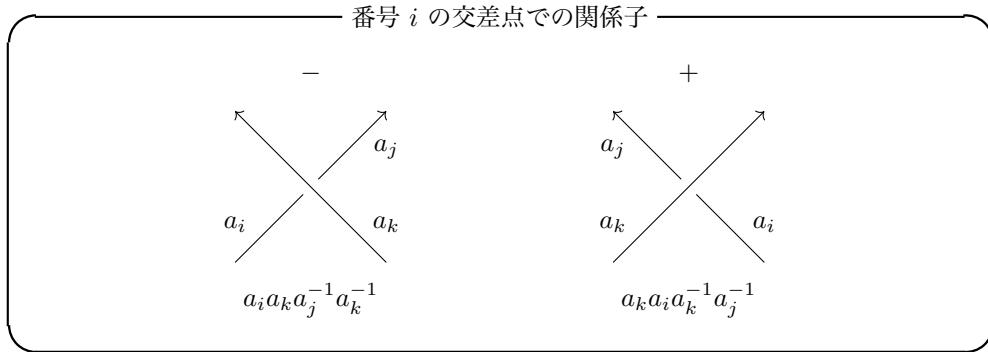
下道 a_i が交差点 i に入るため、ここでは交差点 i から出る下道を探します。まず、 G_c の $-i$ を含む成分リストを円環にして、数 $-i$ を基点にして順方向で隣の数から順番に調べます。ここで j を検証すべき成分リストの数とします。このときに数 j が負の数であれば求める a_{-j} が交差点 i から出る下道になりますが、 j が正の数であれば順方向で j の隣の数を調べます。この操作を繰り返して成分リストを一周して $-i$ に戻ったときは a_i が交差点 i から出る下道になります。このようにして交差点 i に関する二つの下道が求められます。次に交差点 i を通過する上道の探索では、 i が成分リストに現われている箇所を探しますが、 i が $-i$ を含む成分リストに含まれていないことが絡み目で生じます。このときはもう一つ別の G_c の成分リストに i があるため、この i を含む成分リストで上道の検索を行います。ここで成分リストを円環にして i を基点とし、 i の隣の順方向の数 k を調べます。ここで k が負の数であれば a_{-k} が求める上道になりますが、 k が正の数であれば順方向で k の隣の数を調べます。以降、該当する数がなくて i に戻ったときは a_i が求める上道になります。

- i が正の数の場合:

交差点 i を通過する上道に対応することが判りますが、この上道が何であるかは、

この上道が潜り込む交差点番号を探さなければなりません。そのために i が含まれる G_s の成分リストを円環状にして i を基点として順方向で i の隣の数 k を調べます。ここで k が負の数であれば、この $-k$ が求める交差点番号で、上道が a_{-k} であることが判りますが、そうでなければ k の順方向で隣の数を調べます。この操作を繰り返して i に戻ったときは上道が a_i になります。次に下道を探します。そのため $-i$ を成分リストで探しますが、 $-i$ が成分リストになれば、正規ガウス・コード G_s で i を含む別の成分リストで検証を行います。なお、交差点 i に潜り込む下道が a_i であることはガウス・コードの生成方法から判るために、交差点 i から出る下道を探すことになります。これは i を基点に順方向で隣の数を j とすると、 j が負の数であれば a_{-j} が求める下道になります。 j が正の数であれば、 j の順方向で隣の数の検証になります。この手順を繰り替えして i に戻ったときは a_i が求める下道になります。

ここで説明したように、交差点番号から順方向で最初に現れた数が道の番号で、上道の場合と下道の場合で絡み目の成分を指定し、その成分で検査を行うようにプログラムを構築します。このようにして求めた上道と下道の情報に加え、交差点の符号の情報から交差点が次で復元できます：



では、図 6.11 に示す三葉結び目で説明しましょう。

1. 拡張ガウス・コードの計算:

結び目にはあらかじめ向きを入れて図中の☆を基点とします。この基点から向きに沿って結び目を一周する際に初めて通過する交差点で交差状況（上道（+）か下道（-）か）、二度目に通過するときに交差点符号（+か-）を交差点番号に付与します。ここでは $[-1, 3, -2, -1, -3, -2]$ が拡張ガウス・コードとして得られます。

2. 結び目の基本的な情報の収集:

交差点集合 S_c は $\{1, 2, 3\}$ 、交差点リスト L_c は $[1, 2, 3]$ 、平坦化ガウス・コード L_f が $[-1, 3, -2, -1, -3, -2]$ になります。

3. 交差点の符号と正規のガウス・コードの計算;

平坦化ガウス・コードをひっくり返して $[-2, -3, -1, -2, 3, -1]$, このリストで最初に現れた交差点番号に関連する数の符号が交差点の符号になることから符号リスト $L_s = [-1, -1, -1]$ を得ます。つぎに平坦化ガウス・コードの最初に現れた交差点番号の符号に -1 をかけた数で二度目に現れた数を置き換えると正規ガウス・コード $G_c = [[-1, 3, -2, 1, -3, 2]]$ が得られます。

4. 交差点の復元:

交差点番号 1 については成分リストで -1 を探し, 見つけた 1 から順方向で最初の負の数を探します。ここで -2 が該当する数であることから, 交差点番号 1 から出る下道は a_2 , 1 を成分リストで探し, 1 の順方向で最初に出る負の数 -3 があるため交差点番号 1 を通過する上道は a_3 . 以上をリストとして表現すると $[1, 2, 3, -1]$ が交差点番号 1 の関係子の情報になります。交差点番号 2 では -2 の順方向にある最も近い負の数が -3 のために交差点 2 を出る下道は a_3 , 2 の順方向で最も近い負の数は, 成分リストを円環で考えると -1 になるため上道は a_1 . これらをリストで表現すると $[2, 3, 1, -1]$ になります。絡み目/結び目の基本群は交差点の総数を n とするとときに交差点の関係子は $n - 1$ 個あれば十分で, 三葉結び目の場合は交差点数が 3 のため, ここで求めた 2 つで十分です。

ここで求めた関係子の情報と関係子の構築図から, この三葉結び目の群の表示として

$$\langle a_1, a_2, a_3 \mid a_3 a_1 a_3^{-1} a_2^{-1}, a_2 a_3 a_2^{-1} a_1^{-1} \rangle$$

が得られます。

前述のように SageMath では 7.2 から結び目理論パッケージで拡張ガウス・コードが提供され, この拡張ガウス・コードで結び目や絡み目を定義すると, それらの描画まで行えます。しかし, 単純にパッケージを使うだけでは面白くないために上述の手続きを SageMath で表現してみましょう。なお, SageMath で行わせる処理は基本的に Python のリストの処理が中心です:

```
def checkPlusOnly(L):
    if True in map(lambda(x):x<0, L):
        return(False)
    else:
        return(True)

def checkMinusOnly(L):
    if True in map(lambda(x):x>0, L):
        return False
    else:
        return True
```

```

def flattenGaussCode(GaussCode):
    return(flatten(GaussCode))

def getCrossings(GaussCode):
    L_f = flatten(GaussCode)
    AL_f = map(lambda(x):abs(x), L_f)
    L_c = list(set(AL_f))
    return L_c

def getSignsOfCrossings(L_c, L_f):
    L_s = []
    RL_f = L_f[-1::-1]
    ARL_f = map(lambda(x):abs(x), RL_f)
    for i in L_c:
        n = RAL_f.index(i)
        L_s.append(sign(RL_f(n)))
    return L_s

def getGenerators(GaussCode):
    L_c = getCrossings(GaussCode)
    n = len(L_c)
    Generators = ""
    for i in L_c:
        if Generators=="":
            Generators = "a_" + str(i)
        else:
            Generators = Generators + ", a_" + str(i)
    return(Generators)

```

最初の `checkPlusOnly()` と `checkMinusOnly()` は与えられたリストの成分が全て正, あるいは負であれば `True`, それ以外で `Faulse` を返す函数で, 自明な成分の検出で用います。つぎの函数 `flattenGaussCode()` はガウス・コードを平坦化する函数ですが, 実はこのような函数として `SageMath` には函数 `flatten()` が用意されており, それを単に呼出すだけです^{*3}。函数 `getCrossings()` は与えられたガウス・コードから交差点リストを返す函数です。この函数は数値リストを集合型, つぎにリスト型への変換で交差点リストを生成しています。函数 `getSignsOfCrossings()` は交差点リスト L_c と平坦化ガウス・コード L_f を引数として交差点符号リスト L_s を返す函数です。平坦化ガウス・コードに二度目に現われる交差点番号に関わる数の符号が交差点の符号であることから, この函数では平坦化ガ

^{*3} Python と NumPy に函数 `flatten()` はありません。*Mathematica* に同様の函数があるためにその影響でしょうか。

ウス・コードを逆順にしたリスト RL_f とその絶対値のリスト ARL_f を生成し, メソッド `index()` を使って交差点番号 i の ARL_f での位置を検出, その位置情報から RL_f の符号を求めることで符号リスト L_s を生成しています. 関数 `getGenerators()` は拡張ガウス・コードから結び目群の生成元を文字列として返す函数です. ここでの処理は交差点と道が一対一に対応し, さらにそれらの道が結び目群の生成元になることを利用しています. なお生成元は ‘ a_i ’ の書式です.

つぎにガウス・コードから成分の分析や関係子を取り出す函数を示します:

```
def transGaussCode(ExGaussCode):
    G_c = []
    n_s = map(len, ExGaussCode)
    L_c = getCrossing(ExGaussCode)
    L_f = flatten(ExGaussCode)
    AL_f = map(lambda(x):abs(x), L_f)
    RAL_f = AL_f[-1::-1]
    n_s = map(len, ExGaussCode)
    for i in L_c:
        n_i = AL_f.index(i)
        rn_i = RAL_f.index(i)
        L_f[-1-rn_i] = -sign(L_f[n_i])*i
    for j in n_s:
        L_t = []
        for k in range(0, j):
            L_t.append(L_f[0])
            L_f.pop(0)
        G_c.append(L_t)
    return [G_c, L_c, L_s]

def moveTrivialComponent(G_c, L_c, L_s):
    nG_c = []
    L_t = []
    for C in G_c:
        if checkPlusOnly(C):
            L_t.append(C)
        if checkMinusOnly(C):
            L_t.append(map(lambda(x):abs(x),C))
    T_f = list(set(flatten(L_t)))
    for C in G_c:
        for i in T_f:
            j = -i
            if i in C: C.remove(i)
            if j in C: C.remove(j)
```

```

L_s.remove(L_s[L_c.index(i)])
L_c.remove(i)
nG_c.append(C)
return [nG_c, L_c, L_s]

def relatorsOfCrossings(G_c, L_c, L_s):
    L_r = []
    L_f = flatten(G_c)
    L_c = L_c.pop(-1)
    n_c = len(L_c)
    for h in range(0, n_c):
        i = L_c[h]
        L_t = [i]
        for C in G_C:
            if -i in C: A = C
            if i in C: B = C
        p_i = A.index(-i)
        for j in A[p_i+1:]+A[0:p_i+1]:
            if j<0:
                L_t.append(abs(j))
                break
        q_i = B.index(i)
        for k in B[q_i+1:]+B[0:q_i]:
            if k<0:
                L_t.append(abs(k))
                break
        L_r.append(L_t.append(L_s[h]))
    return L_r

```

函数 `transGaussCode()` は拡張ガウス・コードから正規のガウス・コード G_c , 交差点リスト L_c と対応する交差点符号リスト L_s を出力します。函数 `moveTrivialComponent()` は自明な成分をライデマイスター移動の II に対応する絡み目成分の変形で交点を解消する操作ですが、ここでの考え方はガウス・コードが全て正、あるいは負の成分を検出し、そこに含まれる交差点を除去する限定された状況下での交差の解消です。函数 `relatorOfCrossings()` は正規のガウス・コード G_c , 交差点リスト L_c と交差点符号リスト L_s から交差点の符号 L_s を引数とし、交差点リスト L_c に対応する関係子を構築する上で必要な交差点に向かう下道、交差点から出る下道、交差点を通過する上道と交差点の符号の 4 成分で構成されるリストを成分とするリスト L_r を返します。

最後に群の生成に関係する函数を示します:

```
def representationLinkGroup(Generators, L_c, L_r):
```

```

Relators = []
F = FreeGroup(Generators)
for r in L_r:
    [i1, j1, k1, sgn] = r
    [i, j, k] = map(lambda(x):L_c.index(x)+1,[i1,j1,k1])
    if sgn>0:
        Relators.append(F([k,i,-k,-j]))
    else:
        Relators.append(F([i,k,-j,-k]))
return (F/Relators)

def LinkGroup(ExGaussCode):
    G_c = []
    L_c = []
    L_s = []
    L_r = []
    Relators = []
    [G_c, L_c, L_s] = transGaussCode(ExGaussCode)
    [G_c, L_c, L_s] = removeTrivialComponent(G_c, L_c, L_s)
    Generators = getGenerators(G_c)
    F = FreeGroup(Generators)
    L_r = relatorOfCrossings(G_c, L_c, L_s)
    G = representationLinkGroup(Generators, L_c, L_r)
    return [G_c, L_c, L_s, G]

```

関係子の計算は函数 representationLinkGroup() で処理しますが、一般的に関係子は通常の多項式環ではなく自由群上で定義しなければなりません。SageMath では特に環の指定が行われない場合、可換な多項式環が用いられ、変数や項の並び換えや簡易化が行われます、その結果、関係子が勝手に簡約化され図形的な意味を壊されてしまいます。このことを防ぐために自由群 F とその商群を定義し、その中で関係子を設定する必要があります。ここでは自由群 F を内部で定義し、その F の元として関係子をリストの形で定めています。なお、メソッド FreeGroup() ではその生成元の指定は Python 流儀の 0 からではなく 1 から開始します。そのため交差点のリストを利用して関係子を表現するリストの成分との対照を行い、それを基に関係子を定義します。そして、絡み目軍は自由群の帰結群による商群として与えられます。最後の函数 calcLinkGroup() はこれらの函数をまとめて拡張ガウス・コードからガウス・コード、交差点番号と交差点の符号、そして、絡み目群を返却する函数です。

実際の計算例を以下に示します：

```
sage: EGC_Trefoil = [[-1, 3, -2, -1, -3, -2]]
```

```
sage: KG_Trefoil = LinkGroup(EGC_Trefoil)
sage: KG_Trefoil
([[-1, 3, -2, 1, -3, 2]],
 [[1, -1], [3, -1], [2, -1]],
 Finitely presented group < a_1, a_2, a_3 | a_3*a_1*a_3^-1*a_2^-1,
 a_2*a_3*a_2^-1*a_1^-1 >)
sage: EGC_FigureEight = [-1, 3, -2, 4, 3, 1, 4, 2]
sage: KG_FigureEight = KnotGroup(EGC_FigureEight)
sage: KG_FigureEight
([[-1, 3, -2, 4, -3, 1, -4, 2]],
 [[3, 1], [1, 1], [4, 1], [2, 1]],
 Finitely presented group < a_1, a_2, a_3, a_4 | a_2*a_3*a_2^-1*a_4^-1,
 a_4*a_1*a_4^-1*a_2^-1, a_3*a_4*a_3^-1*a_1^-1 >)
```

と、このように絡み目/結び目の拡張ガウス・コードで初期化を行い、通常のガウス・コード、各交差点での符号、絡み目群のリストを返却する函数ができました。

6.9 LinkClass クラス

拡張ガウス・コードを基に基本群を生成するプログラムを構築しましたが、この拡張ガウス・コードは絡み目の正則射影図の交差点数、交差点での符号等の数多くある正則射影図の属性の一つです。しかし、この拡張ガウス・コードだけさえあれば正則射影図の復元も可能なために、この拡張ガウス・コードが正則射影図を語る上で最も本質的な情報源であることが判ります。そこで、絡み目のクラス定義で、この拡張ガウス・コードをその中核に据えることにします。

絡み目のクラス LinkClass はどのようなものであるべきでしょうか？まず、このクラスは当然、拡張ガウス・コードを与えることで初期化されるものとすべきです。そして、このクラスでは拡張ガウス・コードから正規のガウス・コードや交差点の情報を含むべきです。そして、基本群もその基本的な属性として保持すべきでしょう。このことから、絡み目のクラスとして持つべき属性が定まります。つまり、必要とされる属性としては

- 拡張ガウス・コード
- ガウス・コード
- ザイフェルト系
- 交差点
- 交差点での符号
- 関係子
- 成分数

が挙げられるでしょう。ここでガウス・コードは拡張ガウス・コードから交差点の符号情報を除去し、交差点での道の上下関係のみの正規のガウス・コード、そしてザイフェルト系はザイフェルト円周とその橋の情報です。これらは絡み目の最も基本的な情報と言えるでしょう。また、その結び目が何であるかオブジェクトとして表示しなければならないときに、これらの情報の幾つかを表示すべきでしょう。これらのことから、まず、クラスの属性と初期化メソッド、公式の表示を定めるメソッドを次で定義します:

```
import sqlite3
import networkx as nx
import matplotlib.pyplot as plt
from sage.structure.element import RingElement

class LinkClass(RingElement):
    ExGaussCodes = []
    GaussCodes = []
    SeifertCircles = []
    Crossings = []
    Signs = []
    Generators = ''
    Relators = []
    Components = 0
    Group = ''
    SQLite3_DB = ''

    def __init__(self, ExGaussCodes=None):
        if ExGaussCodes is not None:
            self.ExGaussCodes = ExGaussCodes
            self.Components = len(ExGaussCodes)
            self.getCrossings()
            self.getSignsOfCrossings()
            self.renumberingLink()
            self.transGaussCodes()

    def __repr__(self):
        print self.ExGaussCodes
        print self.GaussCodes
        print self.Crossings
        print self.Signs
        return "OK"
```

ここで示した class 文は、クラス LinkClass の定義で、クラス LinkClass の最も基本的なメソッドを包含する部位です。まず最初の import 文でクラスが必要とするパッケージとモジュールの読み込みを行います。ここでの記述は通常の Python プログラムとの

違いはありません。import文で読み込むモジュールは SQLite3, NetworkX, matplotlib からは plot, sage.structure.element から RingElement です。ここで、SQLite3 は軽量RDBで絡み目不变量の計算で現れる中間的な正則射影図の保管に用います。この理由は不变量の計算で莫大な中間データが生成され、これらをリストや配列で保存することが妥当な処理と言えないためです。NetworkX はグラフ理論のためのパッケージですが、ここではザイフェルト系をグラフとして表現するために用い、このザイフェルト系の可視化で matplotlib の plot を用います。つぎに ‘class LinkClass(RingElement)’ はクラス RingElement を継承することを意味する class 節です。この LinkDiagram クラスの定義では SageMath で定義されたクラス RingElement を継承することで代数的な構造を手に入れるためです。なお、クラスの継承で特殊メソッドの上書きを行いますが、SageObject のサブクラスでは利用者の上書きのために別のメソッドが用意されていることに注意が必要です。たとえば、オブジェクトの公式の表示を定める特殊メソッドとして通常は特殊メソッド __repr__() の上書きを行いますが、SageObject では別にメソッド _repre_() が用意されており、こちらを用います。この他に代数環を表現する SageObject のサブクラス RingElement の継承では、その積演算 “*” と和演算 “+” についても通常の特殊メソッド __mul__() や __add__() を上書きするのではなく、別の用意されているメソッド _mul_() や _add_() の上書きを行います。このことは代数的構造の話でその詳細を述べることとし、拡張ガウス・コードを表現するリストの処理を行うメソッドについて述べることにしましょう。

6.9.1 初期化に関わるメソッド

ここではクラス LinkClass のガウス・コードの処理に係るメソッドについて述べます。なお、ここで述べるメソッドは絡み目群の計算を行う函数をメソッドに書き直したもので、なお、Python のメソッドの定義では第一引数に必ず “self” を記載しますが、実際の利用では引数 “self” は記載しません。また、属性を書き直すメソッドであれば return 文を利用する必要はありません：

```

def checkPlusOnly(self, C):
    if True in map(lambda(x):x<0, C):
        return(False)
    else:
        return(True)

def checkMinusOnly(self, C):
    if True in map(lambda(x):x>0, C):
        return False
    else:
        return True

```

```
def getCrossings(self):
    L_f = flatten(self.ExGaussCodes)
    AL_f = map(lambda(x):abs(x), L_f)
    self.Crossings = list(set(AL_f))

def getSignsOfCrossings(self):
    self.Signs = []
    L_f = flatten(self.ExGaussCodes)
    RL_f = L_f[-1::-1]
    ARL_f = map(lambda(x):abs(x), RL_f)
    for i in self.Crossings:
        n = ARL_f.index(i)
        self.Signs.append(sign(RL_f[n]))

def transGaussCodes(self):
    GaussCodes = []
    n_s = map(len, self.ExGaussCodes)
    self.getCrossings()
    Crossings = self.Crossings
    L_f = flatten(self.ExGaussCodes)
    self.getSignsOfCrossings()
    AL_f = map(lambda(x):abs(x), L_f)
    RAL_f = AL_f[-1::-1]
    n_s = map(len, self.ExGaussCodes)
    for i in Crossings:
        n_i = AL_f.index(i)
        rn_i = RAL_f.index(i)
        L_f[-1-rn_i] = -sign(L_f[n_i])*i
    for j in n_s:
        L_t = []
        for k in range(0, j):
            L_t.append(L_f[0])
            L_f.pop(0)
        GaussCodes.append(L_t)
    self.GaussCodes = GaussCodes

def moveTrivialComponents(self):
    nExGaussCodes = []
    nGaussCodes = []
    L_t = []
    for i in range(0, self.Components):
        C = self.GaussCodes[i]
        if C!=[]:
```

```

        if self.checkPlusOnly(C):
            L_t.append(map(lambda(x):abs(x),C))
        if self.checkMinusOnly(C):
            L_t.append(map(lambda(x):abs(x),C))
T_f = list(set(flatten(L_t)))
for k in range(0, self.Components):
    C = self.ExGaussCodes[k]
    D = self.GaussCodes[k]
    for i in T_f:
        j = -i
        if i in C: C.remove(i)
        if i in D: D.remove(i)
        if j in C: C.remove(j)
        if j in D: D.remove(j)
        if i in self.Crossings:
            self.Signs.remove(self.Signs[self.Crossings.index(i)])
            self.Crossings.remove(i)
    nExGaussCodes.append(C)
    nGaussCodes.append(D)
self.ExGaussCodes = nExGaussCodes
self.GaussCodes = nGaussCodes

def renumberingLink(self):
    G_c = []
    L_f = flatten(self.ExGaussCodes)
    RL_f = L_f[-1::-1]
    nL_f = copy(L_f)
    L_c = self.Crossings
    L_s = self.Signs
    L_n = map(len, self.ExGaussCodes)
    n = len(L_c)
    self.Crossings = [1..n]
    for k in self.Crossings:
        i = L_c[k-1]
        j = -i
        if i in L_f: nL_f[L_f.index(i)] = k
        if i in RL_f: nL_f[-RL_f.index(i)-1] = k
        if j in L_f: nL_f[L_f.index(j)] = -k
        if j in RL_f: nL_f[-RL_f.index(j)-1] = -k
    for n_i in L_n:
        C = []
        for j in range(0, n_i):
            C.append(nL_f[0])
            nL_f.pop(0)

```

```

        G_c.append(C)
        self.ExGaussCodes = G_c

    def getGenerators(self):
        Generators = ""
        for i in self.Crossings:
            if Generators=="":
                Generators = "a_" + str(i)
            else:
                Generators = Generators + ", a_" + str(i)
        self.Generators = Generators

    def getRelators(self):
        self.Relators = []
        L_f = flatten(self.GaussCodes)
        n_c = len(self.Crossings)
        for h in range(0, n_c-1):
            i = self.Crossings[h]
            L_t = [i]
            for C in self.GaussCodes:
                if -i in C: A = C
                if i in C: B = C
            p_i = A.index(-i)
            for j in A[p_i+1:]+A[0:p_i+1]:
                if j<0:
                    L_t.append(-j)
                    break
            q_i = B.index(i)
            for k in B[q_i+1:]+B[0:q_i]:
                if k<0:
                    L_t.append(-k)
                    break
            L_t.append(self.Signs[h])
            self.Relators.append(L_t)

    def getGroup(self):
        Relators = []
        F = FreeGroup(self.Generators)
        for r in self.Relators:
            [i1, j1, k1, sgn] = r
            [i, j, k] = map(lambda(x): self.Crossings.index(x)+1,[i1,j1,k1])
            if sgn>0:
                Relators.append(F([k,i,-k,-j]))
            else:

```

```
Relators.append(F([i,k,-j,-k]))
self.Group = F/Relators
```

ここで新たに加わったメソッドが `renumberingLink()` です。このメソッドを導入した理由は、クラス `LinkClass` に代数的構造を入れる際にガウス・コードの交差点番号の振り直しの必要性が生じるためです。この番号の振り直しでは絡み目の正則射影図の交差点の総数が n のときに交差点番号を $1, 2, \dots, n$ を割り当て直すもので、メソッド `moveTrivialComponents()` で交点の解消で歯抜けになったときの対処も含めています。もう一つ加わるメソッドがメソッド `numberShift()` で、メソッド `renumberingLink()` による番号の振り直しで「正規化」した絡み目 L_1 と L_2 に対し、連結和等の演算子“ \circ ”による演算 $L_1 \circ L_2$ を計算するとき、生成されるリストは集合としては二つの絡み目の交差点の和集合になるために番号の衝突を避けるために第二引数 L_2 の交差点番号を L_1 の交差点数が n_1 のときに n_1 だけ移動させるメソッドです。これらのメソッドはリストの番号の振り直しでしかありませんが、ガウス・コードの一致が絡み目の同一性をある程度、保証することを意図しています。

6.9.2 導入すべき代数的構造

この `LinkClass` に入る「**代数的構造**」のことを述べておきましょう。クラス `LinkClass` ではクラス `RingElement` を継承しようとしてます。これは絡み目の代数的な構造に注目した結果です。ここで新たに定義しようとするクラスに適切な「**代数的構造**」、すなわち代数的な「**枠組**」を与えるためには「**対象が何であるか**」、それらの対象間にある「**演算がどのようなもの**」であるかを適切に「**語ること**」をしなければなりません。さて、このクラスの対象は何でしょうか？ここで定義するクラスは絡み目を表現します。この絡み目は3次元空間内部の曲線をそのまま扱うのではなく、2次元平面へ射影し、二重点（交差点）の情報を付加した正則射影図です。そして前述のプログラムはその正則射影図から読み取れる拡張ガウス・コードを基にしてプログラムを記述しています。だから対象は拡張ガウス・コードであるべきです。これで対象が決まりました。さて、正則射影図には前述のように連結和と直和の二つの演算があります。どちらも「**和**」という言葉がありますが、まず、連結和はどのように表現すべきでしょうか？ここで絡み目の連結和は一つの成分に対して行われる処理のために結び目の連結和として捉えられます。ここで結び目の連結和 $K_1 \# K_2$ は双方の起点で処理を行うのであれば最初に K_1 を回って次に K_2 を回る形になるために単純にリストの結合として捉えられます。このときに単位元は空リスト `[]` で表現できますが、自明な結び目は交差点をもたないことから、連結和の単位元を空リスト `[]` で表現することに問題が全くないことが分かります。また、絡み目については、どの成分で連結和を取るかということが問題になりますが、ここで絡み目 $L = \cup_{i=1}^m L_i$ を単なる円周の集合ではなく順序対 $\langle L_1, L_2, \dots, L_m \rangle$ として考え、連結和は双方の第一成分で行う演算と

して定義します。なお、結び目の連結和は可換であるため、この絡み目の連結和も可換であることが判ります。そして、連結和の単位元は自明な結び目で、自明な結び目のガウス・コードは空リストのリスト ‘[[]]’ で表現され、この連結和が代数的構造の和を定めることができます。さて、絡み目のもう一つの演算である直和を

$$\langle L_1^1, L_2^1, \dots, L_m^1 \rangle \oplus \langle L_1^2, L_2^2, \dots, L_n^2 \rangle = \langle L_1^1, \dots, L_m^1, L_1^2, \dots, L_n^2 \rangle$$

で定めましょう。このように直和は集合としては和集合ですが、絡み目は成分に順序が入っているために直和には可換性がありません。そして、この直和の単位元として空集合 \emptyset があり、Python では ‘{}’ が対応するでしょう。さて、このように絡み目の集合は単位元を有する二つの演算を持ちますが、これに類似したものに何があるでしょうか？演算が二つあるものに自然数 \mathbb{N} があります。実際、自然数 \mathbb{N} には式 ‘ $1 + 1$ ’ にある和 “+”，式 ‘ 2×3 ’ にある積 “ \times ” の二つの演算がありますが全て可換です。では、2 次の正方行列の集合 $M(2)$ はどうでしょうか？こちらは和 “+” は可換ですが 積 “.” は可換ではありません。そして、2 次の正方行列の集合は環と呼ばれる代数的構造を持ちますが、この 2 次の正方行列がモデルになりそうです。そこで絡み目の連結和は和演算 “+”，直和は可換性がないために積演算 “*” として表現しましょう。そうなると代数的構造として和と積の二つの演算を持つ環が妥当と判断できるでしょう^{*4}。これが抽象基底クラス RingElement を継承することにした理由です。これは演算それ自体の定義ではなく、その演算が持つ性質を利用するときに威力を発揮します。

ここでクラス RingElement を継承することで環として必要な性質を継承するだけに必要なメソッドや属性が揃うだけで、具体的なことはそれだけではまだ何も決っていません。上位のクラスのメソッドや属性を利用者がきちんと継承する側に合せて再定義しなければなりません。それではこれらの演算をどのように入れればよいでしょうか？これが数式処理 Maxima であれば自由自在に連結和 “#” や直和 “⊕” を定義することができますが、このように万事が気軽に見えることは継承すべきクラスというものがないためです。ところが、Python では和 “+” や積 “*” 演算は特殊メソッドと呼ばれるメソッドで表現されており、それらの上書きで自分が定義する対象の演算が定義できるようになっています。具体的には和演算 “+” の定義は特殊メソッド `__add__()` で、積演算 “*” の定義は特殊メソッド `__mul__()` の上書きを行うと和や積を自分のクラスに適合するように定義し直すことができます。絡み目の連結和と直和の表現も、これらの特殊メソッドを利用すれば良さそうですが、ここで注意が必要になります。まず、演算の表現は要するにリストの処理に過ぎないために、その記載に問題はなさそうです。そして、連結和の可換性ですが、二つの演算結果を比較することを現時点では考えておらず、それで実用上の問題がないために

^{*4} より正確には絡み目の集合は半環 (semiring) の構造を入れることができます。

不問にしましょう。実用上の問題になるのは結合律です。まず、この結合律が充たされれば $a \# b \# c$ のように二項以上の計算式の記載が可能になりますが、Python の特殊メソッド `__add__()` や `__mul__()` を上書きしても、これらの二項演算子は結合律を充しません。そのために $a + b + c$ や $a * b * c$ のような式はエラーになります。なぜなら、これらの式の演算子は左右二つの演算子に対する演算子であり、 $a + (b + c) = (a + b) + c$ や $a * (b * c) = (a * b) * c$ といった結合律を前提にしているためです。だから結合律を入れなければ、このクラスのためだけに自力で結合律を構築するか、既存の SageMath のクラスを流用するかを選択しなければなりません。ところで、自力で結合律を構築するのであれば「**車輪の再発明**」を行う妥当性が問われますが、ここでは単に結合律を入れたいだけで、SageMath にあるものを利用しない理由になりません。そこで SageMath のどの代数的構造を入れるかという問題になりますが、ここでは演算が二つで一方が可換、もう一方が非可換であり、各演算に対して半群になることから、環を表現する抽象基底クラス `RingElement` で十分と判断できます。この抽象基底クラス `RingElement` を継承するために `import` 文で `sage.structure.element` から `RingElement` モジュールをあらかじめ読み込んでいる理由です。では、SageMath の代数的構造を与える抽象基底クラスを継承してしまえば、和や積の特殊メソッドを上書きしてしまえば良いでしょうか？SageMath の抽象基底クラスには書き換えるべきメソッドがちゃんと用意されています。ただ、その書き換えを行うメソッドは Python の特殊メソッドの名前とは文字“`_`”が一つ少なくなっています。

6.9.3 書換えるべきメソッドについて

これでクラス `LinkClass` の骨子が定まりました。では、与えられた 絡み目の拡張ガウス・コードをこのクラスのインスタンスとするにはどうすればよいでしょうか？このクラスのインスタンス化で拡張ガウス・コードを引数として指示すれば、その拡張ガウス・コードに対応するインスタンスを生成するようにすると良いでしょう。そのため `LinkDiagram` クラスは正則射影図に対応するインスタンス生成で、その正則射影図の拡張ガウス・コードを成分とするリストを一つ取ることにします。ここで拡張ガウス・コードは上述の方法で構築した整数のリストで、結び目のときは各成分の拡張ガウス・コードから構成されることになります。そこでインスタンス化の際の引数としては拡張ガウス・コードのリストを与えるようにすれば結び目の場合でも問題ありません。通常、Python のインスタンスの初期化では特殊メソッド `__init__()` が用いられますが、結び目の半群としての構造を入れたいがためにクラス `RingElement` を継承することにしています。このクラスには利用者が書換えるべき特殊メソッドの代用として文字 `_` が一つだけのメソッドが準備されており、インスタンスの初期化では `_init_()` を用います。このメソッド `_init_()` の記載内容はメソッド `__init__` を用いるときと記述に違いはありません。今

回は拡張ガウス・コードのリストを一つだけ取るために `self` 以外の明示的な引数として拡張ガウス・コードのリスト `ExGaussCodes` を記載しておきます。ところで自明な結び目も一々 ‘`[]`’ で与えて初期化するのもどうでしょう？それよりも引数が与えられなければ自明な結び目のデータを生成するようにしましょう。このことはメソッド `_init_()` の引数 `ExGaussCodes` に既定値として `None` を設定することを ‘`ExGaussCodes=None`’ と記載することで行えます。それからメソッド内部で変数 `ExGaussCodes` の値で自明な結び目か、まともな処理を必要とするかに分岐するようにすれば良いのです。そして、このクラスの初期化で与えられた拡張ガウス・コードのリストから交差点の上下関係のみの正規カウス・コードと符号付き交差点のリスト、絡み目の成分数、ザイフェルト円周のリストを生成するようにしましょう。このように変数 `GaussCodes`, `Crossings`, `Components` にはインスタンス化の時点で計算した値がそれぞれ割り当てられます。ただし、変数 `SQL` は多項式不变量の処理で SQLite3 の RDB の情報を載せるために準備したもののためにメソッド `_init_()` では指示しません。

それからインスタンスの情報を得るために一々、メソッドに訴えるよりもインスタンスの名前を指示したときに分かり易い書式で表示させたいものです。このようなことを実現させる Python の特殊メソッドが `__repr__()` です。ただし、SageMath の `sage.structure.element` のクラスを継承するときは Python の特殊メソッドではなく、あらかじめ準備されたメソッドの上書きを行うことがあります。ここでは環のクラス `RingElement` を用いますが、この場合はメソッド `_repr_()` の上書きで対処します。ここで表示させる内容はガウス・コードとその交差点のリスト、成分数とザイフェルト円周のリストを表示させることにしますが、単純に指定した書式の文字列を `return` 文で返せばよいのです。

次に具体的な演算を定義しなければなりません。実際、代数的な構造を入れたとはいえ、具体的な演算は何一つ決まっていないからです。ここで定義すべき演算は連結和と直和です。まず連結和に関しては代数的構造について考察したように、ここでの処理は与えられた拡張ガウス・コードのリストの先頭の成分に対して行うものとします。この演算については前述のようにリストの結合を基とすればよいことが分かっていますが単純な結合ではありません。図 6.3 に三葉結び目と八の字結び目の連結和の例を示していますが、連結和は直感的には左右に結び目を並べて双方の自明な紐の箇所を外して結びつける操作で、それを忠実に表現しようとすると和 “+” 右辺の被演算子の結び目に関して交差点番号の付け直しが必要になります。そこで番号の振替には内部的な函数として函数 `numberShift()` を構築し、左右の被演算子に対して番号の付け替えを行った上で拡張ガウス・コードを構築するようにします。ここで和演算の実装は通常は二項演算子 “+” を定義するメソッド `__add__()` の上書きで実現できます。ただし、このメソッドは単純に二項演算子を

定めるだけで、結合律までも定めるものではありません。ここでは RingElement クラスを継承するようにしたためにメソッド `_add_()` に演算を定義します。ただし、メソッド `_add_()` で記載する内容はメソッド `__add__()` に記載するものと違いはありません。このメソッドには二つの被演算子に対応する引数が必要になります。被演算子の一つはメソッドの定義で必要な `self`、もう一つは同一クラスの別インスタンスに対応する `other` で、`self` と `other` の `ExGaussCodes` の第一成分に対してリストの和 “+” を行い、それで得られた拡張ガウス・コードを使って新たなインスタンスを返すようにしています。これらのことば直和についても同様です。この直和はその非可換性から積演算を用いることにします。ここで積演算は特殊メソッド `__mul__()` の上書きで一般的には行いますが、`RingElement` クラスではあらかじめメソッド `_mul_()` が用意されているのでそちらを用います。記述内容はメソッド `__mul__()` と同様です。このように `RingElement` クラスのメソッドを上書きすることで効なく結合律を充たす演算にすることができます：

```

def _add_(self, other):
    m = len(self.Crossings)
    n = len(other.Crossings)
    ExGaussCodes = []
    if self.ExGaussCodes[0]==[]:
        ExGaussCodes = other.numberShift(m)
        for i in self.ExGaussCodes[1:]:
            ExGaussCodes.append(i)
        return(LinkClass(ExGaussCodes))
    else:
        if other.ExGaussCodes[0]==[]:
            ExGaussCodes = other.numberShift(n)
            for i in other.ExGaussCodes[1:]:
                ExGaussCodes.append(i)
            return(LinkClass(ExGaussCodes))
        else:
            fa = map(abs, flatten(self.ExGaussCodes))
            bs = max(fa) - min(fa) + 2
            "1st link starts from 1."
            a = self.numberShift(1)
            "2nd link starts bs."
            b = other.numberShift(bs)
            tb = b[0][-1]
            rb = b[0][0:-1]
            ab = abs(tb)
            arb = map(abs, rb)
            if ab in arb:
                n0 = arb.index(ab)
                cb = rb[n0]

```

```

rb[n0] = tb
tb = -cb
ExGaussCodes.append(a[0] + [tb] + rb)
for i in a[1:]:
    ExGaussCodes.append(i)
for i in b[1:]:
    ExGaussCodes.append(i)
return(LinkClass(ExGaussCodes))

def __mul__(self, other):
    ExGaussCodes = []
    if len(self.ExGaussCodes)==0:
        ExGaussCodes.append([])
        b = other.numberShift(1)
        for i in b:
            ExGaussCodes.append(i)
    else:
        if len(other.ExGaussCodes)==0:
            b = self.numberShift(1)
            ExGaussCodes.append([])
        else:
            a = self.numberShift(1)
            fa = map(abs, flatten(a))
            bs = max(fa) - min(fa) + 2
            b = other.numberShift(bs)
            for i in a:
                ExGaussCodes.append(i)
            for j in b:
                ExGaussCodes.append(j)
    return(LinkClass(ExGaussCodes))

def numberShift(self, n=None):
    ExGaussCodes = []
    eG_c = self.ExGaussCodes
    m = min(map(abs, flatten(eG_c)))
    if n is not None and n>0:
        bs = m - n
    else:
        bs = m
    for C in eG_c:
        D = []
        for i in C:

```

```

        D.append(i - sign(i) * bs)
ExGaussCodes.append(D)
self.ExGaussCodes = ExGaussCodes

```

6.9.4 メソッドの記述について

このクラス LinkDiagram に成分の取り出しや削除、成分の鏡像を生成するメソッドも入れておきましょう：

```

def linkComponent(self, n=None):
    ExGaussCode = []
    if n is not None:
        k = 0
        if n < self.Components:
            xgssc = self.ExGaussCodes[n]
            axgssc = map(abs, xgssc)
            crssngs = list(set(axgssc))
            chk = map(lambda (x):axgssc.count(x)==2, axgssc)
            for i in chk:
                if i:
                    ExGaussCode.append(xgssc[k])
            k = k + 1
    return(Diagram([ExGaussCode]))


def delLinkComponent(self, n=None):
    dcrssng = []
    xgsscs = self.ExGaussCodes
    if n is not None:
        if n < self.Components:
            exgssc = xgsscs[n]
            xgsscs.pop(n)
            if len(exgssc)>0:
                aexgssc = map(abs, exgssc)
                chk = map(lambda (x):aexgssc.count(x)==1, aexgssc)
                k = 0
                for i in chk:
                    if i:
                        dcrssng.append(aexgssc[k])
                k = k + 1
            for i in dcrssng:
                for x in xgsscs:
                    if i in x:
                        x.remove(i)

```

```

        if -i in x:
            x.remove(-i)
    return(LinkDiagram(xgsscs))

def mirrorImage(self, n=None):
    ExGaussCodes = []
    if n is None or n<0 or n>self.Components:
        for x in self.ExGaussCodes:
            ExGaussCodes.append(map(lambda(i):-i, x))
    else:
        xgausscode = map(lambda(z):-z, self.ExGaussCodes[n])
        xcrssngs = map(lambda(x):abs(x), xgausscode)
        crssngs = list(set(xcrssngs))
        cnt = map(lambda(z): xcrssngs.count(z)==1, crssngs)
        w = range(0,len(cnt))
        if sum(cnt)>0:
            crps = list(set(map(lambda(z):cnt[z]*crssngs[z],w)))
            for x in self.ExGaussCodes[0:n]:
                ExGaussCodes.append(x)
                for i in crps:
                    if i in xgausscode:
                        n = xgausscode.index(i)
                        xgausscode[n] = -i
                    else:
                        if -i in xgausscode:
                            n = xgausscode.index(-i)
                            xgausscode[n] = i
            ExGaussCodes.append(xgausscode)
            for x in self.ExGaussCodes[n+1:]:
                ExGaussCodes.append(x)
                for i in crps:
                    if i in xgausscode:
                        n = xgausscode.index(i)
                        xgausscode[n] = -i
                    else:
                        if -i in xgausscode:
                            n = xgausscode.index(-i)
                            xgausscode[n] = i
            else:
                for x in self.ExGaussCodes[0:n-1]:
                    ExGaussCodes.append(x)
                ExGaussCodes.append(map(lambda(z):-z, xgausscode))
                for x in self.ExGaussCodes[n+1:]:
                    ExGaussCodes.append(x)

```

```
return(LinkDiagram(ExGaussCodes))
```

これらのメソッドは整数 n をオプションとするものです。整数 n が指定されなかったときの処理はそれぞれ異なります。まずメソッド `link_component()` は整数で指定した番号の絡み目の成分を取り出します。この番号は拡張ガウス・コードのリストの添字に対応します。整数 n が与えられないときと成分数を超過したときは自明な結び目に対応する空リスト `[]` を返します。つぎのメソッド `del_link_component()` は指定した番号の成分を絡み目から削除します。無指定のときは削除を一切行なわずに元と同じ拡張ガウス・コードを持つインスタンスを返却します。メソッド `mirror_image()` は指定した成分のみを鏡像にしたインスタンスを返し、整数が指示されていなければ全ての成分を鏡像にしたインスタンスを返します。

さて、次に拡張ガウス・コードから交差点の符号の情報の無い正規のガウス・コードと交差点の情報を生成するメソッドを構築しておきましょう：

```
def gaussCode(self, ExGaussCode):
    GaussCode = []
    crssngs = copy(self.Crossings)
    listCrossings = list(set(map(abs, ExGaussCode)))
    for x in ExGaussCode:
        i = abs(x)
        s = sign(x)
        if i in listCrossings:
            listCrossings.remove(i)
            if -i in self.Crossings:
                crssngs[crssngs.index(-i)] = s*i
                GaussCode.append(i)
            else:
                if i in self.Crossings:
                    crssngs[crssngs.index(i)] = s*i
                    GaussCode.append(-i)
                else:
                    crssngs.append(x)
                    GaussCode.append(x)
        else:
            if x in GaussCode:
                GaussCode.append(-x)
            else:
                GaussCode.append(x)
                if -i in crssngs:
                    crssngs[crssngs.index(-i)] = s*i
                else:
```

```

        if i in crssngs:
            crssngs[crssngs.index(i)] = s*i
    return ([GaussCode, crssngs])

```

このメソッド gauss_code() は拡張ガウス・コードをリストとして先頭から解釈し、初めて現れた交差点はそのままにし、二度目に現れた交差点で符号と番号を交差点のデータに追加し、ガウス・コードには前回現れたときの逆の符号を与えるという処理を行っていますが、この処理は結び目でも絡み目でも違ひはありません。

次にザイフェルト系の円周を求めるメソッドを定義します:

```

def seifert_circles(self):
    GaussCodes = self.GaussCodes
    Crossings = self.Crossings
    newCrossings = copy(Crossings)
    for i in Crossings:
        newGaussCodes = []
        newCrossings.remove(i)
        gcds = []
        gpr = []
        sgn = sign(i)
        ai = abs(i)
        mi = -ai
        for x in GaussCodes:
            if not(i in x or -i in x):
                gcds.append(x)
            else:
                gpr.append(x)
        if len(gpr)==1:
            GaussCode = gpr[0]
            p0 = GaussCode.index(mi)
            p1 = GaussCode.index(ai)
            if p0<p1:
                newGaussCodes.append(GaussCode[0:p0]+[i]+GaussCode[p1+1:])
                newGaussCodes.append(GaussCode[p0+1:p1]+[i])
            else:
                newGaussCodes.append([i]+GaussCode[p1+1:p0])
                newGaussCodes.append(GaussCode[p0+1:]+GaussCode[0:p1]+[i])
        else:
            if len(gpr)==2:
                if ai in gpr[0]:
                    ga = gpr[1]

```

```

        gb = gpr[0]
    else:
        ga = gpr[0]
        gb = gpr[1]
    p0 = ga.index(mi)
    p1 = gb.index(ai)
    px = [i]+gb[p1+1:]+gb[0:p1]+[i]+ga[p0+1:]+ga[0:p0]
    newGaussCodes.append(px)
    for j in gcds:
        newGaussCodes.append(j)
    GaussCodes = newGaussCodes
    return(GaussCodes)

```

このメソッド `seifert_circles()` は各交差点での解消処理を行います。のちの不変量の計算では上道と下道の繋ぎ替えで向きを変更する処理が入りますが、ここでの処理は向きの逆転ではなく単純な繋ぎ替え処理が中心です。そして出力はザイフェルト円周を構成する交差点番号に交差点の符号を付けたもののリストです。この番号は絡み目の各成分に与えた向きに従って並んだものです。

6.9.5 ザイフェルト系の可視化について

そしてザイフェルト系の可視化を行えるようにメソッドも定義しておきましょう。このメソッドではグラフの定義に NetworkX、グラフの描画で matplotlib から `plot` を利用します：

```

def draw_seifert_circles(self, file=None, layout=None):
    G = nx.MultiDiGraph()
    G.clear()
    scs = self.SeifertCircles
    nodes = []
    nodelist = []
    fsc = []
    edges = []
    a = ''
    b = ''
    asc = 97
    for x in scs:
        b = ''
        tmp = []
        for i in x:
            a = b
            ai = abs(i)
            fsc.append(ai)

```

```
b = chr(asc) + str(ai)
nodes.append(b)
tmp.append(b)
if len(a)>0:
    edges.append((a,b))
    G.add_edge(a,b,weight=0.5, sign=0)
a = chr(asc) + str(abs(x[0]))
nodelist.append(tmp)
edges.append((b,a))
G.add_edge(b,a,weight=0.5,sign=0)
asc = asc + 1
G.add_nodes_from(nodes)
for j in self.Crossings:
    sj = sign(j)
    aj = abs(j)
    p0 = fsc.index(aj)
    p1 = fsc[p0+1:].index(aj) + p0 + 1
    G.add_edge(nodes[p0],nodes[p1],weight= -1, sign=sj)
"The classifications of edges into 3 types with sign."
sc=[(u,v) for (u,v,d) in G.edges(data=True) if d['sign']==0]
pb=[(u,v) for (u,v,d) in G.edges(data=True) if d['sign']==1]
mb=[(u,v) for (u,v,d) in G.edges(data=True) if d['sign']==-1]
plt.clf()
if layout is None:
    pos = nx.shell_layout(G)
else:
    if layout=="circular":
        pos = nx.circular_layout(G)
    else:
        if layout=="spring":
            pos = nx.spring_layout(G)
        else:
            pos = nx.shell_layout(G)
for i in nodelist:
    nx.draw_networkx_nodes(G,pos ,nodelist=i ,node_size=200,
                           alpha=0.8,color='r')
    nx.draw_networkx_edges(G,pos ,edgelist=sc ,width=1.5)
    nx.draw_networkx_edges(G,pos ,edgelist=pb,style='dashed',
                           edge_color='b',width=1)
    nx.draw_networkx_edges(G,pos ,edgelist=mb,style='dotted',
                           edge_color='g',width=1)
    nx.draw_networkx_labels(G,pos ,font_size=8,
                           font_family='sans-serif')
plt.axis('off')
```

```

plt.show()
if file is not None:
    plt.savefig(file)
return(G)

```

このメソッドで行う可視化は絡み目そのものの可視化ではありませんが、ザイフェルト系は絡み目を境界とする曲面の設計図そのものになるので、その中心的な存在であるザイフェルト系の可視化によって絡み目の構造が把握できることになります。ただし、ザイフェルト系の可視化で真面目に各交差点の位置を決めて描くことは、絡み目の交差点数や成分数が増えれば非常に困難になることが予想されます。そこで、ここでは「車輪の再発明をしない」という SageMath の基本方針を尊重して SageMath に含まれているパッケージで解決します。ここで、ザイフェルト円周が基本的に孤立した円周であることから、ザイフェルト系を有向グラフとして定義して表示すればどうでしょうか？SageMath には都合の良いことにグラフ理論向けのパッケージ NetworkX を標準で含んでいるのでこれを使わない手はありません。

まず、ザイフェルト円周を有向グラフとして可視化するためには、交差点をグラフの節点 (node) として定義し、それから各交差点を繋ぐ道を辺 (edge) として定義しなければなりません。ここで注意することはザイフェルト円周の系では同じ交差点が二つの円周上に別々に現われるので、番号をそのまま節点にすると何を描いているのか不明になる恐れがあります。そこでザイフェルト系の円周単位で節点をグループに分けします。この分類はザイフェルト曲面の構成で貼られる円盤に直接対応しますが、たとえば円盤 ‘a’ の境界となるザイフェルト円周に属する節点の先頭に ‘a’ を配置すると節点の名前とその意味がより判り易くなります。そこで、函数 char() を使って整数から ASCII 文字を生成し、この文字を交差点番号を函数 str() で文字列として先頭に追加することで節点の名前を定めましょう。あとは節点のリストをまとめてグラフの節点として定義するためにメソッド add_nodes_from() を用います。ただ、これではでは節点を定義するだけです。そうではなく節点をザイフェルト円周単位で扱いたいので、nodelist に節点を円周単位のリストとして保存しておきます。一方のグラフの各辺は一括処理を行うよりもグループ単位で処理する方が効率的なので個別にメソッド add_edge() で定義します。この辺の定義では辺に重みを ‘weight’ で、種類を明記するために ‘sign’ を指定し、ザイフェルト円周上の辺、符号 +1 の交差点、符号 -1 の交差点が区分できるように設定しておきます。ここでの重みは可視化で節点配置を行う spiral_layout で意味があり、負の数であれば排斥、正の数であれば引力として系が安定するように節点の配置計算を行います。また sign は辺の色の設定で用います。さてザイフェルト円周を構成する節点と辺がこれで定義できたので今度は交差点自体も定義しておきましょう。この交差点はザイフェルト円周系では向きを持った線分として表現されています。そこでこの線分を重さを持った辺として表現します。つまり、

符号 ± 1 の交差点を表現する辺なら-1, ザイフェルト円周を構成する辺には 0.5 を設定しておきます。そして、ザイフェルト円周を構成する節点を節点リストとして纏めておきます。それから NetworkX ではグラフ可視化で節点の配置をレイアウトで指定することができます。ここで可能なレイアウトには circular_layout, shell_layout, spring_layout と special_layout の 4 通りあり、circular は全ての節点を円周上、shell は螺旋上に節点を並べるために重みはさほどの意味を持ちませんが、spring は辺の重みを利用して節点の配置を行います。ただし、spring では初期節点配置をランダムに配置して、位置エネルギーの計算を行う方法のために毎度、図式の配置が異なります。その結果、絡み目の節点の良い配置が得られたり、不可解な配置になったりと安定しない問題があります。この点は非常に悩ましいことです。そのためにこのメソッドでは spring, shell, circle を必要に応じて選択可能にし、無指定の場合は shell を選択するようにしています。また、グラフを構成する辺に関してもグループ分けを行いますが、ここでは sgn でグループ分けを行い、このグループ分けした辺に対してメソッド draw_network_edges() で辺の線の太さ、色、形状を指定しています。ここでは-1 の符号を持つ交差点を表現する辺を緑の点線、+1 の符号を持つ交差点を表現する辺を青の破線で表現し、ザイフェルト円周を構成する辺は黒の実線とします。それから NetworkX のグラフ可視化では matplotlib の plot モジュールを用います。このことが LinkDiagram クラスの冒頭で matplotlib から plot を import していた理由です。このメソッドを定義する際に注意することは plot のメソッド clf() で描画の初期化を行わないと古い描画が残って再度表示されてしまうことです。そのためにここではレイアウトの指定を行う前に plt.clf() で描画の初期化を行っています。

ここで実例を示しておきます。最初に半群としての性質を見ておきましょう：

```
sage: load("LinkDiagrams.py")
sage: K3_1 = LinkDiagram([-1,3,-2,1,3,2])
sage: K3_1
GaussCodes:[[-1, 3, -2, 1, -3, 2]]
Crossings:[1, 3, 2]
SeifertCircles:[[3, 2, 1], [1, 3, 2]]
Components: 1

sage: Duo_K3_1 = K3_1 + K3_1
sage: Trio_K3_1 = Tri_K3_1 = K3_1 + K3_1 + K3_1
sage: Duo_k3_1
GaussCodes:[[-1, 3, -2, 1, -3, 2, 5, -4, 6, -5, 4, -6]]
Crossings:[1, 3, 2, 5, 4, 6]
SeifertCircles:[[4, 6, 5], [1, 3, 2, 5, 4, 6], [1, 3, 2]]
Components: 1
```

```
sage: Trio_K3_1
GaussCodes:[[-1, 3, -2, 1, -3, 2, 5, -4, 6, -5, 4, -6, 8, -7, 9, -8, 7, -9]]
Crossings:[1, 3, 2, 5, 4, 6, 8, 7, 9]
SeifertCircles:[[7, 9, 8], [1, 3, 2, 5, 4, 6, 8, 7, 9], [4, 6, 5], [1, 3, 2]]
Components: 1
```

最初の load 文で LinkDiagram クラスの読み込みを行った後、結び目 K3_1 を定義し、その連結和の計算を行っています。この結び目 K3_1 は今迄、何度も出ている三葉結び目ですが、演算 “+” はこのように結合律を充します。これが Python の特殊メソッド `__add__()` の上書きであれば Trio_K3_1 の計算でエラーになります。その意味で代数構造が上手く導入できたと言えるでしょう。

次にザイフェルト系の描画に移りましょう。ここでのザイフェルト円周の例は図 6.12 に示す絡み目を使います。この絡み目は符号が +1 の三葉結び目と -1 の捻りが入った自明な結び目が絡んだものです。この絡み目のザイフェルト系を手書きで構築したものを図 6.13 に示しておきますが、このザイフェルト系は 4 個の円周で構成され、特に交差点番号 6 を持つ円周では交差点 6 を一つだけ持つ円周が現われます。この図では交差点を置換える帯は負の交差であれば緑の点線、正の交差であれば青の破線として表現しています。この例では三葉結び目側の交差点は全て +1 なので青の破線になりますが、捩れの入った自明な結び目側は交差点の符号が全て -1 であるために緑の点線で表現されることになります。つぎに LinkDiagram クラスでの処理を見てみましょう：

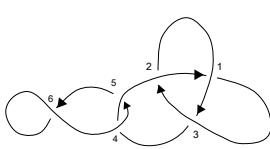


図 6.12 扱っている絡み目

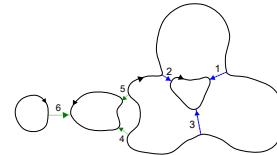


図 6.13 手書きのザイフェルト系

```
sage: d2 = LinkDiagram([[-1,3,-2,1,3,-4,5,2],[-4,-5,-6,-6]])
sage: d2
GaussCodes:[[-1, 3, -2, 1, -3, -4, 5, 2], [4, -5, -6, 6]]
Crossings:[1, 3, 2, -4, -5, 6]
SeifertCircles:[[-6, -4, -5], [-6], [-4, -5, 2, 1, 3], [3, 2, 1]]
Components: 2
sage: g2 = d2.draw_seifert_circles(file="d2_test.png", layout=spring)
```

LinkDiagram クラスでは絡み目の拡張ガウス・コードをその引数として与えると絡み目に対応する LinkDiagram クラスのインスタンスを生成します。ここでインスタンス名

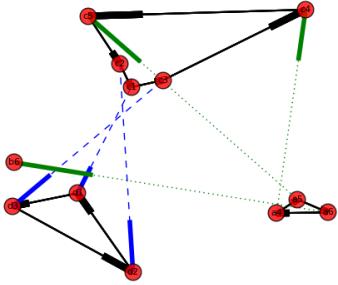


図 6.14 layout=spring による描画

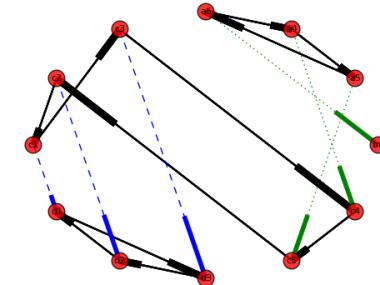


図 6.15 layout=circular による描画

を入力するとガウス・コードやザイフェルト円周等の情報が表示されますが、この機能は RingElement クラスのメソッド `_repr_()` の上書きで実現しています。ザイフェルト円周の可視化はメソッド `drawSeifertCircles()` で行います。ここでは引数にファイル名と節点のレイアウトを指定しているために画像ファイルの出力と節点のレイアウトを行います。ここで出力した画像を図 6.14 と図 6.15 に示します。ザイフェルト円周は黒で交差点を表現する線分よりも太く表示し、交差点は符号が +1 であれば青の破線、符号が -1 であれば緑の点線としています。それからザイフェルト円周はそれら円周のリストから順番に ‘a’, ‘b’, ‘c’, ‘d’ … と割り当て、各節点にはグループを表現するアルファベットを交差点番号の先頭に置いた名前にしています。レイアウトの指示は多少の試行錯誤が必要になるかもしれません。図 6.14 では `spring` を設定したためにザイフェルト円周はグループ単位で適度に散らばった形で表示されます。これが `circular` や `shell` であれば円周上に接点が配置されます。実際、図 6.15 ではレイアウトとして `circular` を選択しているので接点は全て单一の円周上に載ります。どのレイアウトがザイフェルト系の構造を分かり易く示しているかは実際に幾つか試して最適なものを選ぶことになります。ただし、節点が円周上に並ぶものであれば `circular`、ザイフェルト円周が共通の中心点を持つものであれば `shell`、それ以外の一般的なものは `spring` を試すと良いでしょう。

この LinkDiagram クラスでは拡張ガウス・コードを基本としているので、先程の基本群を計算する函数もこのクラスに追加することが容易です。さて、このように絡み目固有の群の構成や、ザイフェルト系の可視化ができるようになりました。そうすると絡み目の構造をザイフェルト系で観察し、それらの分類を群で把握したいものです。具体的には三葉結び目と八の字結び目の基本群を構築したので両者が同じ結び目なのかどうかを調べたいところですが、そのときに「**語の問題**」が生じます。

6.9.6 語の問題

結び目/絡み目の基本群や組紐群の表示は機械的に計算することが容易ですが、二つの結び目/絡み目の群が与えられたときに、それらか同じものかどうかの判別が難しいという側面があります。これは群論で「**語の問題 (word problem)**」と呼ばれるものです。ここで二つの群に対する同値性の検証で次に示す「**Tietze 変換**」と呼ばれる操作で移り合える群は同じものであることが知られています：

Tietze 変換

I	$\langle x_1, \dots, x_n \mid r_1, \dots, r_m \rangle$	\Rightarrow	$\langle x_1, \dots, x_n \mid r_1, \dots, r_m, u \rangle$
			$u \in R$
I'	$\langle x_1, \dots, x_n \mid r_1, \dots, r_m, u \rangle$	\Rightarrow	$\langle x_1, \dots, x_n \mid r_1, \dots, r_m \rangle$
			$u \in R$
II	$\langle x_1, \dots, x_n \mid r_1, \dots, r_m \rangle$	\Rightarrow	$\langle x_1, \dots, x_n, y \mid r_1, \dots, r_m, y\zeta^{-1} \rangle$
			$y \notin \{x_1, \dots, x_n\}, u \in F$
II'	$\langle x_1, \dots, x_n, y \mid r_1, \dots, r_m, y\zeta^{-1} \rangle$	\Rightarrow	$\langle x_1, \dots, x_n \mid r_1, \dots, r_m \rangle$
			$y \notin \{x_1, \dots, x_n\}, u \in F$

ここで群 F の表示を $\langle x_1, \dots, x_n \mid r_1, \dots, r_m \rangle$ とし、関係子 $\{r_1, \dots, r_m\}$ が自由群 $\langle x_1, \dots, x_n \rangle$ 内で生成する帰結群を R と表記しています。ここでは Fox の本 [23] の P.59 にある例題を使って、二つの群の表示 $\langle x, y, z \mid xyz(yzx)^{-1} \rangle$ と $\langle x, y, a \mid xa(ax)^{-1} \rangle$ が同じ群の表示であることを確認してみましょう：

Tietze 変換の例

$$\begin{aligned}
 \langle x, y, z \mid xyz(yzx)^{-1} \rangle &\xrightarrow{\text{II}} \langle x, y, z, a \mid xyz(yzx)^{-1}, a(yz)^{-1} \rangle \\
 &\xrightarrow{\text{I}} \langle x, y, z, a \mid xa(ax)^{-1}, a(yz)^{-1}, xyz(yzx)^{-1} \rangle \\
 &\xrightarrow{\text{I}'} \langle x, y, z, a \mid xa(ax)^{-1}, a(yz)^{-1} \rangle \\
 &\xrightarrow{\text{I}} \langle x, y, z, a \mid xa(ax)^{-1}, z(y^{-1}a)^{-1}, a(yz)^{-1} \rangle \\
 &\xrightarrow{\text{I}'} \langle x, y, z, a \mid xa(ax)^{-1}, z(y^{-1}a)^{-1} \rangle \\
 &\xrightarrow{\text{II}'} \langle x, y, a \mid xa(ax)^{-1} \rangle
 \end{aligned}$$

このように有限回の Tietze 変換の列によって二つの群の表示が同値であることが示せますが、この手法の難点は Tietze 変換を見つけなければ二つの群の同値性が主張できず、そのような Tietze 変換の列を見付けられないと Tietze 変換が構築できないことが別問題であるため、Tietze 変換が判らないことが別の群の表示であると結論付けられない点

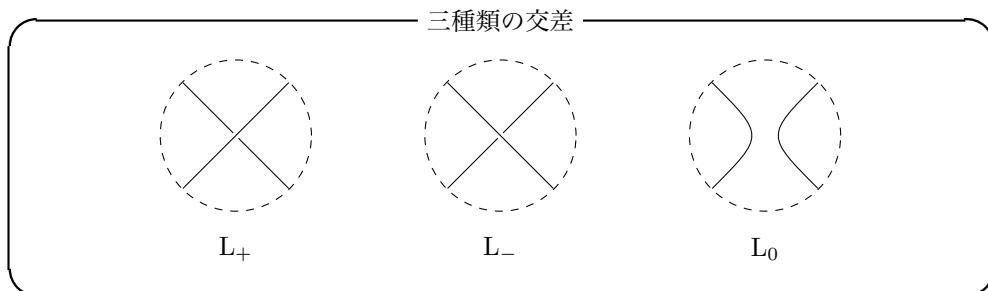
です。この点はライデマイスター移動と似ていて、いずれにせよ計算と比較が容易な不变量が望まれる理由になります。

6.10 多項式不变量

このように群の表示を使って結び目/絡み目の同値性を判断することは簡単なことではありません。そこで多少情報が落ちても使い勝手の良い値で結び目/絡み目の分類がどこまでできるかやってみることになります。そこで、結び目/絡み目からさまざまな多項式を生成し、これらの多項式を使って判別するということが行われています。これらの多項式の計算方法では補空間の基本群を直接使う代数的な方法、結び目/絡み目のザイフェルト曲面を貼って求める幾何学的な方法がありますが、正則射影図の局所的な交差点の状態を変更することで得られる正則射影図の多項式の間に成立する「スケイン関係式」と呼ばれる関係式から求める手法があります。なお、自明な結び目（正則射影図で交差点を持たないもの）の多項式不变量が 1 になるように正規化されます。

6.10.1 スケイン関係式

向き付けられた結び目/絡み目の正則射影図に対して「局所的な交差の変更」として次の三種類の交差を考えます：



この三種類の交差に基づく関係式を「スケイン関係式 (skein relation)」^{*5}と呼びます。この関係式から得られる多項式は z^{-3} 等の負の次数を許容するもので、より正確には「ローラン多項式 (Laurent polynomial)」と呼ばれる多項式です。そして、ライデマイスター移動で不变になる多項式を結び目/絡み目のスケイン多項式と呼びます。スケイン多項式としては以下の関係式を充たす多項式がその代表として挙げられます：

■ジョーンズ多項式 (Jones polynomial): $t^{-1}V_{L_+}(t) - tV_{L_-}(t) = (t^{\frac{1}{2}} - t^{-\frac{1}{2}})V_{L_0}(t)$

■コンウェイ多項式 (Conway polynomial): $\nabla_{L_+}(z) + \nabla_{L_-}(z) = z\nabla_{L_0}(z)$

^{*5} skein は縫糸の意味です。

■アレキサンダー多項式 (Alexander polynomial): $\Delta_{L_+}(t) + \Delta_{L_-}(t) = (t^{\frac{1}{2}} - t^{-\frac{1}{2}})\Delta_{L_0}(t)$

■ホンフリー多項式 (HOMFLY polynomial): $xP_{L_+}(x, t) - tP_{L_-}(x, t) = P_{L_0}(x, t)$

これらの多項式は自明な結び目なら 1 になる性質がありますが、1 に等しいときに自明な結び目になるかどうかは別問題です。また、コンウェイ多項式とアレキサンダー多項式は、変数については $z \leftrightarrow t^{\frac{1}{2}} - t^{-\frac{1}{2}}$ で互いに移りあえるという性質があります。またホンフリー多項式の由来は多項式を発見した人の名前の頭文字 (Hoste, Ocneanu, Morton, Freyd, Likorish, Yetter) を並べたものです。これらのスケイン多項式は絡み目 L_1, L_2 の連結和 $L_1 + L_2$ に対しては L_1 と L_2 のスケイン多項式の積になることが判ります。

なお、SageMath の BraidGroup パッケージで実装されている多項式はジョーンズ多項式とアレキサンダー多項式です。歴史的にはアレキサンダー多項式が古く、ジョーンズ多項式は 1980 年代に現われた強力な不变量です。

6.10.2 ジョーンズ多項式

ジョーンズ多項式は作用素環の研究から得られたもので、やがてカウフマンのブラケット多項式で表現され、それからコンウェイ多項式と同様のスケイン関係式からも求められるように定式化が行われました。ジョーンズ多項式は後述のアレキサンダー多項式よりも結び目の分類で強力な多項式で、2016 年の現時点でも非自明な結び目で多項式が 1 になるものが発見されていません。なお、SageMath の BraidGroup パッケージは、その関係からオプションの指定がなければカウフマンのブラケット多項式の流儀で表示され、オプションの指定があればスケイン形式から得られる多項式を出力します。

6.10.3 アレキサンダー多項式

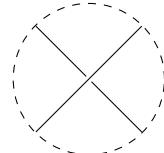
アレキサンダー多項式は古典的な結び目多項式です。スケイン多項式として認知されるまでは結び目群の表示から計算したり、ザイフェルト曲面を使って求めていました。この求め方からも判るように基本群と密接な関連を持っていますが、機械的に求められる手法が、結び目群の表示から求める方法で、結び目群の n 個の関係子をフォックス微分と呼ばれる特殊な「**微分**」を使って $n \times n + 1$ の大きさのフォックス微分によるヤコビ行列を計算し、このヤコビ行列から $n \times n$ の行列を取り出して、それらの行列式の最小公約数として(1 次の)アレキサンダー多項式が得られます。この多項式の幾何学的解釈は補空間をザイフェルト曲面で切り開いた空間の「**普遍被覆空間**」と呼ばれる空間の構成に密接に関係します。ただし、このアレキサンダー多項式は他の結び目多項式と比較して非力で、たとえばアレキサンダー多項式が 1 になる非自明な結び目として「**樹下・寺坂結び目**」が著名

です。

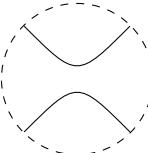
6.10.4 カウフマンのブラケット多項式

カウフマンのブラケット多項式は上述のスケイン関係式と別の交差関係を用います。なお、この多項式では結び目に向きを入れる必要はありません：

—— ブラケット多項式で用いる交差 ——



L

L₀L_∞

カウフマンのブラケット多項式は次の性質を充します：

—— カウフマンのブラケット多項式の性質 ——

1. $\langle O \rangle = 1$
2. $\langle L \sqcup O \rangle = (-A^2 - A^{-2})\langle L \rangle$
3. $\langle L \rangle = A\langle L_0 \rangle + A^{-1}\langle L_\infty \rangle$

カウフマンのブラケット多項式はライデマイスター移動の TYPE II と TYPE III に関して変化がありませんが、TYPE I の移動については $A^{\pm 3}$ だけ違いが生じます。実際に TYPE I の計算をしてみましょう：

$$\langle \text{○} \text{○} \rangle = A \langle \text{○} \text{○} \rangle + A^{-1} \langle \text{○} \text{○} \rangle = A(-A^2 - A^{-2}) \langle \text{○} \text{○} \rangle + A^{-1} \langle \text{○} \text{○} \rangle = -A^3 \langle \text{○} \text{○} \rangle$$

$$\langle \text{○} \text{○} \rangle = A \langle \text{○} \text{○} \rangle + A^{-1} \langle \text{○} \text{○} \rangle = A \langle \text{○} \text{○} \rangle + A^{-1}(-A^2 - A^{-2}) \langle \text{○} \text{○} \rangle = -A^{-3} \langle \text{○} \text{○} \rangle$$

ここで $\text{○} \text{○}$ の交差点の符号が 1, $\text{○} \text{○}$ の交差点の符号が -1 と幕の次数の符号が交差点の符号に対応していることが判ります。だから絡み目 L の正規射影 \mathcal{D}_L のカウフマンのブラケット多項式にあらかじめ捻れ $w(\mathcal{D}_L)$ を使って $(-A^{-3})^{-w(\mathcal{D}_L)}$ をかけてしまえばどうでしょう？すると TYPE I で変化がなくなり、他の TYPE II と TYPE III では総符号数に変化が生じないためにライマイスター移動で変化がなくなります。つまり、同値な正規射影図に対しては同じ量になるために絡み目 L の不变量になることが判ります。

以上から、次で多項式 $J(L)$ を定めます:

$$J(L) \stackrel{\text{Def.}}{=} (-A^{-3})^{-w(\mathcal{D}_L)} \langle \mathcal{D}_L \rangle$$

この多項式 $J(L)$ は変数 A の置換によってジョーンズ多項式 V_L に変換することができます。つまり次の関係式を充します:

$$J(L) = V_L(A^4)$$

6.11 カウフマンのブラケット多項式を計算するプログラム

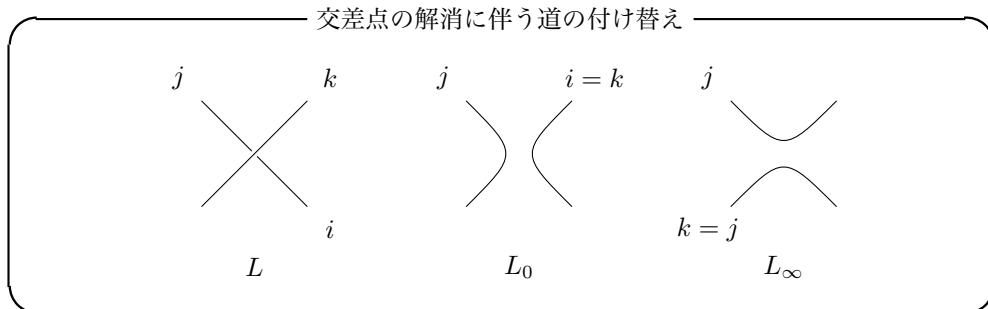
カウフマンの多項式の計算は非常に機械的に行えます。実際、 L から L_0, L_∞ への移行で交差が一つ解消しており、この交差一つの解消で二つの正則射影図が得られます。この操作を次の交差点で行うと 2×2 で 4 個の正則射影図が、さらに次の段階では 4×2 、すなわち 8 個の正則射影図が得られます。そして n 段目では 2^n 個の正則射影図が得られます。その一方で、各段で交差点が消去されるので交差点が n 個の正則射影図であれば最終的には n 段目で 2^n 個の互いに交差しない自明な結び目を成分とする絡み目の正則射影図が得られます。ここで自明な結び目を $-A^2 - A^{-2}$ で置き換える、その段に至るまでの経路から A と A^{-1} を乗じて総和を計算すればカウフマンのブラケット多項式が計算できることになります。この点はスケイン関係よりも非常に明瞭で機械的な処理になっているので、その意味では計算し易いものになっています。実際、スケイン関係で交差を入れ替えた K_- の交差が減るのかどうかは実際にそれを判別するプログラムを構築しない限りは機械的に判断はできません。だから処理を行うたびに全体がより簡易なものになる保証がなく、それを確認する手続きが別途必要になるためです。とはいえば隣り合った交差点の符号を合わせてしまえばライデマイスター移動の Type II によって二つの交差点が外せるために、このような戦略性を持った処理を行わなければなりません。ところがカウフマンのブラケット多項式の計算では交差点であればどこでも必ず交差点を減らすことができるのです。

では、カウフマンのブラケット多項式を計算するために必要なものは何でしょうか？ これはガウス・コードと各交差点での符号の情報だけで十分です。実際、 L から L_0, L_∞ の構築は実質的に指定した交差点番号をガウス・コードから削除し、その付近での道の付け替えで済むからです。そして前述の LinkDiagram クラスではザイフェルト系を求めるメソッドを構築しており、このメソッドも交差点の解消で道の向きを変更しない場合の処理に対応しているのです。だから残りを構築すれば交差点の解消によって生成される絡み目や結び目のガウス・コードが構築できることになります。そこで、この多項式を計算する

プログラムを LinkDiagram クラスのメソッドとして定義することにしましょう。まず最初に上道と下道の繋ぎ合わせで道の向きを変更することで交差点の符号が逆転するときのプログラムを構築しておきましょう。

そこでまず、正規のガウス・コードのリストと交差点の符号を保持したリストで構成されたリストを正則射影図を代表する対象として「Diagram」と呼ぶことにします。そして Diagram を成分とするリストを「Diagrams」と呼びます。ただ Diagrams はカウフマンのブラケット多項式の計算過程で生じるもので基本は Diagram です。また LinkDiagram クラスのインスタンスとの違いは、Diagram は LinkDiagram クラスの正規ガウス・コードと交差点リストで構成され、多項式の計算で大量に生成される中間的な絡み目を表現する中間的なデータであるということです。

次に正則射影図の交差を解消する函数を作成します。ここで重要なのは交差点の解消によって結び目が絡み目に変形される可能性があることです。この状況を次の図で説明しましょう：



この図では交差点番号 i の解消を示していますが、その実態は交差点番号 i における上道と下道の繋ぎかえです。ガウス・コードでは a_i の下道が右下から交差点 i に向かい、交差点 i を過ぎてから道 a_j になります。一方で上道 a_k は交差点 i の符号にしたがって +1 であれば左側から右側へ、-1 であれば逆の右側から左側へと交差点に向かいます。ここで道 a_i, a_j, a_k が同じ絡み目の成分であれば図の i の方向から入って j の方向に出ると必ず k の道を通りますが、交差点 j が絡み目の別成分との交差で生じるのであれば i から j を通過しても k を通過することはありません。

さて、 L_0 の交差点の解消では上道 a_k を交差点 i で二つに分割し、下道 a_i と上道 a_k の上側半分を繋ぎ、下道 a_j と上道 a_k の下半分を繋ぎます。このように上道と下道を繋ぎ合わせる操作であるために上道と下道が別成分であれば一つの成分に融合される処理であることが分かります。問題になるのはこれら上道と下道が同じ成分にある場合です。このとき交差点 i の符号で事情が異なります。まず交差点 i の符号が +1 であれば上道 a_k は左

下から右上に向かいます。そして右上からやがて下道 a_i を通って交差点 i に戻るので L_0 の処理では左右に成分が分離することが分かります。ところが L_∞ の処理では a_i を通つて上道 a_k の下半分を逆走するとやがて下道 a_j を逆走し、それから上道 a_k の上半分を通過して下道 a_i に戻るために成分の変化が生じません。ところが交差点 i の符号が -1 であれば上道 a_k は右上から左下に向かうために左下側と下道 a_i が繋り、 L_0 にて成分の変化は生じないものの L_∞ にて上下の成分の分割が生じます。このように成分の変化が交差点の符号に依存して生じることと、交差点の解消で上道を逆走するがあるために関連する交差点で符号の反転が生じます。

そこで、ここでは函数を交差点の状況に応じて二つに分けます。一つは交差点が同一成分の道による場合で、もう一つが別成分との交差によって生じる交差点の場合の処理です。上道と下道の融合が生じるために後者では成分が必ず一つ減りますが、前者は成分が変化しないものと一つ増えるものの二種類があります。ここでは前者の同一成分の道による交差点の解消を行う函数 crossing_change_a() を示します:

```
def crossing_change_a(connectedDiagram):
    [GaussCodes, Crossings] = connectedDiagram
    GaussCode = GaussCodes[0]
    n = Crossings[0]
    s = sign(n)
    i = abs(n)
    sa = Crossings[1:]
    sb = copy(sa)
    x = []
    p0 = GaussCode.index(-i)
    p1 = GaussCode.index(i)
    "splitting"
    if p1>p0:
        a1 = GaussCode[p0+1:p1]
        a2 = GaussCode[p1+1:] + GaussCode[:p0]
    else:
        a1 = GaussCode[p0+1:] + GaussCode[:p1]
        a2 = GaussCode[p1+1:p0]
    "fusion"
    x = a1[-1::-1]
    sb = toggle_crossings(sb, x)
    a0 = [a2 + x]
    pa = [[a1, a2], sa]
    ma = [a0, sb]
    if s==1:
        z = [pa, ma]
```

```

else:
    z = [ma, pa]
    return(z)

def toggle_crossings(Crossings, GaussCode):
    newCrossings = []
    list_crossings = list(map(abs, GaussCode))
    ordr = map(abs, Crossings)
    for n in Crossings:
        i = abs(n)
        m = n
        if i in list_crossings:
            if list_crossings.count(i) == 1:
                m = -n
        newCrossings.append(m)
    return(newCrossings)

```

ここで注意すべきことは削除する交差点の符号が $+1$ のときに L_0 で絡み目の成分が一つ増え、削除する交差点の符号が -1 のときに L_∞ で絡み目の成分が一つ増えることです。そして削除する交差点の符号が $+1$ のときに L_∞ の生成で上道側で削除する交差点の上から出てまた戻るまでの部分の各交差点で交差の符号が逆になります。これは削除する交差点の符号が -1 であれば L_0 で同様の上道側の処理が必要になります。この処理を行う函数が函数 `toggle_crossings()` です。この函数で交差点の符号の反転を行うのは他の成分との交差によって生じる交差点のみです。つまり上道と下道が同じ成分にあれば上道と下道の向きの反転が双方に生じるために符号の反転が生じません。しかし、上道と下道が別成分であればどちらか一方の道の向きのみが反転するために交差点の反転が生じることになります。

次に交差点の後に絡み目の二つの成分に交差点番号が配置されているときに交差点の解消を行う函数 `crossing_change_b()` を示します。この函数による処理では絡み目の成分変化はありませんが、一部の交差点で符号の反転が生じます:

```

def crossing_change_b(disjointDiagram):
    [GaussCodes, Crossings] = disjointDiagram
    z = []
    n = Crossings[0]
    s = sign(n)
    i = abs(n)
    sa = Crossings[1:]
    sb = copy(sa)
    if -i in GaussCodes[0] and i in GaussCodes[1]:

```

```

CodeA = GaussCodes[0]
CodeB = GaussCodes[1]
else:
    CodeA = GaussCodes[1]
    CodeB = GaussCodes[0]
p0 = CodeA.index(-i)
p1 = CodeB.index(i)
px = CodeB[p1+1:] + CodeB[:p1]
pa = [[CodeA[0:p0] + px + CodeA[p0+1::]], sa]
x = px[-1::-1]
sb = toggle_crossings(sb, x)
pb = [[CodeA[0:p0] + x + CodeA[p0+1::]], sb]
if s==1:
    z = [pa, pb]
else:
    z = [pb, pa]
return(z)

```

つぎにこれらを統括する交差の解消を行う函数 crossing_change() を示します。この函数では交差点がある一つの成分にあるかどうかは正則射影図を表現する対象 LinkDiagram の正規ガウス・コードに符号が異なった番号が含まれるかどうかで判別可能です。一つの正規ガウス・コードに正負の番号が含まれていれば函数 crossing_change_a() へ、正負のどちらか一方しか含まれないときは函数 crossing_change_b() で処理を行うようになります:

```

def crossing_change(Diagram):
    [GaussCodes, Crossings] = Diagram
    i = abs(Crossings[0])
    y = []
    cpl = []
    newGaussCodes = []
    Diagrams = []
    for x in GaussCodes:
        if not(-i in x or i in x):
            newGaussCodes.append(x)
        else:
            if -i in x and i in x:
                y = crossing_change_a([[x], Crossings])
            else:
                cpl.append(x)
    if len(cpl)>0:
        y = crossing_change_b([cpl, Crossings])
    Diagrams = [[y[0][0]+newGaussCodes, y[0][1]], [y[1][0]+newGaussCodes, y[1][1]]]

```

```
return(Diagrams)
```

これで交差の解消を行った函数ができました。この函数の引数は Diagram ですが、出力は L_0 と L_∞ に対応する Diagram を成分とするリストの Diagrams になります。これらのデータは LinkDiagram クラスそのものにはなりません。これらのデータはあくまでも計算の過程で生じた中間的な产物であるためです。

この交差点の解消は与えられた絡み目の交差点のリストに従って一斉に行うことになり、この操作によって生成される正則射影図を整理して上手く並べると、各正則射影図から二本の分枝が生じる樹形図として整理することができます。そして、函数 crossing_changes_a() と crossing_changes_b() が出力する Diagrams の第一成分が A を乗ずるものに、第二成分が A^{-1} を乗ずるものに対応します。これは樹形図で左側に L_0 、右側に L_∞ を置く表記に意図的に合わせたためです。さらに Diagram のリスト Diagrams の添字を二進数で表示すると 添字 0 の箇所で A を乗じ、添字 1 の箇所で A^{-1} を乗じることに対応していることが分かります。そして、自明な結び目では交差点が存在しないために拡張、正規ともに自明な結び目のガウス・コードは空リスト ‘[]’ が対応します。このことから最後に函数 crossing_change() で得られたリスト Diagrams は空リストのみで構成され、各 Diagram の添字を二進数表示したものが A と A^{-1} の積の情報で、各 Diagram のガウス・コードの空リストの総数から 1 を減じたものが $-A^2 - A^{-2}$ の幕の次数の情報になります。これらを利用して Diagrams からカウフマンのブラケット多項式の計算を行う函数 calc_Kauffman_Bracket() が次のように構築されます：

```
def kauffman_bracket(linkDiagram, LinkName=None, DB=None):
    var('A')
    Diagrams = [[linkDiagram.GaussCodes, linkDiagram.Crossings]]
    dtable = "diagrams"
    kbptable = "kauffman_bracket_table"
    As = []
    Fs = []
    KBIK = []
    Stage = 0
    if DB is not None and LinkName is not None:
        connect_db(DB, [dtable, "LinkName text", "Stage int", "Position text",
                        "GaussCodes text", "Crossings text"]])
        insert_diagrams2table(DB, dtable, LinkName, Stage, Diagrams)
    cps = linkDiagram.Crossings
    for i in cps:
        Stage = Stage + 1
        tmp = map(crossing_change, Diagrams)
        Diagrams = []
```

```

for j in tmp:
    Diagrams = Diagrams + j
Crossing = abs(i)
if DB is not None and LinkName is not None:
    insert_diagrams2table(DB, dtable, LinkName, Stage, Diagrams)
for i in Diagrams:
    j = len(i[0]) - 1
    KBTIK.append((-A**2-A**(-2))**j)
n = len(KBTIK)
m = len(Integer(n-1).digits(2))
Polynomial = 0
for i in range(0,n):
    As.append(sum(Integer(i).digits(2)))
for i in As:
    Fs.append(A**(m-2*i))
for i in range(0,n):
    Polynomial = Polynomial + KBTIK[i]*Fs[i]
Polynomial = expand(Polynomial)
if DB is not None and LinkName is not None:
    connect_db(DB, [[kbptable, "LinkName text", "CrossingNumber int",
                     "GaussCodes text", "Crossings text", "Polynomial text"]])
    insert_kauffman_bracket2table(DB, kbptable, LinkName, Diagrams, Polynomial)
return(Polynomial)

def kauffman_bracket_polynomial(LinkDiagram, KnotName=None, DB=None):
    var('A')
    Polynomial = kauffman_bracket(LinkDiagram, KnotName, DB)
    w = sum(map(sign, LinkDiagram.Crossings))
    return(expand((( -A)^3)^(-w)*Polynomial))

```

この処理では樹形図の情報が最終的な計算結果リストの位置と対応が、その位置情報を2進数表示したときの次数リストと一致していることを利用しています。この2進数への変換はメソッド`digits()`を用いますが、このメソッドはSageMathの`Integer`型のものであるのに対し、関数`range()`が生成するリストの成分はPythonの`int`型であるため、そのままでは利用できません。同一表示で型が異なっているのも厄介ですが、ここでは`Integer()`で型を`int`型から`Integer`型に変換して処理を行っています。SageMathでは整数の利用でこのようなことが生じ易いので注意が必要です。

計算例を以下に示しておきましょう：

```

sage: K3_1 = LinkDiagram([-1,3,-2,1,3,2])
sage: Duo_K3_1 = K3_1 + K3_1
sage: Trio = K3_1 + K3_1 + K3_1

```

```
sage: Ans = map(factor ,map(kauffman_bracket , [K3_1, Duo_K3_1, Trio_K3_1]))
sage: for i in Ans:
....:     print i
....:
-(A^12 + A^4 - 1)/A^7
(A^12 + A^4 - 1)^2/A^14
-(A^12 + A^4 - 1)^3/A^21
```

と、ガウス・コードと交差点符号情報だけで結び目/絡み目に関連する多項式が計算できます。この計算では演算子の結合律が用いられており、連結和によって多項式が連結和の成分の積になっていることが確認できます。

なお、この計算では交差点が n 個あれば実に $\sum_{i=0}^n 2^i = 2^{n+1} - 1$ 個の絡み目が現われます。交差点数が 3 個の三葉結び目でさえ 15 個の絡み目が現われるため、これらのデータをリストで保管することは賢明なことではありません。だから、ここでの函数も局所変数に一時的に蓄える以上のことはしていません。そこで SageMath に最初から包含されている RDBM の SQLite3 を利用しましょう。SQLite3 は軽量で、その利用のための前準備が不要の RDBM です。だからこそ、この計算処理で生じるデータを蓄えることに適しています。ここでは二つの表を使うことにしましょう。一つは絡み目の名前、絡み目の交差点数、計算した絡み目のガウス・コードと交差点の情報、それと計算したカウフマンのブラケット多項式を属性とする表、もう一つの表をロルフセンの結び目表の名前、消去した交差点番号、樹形図での絡み目の位置、絡み目のガウス・コードと交差点の情報を属性とする表とします。ここで前者の表が計算結果、後者の表は計算過程に関するものになります。では SQLite3 で最初に DB と上記の表をあらかじめ生成しておきましょう。ここで SQLite3 を Python から利用するためのモジュール sqlite3 を読み込んでおく必要がありますが、SageMath にはパッケージに包含されているので import 文で読込むだけです。

それからデータベースを開いて表を生成する函数 connect_db() を次で定義します:

```
def connect_db(DB, tables):
    if len(tables)>0:
        cursor = sqlite3.connect(DB)
        sql = "SELECT * FROM sqlite_master WHERE TYPE=='table'"
        csr1 = cursor.execute(sql)
        ans = flatten(csr1.fetchall())
        for i in tables:
            tname = unicode(i[0])
            if not(tname in ans) and len(i)>2:
                sql0 = "CREATE TABLE " + tname + " "
                sql1 = "(" + unicode(i[1])
```

```

        for k in i[2:]:
            sql1 = sql1 + "," + unicode(k)
            sql1 = sql1 + ")"
            cursor.execute(sql0 + sql1)
        cursor.commit()
        return(1)
    else:
        return(0)

```

この函数では指定された DB に接続し, SQLite3 の DB 単位に唯一作成される表 sqlite_master から表を検索し, リスト tables に同一名の表が無いか検索し, 表が存在しなければ表を生成する処理を行います. なお, cursor オブジェクト情報を一旦蓄え, メソッド commit() を実行することで初めて処理が確定されます. この commit を忘れていると cursor オブジェクトをメソッド close() で閉じた途端に情報が消えてしまうので注意が必要です.

```

def insert_diagrams2table(DBName, TableName, LinkName, Stage, Diagrams):
    cursor = sqlite3.connect(DBName)
    sql = "insert into " + TableName + " values (?, ?, ?, ?, ?, ?)"
    m = 0
    for i in Diagrams:
        Position = number2position(m, Stage)
        m = m + 1
        GaussCodes = str(i[0])
        Crossings = str(i[1])
        cursor.execute(sql, (LinkName, int(Stage), Position, GaussCodes, Crossings))
    cursor.commit()
    cursor.close()

def insert_kauffman_bracket2table(DBName, TableName, LinkName, Diagrams, Polynomial):
    cursor = sqlite3.connect(DBName)
    sql = "insert into " + TableName + " values (?, ?, ?, ?, ?, ?)"
    GaussCodes = str(Diagrams[0])
    Crossings = str(Diagrams[1])
    CrossingNumber = int(len(Diagrams[1]))
    KBP = str(Polynomial)
    cursor.execute(sql, (LinkName, CrossingNumber, GaussCodes, Crossings, KBP))
    cursor.commit()
    cursor.close()

```

まず, 位置を定める函数が number2position() です. この函数は引数として Diagrams の位置と交差点を消去する段階にそれぞれ対応する int 型の整数を取ります. この函数は位置情報として Diagrams での絡み目の情報の位置を 2 進数表記した文字列として返し,

そのときの表示桁数を交差点を消去する段階を表現する数値が対応します。たとえば三葉結び目の段階は‘0’で一つの交差点の消去を行うと段階は‘1’。このときの L_0 が‘0’, L_∞ が‘1’になり、次の段階‘2’では L_0 派生のものが‘00’, ‘01’, L_∞ 派生のものが‘01’と‘11’になるというあんばいです。この位置の情報が A と A^{-1} の積に対応します。実際, ‘0’が A , ‘1’が A^{-1} に対応します。この処理は関数 kauffman_bracket() で用いています。それから関数 insert_diagrams2table() で表 diagrams に計算過程で現われる絡み目の情報を登録し、関数 insert_kauffman_bracketTable() で結び目の名前、ガウス・コードと交差点で構成された正則射影図の情報とカウフマンのブラケット多項式を登録します。

ここで実際の計算例を示しますが、図示すると以下の射影図を続々と生成しているのです：

```
sage: K3_1 = LinkDiagram([-1,3,-2,1,3,2])
sage: kauffman_bracket(K3_1,DB="/Users/yokotahiroshi/MyKnotDB.db",LinkName="Trefoil")
-A^5 - 1/A^3 + 1/A^7

sage: conn=sqlite3.connect("/Users/yokotahiroshi/MyKnotDB.db")
sage: cur1 = conn.cursor()
sage: bf1 = cur1.execute("select * from diagrams")
sage: ans=bf1.fetchall()
....: for i in ans:
....:     print i
....:
(u'Trefoil', 0, u'', u'[-1, 3, -2, 1, -3, 2]', u'[1, 3, 2]')
(u'Trefoil', 1, u'0', u'[3, -2], [-3, 2]', u'[3, 2]')
(u'Trefoil', 1, u'1', u'[-3, 2, -2, 3]', u'[-3, -2]')
(u'Trefoil', 2, u'00', u'[-2, 2]', u'[2]')
(u'Trefoil', 2, u'01', u'[-2, 2]', u'[-2]')
(u'Trefoil', 2, u'10', u'[-2, 2]', u'[-2]')
(u'Trefoil', 2, u'11', u'[[2, -2], []]', u'[-2]')
(u'Trefoil', 3, u'000', u'[], []', u'[]')
(u'Trefoil', 3, u'001', u'[], []', u'[]')
(u'Trefoil', 3, u'010', u'[], []', u'[]')
(u'Trefoil', 3, u'011', u'[], []', u'[]')
(u'Trefoil', 3, u'100', u'[], []', u'[]')
(u'Trefoil', 3, u'101', u'[], []', u'[]')
(u'Trefoil', 3, u'110', u'[], []', u'[]')
(u'Trefoil', 3, u'111', u'[], []', u'[]')

sage: bf1 = cur1.execute("select * from diagrams where Stage==2")
sage: ans=bf1.fetchall()
sage: for i in ans:
```

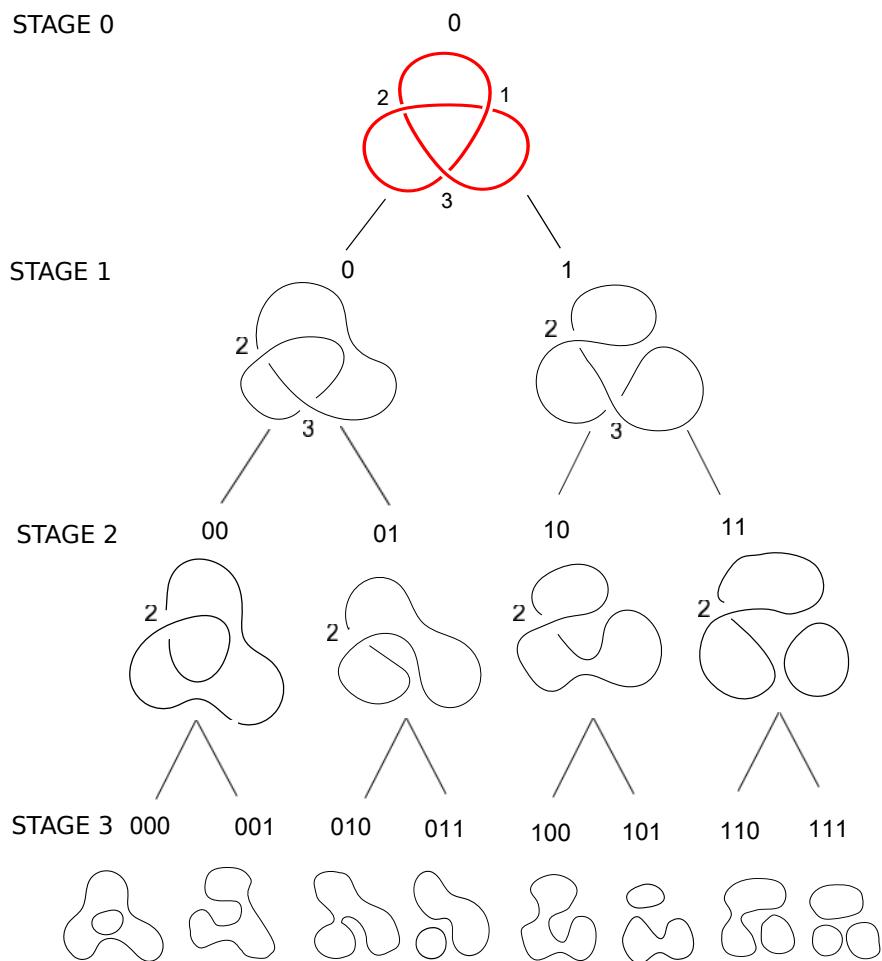


図 6.16 三葉結び目の処理

```
....:     print i
....:
(u'Trefoil', 2, u'00', u'[-2, 2]', u'[2]')
(u'Trefoil', 2, u'01', u'[-2, 2]', u'[-2]')
(u'Trefoil', 2, u'10', u'[-2, 2]', u'[-2]')
(u'Trefoil', 2, u'11', u'[[2, -2], []]', u'[-2]')
sage: bf2 = cur1.execute("select * from kauffman_bracket_table")
sage: ans2 = bf2.fetchall()
sage: for i in ans2:
....:     print i
(u'Trefoil', 2, u'[[[], []], []]', u'[[[], []]]', u'-A^5 - 1/A^3 + 1/A^7')
```

と、このようにデータの検索が行えるのです。ここでやっていることの詳細を解説すると、`kauffman_bracket()` では DB と LinkName にデータベースの情報と絡み目の名前を指定するとカウフマンのブラケットをデータベースに書き込みます。ここで `kauffman_bracket()` フィルでは書き込むテーブルは現時点では固定しています。このテーブルは分解の様子を書き込む `diagrams` テーブルと多項式を書き込む `kauffman_bracket_table` の二つです。`kauffman_bracket()` フィルではテーブルを初期化することではなく、繰り返し書き込むだけです。

あとがき

まだ本文も出来上がってはいませんが、最近思うことを幾つか。まず、ここ最近は個々で生み出されたものが何かと思わぬ形で統合された形で現われるということでしょうか。

Maxima 本を 2004-2006 にかけて執筆していたころが牧画的にさえ思えます。あのころは今と比べて Maxima と他の関連するアプリケーションについて述べればよかつたためです。ところが現在は、ビックデータやクラウド等、より大規模で統合的なものへと移り、解説も一つの言語、話題では済まなくなっています。

Modelica 言語を調べるにつけ、「ものづくり」と切株を守っている間に世界は派手に動いていたのだと実感する今日この頃です。

参考文献

- [1] アリストテレス, アリストテレス全集 1 カテゴリー論・命題論, 岩波書店, 2013.
- [2] アリストテレス, 形而上学(上下), 岩波文庫.
- [3] アリストテレス, (旧)アリストテレス全集 2, トピカ・詭弁論駁論, 岩波書店, 1987.
- [4] アリストテレス, (新)アリストテレス全集 2, 分析論前書・分析論後書, 岩波書店, 2014.
- [5] 飯田隆, 言語哲学大全 I 論理と言語, 効草書房, 1987.
- [6] 井筒俊彦, イスラーム思想史, 中公文庫, 中央公論社, 1991.
- [7] 今道友信, アリストテレス, 講談社学術文庫, 2004.
- [8] 上坂吉則, ニューロコンピューティングの数学的基礎, 近代科学社, 1993.
- [9] 大石進一, 精度保証付き数値計算, コロナ社, 2000.
- [10] 大畠明(著), 吉田勝久(監修), モデルベース開発のための複合物理領域モデリング-なぜ、奇妙なモデルが出来てしまうのか?- (MBD Lab Series), TechShare, 2012.
- [11] 桂田祐史, IEEE754 倍精度浮動小数点数のフォーマット
http://www.math.meiji.ac.jp/~mk/lab0/text/ieee_format/
- [12] 河内明夫編, 結び目理論, シュプリンガー・フェアラーク東京, 1990.
- [13] 後藤和茂, BLAS の概要 (http://jasp.ism.ac.jp/kinou2sg/contents/RTutorial_Goto1211.pdf), 2006.
- [14] 柴田有, グノーシスと古代宇宙論, 効草書房, 1982.
- [15] 清水哲郎, オッカムの言語哲学, 効草書房, 1990.
- [16] 清水義夫, 圏論による論理学 高階論理とトポス, 東京大学出版会, 2007.
- [17] 新開謙三, 疑微分作用素 - 偏微分方程式解法への応用-, 裳華房, 1994.
- [18] 杉晴夫, 神経とシナプスの科学 現代脳科学の源流, ブルーバックス, 講談社, 2015
- [19] 藤野登, 論理学 -伝統的形式論理学-, 内田老鶴園, 2003.
- [20] デーデキント著, 河野伊三郎訳, 数について 連続性と数の本質, 岩波文庫, 1996.
 Project Gutenberg による英訳 (Essays on the Theory of Numbers):
<http://www.gutenberg.org/etext/21016>
- [21] 田中尚夫, 選択公理と数学, 星雲社, 1987.

- [22] 聖トマス, 形而上学叙説, 岩波書店, 1935.
- [23] クロウエル, フォックス, 結び目理論入門, 現代数学全書, 岩波書店, 1989.
- [24] 平井有三, はじめてのパターン認識, 森北出版, 2012.
- [25] プラトン (著), 藤沢 令夫 (訳), 国家, 岩波文庫, 岩波書店, 1976.
- [26] フレーゲ, フレーゲ著作集 1 概念記法, 効草書房, 1999.
- [27] フレーゲ, フレーゲ著作集 3 算術の基本法則, 効草書房, 2000.
- [28] ポアンカレ (著), 吉田洋一 (訳), 科学と方法, 岩波文庫, 岩波書店, 1953.
- [29] ボエティウス (著), 永嶋哲也 (訳註), ボエティウス「イサゴーゲー第二註解」,
http://www002.upp.so-net.ne.jp/tetsu/study/t01_boepor.pdf
- [30] M.Lynne Murphy, Ane. Koskela, 意味論キーテーム辞典, 開拓社, 2015
- [31] 皆本晃弥, IEEE754 と数値計算,
<http://www.ma.is.saga-u.ac.jp/minamoto/doc/kyudai.pdf>
- [32] 山内志朗, 普遍論争, 平凡社ライブラリー, 2008
- [33] 横田博史, はじめての Maxima, I/O Books, 工学社, 2006.
- [34] 横田博史, はじめての Maxima 改訂 α 版 (MathLibre に収録)
- [35] 横田博史, 数値計算・可視化ツール Yorick, I/O Books, 工学社, 2010.
- [36] 吉田光邦, 錬金術 - 仙術と科学の間 -, 中央公論新社, 2014.
- [37] J.Barnes, PORPHYRY INTRODUCTION, Oxford University Press, 2006.
- [38] Bell, Toposes and the Local Set Theories, Dover, .
- [39] Birman, Braid groups and mapping class groups, Ann. Math, Princeton University Press, 1963.
- [40] Boethius, Isagoge, <http://www.forumromanum.org/literature/boethius/isag.html>
- [41] G. J. Brose, MATLAB 数値解析, Ohmsha, 1998.
- [42] Haigh, An interview with Jack J. Dongarra,
http://history.siam.org/pdfs2/Dongarra_%20returned_SIAM_copy.pdf, 2004.
- [43] Goodger, reStructuredText ディレクティブ,
<http://docutils.sphinx-users.jp/docutils/docs/ref/rst/directives.html>
- [44] An interview with Charles L. Lawson,
http://history.siam.org/pdfs2/Lawson_final.pdf, 2004
- [45] Mac Lane, The Category theory for working Mathematician, Springer
- [46] Mac Lane, Moerdijk, Sheaves in Geometry and Logic, A First Introduction to Topos Theory, Springer Verlag, 1992.
- [47] Porphyry, Introduction(Isagoge) to the logical Categories of Aristotle,
http://www.ccel.org/ccel/pearse/morefathers/files/poaphyry_isagogue_01_intro.htm
- [48] Porphyry, Letter to Marcella,

- http://www.tertullian.org/fathers/porphry_marcella_02_text.htm
- [49] B.Russell, The Principles of Mathematics,W.W.Norton & Company,Inc.,1996.
- [50] B.Russell & A.N.Whitehead,Principia Mathematica to *56, Cambridge Mathematical Library,Cambridge University Press,1997.
- [51] Diving into Python: <http://www.diveintopython.net/toc/index.html>
- [52] MathWorks 日本: <http://www.mathworks.co.jp/>
- [53] アテナイの学堂 <http://ja.wikipedia.org/wiki/アテナイの学堂>