

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 -

USO803 版

横田博史

平成 29 年 09 月 19 日 (火)

数式処理システム SageMath への招待 ©(2017) 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 が与えているのです。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なのです。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なのです。

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 USO800 版なのです。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、そのためにはこの文書の二次配布はご遠慮願います。とはいえた秘密の文書ではありません。GitHub で公開しているので、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の使い方	10
1.3 この本の方針	24
第2章 オブジェクト指向について	25
2.1 SageMathの中核としてのPython	26
2.2 オブジェクト指向プログラミングの哲学的側面	27
2.3 判断と推論	44
2.4 集合論について	50
2.5 関数と関係	58
2.6 代数的構造について	61
2.7 異 (Category)	68
2.8 トポス (Topos)	105
2.9 トポスの基本定理	113
2.10 高階論理 λ -h.o.l. とトポス	116
第3章 Pythonについて	125
3.1 Pythonとは?	126
3.2 Pythonの特徴	127
3.3 有理数を構築してみよう	140
3.4 バッカス・ナウア記法 (BNF)について	147
3.5 Pythonの字句解析について	151
3.6 Pythonの式と文	162
3.7 複合文	182
3.8 オブジェクトについて	185
3.9 特殊メソッド	211
3.10 記述子 (descriptor)	218

3.11	クラス属性の参照について	220
3.12	名前空間とスコープ	224
3.13	名前付けと束縛	225
3.14	例外	227
第 4 章 数値行列処理即席講座		235
4.1	はじめに	235
4.2	Numpy について	236
4.3	実数の表現	238
4.4	数値行列ライブラリについて	244
4.5	MATLAB 系言語と Numpy の速習	264
4.6	配列の演算について	281
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	299
5.3	SageMath の数の構成	305
5.4	SageMath のオブジェクト	308
5.5	SageObject の各階層について	310
第 6 章 SQLite について		315
6.1	SQLite 速習	316
6.2	SQLite のキーワード	320
6.3	SageMath から SQLite を使う	321
第 7 章 結び目理論への適用		325
7.1	概要	325
7.2	結び目/絡み目とは	325
7.3	正則射影図	327
7.4	結び目/絡み目の同値性	329
7.5	群について	331
7.6	結び目/絡み目を表現する群	336
7.7	組紐群と置換群	343
7.8	ガウス・コード	349
7.9	LinkClass クラス	359
7.10	多項式不变量	382
7.11	カウフマンのブラケット多項式を計算するプログラム	385

第 8 章	CoCalc について	399
8.1	CoCalc とは	400
8.2	利用者登録について	401
8.3	基本設定	402
8.4	プロジェクト	403
8.5	ファイルのアップロードとダウンロード	404
8.6	端末, Jupiter, L ^A T _E X の利用について	407
8.7	Docker コンテナとしての CoCalc	411
第 9 章	SageMath の拡張	413
9.1	sagemath からのパッケージ入手	414
9.2	一般の Python パッケージのインストール	414
9.3	GNU R のパッケージのインストール	414
参考文献		415

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。この SageMath は商用の数式処理システム *Mathematica* と数値行列処理システムの MATLAB の代替となる OSS の数学環境の実現を目的にしていますが、そのために SageMath の開発で採用した手法・手段が非常にユニークであることに尽きます。実際、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研究目的のために開発された高機能の OSS のアプリケーションが多数存在していることに加え、それらを動作させるための計算機環境も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。

まず、OSS のアプリケーションには研究機関等の研究の成果として一般に公開したものがあり、研究者が関心を持っている分野では非常に優れた性能を持っています。しかし、限られた用途と利用者を対象とす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、専門家でも使いにくかつたり、逆に、汎用性があり、そのユーザ・インターフェイスを含めた使い勝手が良かったとしても、前提とされる知識が過剰に要求されるといった理由に加え、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多々あります。とは言え、他の分野に転用が全く効かないという本質的に特殊なものは少なく、他の分野にも転用が効くものが沢山あります。そこで、そのようなアプリケーションの入出力を抽象化することで汎用性を持たせ、処理言語やデータ構造も Python で統一し、それらのアプリケーションのインターフェイスを構築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でしかありません。そして、新たに定義したデータにしても当初の専門分野のみの観点で定義したものから、より広範囲な分野の観点を加えた一般的なものとして再定義された結果、入出力データが Python で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るだけでなく、Python を共通基盤にしたことで本来の専門家にとっても統一的な操作環境が得られ、しかも、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ることになります。

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 SageMath のやり方が十分に実用的になった背景に、現在の計算機環境が従来と比べて格段に贅沢な環境にあるという現実があります。実際、2017年現在の携帯電話のCPUでさえも複数のコアを持って1GHz以上の動作周波数で動作し、加えて2GB以上のメモリ、最低でも16GB程度の記憶媒体を持ち、さらに高速ネットワークに当然のように接続可能な環境があります。このような「**贅沢な環境**」になる以前の環境ではアセンブラーを駆使するといった工夫でプログラムサイズを可能な限り小さくしたり、最適化を行う必要がありました。しかし、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余程下手なプログラムを組まなければ実用的な処理速度で動作します。おまけに職人技で最適化したシステムよりも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。

1.1.2 SageMath の中核としての Python

SageMathは基本的にPythonで記述されており、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を取り入れられた言語で、さらに「**多重模範言語 (multi-paradigm 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。ちなみに、オブジェクト指向プログラミングを導入した数式処理システムでは、いきなり数式を入力して処理が行えるとは限りません。このことを多項式の因式分解という処理を通して考えてみましょう。オブジェクト指向プログラムに対応した数式処理システムで多項式の因式分解を行うために最初にすべきことは、その多項式が所属すべき「**多項式のクラス**」の「**インスタンス**」として多項式を生成することです。これによってインスタンスの因式分解を実行する函数、すなわち、「**メソッド**」が利用可能になります。この多項式のインスタンス化で、数式処理システムによっては、これから扱う多項式の係数が何で、変数が何であるかをあらかじめ指定しなければならないものがあります。このような処理は「**整数係数の多項式の因式分解を素早く行いたい**」と思う利用者にとっては余計な作業で、おまけに、その定義が覚え難いものであればより一層、面倒な障壁になるでしょう。これを真面目に処理している数式処理ソフトにSingularがあります。このSingularでは「**環**」と呼ばれる数学上の対象を「**世界**」として定めて多項式の処理がはじめて行えますが、何も指定しなければ整数の四則演算しかできません。こうした準備は前提になる概念の知識があればごく自然なことですが、そうでなければ何かと事前に設定をしなければならない「**堅苦しい言語**」で、おまけに多項式の展開を覚えたての中学生が計算機で多項式の展開をさせるときに「**環**」の概念を要求する事態になります。このことはただでさえ堅苦しい言語に数学的な概念というさらに厄介な障壁(?)が加わることに

*1 だからといって高速処理への要求がなくなるという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。

なります。もちろん、こうした処理を「**魔法の呪文**」だと思ってしまう手もありますが、いずれにせよ道具を使う都度、「**呪文**」で呼び出すという不可解な手間が増えることになります。ところがPythonは前述のように多重模範の言語であるお陰でメソッドを通常の函数のように扱うことができたり、利用する变数を宣言しておけばそのまま多項式を入力できる等と命令言語のBasicのような扱いが可能と、オブジェクト指向プログラミング言語としての侧面を全面に出さずに利用できます。

実際にSageMathで多項式の因子分解を行ってみましょう：

```
sage: a = x^4 + 4*x^3 + 6*x^2 + 4*x + 1
sage: a.factor()
(x + 1)^4
sage: factor(a)
(x + 1)^4
sage: type(a)
<type 'sage.symbolic.expression.Expression'>
sage: type(x)
<type 'sage.symbolic.expression.Expression'>
```

ここで式の因子分解では式の因子分解を行うメソッドfactor()を利用して‘a.factor()’で分解したり、函数として‘factor(a)’で分解させています。どちらも「**多項式環**」は表に出ていません。とは言え、多項式環が表から見えないだけで実際は厳として存在しています。実際、SageMathが立ち上がった時点で变数xの多項式環が定義されており、このことはtype(a)とtype(x)の結果が‘<type 'sage.symbolic.expression.Expression'>’と返却されることから容易に判ります。この返却値の意味することはSageMathではあらかじめ变数xの多項式環が定義されているということです。また、SageMathでは变数x以外は未定義ですが、变数x以外の变数を持つ多項式を入力するときは必要な变数を变数宣言であらかじめ増やしておきます。このように計算機代わりに利用する程度であればオブジェクト指向プログラミング言語であることを意識しなくとも使えます。

では、面倒なことが多そうなオブジェクト指向の言語には一体どのような有り難さがあるのでしょうか？その大きなご利益の一つが「**継承**」と呼ばれる仕組です。つまり、新しいクラスを構築する際に既存のクラスを雛形にすると、その既存のクラスに付随する属性やメソッドをそのまま新しいクラスのメソッドとして使えるという機能です。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、複素数は実数の対として表現可能なことから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形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を定めることができます。このことはライブラリを構築しても、そのライブラリ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。このように継承をうまく利用することでソフトウェア資産を有効に、しかも効率的に利用することができます。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したクラスを選択すればよく、最適のクラスがなくても類似の概念を継承することで新たなクラスを定義しさえすればよいのです。このように日々、数学の諸問題への対処が将来への資産として生かせることが SageMath の強みです。

1.1.3 電池込みだよ

SageMath は Python を基盤に数学に関する OSS を徹底して統合した豪華なシステムです。この SageMath に類似したものに Continuum Analytics 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、OSS のアプリケーションやライブラリを纏めたもので、それらに SageMath と重なるものが多くあること、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の度合いは大きく異なります。たとえば、Python には数式処理モジュール SymPy があり、このモジュールは SageMath と Anaconda の双方に含まれています。ここで Anaconda の SymPy は Numpy や Matplotlib 同様に Anaconda を構成する Python モジュールの一つですが、SageMath では Python を使って新たに数学対象のクラスを構築し、SymPy は数式の表現を受け持つ部分として取り込まれています。この SymPy も多項式の展開・因子分解、初等函数の微分・積分といった数式処理能力を持っていますが、SageMath では Common Lisp で記述された汎用の数式処理 **Maxima** を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 Macsyma の OSS 版で、文脈 (Context) 等の興味深い機能を持っているものの、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いないため、可換環の処理は **Singular**、数論は **PARI/GP**、有限群論は **GAP** といった OSS の数式処理を SageMath は組み込んでいます。しかし、利用者にとっては 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でしかありません。その一方で、必要に応じてこれらの専門ツールを SageMath と独立して利用したり、SageMath 内部で明示的に使うこともできます。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。

つぎに数値計算についても述べておきましょう。通常、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の処理の遅さです。これは数値演算で任意精度の演算が可能であることに加え、数値行列処理に特化した MATLAB 系言語の数値行列を処理するための構文と比べて格段に貧弱である点、数式処理に注力した結果、数値行列処理のための専用のライブラリが未整備になり易いことが挙げられます。この弱点に対し、SageMath は任意精度の数値計算では GMP を利用し、数値行列処理に対してはモジュール Numpy を利用します。この Numpy は Python のリスト型オブジェクトを強化し、多次元配列と関連する演算を Python に導入するモジュールで、数値行列処理向けのライブラリも組込まれています。ここで数値行列処理向けのライブラリの FORTRAN や C 向けに記述されたもので、その中で「**LAPACK(Linear Algebra PAKage)**」が著名で、この LAPACK は「**BLAS(Basic Linear Algebra Subprograms)**」^{*2} と呼ばれるベクトルと行列の基本的な計算を効率良く処理することを目的としたルーチン群上に構築されており、この BLAS の性能が LAPACK の性能に直接影響します。そのためにさまざまな計算機環境に対して最適化を行った BLAS があります。まず、最適化が行われた BLAS で無課金で利用可能な「GotoBLAS2」の後継である「**OpenBLAS**」と「**ATLAS(Automatically Tuned ALgebra Software)**」、商用のものに Intel の「**MKL(Math Kernel Library)**」や AMD の「**ACML(AMD Core Math Library)**」があり、SageMath には OpenBLAS が組込まれています。これに加えて 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 があり、このパッケージも SageMath に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 に匹敵する数値行列処理能力を SageMath は兼ね備えることになります。

SageMath はノートブック形式のユーザ・インターフェイスとして Jupyter を採用しており、数式のレンダリングでは jsMath が用いられています。また、ワークシートの共用也可能なためにネットワーク上で協調作業ができます。また、仮想端末上では IPython ベースでテキストによる数式のプリティプリント（アスキーアートによる数式表示!）が行われます。その他にも多くのアプリケーションとライブラリが SageMath に有機的に取り込まれており、SageMath は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ています。さらにクラウドベースの商用サービスの CoCalc^{*3}があり、無課金であっても SageMath 本体、さらには LaTeX や reST の文書作成環境を一通り利用できます。このように SageMath は既存の数式処理システムと比較しても機能的に見劣りしないどころか、

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

1.1 背景

その柔軟性の高さで勝るシステムです。そのために SageMath は数式処理システムとして安易に使うも良ければ、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)」^{*4}を彷彿させるものです。

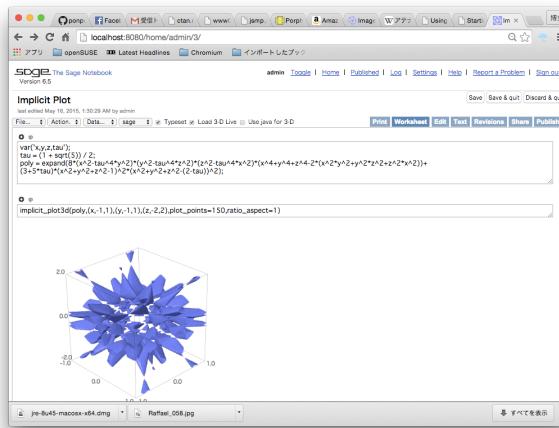


図 1.1 ウエブ・ブラウザを用いた SageMath のノートブック

1.1.4 SageMath が使える環境は？

SageMath は基本的に UNIX 環境で動作します。ここで UNIX 環境には Solaris, FreeBSD や Apple の macOS(OSX), それと UNIX 互換の環境として各種 LINUX ディストリビューションがあります。また、macOS 版の SageMath 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ますが、アプリケーション版 Sage.app を起動すると Finder 上に SageMath のアイコンが現れて図 1.2 に示すように、そこから選べるメニューに 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

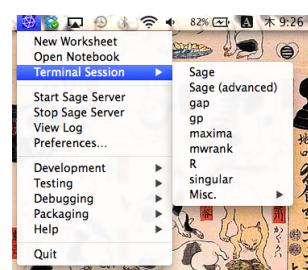


図 1.2 Sage.app のメニュー

^{*4} 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。

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。なお、MS-Windows 上では現時点では正規版はありません。ただし、Cygnus 環境上のインストーラが開発されたり、MS-Windows 10 で Linux バイナリが実行可能になるといった動きがあるために今後に注目といったところでしょう。

ところで SageMath が Linux 上で動作するとは言え、Linux にはディストリビューションの違いがあり、さらには同じディストリビューションでもバージョンの違い、同じバージョンでも個々のライブラリ等のアップデートの違いといった些細な違いがあります。SageMath のような複合的なシステムでは、ライブラリ等の整合性の問題からインストールに障害が出易く、さらに苦労して構築した環境が実際に正常に動作するかどうかといったこと、おまけに正常に動作している環境を各種ソフトウェアの更新に合せて保持しなければならないといった困難な課題が生じます。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を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下げることができます。さらに MS-Windows 環境のように UNIX 環境と比べてあまりにも異質な環境に無理に移植せずに仮想化環境 (VMware や Oracle VirtualBox) を活用しています。この場合はウェブ・ブラウザを用いて MS-Windows 上の仮想計算機環境で動作している事情は末端の利用者には判りません。また、仮想計算機よりも効率的な方法として Docker のコンテナとして配布する手段もあります。Docker のコンテナとして SageMath のクラウド環境版である CoCalc^{*5} の Docker イメージがあります。こちらはインストールに 8GB 程のディスク容量を必要としますが、GNU Octave や Julia, LaTeX や reST の文章作成環境といった環境が一通り揃った非常に贅沢な環境です。こちらの方法であれば Linux 環境であれば仮想計算機よりも処理が軽快であり、macOS や MS-Windows 環境でもアプリケーションとして使い易いでしょう。ただし、CoCalc そのものと比較すると、2017 年 6 月現在では、とりあえず SageMath と Markdown 編集環境の利用で問題がない縮小版といった状況です。ちなみに SageMath 自体はソースファイルで 280MB 程度、バイナリ版で 700MB、MS-Windows 版になると仮想計算機込みで 1GB 程度と大きなシステムです。また、CoCalc の Docker イメージになると 8GB のディスク容量を必要とします。とは言え、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り、その入手がさほどの負担にならなくなっています。実際、HD 画質の動画配信なら 1GB の容量は 2 時間程度の映画程度の大きさでしかありません。

^{*5} SageMathCloud を 2017 年 5 月に改名したものです。

また、計算機のディスクや処理能力に余裕があれば、いつそのこと MathLibre^{*6}を導入されることは如何でしょうか？ 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も含まれています。 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付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^{*7}。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。

もし、SageMath をインストールすることや仮想計算機が利用できなければ、最後の手段として CoCalc^{*8}を使ってみましょう！ この CoCalc は §8 で説明しますが、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することができます。こちらは L^AT_EX 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。

1.1.5 SageMath の情報

SageMath の情報は本家サイト: <http://www.sagemath.org> から得られます。また、日本語の情報源として Google Group に「**Sage Japan**」もあり、こちらはあまり活発な活

^{*6} 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。

^{*7} 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。

^{*8} <https://cocalc.com>

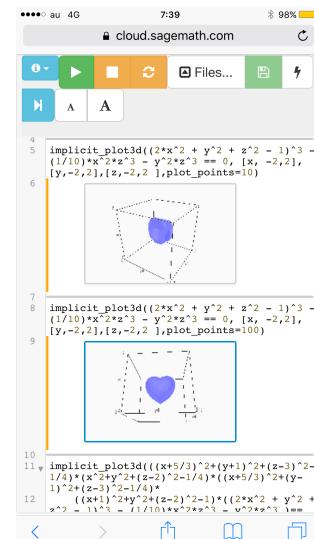


図 1.3 スマートフォンで利用

動をしておらず、今後に期待といった状態にあるため、この本を読まれた方は是非参加して盛り上げて頂ければと思います。さらに日本でも **Sage Days in Japan** というワークショップが開かれることができます。それから書籍には「群論の味わい-置換群で解き明かすルービックキューブと 15 パズル」が販売されている程度で、その他には「Sage for Newbies」の抄訳の「はじめての Sage」が WEB で公開されています^{*9}。この文書は MathLibre(KNOPPIX/Math) の DVD にも収録されており、SageMath に限らず計算機一般の話もあって非常に面白い文書です。

1.2 SageMath の使い方

1.2.1 SageMath のユーザ・インターフェイス

さて、ここでは 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、もしもそうでなければ CoCalc を試してみると良いでしょう。まず、最初に SageMath を立ち上げましょう。UNIX 系の OS であれば仮想端末上で `sage` と入力してください。また、Mac で macOS 版を導入しているのなら通常のアプリケーションと同様にランチャから Sage のアイコンをクリックすれば立ち上がります。すると Python のシェルである IPython が立ち上ります。また、macOS 版であれば Safari 等のブラウザを利用したノートブック形式で立ち上がりますが、こちらは Jupyter のノートブックを利用したものです。まず、IPython は標準の Python よりも履歴機能、ログ出力や GUI 等の機能が通常の Python インタプリタよりも強化されたシェルであり、Jupyter は IPython を基に Python 依存のコードを取り除いて、Python に限定されずに GNU Octave や GNU R といったアプリケーションのノートブック形式のフロントエンドとして利用ができます。ここでは仮想端末版の様子を示しておきます：

SageMath version 7.2, Release Date: 2016-05-15	
Type "notebook()" for the browser-based notebook interface.	
Type "help()" for help.	

sage: p1 = (x + 1)^3	
sage: p1	
(x + 1)^3	

ここでの ‘sage:’ は仮想端末で利用するときの SageMath のプロンプトです。それに続いて式の入力をを行い、入力式の評価は仮想端末では `Enter` キーで行います。ノートブック形式の場合はセル内に式を入力して `Shift+Enter` で入力の評価を行います。これは数式処理

^{*9} 筆者のサイトから辿って入手することができます。

理の *Mathematica* と同様のセルの評価方法です。また、SageMath には Maxima のような式の末尾を示す記号がありません。Maxima の式の末尾の記号 “;” を SageMath の入力行末尾に置くと結果のエコーバックを行いません。また値の割当は記号 “=” を用い、この際にエコーバックは行われません。これは Python のオブジェクトでは print 文^{*10}で表示される「公式の文字列」と呼ばれる文字列をオブジェクトの属性を使って定めることができます。この文字列が定められていないときはオブジェクトが配置された番地に依存する文字列が表示されるためでしょう。なお、この例では変数 p1 に ‘ $(x + 1)^3$ ’ を割当てており、この SageMath の式は数式 $(x + 1)^3$ に対応します。ちなみに、Python の幂乗の演算子は演算子 “**” で、演算子 “^” はビット単位の演算子の排他的論理和 XOR に相当します。このように SageMath では演算子 “^” は SageMath の数学的対象の幂乗の演算子として扱われますが、演算子 “^” が幂乗の演算子として置換えられていません。この点を忘れて演算子 “^” を本来の Python オブジェクトに使うと本来の XOR の意味で処理されることに注意してください。たとえば for 文で変数 i でループをまわすときに変数 i に束縛される整数は Python の整数型であり、SageMath の整数ではありません。そのため字面は同一でも SageMath の整数のメソッドが使えないといった一見すると意味不明なエラーに悩まされる羽目になります。

では p1 にはどのようなオブジェクトが束縛されているのでしょうか？このオブジェクトの型を調べるために函数 type() が使えます：

```
sage: type(p1)
sage.symbolic.expression.Expression
```

函数 type() の結果から変数 p1 には ‘symbolic.expression.Expression’ というクラスのオブジェクトが束縛されていることが判ります。では、このクラスのオブジェクトでどのような処理が可能でしょうか？それを簡単に知る方法があります。SageMath に `p1.` と入力して `[TAB]` キーを押して下さい。すると下記のように一覧が表示されます：

```
sage: p1.
Display all 190 possibilities? (y or n)
p1.N                      p1.gamma                  p1.op
p1.Order                   p1.gcd                     p1.operands
— 略 —
p1.full_simplify           p1.numerator_denominator
p1.function                p1.numerical_approx
sage: p1.
```

^{*10} Python 3.x で函数 print()

途中を省略していますが、最初に記述があるように 190 個程のメソッドや属性の一覧が表示されます。この機能は IPython の命令補完機能を利用したもので、Python の構文や函数、メソッドや属性の補完が TAB キーを併用することで行えます。この機能は後述のウェブ・ブラウザをユーザ・インターフェイスとして用いるときでも同様の表示が行われます。この機能は函数 dir() の機能を利用したもので、途中まで入力した文字列に適合する「**名前空間**」中の名前の一覧を出力しています。ここで名前空間が何であるかはあとで説明することにして、とりあえず ‘p1.expa’ と入力して TAB キーを押してみて下さい：

```
sage: p2 = p1.expa
p1.expand          p1.expand_log      p1.expand_rational  p1.expand_trig
sage: p2 = p1.expand
```

今度は ‘p1.expand’ と補完され、この文字列を先頭に持つ幾つかの候補が表示されます。このように SageMath のユーザ・インターフェイスには TAB キーを用いた入力の補完機能があり、名前を補完したり、候補が複数存在するときに候補を表示します。ここでやろうとしていることは名前 p1 に束縛された多項式の展開ですが、この一覧に含まれている ‘expand_log’ は指数函数を含む式の展開、‘expand_trig’ は三角函数を含む式の展開に適した処理を行うメソッドで、残りの ‘expand’ か ‘expand_rarional’ が妥当です。そこでメソッド expand() で p1 を処理しましょう：

```
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
```

ここでの式の展開はオブジェクトの情報だけで式の展開が行えるために ‘p1.expand()’ で展開することができます。ところで、Python は多重規範言語であるために、命令として ‘expand(p1)’ と記述することもできます。このように TAB キーを使った入力補完機能が使えます。

次に $x^2 - y^2$ の因子分解を SageMath で試してみましょう：

```
sage: (x^2-y^2).factor()
```

```
NameError                                     Traceback (most recent call last)
<ipython-input-1-93bdb3cd19b8> in <module>()
      1 (x**Integer(2)-y**Integer(2)).factor()
NameError: name 'y' is not defined
```

今度はエラーが出ました！このエラーの内容は「**名前 y が何なのか判らない**」という意味です。ところで、名前 x については何も文句を言っていません。これはなぜでしょうか？

そこで函数 type() で名前 x の型を確認しておきましょう:

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

```
NameError Traceback (most recent call last)
<ipython-input-6-6848ec1ead4> in <module>()
      1 type(y)

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

まず、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が、名前 y には何も束縛されていないことが ‘NameError:’ に続く ‘name 'y' is not defined’ という文言から判ります。ちなみにここで発生したものは「例外」と呼ばれるオブジェクトで、例外処理と呼ばれる処理を定義することで例外に対応した処理が可能になります。さて、ここで「**NameError**」や「**name 'y'**」という表記には Python の「**名前空間 (name space)**」が関係しています。つまり、Python が扱うデータは全てオブジェクトとして捉えられますが、これらオブジェクトの名札に対応するものがここでの名前です。そして、名前とオブジェクトの対応関係を「**名前空間**」と呼びます。なお、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、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。ここで生じたエラーの理由は変数 y に対応するオブジェクトがないためで、変数 x でエラーが出なかった理由は名前 x に対応するオブジェクトが存在しているためです。では新たに利用する変数の定義はどうすればよいでしょうか？この場合は函数 var()^{*11}を使って名前 y が式の変数であることを SageMath に教えてやれば良いのです：

```
sage: var('y')
y
sage: (x^2-y^2).factor()
(x + y)*(x - y)
```

‘var('y')’ で名前 y が名前空間に変数として加えることで名前 y を含む式を入力してもエラーが出なくなります。なお、複数の名前を変数として新たに登録するときは単純に函数 var() の引数として文字列のリストを与えます：

```
sage: var(['x1', 'x2'])
(x1, x2)
sage: (x1^6-x2^3).factor()
(x1^4 + x1^2*x2 + x2^2)*(x1^2 - x2)
```

^{*11} 函数 var() は SymPy 由来の函数です。

ここで 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’]” のように括ったものです。なお、SageMath のリストの生成は使い易いように Python のリストの生成機能が拡張されています。たとえば、1 から 10 までの自然数のリストの生成は ‘[1..10]’ で行うことができます:

```
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 SageMath 独自のもので、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 から開始することに注意して下さい。そのために 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ので注意が必要です。たとえば ‘L[0:5]’ で添字が 0 から 4 までの名前 L に束縛されたリストを返却します。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。と、SageMath は MATLAB に類似の方法で多次元の配列操作が可能であり、あまり数値行列の操作が強くなかった従来の数式処理システムと比較し、数値行列処理システムの MATLAB と大差がない機能と処理速度を持っています。

次に代数方程式を解いてみましょう。SageMath では代数方程式を函数 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^{*12}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。

では最後に ‘`help(expand)`’ と入力してみましょう。こうすることで函数やクラス、モジュール等のオブジェクトの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。この Python のヘルプ文書の組込の方法は Matlab 系の言語でも見られる方法ですが、Python では文書文字列に関しても「**PEP**」と呼ばれる規約があります (PEP-257 等)。なお、函数 `help()` は文書文字列の表示を行う Python 組込函数ですが、前述の Jupyter^{*13} では記号 “`?`” もオンラインヘルプとして使えます。たとえば函数 `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 “`?`” を追記します。ただし、この記号 “`?`” を使ったオンラインヘルプは SageMath のフロントエンドの機能のために Python のインタプリタで使えません。

^{*12} 整数係数の多項式が 0 になる数です。代表的数として整数、有理数、純虚数や n 乗根が挙げられます。

^{*13} Jupyter notebook と matplotlib の組合せは MATLAB 利用者にとっても馴染易い環境になります。

1.2.2 ノートブック形式のユーザインターフェイス

SageMath にはよりモダンな環境があります。この環境はノートブックを模し、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。このノートブックを実現するために Jupyter Notebook が用いられており、おおよその操作は Jupyter Notebook に準じます。

ノートブック形式の UI で利用する場合は仮想端末上で `sage -notebook` と入力するか、仮想端末上で SageMath を利用しているときは `notebook()` と入力するとウェブ・ブラウザが立ち上がって SageMath のノートブックが表示されます。MathLibre 上で SageMath を使用するのであれば \sqrt{Math} メニュー や Launcher から SageMath を選択すればノートブック形式の SageMath が立ち上がります。また macOS 上の Sage.app を利用しているのであれば、普通の macOS アプリケーションと同様に Launchpad から呼び出せばウェブ・ブラウザを起動して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、そこからノートブック形式で SageMath を開いたり、仮想端末から SageMath を起動することもできます。なお、ノートブック形式で SageMath をはじめて立上げるとパスワード設定が要求されます。SageMath のノートブックが立ち上がった時点で `New Worksheet` ボタンを押すとノートブック名を設定するダイアログが表示され、それからワークシートへの入力ができるようになります。ここで jsMath がインストールされた環境であれば、Typeset のチェックボックスにチェックを入れておけば数式が綺麗にレンダリングされます。また、計算機（あるいは携帯電話!）がインターネットに接続しているのであれば CoCalc (<https://cocalc.com/>) に接続してみましょう。このサイトは JavaScript に対応したウェブ・ブラウザであれば式の表示や 2 次元グラフとそのアニメーションが利用可能で、PC だけではなく Android 携帯、iPhone 等の iOS 機器でも利用できます。この CoCalc を利用するためには利用者登録が必要ですが、無課金で利用することができます。この CoCalc の詳細は §8 を参照して下さい。

さて、SageMath のノートブックで式を評価するためには Mathematica のフロントエンドと同様に、式をセルに入力したのちに `Shift+Enter` でその式の評価を行います。すると計算結果が入力の下に表示されます。ここで数式は既定値としては昔風のキャラクタを用いた数式の表示となります。jsMath が利用可能な環境であれば上の Typeset にチェックを入れると数式が綺麗にレンダリングされます。そして、このワークシートを開いたり、ワークシートを複数の利用者間で共有することもできます。

1.2.3 画像の簡単な処理

ここでは SageMath が有用な小道具が詰まった Python 環境であることを示すために画像のちょっとした処理を行います。まず、SageMath は Python を Jupyter や IPython の環境で利用している状況と大差ありません。したがって基本的な操作やプログラミングの骨子は Python そのものです。だから、SageMath で必要なモジュールの読み込みは Python と同様に import 文を用います。ここで Python で画像を扱う方法として二つの代表的な方法があります。一つが PIL(Python Imaging Library) を用いる方法、もう一つが Matplotlib を用いる方法で、双方とも SageMath に含まれています。ただし、PIL で生成した画像オブジェクトと Matplotlib で生成した画像オブジェクトは異なったもので、Matplotlib で生成した画像オブジェクトは Matplotlib が Numpy 上で構築されているために、Numpy の多次元配列オブジェクトである ndarray 型のオブジェクトになります。この Matplotlib は商用の数値行列処理システム MATLAB を強く意識したパッケージであるために MATLAB に類似した処理が行えます。

では Matplotlib を使って画像の読み込みを行いましょう：

```
sage: import matplotlib.pyplot as plt
sage: from matplotlib import image as mi
sage: i1=mi.imread('Documents/ScuolaDiAtene.png')
sage: matrix_plot(i1).show()
sage: type(i1)
<type 'numpy.ndarray'>
sage: i1.shape
(509, 800, 3)
```

この例では画像の描画用に Matplotlib パッケージのモジュール pyplot を ‘plt’、画像読み込みモジュール image を ‘mi’ と読み替えておきます。ここで一方が ‘import ...’ でもう一方が ‘from ...’ になっていますが、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。つまり、Python のパッケージやモジュールは階層構造を持っており、パッケージ Matplotlib の直下に pyplot と image というモジュールがあります。最初の ‘import ...’ では ‘matplotlib.pyplot’ とありますが、このような形で直接、モジュール pyplot の読み込みを行っています。一方の ‘from ...’ の例で、‘from matplotlib’ からパッケージを指定し、その下にあるモジュール image を読み込みという文になっています。さて、ここで ‘as’ を用いてモジュールの読み替えを行っていますが、その理由は次の行で判ります。まず、画像の読み込みで ‘mi.imread()’ としています。これはモジュール image に含まれている函数 imread() を用いるという意味で、本来ならば ‘matplotlib.image.imread()’ としなければならないところを ‘matplotlib.image’ の箇所を ‘mi’ で読み替えることで ‘mi.imread()’ に

できます。さて、画像はカレントディレクトリからみて‘Documents’ディクトリ内にある‘ScuolaDiAtene.png’ファイルです。この画像をここでは函数 `imread()` で読みますが、この函数 `imread()` で読み込まれた画像は多次元の数値配列に変換されます。この画像オブジェクトが Python でどのような代物になっているかは函数 `type()` を使って調べることができます。この例では‘`numpy.ndarray`’という文字列が結果に現われています。これは読み込んだ画像のインスタンス `i1` が NumPy ライブラリで定義された多次元数値配列であることを示しています。そして、この数値配列の具体的な形(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ね。このことから本来の画像は 509×800 の大きさの RGB 画像であることが判ります。というのも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 に対応する数値配列であることを示しています。なお、函数 `imread()` で読み込んだ画像は 0 から 1 までの浮動小数点数で輝度が表現された数値配列として生成されています。それから数値配列を `matrix_plot` を使って図 1.4 のように表示できます：



図 1.4 読込画像の表示例

ここまで処理は SageMath でなくても Python で PIL/Pillow, NumPy や PyLab があればできることです。折角なので SageMath のノートに絵を貼りましょう。これに特殊な処理は不要です。単純にモジュール Image の函数 `save()` を使って画像を保存すると SageMath のノートブック側でその画像を貼ってくれます。とは言え、表示領域を自動調整してくれるようなものではありません：

前述のように SageMath には画像処理パッケージの PIL(Python Imaging Library)が標準で含まれています^{*14}。この PIL を用いた画像データの取り込みと操作を述べましょ

^{*14}PIL から分枝(fork)したものに Pillow (<http://python-imaging.github.io/>) があります。PIL は Python 2.x のみに対応しているだけですが、Pillow は 3.x に対応し、PIL と同様の使い方ができます。

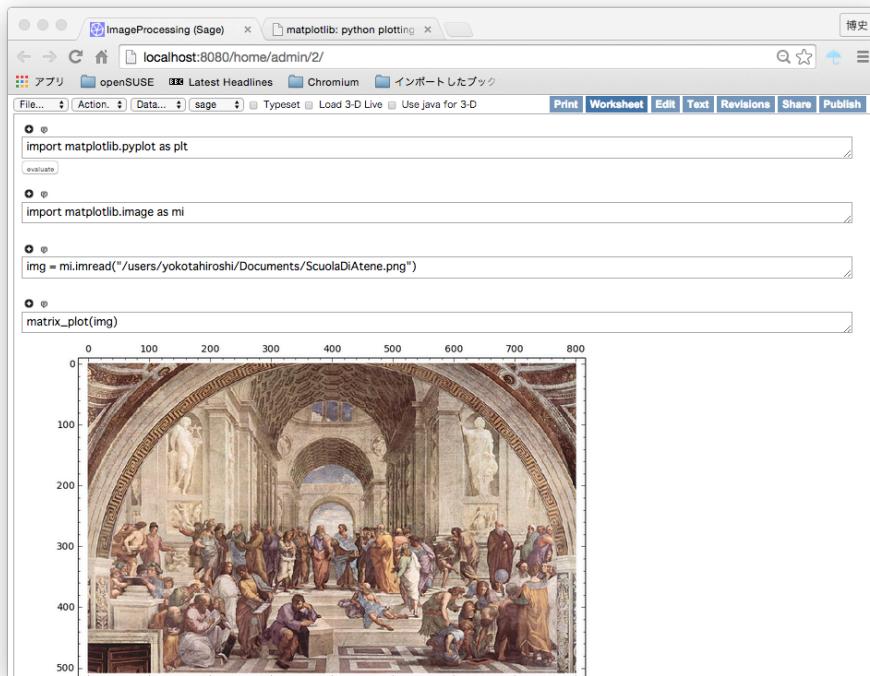


図 1.5 ノートブックへの画像の表示例

う。Matplotlib のときと同様に PIL を用いた画像の読み込み・書き込み、それと画像オブジェクトの表示は、それらの処理を行う関数がモジュール Image に纏められており、あらかじめ `from PIL import Image` でモジュールの読み込みを行っておきます。それから画像の読み込みは関数 `open()` でファイルを指定します。

では PIL で画像の読み込みを行ってみましょう。ここでホームディレクトリ上のディレクトリ Documents に `ScuolaDiAtene.png` という名前の画像ファイルをおいた状態で以下の作業を行います：

```

sage: from PIL import Image
sage: im = Image.open('Documents/ScuolaDiAtene.png')
sage: im.show()
sage: im.save('test.png')

```

この例では最初にモジュールの読み込み、それから画像の読み込みと外部アプリケーションを用いた画像の表示と画像データの保存を行っています。ここでは最初にモジュール Image を PIL から読み込み、それからモジュール Image に含まれる関数 `open()` で画像ファイル

ルの読み込むことで SageMath のオブジェクトとして画像を取り込み, その際に名前 im に束縛します. それからインスタンス im のメソッド show() を用いて読み込んだ画像の表示を行ないます. ちなみに, このメソッド show() は外部アプリケーションを用いて画像の表示を行うメソッドで, 表示に用いる外部アプリケーションを指定できます. ちなみに Linux 版の SageMath で表示アプリケーションとして xv があらかじめ指定されており, macOS 版では命令 open() が用いられるように設定されています. この表示用のアプリケーションの設定はメソッド show() の command オプションで指定できます. たとえば, ImageMagick の命令 display を指定するときは `im.show(command='display')` で display を使って画像オブジェクトの表示が行われます. なお, command オプションでは検索経路上にあるアプリケーションならそのアプリケーション名, そうでなければアプリケーションに至るまでの経路も含む Python の文字列を指定します. また, SageMath をノートブック形式のフロントエンドで利用していればメソッド save() で PNG 形式や BMP 形式で画像オブジェクトを保存するとノートブック側に表示されます.

ところで Matplotlib と異なり PIL には画像データを RGB の行列や配列等に変換して処理する函数がありません. そのために画像データを配列データに変換して MATLAB 系言語が行うように配列上の処理を行いたければ Matplotlib, SciPy や NumPy といったパッケージが別途必要になります.もちろん, これらのライブラリも SageMath には最初から含まれています. まず, Python 上で効率的に多次元の数値配列を扱うためのパッケージとして NumPy があります. この NumPy は Python で多次元の数値配列を扱うためのパッケージ Numeric と後述の数値計算を行うためのパッケージ SciPy の数値配列モジュール Numer を統合したもので, NumPy は SciPy や Matplotlib の基底でもあります. この NumPy の函数 array() で PIL の函数 open() で読み込んだ画像データを RGB の配列データに変換することができます:

```
sage: import numpy as np
sage: imat = np.array(im)
sage: im.size
(800, 509)
sage: imat.shape
(509, 800, 3)
```

ここは最初に NumPy を import で読み込みますが, その際に接頭辞として “numpy” ではなくより短い “np” が使えるように指定しています. それから画像オブジェクトの配列データへの変換は NumPy の函数 array() を用います. 通常は numpy.array() を作用させますが, import 文で “np” と接頭辞を短縮することを宣言しているために np.array() を作用させます. さて, PIL の函数 open() を使って読み込んだ画像オブジェクト im の本来の大きさはメソッド size() で調べられます. このオブジェクト im を array() で NumPy

の配列に変換したときの配列の大きさはメソッド `shape()` で調べられます。最初に `im.size` の結果から画像は縦 509, 横 800 画素の画像であることが判ります。それから `imat.shape` の結果に `im.size` で現われなかった“3”という数がありますが、これは読み込んだ画像が RGB のカラー画像であることを意味し、本来の画像データが 509×800 の大きさの赤 (R), 青 (B), 緑 (G) に対応する配列に分解されていることを意味します。さて、NumPy の配列は MATLAB 系の言語と違い、その添字は 1 ではなく 0 から開始します。つまり、配列 `imat` はその第一成分の添字が 0 から 508 まで、第二成分の添字が 0 から 799 まで、第三成分の添字が 0, 1, 2 になります。

つぎに RGB を個別に数値配列から抜き出してみましょう。まず、`ndarray` 型配列に変換した画像データは画像の大きさの 2 次元数値配列が 3 個含まれていることがメソッド `shape()` の結果から判っています。さて、これら RGB に対応する 3 個の配列は第三の添字を指定することで取り出せます。つまり、第三の添字が 0 の配列が画像の赤の強さ (輝度) に対応し、同様に第三の添字が 1 の配列が画像の青の強さ、そして第三の添字が 2 の配列が緑の強さに対応します。これを SageMath で取り出すときは `imat[:, :, 0]` で赤、`imat[:, :, 1]` で青、`imat[:, :, 2]` で緑の輝度に対応する配列が得られます。ここで用いた添字記号 ‘:’ 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、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ています。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、MATLAB では `a($i:j$)` と表記します。この構文は Python の NumPy の配列でもほぼ同様ですが、Python では配列が 0 から開始するために `a[$i-1:j$]` と表記します。ここで添字の増分を 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)’ と表記します。ちなみに 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。また、配列の添字は “[]” で括らなければなりません。MATLAB で用いられる “()” は函数表記を構成する要素であるために SageMath(Python) の配列で間違って使わないように注意して下さい。このように MATLAB 系言語の表記と微妙に異なる箇所があるために MATLAB に慣れている方は、類似点があるだけに注意が必要です。

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赤色の強度を示す画像になります。この箇所を表示すると次の図 1.6 が得られます：

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

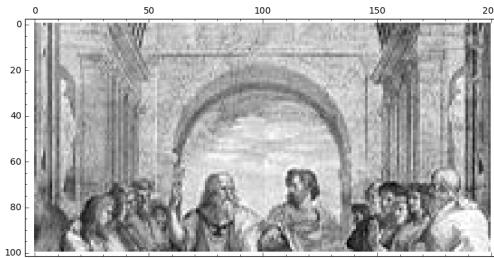


図 1.6 画像の一部切り取り

抜きも、この手法の応用で容易に行えます。たとえば、輝度がある値以上の箇所を取り出すといった方法も配列の処理で容易に行えます。

1.2.4 光線追跡

最後に外部アプリケーションを SageMath に取り込んだ例を挙げておきましょう。ここで紹介するアプリケーションは 光路追跡を行う Tachyon です。この Tachyon には 3 次元分子可視化プログラムの VMD(Visual Molecular Dynamics) が組込まれていますが、この VMD は本来は分子動力学計算アプリケーションのプリ・ポストアプリケーションで、このように SageMath では本来の目的と別の目的で機能が用いられているアプリケーションが多くあります。この Tachyon を SageMath に取り込んだことで通常の函数のように扱うことが可能になります：

```
t6 = Tachyon(camera_center=(0,-4,1), xres = 800, yres = 600,
               raydepth = 12, aspectratio=.75, antialiasing = True)
t6.light((0.02,0.012,0.001), 0.01, (1,0,0))
t6.light((0,0,10), 0.01, (0,0,1))
t6.texture('s', color = (.8,1,1), opacity = .9, specular = .95,
           diffuse = .3, ambient = 0.05)
t6.texture('p', color = (0,0,1), opacity = 1, specular = .2)
t6.sphere((-1,-.57735,-0.7071),1,'s')
t6.sphere((1,-.57735,-0.7071),1,'s')
t6.sphere((0,1.15465,-0.7071),1,'s')
```

```
t6.sphere((0,0,0.9259),1,'s')
t6.plane((0,0,-1.9259),(0,0,1),'p')
t6.show()
```

この例では Tachyon のオブジェクトをあらかじめ生成し、そのオブジェクトに光線追跡すべきオブジェクトや光源の情報を追加してメソッド `show()` ではじめて光線追跡の計算実行と結果表示を行います。ここで生成した画像を図 1.7 に示しておきますが、メソッド `show()` で SageMath を仮想端末上で動かしている場合は外部アプリケーションで、ノートブックであればそのノートブックに画像が表示されます：

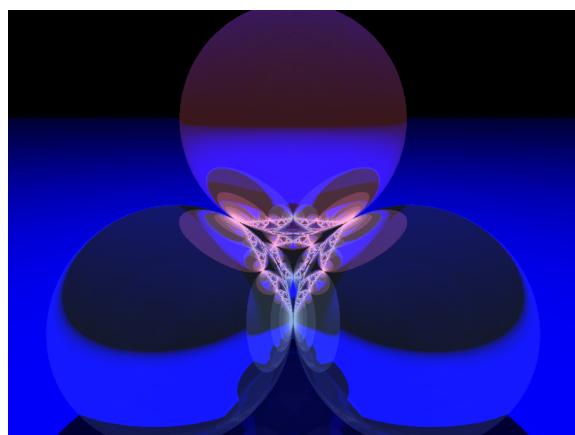


図 1.7 Sage から Tachyon を利用

このように SageMath は雑多なアプリケーションやライブラリへの一種のシェルとして動作していることが理解できるでしょう。そして、Tachyon のように独立したアプリケーションも至って自然な形で SageMath に組込まれています。だから、Python の使い方の基本さえ知っていれば、さまざまなアプリケーションを Python の世界でごく自然に利用できるという点が SageMath の大きな特徴であり、また、その利点でもあります。

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り扱い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。

1.3 この本の方針

さて、この本はどのような方針で進めるべきでしょうか？一つの方法は SageMath のマニュアルを片っ端から翻訳して載せることです。これは一見簡単ですが SageMath を構成する Maxima, Singular といったアプリケーションやライブラリに関する解説も必然的に含むことになって幾ら紙面があっても足りないことになるでしょう。もう一つは理論的背景を解説することです。こちらの関連する数学の基礎事項全てになりかねないために小冊子ではとても無理です。だからと言って高校数学程度の幾つかの例題に対して SageMath がどのように使えるかを試すことであれば別に本にしなくてもワークシートをそのままインターネット上に公開する程度で十分であり、実際、チュートリアルの日本語訳も既に公開されています。

ところで前述の入門書「はじめての Sage」では SageMath の利用者を次の四種類に分類しています：

SageMath 利用者の分類

SageMath 習熟者	Python も SageMath もよく判っている。
SageMath 知見者	Python は知っているが SageMath を少し齧った程度。
SageMath 新米	Python を知らないが、最低一つのプログラム言語を挙げられる。
プログラム作成新米	計算機がどのように動作するのか知らないし、プログラムを組んだこともない。

この分類からも判るように SageMath を使いこなすためには Python の知識が鍵になることが判ります。だからといって **Python というプログラム言語の「A から Ω まで」** をこの本で開陳する必要はないと判断します。そこで、この本では「**SageMath を使ってみたいという人**」を対象にし、SageMath を調べることで Python にも慣れてしまう戦術で進めるつもりです。そのときに「**Python を使って数学をどのように表現しようとして、実際にどう表現しているのか**」ということを探って行きたいと思っています。具体的には SageMath で数学の対象がどのように表現されているかを、PEP と呼ばれる Python の開発で重要な規格書に相当する文書の記載や、実際の実装方法を SageMath のソースファイルを開いて、これらの表現がどのように Maxima などに引渡されているかを確認しようと思います。このことは何時か SageMath の拡張や類似の環境を構築する際に大きく役立つ筈で、この方針こそが SageMath を育てて行くという観点からも重要なことではないかと私は思っています。

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あり、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。このときに「**Python はどのような言語なのか?**」という問い合わせがあるでしょう。Python がどのような言語であるかは 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**Guido・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、処理すべきデータを、そのデータ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クラス固有の値とメソッドと呼ばれるインスタンスのみに作用でき

^{*1} カレル・チャベックの戯曲「ロボット (R.U.R.)」はRossum'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。

る手続を備えており、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で利用ができます。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。

このようにオブジェクト指向プログラミングについて述べることができます。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「鑄型」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 (εικῶν) の世界」と見なしています (c.f. 「洞窟の比喩」 [20]) *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

*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とグノーシス主義になります。

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, 何故そうしたのかという素朴な疑問に対してはプラトンは歯切れが悪くなります。プラトンによると「デーミウールゴス ($\deltaημιουργός$, 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貪欲な神「エロース (Ἔρως , 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, 醜いものや悪い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 ($\tauο \; εν$, 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「**流出現説**」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悪しき世界の創造, 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** ($\Gammaνωσίς$)」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, Ἡρμηνεύτης) 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 ($\Thetaώθ$)^{*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, 錬金術では「**賢者の石**」^{*8}を実際に手にした人物^{*9}とされて



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 ($\Sigmaοφοκλής$) の「ピロクテーテース ($\Phiιλοκτήτης$)」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} 錬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!

います。そのヘルメス文書の一つの「ポイマンドレース (Poimandres)」[12]によると人間は元来、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}から下位の地上に向います。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、地上にてフュシス ($\varphiύσις$, 自然) 内に写った自分の姿に恋してフュシスと愛欲に陥り、「**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**」その結果、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、星辰には支配され、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压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。このときに新プラトン主義のフィルターを介した形でアリストテレス (Αριστοτέλης , Aristotle) の哲学も部分的に受容されます。そして、アリストテレスの哲学の本格的受容は 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。ただし、イスラム文化圏では当初、プロティノスの著書「エンネアデス」^{*15}の影響を強く受けたために新プラトン主義化したものでした。代表的な哲学者としてスコラ哲学でアビケンナ (Avicenna) で知られるイブン・スィーナを挙げておきます。やがて、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, 訳者) として知られるイブン・ルシュド (ibn rušd) が現れます。西ヨーロッパへの伝播ではアベロエスの方がやや早く、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。このようにキリスト教と古代の間には大

^{*10} この宇宙觀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。そして新たな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。

^{*15} プロティノスの遺稿をエイサゴーゲーの著者であるポルフェリオスが編集したものです。

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。

このように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、我々が扱う対象は「**どのようなものであるかを語れるもの**」で、クラスは対象が「**何であるかを語るもの**」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。

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、こちらはイデアのように超越的ではありません。この概念は、その対象を特徴付ける「**微表**」、つまり、「**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。要するに「**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、それを特徴付ける形や色や機能をまとめたものです。すなわち、概念は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 (εἰδος)**」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。また、概念は「**名辞 (term)**」としても現れますが、名辞はあくまでも概念が載る器であって概念そのものではありません。このよう概念は人間が認知し得る具体的なものから出発するため、人間と無関係にどこかに実在し、永遠不滅で超越的な存在のイデアと異なります。そして概念は我々が対象をどのように語るかということ、つまり、説明規定であり、だからこそ、その対象への理解が深まることで語られることの内容が深まることも理解できるでしょう^{*16}。そして、この「**それが何であるか?**」や「**それがどのようなものであるか?**」といった問いかけにより深く考察した人物がアリストテレス (Αριστοτέλης) です。

ここで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

^{*16} 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実に興味深いことです。

ファエロ・サンティ (Raffaello Santi) の有名な絵画「アテナイの学堂」[47]^{*17}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います。すなわち、イデアは地上の個体から超越し、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^{*18}。

とはいっても、イデアが天界に安住する存在で、概念が地べたを這いずり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。この複数の主語の述語になれる性質のことを「普遍」と呼びます。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、「猫」という「概念」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ためです。

ところで「猫」という括り (あつまり) に対して「三毛猫」、「黒猫」、「白猫」、「虎猫」等の毛並で分類することもできます。これらは「猫」の毛並について述べたもので、こちらは「猫」という概念と比べて個々の猫を区分することにより詳細に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

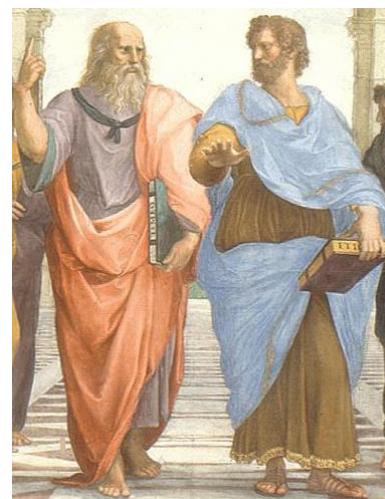


図 2.2 アテネの学堂より: プラトンとアリストテレス

*17 イタリア・ルネサンスにて前述のヘルメス文書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」と呼ばれています。これを契機に新プラトン主義は大きな影響をルネッサンスに与え、「アテナイの学堂」もその影響を受けた作品の一つです。

*18 ここからも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。

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になります。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。そして、「**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。そして種の違いを示す微表(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、上限である最上位の概念を「**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**」等の著作で述べておらず、概念は物事を説明する一方で普遍性を目指す性質を持ちます。

と、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あっても、同じことを主張しているように思えますが、実際、イデア論が秘儀と化した新プラトン主義の哲学者達はアリストテレスを「**師の思想を秘匿するために批判していた**」と捉えるようになります。さらに新プラトン主義の創始者であるプロティノスの弟子のポルフュリオス (*Πορφύριος*, Porphyry of Tyre) が記述した「**手引 (エイサゴー^{ゲー}, Eἰσαγόγη, Isagoge [32])**」^{*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^{*22}、この「**手引**」は(新プラトン主義の) 哲学を学ぶにあたって最初に読むべき本とされ、アリストテレスの哲学に新プラトン主義の夾雜物が含まれる原因のひとつになります。この「**手引き**」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ますが、それが西ローマ帝国崩壊の混乱ののち

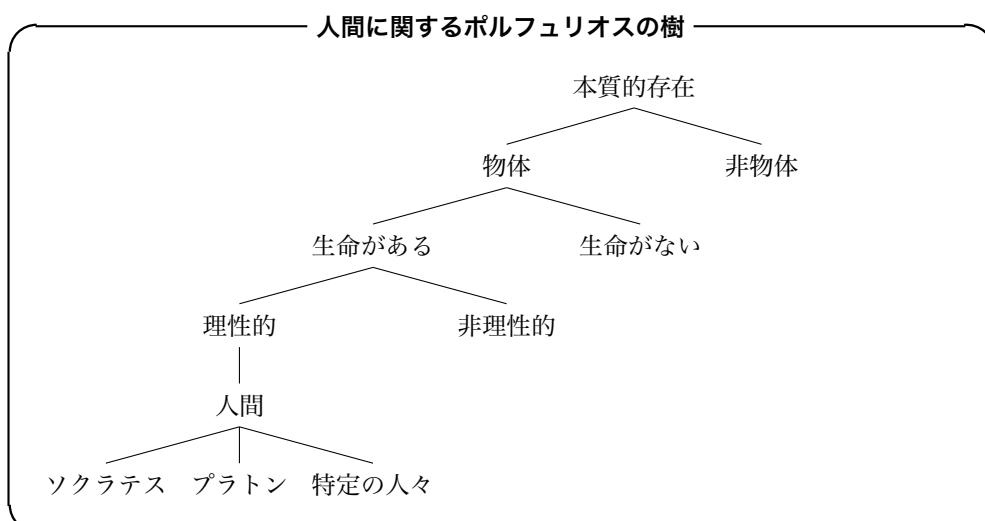
^{*19} 類と種の関係を上位概念と下位概念として述べていますが、「**種類**」という言葉があるように類 (genus) と種 (species) は分類学では属 (genus) と種 (species) に対応し、種は類の直下の概念としての性格があります。

^{*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 [41] とバーンズ (Barnes) がギリシャ語文献から翻訳したもの [32] 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への入門書としての立場で、さらに詳細な解説があります。

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します。この「手引」によると「**ものごとを語る**」ということには「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。ここで「類 (genus)」と「種 (species)」の語源であるギリシャ語の「γένος」と「εἶδος」は共に「形」という意味があり、こ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ます。そして類や種は「**それが何であるか?**」という問への回答であり、「種差」と「**特有性**」、それから「**偶有性**」が「**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」といった属性です。「**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、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が偶有性で、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**」でもあり、感覚されるものは変化するものもあります。その意味でも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。逆に永遠不滅で変化し得ないものは知性の対象となる実体そのものです。

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**」による分類は大きな影響をさまざまな分野に与えています。その一つに「**ポルフュリオスの樹 (Arbor Porphyrianae)**」があります：



ここでは人間に関するポリフェ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「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。逆に名辞 Y について「X を充すものが Y である」とも言える筈です。この操作を「(Y 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、それと外延的な定義として「**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) を与えることを指します。

ところで、キュニコス (犬儒) 派のアンティステネス (*Ἀντισθένες*, 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、「馬は認めても馬性を認めない」と類や種といった概念を認めない立場です。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^{*22}。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。

^{*22} ここで説明規定を後述の圈論での図式と見なすときに圈が完備であればその極限が存在します。つまり、完備な圈であれば個体を説明規定で言い尽くすことができると考えられるでしょう。

2.2.4 形相 ($\epsilon\text{i}\delta\text{o}\varsigma$)

ところでイデア論にもいろいろと問題があります。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と「**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、これらに対して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ことになると今度は「**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、それぞれの瞬間にもイデアがあり、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できたとしても、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** ($\epsilon\text{i}\delta\text{o}\varsigma$, eidos)」はプラトンの「**イデア** ($i\delta\text{e}\alpha$)」のような「**個体から離れた存在** ($\chi\text{o}\rho\iota\sigma\tau\alpha$)」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** ($\text{v}\lambda\eta$)」との「**結合体** ($\sigma\text{u}\nu\text{o}\lambda\text{o}\nu$)」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、現代の科学でも結局、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あり、この流儀はアリストテレスの考察にその源流があることが判ります。だからこそアリストテレスが「万

*23 形而上学 [2] 第一卷九章

「学の祖」と呼ばれるゆえんです。

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。

ただ、この形相があるから金は金であり、鉄が鉄であるとするなら、そ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であり、鍊金術の究極の目的がそれができる「**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のもこのような自然観があつてのことです。

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の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「**事物のあとの普遍**」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「**事物の前の普遍**」と呼ばれる普遍です。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。さらにアリストテレスの範疇論等の入門書であるポルフュリオスの「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*) 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なのか - 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いますが、いわゆる「**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。とはいって「猫」や「三毛猫」に属する個体がとにもかくにも存在するために「**事物のあと普遍**」については中世のスコラ哲学で存在が認められ、もう一方の「**事物の前の普遍**」については否定的です。しかし、普遍論争で門だになった普遍は、概念の存在の在り方や、考察の精密化に伴うものです。

ここで普遍の実在が問題となった背景ですが、アリストテレスが創始し、その後に発展した伝統的論理学で扱う命題には「**存在含意 (external import)**」と呼ばれる条件が付随しています。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一種の暗黙の条件です。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関係しています。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」と置換した場合、日本語の「は」は A と B が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。このように伝統的論理学の命題には主語の隠れた存在性という条件である存在含意が含まれています。たとえば、前述のトマス・アクィナスの「形而上学叙説」[18]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**」とあり、これは命題に存在含意が含まれているために、ある命題が真であれば、その命題の主語も存在しなければなりません。この存在含意は現代の論理学の創始者のフレーゲ (Frege) の概念記法では除外されて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論理学はある意味、千年にも及ぶ神の実在から神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。前述のようにイデアは「**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、これらは「普遍論争」の争点ではなく、「**存在に**

^{*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明治の人々の何気ない凄さを私は感じます。

おける普遍，あるいは「形而上学上の普遍」と呼ばれる普遍が論争の主軸になります。たとえば，「馬」を考えてみましょう。この「馬」という概念の実体は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」と言うことがあっても「A は馬性である」とは言わないので論理学上の普遍ではありません。しかし、「馬」であるためには「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ます。だから、「馬性」と「馬という概念」は別物で、「馬性」は類、種差といった論理学上の普遍でもありませんが、「馬という概念」の属性で、それらに類似したものです。そのために「馬性は馬性に他ならない」と主張するしかなくなります。この議論はイスラム哲学の大家イブン・スィーナ (Avicenna, アビケンナ) が主張^{*25} したもので、この議論は計算機言語の「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の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) は「概念把握された項辞およびそれ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」[13] と概念について述べており、概念を認識する心の働きを加えた考察になっています。本来のアリストテレスの主張は素直に音として発せられた命題を現象として陳述したものと言えるでしょう。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、それこそ重箱の隅をつつくような議論が行われています。このような概念への考察では心が介在するために心理的な表記が現れることになり、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」で、数という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っています。この数の概念に表れる「心理主義」、それと単なる記号とみなすある意味粗雑で素朴な「形式主義」をフレーゲは非難し、「概念記法」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」では論理学から数学を構築するという数学上の「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] によって分類しています。ここで範疇は

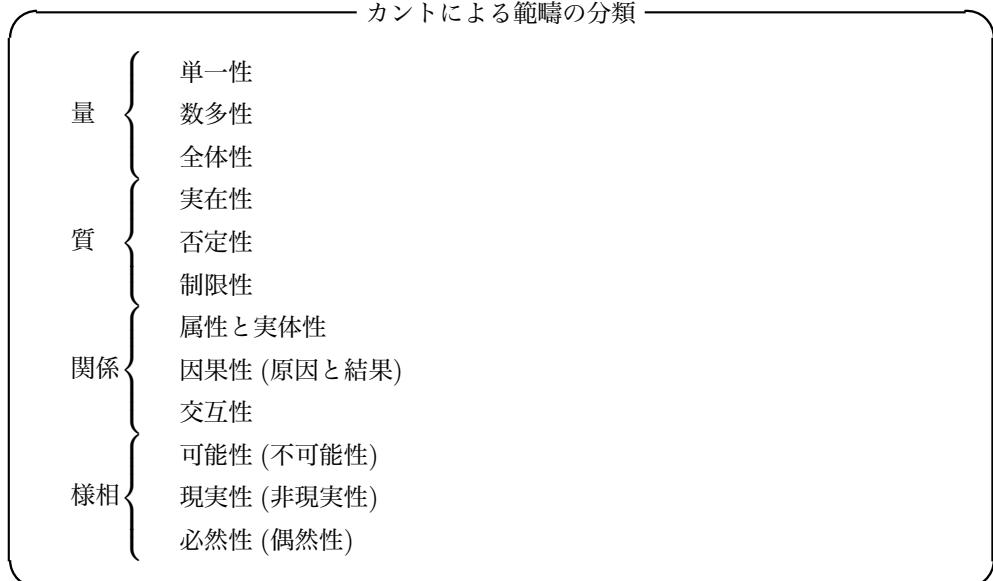
^{*25} Equitas est Equitas tantum.

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 ($\chiατηγορία$) 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 ($\chiατηγορίεσται$) 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：

——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」

ここでの「**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 $\epsilon\iota\omega\alpha$ を名詞化したものに由来し、「**実体**」が訳語として当てられています。このアリストテレスの分類に対してカント (Kant) は量, 質, 関係と様相の 4 級目に分け、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：



アリストテレスの範疇は物事を説明するときの述語の分類ですが、このカントの範疇は後述の物事の判断に対応します。

これら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でのクラスとその構造に深く関わります。

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「**粟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が、内包に関しては下位概念が上位概念より詳細な記述を持ちます。

ここで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」[43] やドイツの数学者フレーゲ (Frege) が独自の図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」[22] といった著作

^{*26} 紀元前 6 世紀頃のクレタのクノッソスの哲学者とのことです。

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」[23]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、ポアンカレはこの非可述的な定義に問題があると述べています ([23],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ますが、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「還元可能性公理」と呼ばれる公理^{*30}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も捉えられます。ここでクラスは、「それがどのようなものなのか」という問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」、

^{*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」[44]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 の表記を用いると $(\exists\varphi).\psi x. \equiv_x .\varphi!x.$ と記述されます。要するに、「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」という都合の良い公理です。

「雨の前に顔を洗うような仕草」, 「ネズミを捕まえる」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入ります。これに似たものとして次に述べる「**継承**」という関係があります。

2.2.9 継承

概念には先程の説明のようにより大きな外延を持つ概念と, より小さな外延を持つ概念があり, より大きな外延を持つ概念を上位概念, 小さな外延を持つ概念のことを下位概念と呼びました。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になります。このことは「上位概念」に含まれる属性をそのまま引き継いで, その概念に新しい「属性」を与えれば新たに「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**」に相当します。

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ができあがることになりかねません。ここで継承関係が一子相伝的な継承であれば, その属性やメソッドが何処から引き継がれたかを探すことが直線的な関係になるために容易です。しかし, 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畏れがあるだけではなく, メソッドや属性の検索という観点からも不利になるためです。実際, クラスを小分けにし過ぎるとどうなるでしょうか? たとえば「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」が入ると素朴に考えても, 猫の毛並だけを

問題にしているのであれば「アジア」, 「東アジア」, 「日本」といった地域はさほど問題にはならず, 冗長でさえあることは理解できるでしょう。さて, このような直系的な継承関係であったとしても, ここで「みけ」が持つ「猫の属性」や「猫の習性」を知りたくなったときにどのようなことが生じるでしょうか? このときに最初に「みけ」が属するクラスから順に調べますね。すると, 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, 後者の継承関係になると「アジアの猫」, 「東アジアの猫」と「日本猫」の三つのクラスを間に挟むため, これらのクラスで検索を行う必要が出てくるのです。このように検索の手間が増えてしまいます。これが複数のクラスを継承する関係であれば, 属性やメソッドの検索により多くの時間を要する可能性が生じることが理解できるでしょう。さらに, この検索の手間の問題だけではなく, この属性やメソッドの検索順位をどのように定めるかで新しいクラスの属性やメソッドが反映されなくなる可能性も出てきます。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であることは言うまでもないでしょう。

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であるかを断定することです。ここで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た12種類の判断について述べることにします。

■量: ここで判断は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 | : 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 |
-

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。

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

■質： ここでの判断は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 」のことを概念「 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： ここでの判断は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： 命題の確實性の関する判断です。

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:

— A. E. I. O. —

- | | | |
|-------------|-----------------------|--------|
| (A) 全称肯定判断: | 「すべての S は P である」 | SAP |
| (E) 全称否定判断: | 「すべての S は P ではない」 | SEP |
| (I) 特称肯定判断: | 「ある S は P である」 | SIP |
| (O) 特称否定判断: | 「ある S は P ではない」 | SO^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、 SAP 等の表記は、これら A.E.I.O. 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、つまり、対応関係が各概念に属する個体全てに分配 (distribute) されるかどうかを表現します。ここで、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にあるとき、つまり、関係が全ての個体に対して「**分配**」されるときで、「**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にあるとは限らない、すなわち、分配され得ないときです。この指示の状況は主辞「 S 」と賓辞「 P 」に「**この**」という言葉をつけることで明瞭になります。たとえば、「全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」と主張できないために「動物」は不周延です。この周延はのちに概念の外延を用いて説明されます。このときに概念 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「**周延される**」と呼んで S^d と表記します。逆に概念 S が概念 P に包含されないときが「**不周延**」であると呼んで S^u と表記します。この周延の状況は主辞と賓辞に対するベン図に類似した「**オイラー図**」でも示せます。

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ることにします。

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で賓辞の「**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	→ 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	→ 全ての神は不老不死である
ある生徒は常識がある	→ 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	→ 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」ということの矛盾概念は「天空を停止」ということです。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」の二種類の生徒に綺麗に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。

A.E.I.O 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 A.E.I.O

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 :	$S^u I^{P^u} \rightarrow S^u O^{\bar{P}^u}$
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。ここで注意することは矛盾概念が日常語のいわゆる「**反対**」ではない事実です。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、「非(不味いラーメン)」には「旨いラーメン」だけではなく「不味くはないがそんなに美味しいもないラーメン」といった微妙なもあるためで、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、との判断と同じ意味を持つ判断を構築する方法で、換質法と同様にとの判断の意味を変えない「**言い換え**」を構成する方法です。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ても、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。では、A.E.I.O. にて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 $S^d A P^u$ であるために主辞と賓辞の位置を入れ替えが可能で、それによって「**すべての S は P である**」から「**ある P は S である**」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 $S^u I P^u$ 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 $S^u O P^d$ となり、 S^u を賓辞の位置に持っ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ないため、この判断の換位はできません。ただし、特称否定判断であっても、換質法で特称肯定判断に変形することは可能です。

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 A.E.I.O

A :	$S^d A P^u$	→	$P^u O P^u$
E :	$S^d E P^d$	→	$P^d E P^d$
I :	$S^u I P^u$	→	$P^u I P^u$
O :	$S^d O P^u$	→	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。ただし、この変形は判断が特殊否定判断になった時点で停止となります。

2.3.4 三段論法

伝統的論理学で後述の三段論法を使った推論を行う際に、二つの前提を「**主語 + 繋辞 + 詳語**」の形式の同じ意味の命題に置換（パラフレーズ）する必要があります。この点は現在も計算機を使う場合は同様で、確実さを重視するのであれば計算機で処理しやすい文章に整形する必要があることは興味深いことです。さて、伝統的論理学の命題は、その主語の存在含意を前提にしているために主語が「**非存在**」のものや存在が不確かな「**仮説**」では「**三段論法 (Syllogism)**」と呼ばれる推論が適用できません。この「**三段論法**」は「**大前提**」、「**小前提**」と「**結論**」の三つの命題から構成される「**推論の規則**」です。次に「**提言三段論法**」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

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[5]。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^{*32}。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見なす大衆によって再三、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^{*33}。

^{*32} 神を νοῦς や ἀρχὴ 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。

^{*33} 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。

ところで、現代の論理学の創始者であるフレーゲは、推論規則として「モダス・ポネンス (Modus Ponens, MP)」のみを公理として導入しています。この MP とは「 A ならば B 」と「 A である」から「 B である」」を推論する規則です。

2.4 集合論について

2.4.1 集合論言語について

クラスはそのインスタンスを成員として含む集まりとしても考えることができます。すると、クラスがどのようなものでなければならないかが問題になります。それに答えるために集合論について述べることにします。まず、クラスに対応する概念は「それが何であるか?」という問に対する説明規定です。つまり、そのものの特徴やものの程度といったことを整理することで概念に到達します。そして、その概念で説明され得るもの集まりが外延です。ここで、フレーゲが概念記法で開始した論理主義は非常に厳密な数学の基礎を与えるかのように見えましたが、どのような命題にも外延が存在しているという素朴な前提のため、「それ自身でないもののあつまり」という命題を考えることで体系内部に矛盾が生じ、このことから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ます。そこからここで述べる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、これらの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の「言語」があり、それが「集合論言語」と呼ばれる言語です。ここでは集合論の論理式で用いる記号について説明しておきましょう：

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: $\vee, \wedge, \supset, \neg, \equiv, \exists, \forall$.
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 ” と対象の同値性を示す記号 “ $=$ ” の他は論理式の論理和 “ \vee ”, 論理積 “ \wedge ”, 否定 “ \neg ” と含意 “ \supset ”, 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。またこの本では a を b で定義することを記号 “ $\stackrel{\text{Def.}}{=}$ ” を導入して ‘ $a \stackrel{\text{Def.}}{=} b$ ’ 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 $A \equiv B \stackrel{\text{Def.}}{=} (A \subset B) \wedge (B \subset A)$ で定義します。また、集合の重要な記号に記号 “ \cup ” や記号 “ \cap ” 等がありますが、これらの記号は後述の集合論の公理から順に定めてゆきますが、これらの集合論の記号系に含まれる記号を用いて、集合論で用いられる論理式を次の形成規則に沿って定義できます:

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 §3.4 で述べる BNF 記法に基く表記で表現可能です。なお、以降の説明では集合論言語 \mathcal{L} の記号や形成方法を用いて、記号 “ \cup ” や記号 “ \cap ” といった記号を必要に応じて定義します。ただし、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、個々の論理式の意味や意義について述べたものではなく、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はありません。それを規定するものが集合論の公理系になります。

2.4.2 集合論の公理系

この集合論言語 \mathcal{L} を用いて集合論の公理系を以下に記しておきましょう:

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 | $\forall x \in u (\neg x = \emptyset) \wedge \forall x, y \in u (\neg x = y \supset x \cap y = \emptyset) \supset \exists v \forall x \in u \exists t (t \in x \wedge t \in v)$ |

では「**集合論の公理系**」で挙げた公理について順番に解説しましょう。

■外延性公理 (Axiom of extensionary) 外延で集合が一意に定まることを保証する公理で、この公理によって以降の公理から存在を保証される集合が一意に定まります。なお、集合の外延は $\{a, b, c, d\}$ のように括弧 $\{ \}$ に集合の構成元、成分あるいは元を a, b, c, d と区切記号 “,” を使って列記します。なお、括弧 $\{ \}$ 内の成分の順番は集合の定義に影響しません。

■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 $\{x, y\}$ と記述します。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、その成分が 3 以上の順序対が構成できます。この順序対の構成方法を以下に纏めておきます：

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 の成分と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 $x \cup y \stackrel{\text{Def.}}{=} \cup\{x, y\}$ によって式 $x \cup y$ 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

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

で定義される記号で、その意味は集合 x の任意の成分を外延として持つ集合の存在を保証します。この公理と外延性公理から唯一存在が保証される集合を「**幂集合**」と呼び、 $\mathfrak{P}(x)$ で集合 x の幂集合を表記します。

■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が保証される集合を「**空集合**」と呼び、記号 “ \emptyset ” で空集合を表記します。

■無限集合公理 (Axiom of infinity set) 無限集合の一つの作り方を定める公理で, v が集合のときに $\emptyset \cup \{v\}$ が集合となることを保証します。さて、空集合公理から空集合 \emptyset は集合です。この空集合と無限集合公理から $\emptyset \cup \{\emptyset\}$ も集合になることが保証されます。さらに $\emptyset \cup \{\emptyset \cup \{\emptyset\}\}$ を構成すると、これ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、この集合の無限列を自然数の定義とすることができます：

自然数の定義

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

この定義では、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自然数が構成可能、すなわち、自然数を体系内に包含していると言えます。

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

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

ここで定義した自然数に「大小関係」を導入することができます。つまり、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 $a \leq b \stackrel{\text{Def.}}{=} a \in b \vee a = b$ で定義します。それによって定義した自然数に大小関係が自然に導入できます。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「後継」、あるいは「後者」と呼びます^{*34}。この自然数については順序数で再度触れます。

*34 自然数の後者関係についてはフレーゲの「概念記法」[21] ではじめて厳密に述べられています。

■置換公理図式 (Axiom schema of replacement) 図式のはじめの $\phi(x, y) = \phi(x, z)$ かつ $y = z$ を $F(x) = y$ で置換えると集合 x の函数 F による像も集合になる公理になると同時に、集合そのものに制約を入れる公理であると言えます。この公理はフレンケルが導入しましたが、本来のツエルメロが入れていた公理は「**分出公理 (Axiom of displacement)**」と呼ばれる次の公理です：

—— 分出公理 (Axiom of displacement) ———

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

分出公理の意味は素朴集合論のように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、置換公理図式から導くことができます。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 $\{x : \phi(x)\}$ を考えてみましょう。この外延はその元がある集合の元であると保証されないために分離公理から集合であるとは断言できません。このような命題の外延のことを「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からです。では素朴集合論で問題となった「**ラッセルの逆理**」をもう一度考えてみましょう。この逆理の本質は外延 $\{x : x \notin x\}$ が素朴集合論の集合から排除できないことで、分出公理を認めると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。しかし、 $x \notin x$ より自分自身を包含しない集合が構成できないため、この命題の外延は集合にならず、この体系から除外できます。

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」[17] 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 $a \in a$ の外延が集合から排除され、このことから集合と集合の元を区別する公理になります。また、この公理から $\dots, x_3 \in x_2, x_2 \in x_1, x_1 \in x_0$ 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点に軽く触れておきましょう。最初に空集合公理と無限公理の二つを認めると自然数が導入できます。このときに大小関係も前述の方法で関係 \in から導入することができますが、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^{*35}。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 [17].

■選択公理 (Axiom of choice) 空集合と異なる集合から、その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” に該当する公理です。この選択公理は他の集

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

合論の各公理から独立した公理で、この公理なしでも「数学」を構築することができます。この選択公理は何かと便利な公理ですが、この公理から非常に厄介な逆理が幾つか導きだせることができます。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：

$$\left\{ \begin{array}{l} A = A_1 \cup A_2 \cup \dots \cup A_n \\ B = B_1 \cup B_2 \cup \dots \cup B_n \end{array} \right.$$

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

この逆理を適用すると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と、それで地球が覆える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^{*36}。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、この公理を認めたときの御利益が圧倒的に大きな公理です。なお、この公理を認めない場合、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以下に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エルメロ・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。

^{*36} 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。

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 $\text{Ord}(u)$ を充すので順序数になります。そして、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 $A \subset A$, 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

つぎに, 重要な概念として, 函数と関係について述べることにします。まず, 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**」であることです。この不飽和の意味ですが, 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 (Church) 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」であれば x^2 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 は $f(\xi, \zeta)$ 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 Curry 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、その構造に关心がない限り、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ブル、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 といった真理値に対応する値を返す函数として捉えることができます。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 (True) か偽 (False) の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 “ \sim ” 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 $\mathbf{N} \times (\mathbf{N} \setminus \{0\})$ 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 が半順序であり、さらに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: $(a R b) \vee (b R a)$

ここで具体的な例を挙げておきましょう。まず、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ではないことが判ります。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ことから大小関係 \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

SageMath は数学的対象を扱いますが、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、演算とその演算の規則を体系的にまとめた性質があります。この性質を算数から導き出してみましょう。まず、算数で最初に扱う数は自然数 $1, 2, 3, \dots$ です。この自然数の計算処理として足算、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、これらの演算は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

$$1 \div 2 = 0 \text{ あまり } 1$$

と商と剰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剰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るとどうなるでしょうか？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**」と呼びます、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

こ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、 $(\mathbb{N}, +)$ ， (\mathbb{N}, \times) と (\mathbb{N}, \div) はマグマになります^{*37}。ここで $(S, *)$ がマグマであるということは集合 S に閉じた二項演算 “ $*$ ” が存在することを意味するだけで、他にどのような性質を持つものは述べていませんが、このマグマという概念を導入したことで、これらの演算がマグマという大枠で語ることができます。

では、これらの演算の詳細を眺めてみましょう。ここで足算 “ $+$ ”，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。この括弧 “ $()$ ” で指示された計算順序に依存しないという性質は「結合律を充たす」という重要な性質です。つまり、集合 S の閉じた演算 “ $*$ ” で任意の $a, b, c \in S$ に対して $(a * b) * c = a * (b * c)$ を充す性質です。ところで、割算 “ \div ” は $(12 \div 6) \div 2 \neq 12 \div (6 \div 2)$ であるために結合律を充たしません。このことから割算が本質的に足算や割算と異なる演算であることが判ります。そこで、閉じた演算を持つという性質と結合律を充たすという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： $a * (b * c) = (a * b) * c$.

具体的には、自然数 \mathbb{N} に対して $(\mathbb{N}, +)$ と (\mathbb{N}, \times) は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はありません。この半群という概念を導入したことで割

^{*37} (\mathbb{N}, \div) は商のみを返す場合です。

算が足算と掛算と本質的に異なる演算として分類できました。このように個々の数学的对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。また、この区分では同じ集合で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 “+” にするか割算 “÷” にするかで半群、あるいはマグマといった異なる概念に分類されます。

ここで足算 “+” と掛算 “×” 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 “÷” は $4 \div 2 \neq 2 \div 4$ であるために可換ではありません。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。

さらに $(\mathbb{N}, +)$ の 0, (\mathbb{N}, \times) の 1 は共に面白い性質をもっています。0 については $a + 0 = 0 + a = a$, 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、単位元を持つ半群のことを「**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。

さて、小学生も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るという操作で、分母が 1 のときは分子のみの表記、整数になります。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^{*38}と言えます。それから分数は演算 “×” 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 “×” 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

^{*38} 同値関係と呼ばれる関係になります。

(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である单系を「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。だから、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) になると群になります。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 “+” と掛算 “ \times ” 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} は足算と掛算という二つの演算を持ち、演算 “+” なら可換群ですが、演算 “ \times ” は演算 “+” の単位元 0 を除く集合 $\mathbb{Z} - \{0\}$ は单系で、群にはなりません。それから足算と掛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算 “ \times ” 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 “*” を持ち、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:

$$1 \quad a * (b + c) = a * b + a * c$$

$$2 \quad (a + b) * c = a * c + b * c$$

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 $(2 \times 3) \times \frac{2}{5} = 2 \times \left(3 \times \frac{2}{5} \right)$ から判るようにな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 “ \times ” の単位元 1 は有理数をそのままにする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

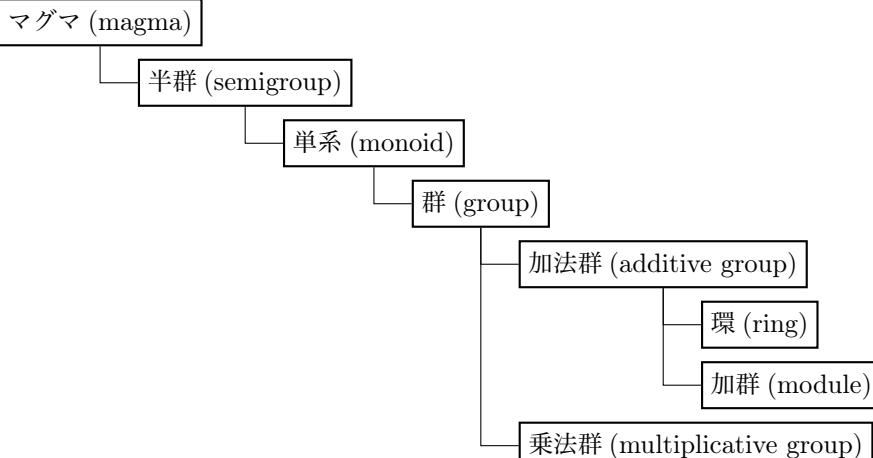
- 集合 $(A, +)$ は可換群である.
- 環 R と集合 A に写像 $*: A \times R \rightarrow A$ が存在する.
- $(x + y) * r = x * r + y * r$
- $x * (r + s) = x * r + x * s$
- $x * (s * r) = (x * s) * r$
- $x * 1 = x$

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 -加群の例になります。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 -加群の条件を満たすために、実ベクトル空間も R -加群であることが分かります。

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、これらの序列はどちらかと言えば発生順に並べた樹形図を次に示しておきましょう：

代数的構造の階層



この樹形図は後述の SageMath の数学的オブジェクト SageObject との関連で再度触れます。

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。この数学用語の「**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たとのことです [1]^{*39}。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[39]によると「**Category**」という言葉はアリストテレスとカント (Kant) の「**カテゴリー論**」に、「**functor**」はカルナップ (Carnap) の著作に由来すると述べています。

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 “=” を用いています。ここでメタグラフの対象は集合の元、矢は写像をそれぞれ抽象化したもので、それらが具体的にどのようなものであるかを述べておらず、あくまでも形式的な定義です。実際、対象とそれらの間の二

^{*39} その注釈書としてポルフェリオス (**Πορφύριος**, Porphyry of Tyre) のエイサゴーゲー [41] が非常に有名です。

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ことも保障されません。

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“ \in ”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} ’ と表記することでお矢 $f : A \rightarrow B$ 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論理学での「**繫辞 (copula)**」と同様の機能を持つものと考えることができます。ここでの連辞は命題「 A は B である」の中の「... は... である」という主語と述語の二項間の関係を表現する機能を持ち、伝統的論理学では繫辭こそが命題を構成するものと考えられていたとのことです [4]。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、変数が二個以上の函数を特に「**関係 (relation)**」、変数の位置を「**項位置**」と呼び、三項以上の変数を持つ関係は二項変数函数を適宜利用することで表現可能なために二項関係を考察すればよい^{*40}と述べ、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。この函数のグラフは点 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、そのうしろの成分が 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 $\langle x, f(x) \rangle$ として記述しましょう。このときには

^{*40} 要するに函数のカリー化 (currying)

グラフ全体は $\{\langle x, f(x) \rangle : x \in A\}$ で外延として記述できます。ここで対象 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。このように XY-グラフは順序対の集合としての性格を持つことになります。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 $\{\langle x, y \rangle : x \in A \wedge y \in B \wedge f x = y\}$ として矢を考えることができます。このときに $x \in A$ に対応する対象 B の元を $f(x)$ あるいは $f x$ と表記し、 $y = f x$ のときに $x \mapsto y$ と表記することで $x \in A$ と $y \in B$ が矢 f を仲立とする関係にあることを示します。ここで空集合 \emptyset を始域とするメタグラフの矢 f を考えると、この矢は空集合 \emptyset でなければならないことが判ります。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、この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、このような操作が可能かどうか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 $(f \circ g) \circ h = \langle \langle f, g \rangle, h \rangle$ をそれぞれ構築できます。これらが一致するかどうかは一般的に正しいと言えませんが、これらの矢の合成が一致すること、すなわち、 $f \circ (g \circ h) = (f \circ g) \circ h$ であることを「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 “()” が不要になり、単に $f \circ g \circ h$ 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), 矢を向きを持つ辺 (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。これらのように図式は対象、矢と省略記号 “...”

で構成され、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ことから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らないことが判ります。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と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：

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

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。ここで矢の合成 $g \circ f$ を矢 h の「分解 (factorization)」とも言います。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

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

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。このように圏論で中心になるものは対象ではなく、あくまでも矢や後述の函手や自然変換に重点があります。

2.7.5 単射, 全射, 同型

写像には単射, 全射, 同相といった性質があります。ところで矢は写像を抽象化したもののため, 矢にも「**単射 (mono)**」, 「**全射 (epi)**」と「**同型 (iso)**」といった性質があります:

—— 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.

ここで矢 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 (cancellable), つまり, 「**左簡約可能 (left cancellable) な矢**」であるとも呼びます。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**」であるとも呼びます。

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, 同射に対応します。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, 圈の対象が通常は集合であるとは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 epi)**: $f \circ g = \text{id}_B$ 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 epi)**」と呼びます.

したがって, 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

製全射かつ单射であることと同値であることが判ります。

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「**単系 (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。そこで、この対象と矢がのあつまりが ZFC 公理系で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「**圏**」と呼び、これらを中心に考察することにします:

—— 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}
- 函数 $\text{dom}, \text{cod}: \mathbf{A} \rightarrow \mathbf{O}$
 $f \in \mathbf{A}$ 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

*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「**籠 (quiver)**」と呼びます。

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は圏と呼ぶ:

- 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単系としての構造を持ちます。

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、矢を全单射写像とする圏 \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、対象はそのまま対象に写し、矢に対しては、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 に対応させるものの、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 $g \circ f$ に対しては矢 $(g \circ f)^{\text{op}}$ 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 ${}^{\text{op}}$ で得られるものを**「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、矢の合成に対しては、矢の双対の順番が逆順になります。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**」と呼び、 \mathcal{C}^{op} と表記します。また、 $\mathcal{C} = \mathcal{C}^{\text{op}}$ を充たす圏 \mathcal{C} のことを「**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)

$$\bullet \rightarrow \bullet$$
- **3** = $\begin{array}{c} \bullet \\ \searrow \downarrow \\ \bullet \end{array}$ (対象が三つで同一矢の他の矢が三つの圏)
- **4** = • \rightrightarrows • (対象が二つで同一矢と異なる矢が二つ)

なお、ここでの **0** と **1** は圏の名称であり、後述の始対象 0 や終対象 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 $A \xrightarrow{\text{id}_A} A$ に繋辞の「... 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 (*Ἀντισθένες*) 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、矢が基点を基点に写す連続写像
- **Cat**: 対象が圏、矢が後述の函手

^{*42} 形而上学 [2] 5 卷 29 章, 1024b34

- \mathcal{C}^{op} : 圈 \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：

————— グロタンディーク宇宙で許容される演算 —————

対集合 { }	$\{u, v\} \stackrel{\text{Def.}}{=} \{(x, y) : x \in u \wedge y \in v\}$
順序対 ⟨ ⟩	$\langle u, v \rangle \stackrel{\text{Def.}}{=} \{\{u\}, \{u, v\}\}$
直積 ×	$u \times v \stackrel{\text{Def.}}{=} \{\langle x, y \rangle : x \in u \wedge y \in v\}$
幂集合 \mathfrak{P}	$\mathfrak{P}(u) \stackrel{\text{Def.}}{=} \{v : v \subset u\}$
和集合 ∪	$\cup u \stackrel{\text{Def.}}{=} \{x x \in u\}$

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 自身は 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がありません。さらに、この 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 の元のことを、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

次に重要な対象として「**始対象 (initial object)**」と「**終対象 (terminal object)**」を定義します：

—— 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。

なお、始対象、終対象は圈によって異なり、これらが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、同型になる矢で一意に定まります。つまり、二つの対照 A, B が圈 \mathcal{C} の始対象(終対象)であれば $A = B$ とは限りませんが $A \cong B$ を充たします。また、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 にそれぞれ写されます。このことから 始対象 0 と終対象 1 が双対関係にあることが分かります。また、圈 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、これを一般化して圈 \mathcal{C} の始対象 1 から対象 $A \in \text{Ob}\mathcal{C}$ への矢を圈 \mathcal{C} の対象 A の「**要素 (element)**」と呼びます。

具体的な例を挙げておきます。圈 Set の始対象 0 は空集合 \emptyset のみです。実際、

$A \in \text{ObSet}$ 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}. 圈 **Rng** の始対象 0 は整数 **Z** になりますが, 圈 **Field** には始対象 0 が存在しません. また, **Set**, **Grp**, **Ab** と **Rng** の各圏の終対象 1 は成分が一つの集合: 「**シングルトン (singleton)**」になります. 特に圏 **Set** では矢 $1 \rightarrow A$ が集合 A の成分を定めます^{*44}. しかし, 圈 **Field** には終対象 1 も存在しません. 圈 **Set** のように始対象 0 と終対象 1 の双方が存在する圏がある一方で圏 **Field** のように始対象も終対象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, 圈 \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良いでしょう. すなわち, 圈 \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:

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

この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)**」 $\text{id}_{\mathcal{C}} : \mathcal{C} \rightarrow \mathcal{C}$ がまず挙げら

*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 圈 **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.

れます。つぎに圏 **Grp** から圏 **Set** への函手で、群の演算に関する事柄を無視して群を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「構造を忘れてしまう函手」のことを「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 $F C \in \text{Ob}\mathcal{D}$
- 圏 \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、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	$\text{id}_A^{\text{op}} = \text{id}_A$
$g \circ f$	$(g \circ f)^{\text{op}} = f^{\text{op}} \circ g^{\text{op}}$
$f: \text{mono}$	$f^{\text{op}}: \text{epi}$
$A:$ 始対象	$A:$ 終対象

また双対函手の性質で $\mathcal{C}^{\text{op}}{}^{\text{op}} = \mathcal{C}$ 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単射になる場合です。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 $\text{Hom}_{\mathcal{C}}(A, B) \xrightarrow{F} \text{Hom}_{\mathcal{D}}(FB, FA)$ が単射になるときで、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。

圏 \mathcal{C} から圏 \mathcal{D} への二つの函手に対しては「**自然変換**」と呼ばれる対応関係が考えられます。まず、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。このように函手 F から G への写像 α で右下の図式を可換にする写像のことを「**自然変換**」と呼び、 $\alpha : F \rightarrow G$ と表記します：

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

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、

圏 \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

$$\begin{array}{ccc} A & \xrightarrow{\quad H_A(A) \quad} & F(A) \\ \downarrow f & \downarrow H_A(f) & \downarrow F(f) \\ B & \xrightarrow{\quad H_A(B) \quad} & F(B) \\ & \xrightarrow{\quad \eta_B \quad} & \end{array}$$

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 $f = H_A(f)(\text{id}_A)$ であることと、この可換図式から $\eta(H_A(f)(\text{id}_A)) = F(f)(\eta_A(\text{id}_A))$ 。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

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

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、 $\theta(a^*) = a$ するために写像 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、 $\text{Nat}(\text{H}^A, F) \cong F(A)$ についても $\text{H}^A(B) = \text{Hom}_{\mathcal{C}}(B, A) = (\text{Hom}_{\mathcal{C}}(A, B))^{\text{op}}$ 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して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 $Y(A) = \text{H}^A$, $A \xrightarrow{f} B$ と $C \xrightarrow{g} A$ に対しては $Y(f)_A(g) = f \circ g$ 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) を定義しなおしておきましょう：

— 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。

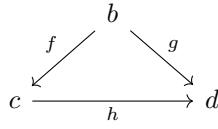
この定義を図式化したものを以下に示しておきます：

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

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。

2.7.11 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



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 $\langle c, f \rangle$ で表記します。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。それから $\langle f, d \rangle$ と $\langle f', d' \rangle$ を考えます。ここで矢 $d \xrightarrow{h} d'$ で次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：

$$\mathcal{C} \xrightarrow{T} \mathcal{C} \xleftarrow{S} \mathcal{D}$$

を考え、 $d \in \mathcal{D}, e \in \mathcal{E}$ 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 圈 \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の可換図式も

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

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。

2.7.12 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と^{*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

^{*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。

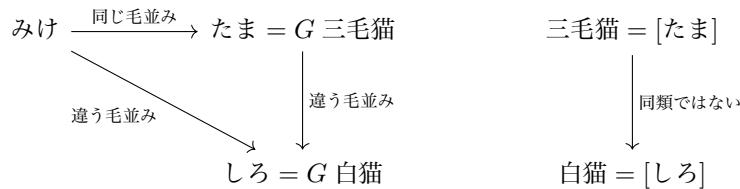
類概念を「毛並み」という種差で分類しようとする意図で導入された概念で、猫は毛並みによって「白猫」、「黒猫」、「虎猫」、「三毛猫」,... と分類できます。ところで、集団の分類とは、その集団の構成員の特徴で纏めた集団を作ることですが、この特徴を種差とすると類概念を種差に対応する種概念で分割することになります。ここで、類概念を種概念で分割したときに、ある種から典型的な構成員を種概念の代表として一つ取り出し、他と種概念と比較することはよくあることです。このときに種の代表とその種の構成員は、その分類の観点から「**同じもの**」として見なされます。このことを整理すると、まず、集合 T をある観点で複数の部分集合 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。

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。

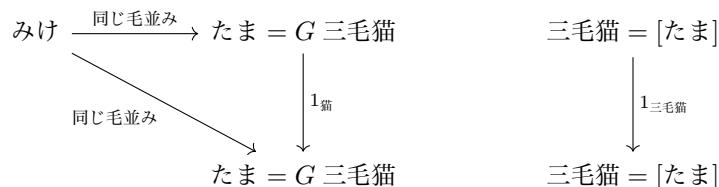
まず、「**猫全体の集合**」を \mathcal{C} 、「**猫の分類の集合**」をここでは $\mathcal{D} = \{\text{三毛猫, 白猫, 黒猫, 虎猫, 斑猫, その他}\}$ とします。このときに \mathcal{C} から \mathcal{D} への写像として包含写像から誘導される写像 F 、 \mathcal{D} から \mathcal{C} への写像として代表を返す写像 G が考えられます。具体的には F は「みけ」を $[みけ]$ に写す写像で、 G は $[みけ]$ を「みけ」に写す写像にそれぞれ対応します。ここで A と B が猫 ($A, B \in \mathcal{C}$) 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} を圈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:



この例は左側の可換図式が猫全体:図 \mathcal{C} での話で、右側が猫の分類:図 \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、図 \mathcal{D} の矢「違う毛並み」は図 \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 によって図 \mathcal{D} の矢:「同じ毛並み」は図 \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:



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:

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

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:

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

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 $p : C \rightarrow X$ 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 [39] 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 $X \in \mathbf{Set}$ 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 **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:

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

2.7.13 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。

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

圈の積を可換図式として表現することができます：

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

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

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

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を意味します。なお、圈を小集合の圈 **Set**

^{*47} この普遍性は後述の極限に関連します。

とするときに対象の積は $A \times B = \{(x, y) | x \in A \wedge y \in B\}$ 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

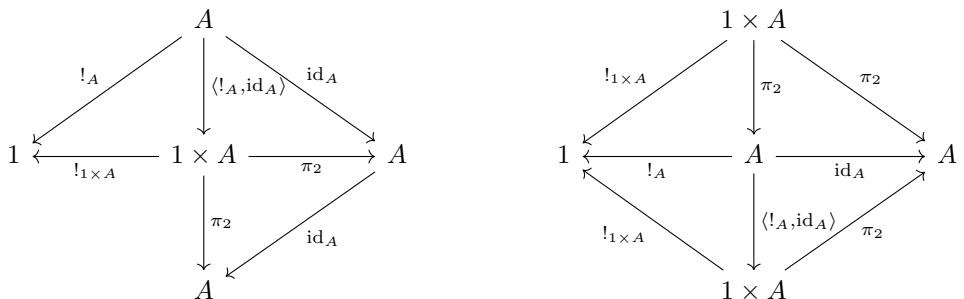
- 可換性: $A \times B \cong B \times A$
- 結合律: $A \times (B \times C) \cong (A \times B) \times C$

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との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：



まず、左の可換図式から $\pi_2 \circ \langle !_A, \text{id}_A \rangle = \text{id}_A$ 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 $1_{\times A} = !_A \circ \pi_2$ 、このことから $\langle !_A, \text{id}_A \rangle \circ \pi_2 = \text{id}_{1 \times A}$ 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 $\langle \text{id}_A, \text{id}_A \rangle$ を特に対角矢と呼び記号 Δ_A で表記される。

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

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：

$$\begin{array}{ccccc} & & C & & \\ & \nearrow f & \uparrow \langle f, g \rangle & \searrow g & \\ A & \xrightarrow{i_1} & A + B & \xleftarrow{i_2} & B \end{array}$$

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 $\text{Ob}\mathcal{C}$ に単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：

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

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：

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

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

に $\text{Hom}(C \times A, B)$ 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。

2.7.14 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 $A \xrightarrow[g]{f} B$ 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:

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

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:

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

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 $A \xrightarrow[g]{f} B$ 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：

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

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

2.7.15 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。

$$\begin{array}{ccccc} & E & & & \\ & \swarrow k & \searrow l & & \\ P & \xrightarrow{f'} & B & & \\ \downarrow g' & & \downarrow g & & \\ A & \xrightarrow{f} & C & & \\ & \searrow h & \swarrow & & \\ & & P & \xrightarrow{f'} & B \\ & & \downarrow g' & & \downarrow g \\ & & A & \xrightarrow{f} & C \end{array}$$

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 \mathbf{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:

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

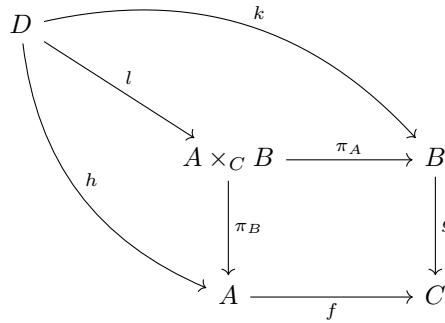
- 次の可換図式において、右側の四角と外側の四角の図式がそれぞれ引き戻しであれば左側の四角の図式も引き戻しになり、また、右側と左側の図式が引き戻しになるときには外側の四角の図式も引き戻しになります:

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

さて、圏 **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 $A, B, C \in \mathbf{Set}$ に対しては $A \times_C B = \{(a, b) \in A \times B \mid f(a) = g(b)\}$ 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



をよくよく考えると次の集合の積と等化が現われます:

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

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。この例は集合の圏 **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 $A \xrightarrow{f} C \xleftarrow{g} B$ に対して次の等化が考えられます：

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

このときには

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

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：

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

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：

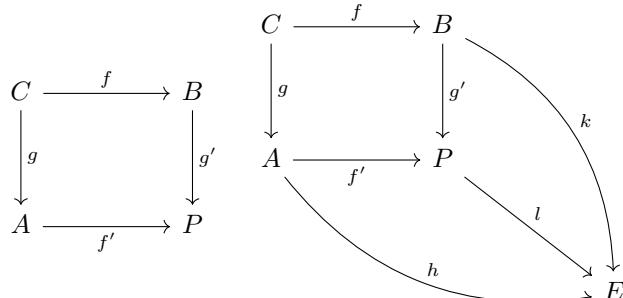
$$\begin{array}{ccccc} & & k & & \\ & E & \swarrow & \searrow & \\ & & \langle h k \rangle & & \\ & & \downarrow & & \\ D & \xrightarrow{e} & A \times B & \xrightarrow{\pi_A} & B \\ \downarrow l & & \downarrow \pi_B & & \downarrow g \\ A & \xrightarrow{f} & C & & \end{array}$$

このことから $A \xleftarrow{\pi_A \circ e} D \xrightarrow{\pi_B \circ e} B$ 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。

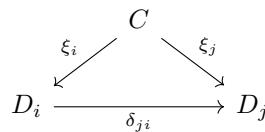
極限と余極限

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が、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



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:

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

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

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

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:

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

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

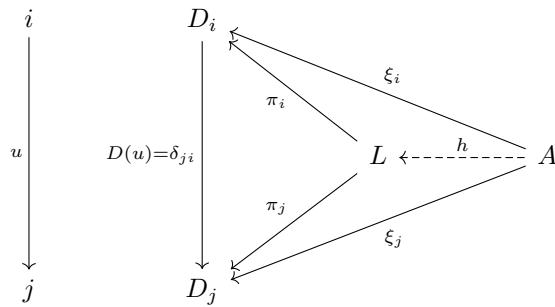
$$\begin{array}{ccc} i & & \\ \downarrow u & & \\ j & & \\ C & \begin{array}{c} \nearrow \pi_i \\ \searrow \pi_j \end{array} & \\ & D_i & \\ & \downarrow Du & \\ & D_j & \\ & \begin{array}{c} \nearrow \rho_i \\ \searrow \rho_j \end{array} & \\ & C & \end{array}$$

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 $\lim_{\rightarrow} D$ 、あるいは $\text{colim } D$ と表記します。

$$\begin{array}{ccccc}
 & i & & & \\
 & \downarrow u & & & \\
 & D(u)=\delta_{ji} & & & \\
 & \downarrow & & & \\
 j & & D_i & \searrow \zeta_i & \\
 & & \downarrow \rho_i & & \\
 & & L & \dashrightarrow k & A \\
 & & \swarrow \zeta_j & & \\
 & & D_j & \nearrow \rho_j &
 \end{array}$$

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、押し出しはその双対の余極限になります。次に対象の積と等価については、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が、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

$$\begin{array}{ccc}
 D_1 & \xrightarrow{\begin{array}{c} F=Df \\ G=Dg \end{array}} & D_2 \\
 \nwarrow \pi_1 & & \searrow \pi_2 \\
 & \lim D &
 \end{array}$$

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

$$\begin{array}{ccc}
 D_1 & \xrightarrow{\begin{array}{c} F=Df \\ G=Dg \end{array}} & D_2 \\
 \nwarrow \pi_1 & & \searrow \pi_2 \\
 & \lim D & \\
 \downarrow h & & \\
 A & &
 \end{array}$$

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:

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

つまり、 $\lim D \xrightarrow{\pi_1} D_1$ 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、圏 \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。

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ない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：

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

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

$$\begin{array}{ccccc} & & A & & \\ & \swarrow \pi_i \circ h & & \searrow \pi_j \circ h & \\ D_i & \xleftarrow{Du} & & \xrightarrow{Dj} & D_j \end{array}$$

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:

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

そこでまず $\pi \circ \xi$ 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

$$\begin{array}{ccccc} & & B & & \\ & \swarrow \pi_i \circ \xi & & \searrow \pi_j \circ \xi & \\ D_i & \xrightarrow{Du} & & & D_j \end{array}$$

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 $Du \circ \pi_i \circ \xi = Du \circ \pi_i \circ h \circ k$, これと $\pi \circ h$ が図式 D の錐であることから $Du \circ \pi_i \circ h \circ k = \pi_j \circ h \circ k = \pi_j \circ \xi$, 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

$$\begin{array}{ccccc} & & B & & \\ & \swarrow \pi_i \circ \xi & \downarrow k & \searrow \pi_j \circ \xi & \\ & A & & & \\ D_i & \xrightarrow{Du} & \pi_i \circ h & \pi_j \circ h & D_j \end{array}$$

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 $A \xrightarrow{(f,g)} B \times B \xleftarrow{\Delta_B} B$ の引き戻し：

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

から $A \xrightarrow[g]{f} B$ の等化 $E \xrightarrow[e]{\Delta_B} A$ 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、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。

^{*48} τόποι の複数形です。

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は「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。

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

この図式の意味ですが、ここで圏 \mathcal{C} を小集合の圏 \mathbf{Set} で説明しておきましょう。このとき、対象 A, B の関係は $A \subset B$ として考えることができます。それから Ω を $\{\text{True}, \text{False}\}$ の二つの真理値の集合としましょう。次に \top は $A \xrightarrow{!_A} 1$ が一意に存在するために単射で、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 との間の矢が単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：

^{*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。

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:

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

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 $\vdash a = a$ すなわち, $\forall x(x = x)$ を真 (True), 概念記法で $\vdash a \neq a$, すなわち, $\forall x(x \neq x)$ を偽 (False) として定義していますが [22], 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ぬために, 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, 「**存在するものを作り出すと言ふこと**, あるいは**存在しないものを作り出すと言ふこと**」が真で, 「**存在するものを作り出さないと言ふこと**, あるいは**存在しないものを作り出すと言ふこと**」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。

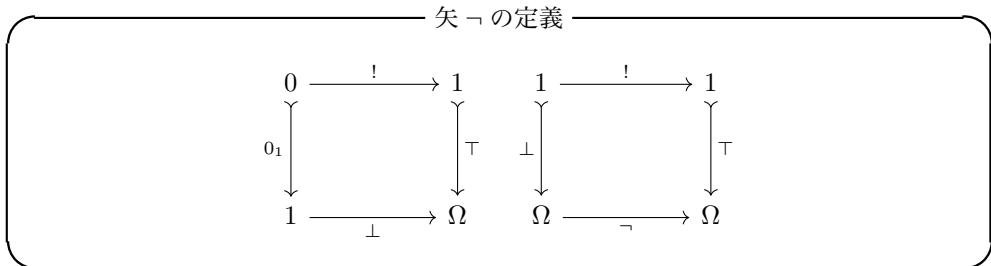
それから圏 \mathcal{C} に部分対象分類子 Ω と付随する矢 $1 \xrightarrow{\top} \Omega$ が存在するときに連言 “ \wedge ”も同値性 “ $=_A$ ” と同様に定義できます:

矢 \wedge の定義

$$\begin{array}{ccc} 1 & \xrightarrow{!} & 1 \\ \langle \top, \top \rangle \downarrow & & \downarrow \top \\ \Omega \times \Omega & \xrightarrow{\wedge} & \Omega \end{array}$$

これを小集合の圏 **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 (True, 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。さらに圈 \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、右側の図式が矢 \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圈 **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、圈 **Set** で始対象 $0 = \emptyset$ 、終対象 $1 = \{*\}$ であるために $0_1 = \emptyset$ 、 $\perp \emptyset = \text{True}$ 、 $\perp \{*\} = \text{False}$ 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。

このように圈 \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、さらに圈 \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圈です。

2.8.3 基本トポス

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

基本トポスの定義

1. 圈 **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. **E** には部分対象分類子 Ω が存在する。

1., 2., 3. を充す圈を「**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.

トポスになる圈として代表的なものとして小集合から構成される圈 \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.

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

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:

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

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:

$$\begin{array}{ccc}
 B^A \times A & \xrightarrow{\text{ev}} & B \\
 \downarrow h' \times \text{id}_A & \searrow h \circ \text{ev} & \downarrow h \\
 B^A \times A & \xrightarrow[\text{ev}]{} & B
 \end{array}$$

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:

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

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 **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:

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

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:

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

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:

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

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.

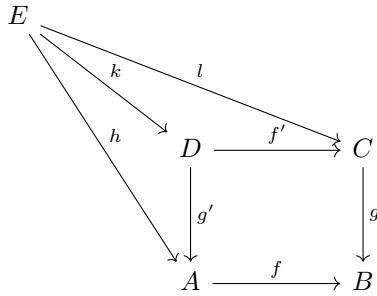
まず, 圈 \mathbf{E} が基本トポスであればコンマ圏 $(\mathbf{E} \downarrow B)$ も基本トポスであることを示しましょう. そのためにはコンマ圏 $(\mathbf{E} \downarrow B)$ で終対象の存在, 積の存在, 幂の存在と部分分類子の存在を示さなければなりません. これらの事項を順番に確認しましょう.

■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:

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

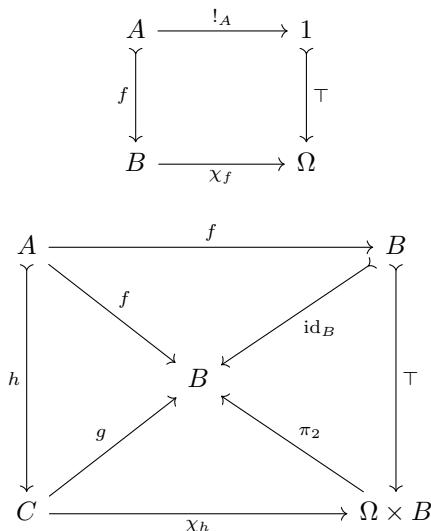
このことから $B \xrightarrow{\text{id}_B} B$ がコンマ圏 $(\mathbf{E} \downarrow B)$ の終対象であることが判ります.

■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:

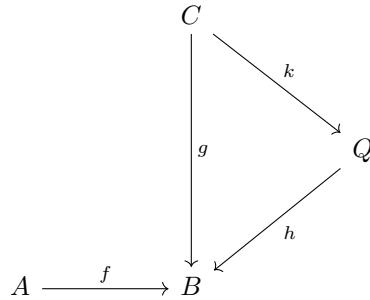


ところで、この引き戻しの可換図式はコンマ圏 $(\mathbf{E} \downarrow B)$ の対象の積の可換図式そのものになります。このときにコンマ圏 $(\mathbf{E} \downarrow B)$ の対象 $D \xrightarrow{f \circ g'} B$ がコンマ圏 $(\mathbf{E} \downarrow B)$ の対象 $A \xrightarrow{f} B$ と $C \xrightarrow{g} B$ の積になります。

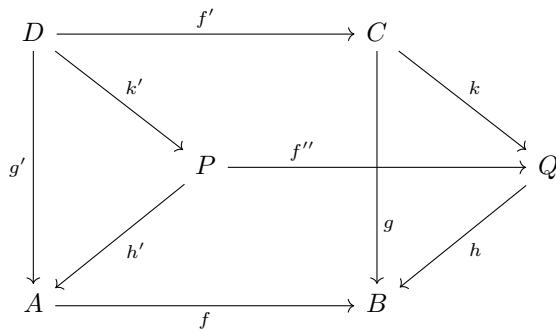
■部分対象分類子の存在: \mathbf{E} が基本トポスであることから部分対象分類子 Ω が存在します：



■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

ここでは「圏論による論理学」[14] に従って函数型高階論理 λ -h.o.l とあるブーリアン・トポス \mathbf{E} との対応関係を紹介します。ここでブーリアン・トポス (Boolean topos) とはトポス \mathbf{E} で部分対象分類子 Ω として真理値 {True, False} とするものです。

2.10.1 高階論理 λ -h.o.l. について

まずこの函数型古典高階論理 λ -h.o.l について簡単に説明しておきます。ここで取り上げる函数型古典高階論理 λ -h.o.l は型を持った λ 計算です。ただ、ここではその λ 表記の利用に留め、それ以上は深入りしません。次に型はまず λ -h.o.l. が扱う対象としていくつかの「領域」を想定しているため、型はその領域の意味を表現するものと言えます。たとえば λ -h.o.l. が扱う命題を構成する対象が構成する領域、命題を λ -h.o.l. で判断した結果、その命題が正しいとか偽であるといった状況を示す真理値で構成される領域といったものです。そういう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 と表記しますが、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 $\langle\alpha\beta\rangle$ でその型を定めることができます。ただし、この写像の型については次の規則を入れることにします：

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。

ここで 1. については単なる略記として認めることは問題がないでしょう。2. については前半の外側の括弧を外した略記も特に問題はないでしょう。後半の括弧を全て外した表記と前半の表記との関係については、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の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、項の場所に注目をしています。それから三変数以上の函数についても二項関係の延長として捉えています。

以上から λ -h.o.l. の型を次で定義します：

 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.

ここでの型 e は言語対象での「**実在物 (entity)**」を指し、真理値の型 t とは別にあることを主張しています。それから写像にも型を導入し、実在物と真理値といった「静的」なものだけではなく、写像の合成によって新たな型を創り出す「動的」な側面を持っています。なお、写像の型の表記については上述の右結合を採用するものとします。

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. 變項: $x_\alpha, y_\beta, z_\gamma, \dots$
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, \top 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. $=_{\alpha\langle\alpha t\rangle}$ は型 $\alpha\langle t \rangle$ 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.	$\wedge_{t\langle tt\rangle}$	$\stackrel{\text{Def.}}{=}$	$\lambda x_t.\lambda y_t.(\lambda f_{t\langle tt\rangle}.(f_{t\langle tt\rangle}TT) = \lambda f_{t\langle tt\rangle}.(f_{t\langle tt\rangle}x_ty_t)))$
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があります. 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	\rightarrow	対象 E
型 t	\rightarrow	対象 Ω
型 $\langle \alpha, \beta \rangle$	\rightarrow	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。

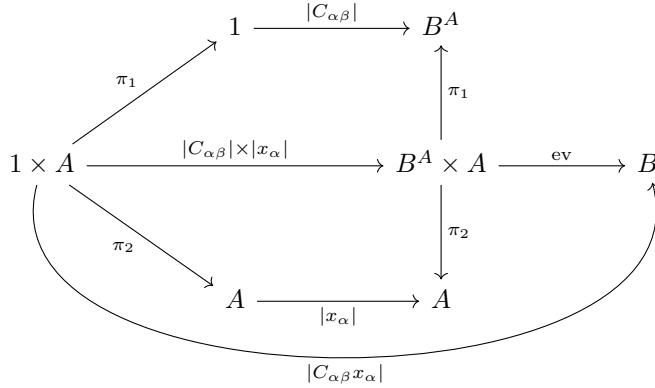
■項 $C_{\alpha\beta}$: $1 \xrightarrow{|C_{\alpha\beta}|} B^A$.

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

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

$|C_{\alpha\beta}D_\alpha|$

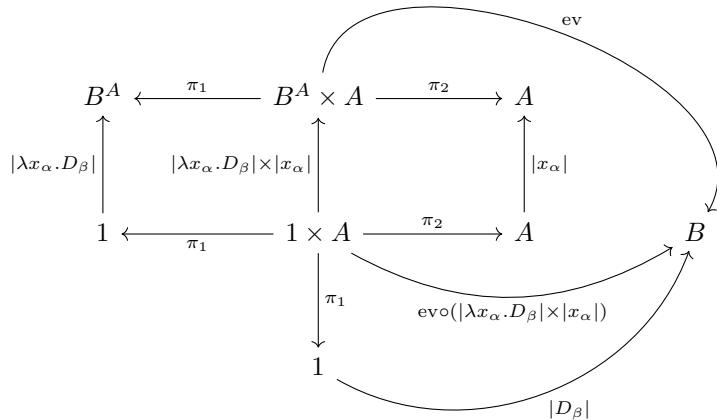
■項 $C_{\alpha\beta}x_\alpha$: $1 \times A \xrightarrow{|C_{\alpha\beta}x_\alpha|} B$. ここで $|C_{\alpha\beta}x_\alpha| = \text{ev} \circ \langle |C_{\alpha\beta}| \times |x_\alpha| \rangle$ になります。これが判る可換図式を以下に示しておきます:



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

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



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 $|\lambda x_\alpha.D_\beta|$ 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:

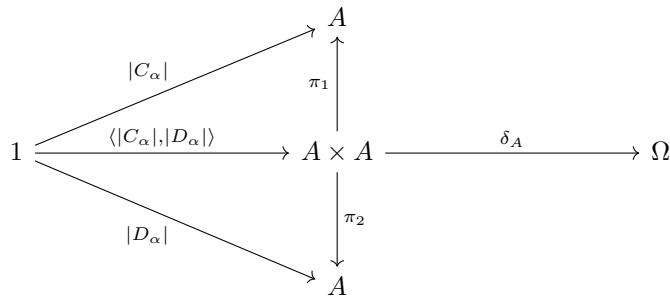
$$\begin{array}{ccccc}
 & B^A & & A & \\
 \pi_1 \leftarrow & B^A \times A & \xrightarrow{\pi_2} & A & \\
 | \lambda x_\alpha . C_{\alpha\beta} x_\alpha | = | C_{\alpha\beta} | \uparrow & | \lambda x_\alpha . C_{\alpha\beta} | \times | x_\alpha | \uparrow & & | x_\alpha | \uparrow & \\
 1 \leftarrow \pi_1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

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 $| \lambda x_\alpha . C_{\alpha\beta} | \times | x_\alpha | = | C_{\alpha\beta} | = | \hat{C_{\alpha\beta}} x_\alpha |$ であることがわかります。

■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 **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 **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 “=” 以外に判断に関するもののがなく、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 ‘ $\neg A$ ’ を $\frac{}{A}$

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」と呼ばれる壮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 $(A_{tt} T_t \wedge A_{tt} F_t) = \forall x_t A_{tt} x_t$

第3章

Pythonについて

3.1 Python とは？

Python は一つの言語仕様で、Java による Jython, .Net Framework/Mono による IronPython, さらには 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 Cython 2.7 を基盤とし、C ライブラリの利用で Cython が用いられています。なお、Python には 2.x 系と 3.x 系がありますが、互換性の問題で双方が並立した状況です。そのために 2.7 系の説明を中心に行なながら、3.x 系のこともその差異が生じる度に説明します。

計算機のアプリケーションにはその処理系に加えて、利用者と処理系との仲立を行うソフトウェアを持つことがあります。この仲立を行うソフトウェアを卵の内部と外部を隔てる境界の殻にたとえて「**シェル (shell)**」と呼びます。このシェルを介して処理系とやり取りするために一般の利用者にとってシェルがそのアプリケーション本体のように見えます^{*2}。Python の代表的なシェルに CPython に付属するシェル、IPython や Jupyter^{*3} があります。ここで CPython に付属のシェルは「**インタプリタ (interpreter)**」と呼ばれ、そのユーザ・インターフェイスは行単位で命令文を入力・処理する「**CLI(Command Line Interface)**」です。それに対して IPython や Jupyter はより高度な編集・履歴機能に加え、ウェブ・ブラウザを利用し、グラフィックス・オブジェクトも表示可能なノートブック形式のユーザ・インターフェイスが実現されています。

この章では最初に Python 2.x 系の概要を述べ、それから例として有理数の定義を行います。それから、「Python 言語リファレンス」^{*4}にしたがって Python(2.x) の言語的な説明を行い、3.x 系と違いがある箇所ではそのことについても言及します。なお、Python の実例で CPython のインタプリタを用いますが、SageMath で IPython がシェルとして用いられ、SageMathCloud では標準の IPython に加えて Jupyter も選べるため、必要に応じて IPython 等のシェルについても説明します。

^{*1} 正確には Python に幾つかの制約を加えた RPython です。

^{*2} 店の窓口の印象が店全体の印象になることに似ています。

^{*3} Jupyter は IPython の後継で、Python に依存する箇所を切り分けて Haskell, Julia, GNU Octave や GNU R 等に対応したウェブ・ブラウザを使ったノートブック形式のユーザ・インターフェイスを実現しています。

^{*4} Python 言語リファレンス: <http://docs.python.jp/2/reference/index.html>, 2.x 系向けの文書です。

3.2 Python の特徴

3.2.1 多重模範言語

Python はクラスに基づくオブジェクト指向プログラミング言語 (OOPL) と呼ばれる言語の一つですが、インターフェリタを使えば対話的に処理が行え、プログラミングも BASIC のように命令型風に、あるいは LISP のように函数型風に、それから C のように手続言語風にとさまざまな様式で行えます。この Python のように目的に応じてさまざまなプログラミング様式が選べる言語のことを「**多重模範言語 (Multiparadigm programming language)**」と呼びます。この性質は Python に大きな自由度を持たせることになります。

3.2.2 簡素化された構文

Wikipedia に「核となる構文や文法が必要最低限に抑えられている」とあるように 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うようにしています。実際、§3.6 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、Perl^{*5} や Mathematica で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がやり難くなる一方で、常識的なプログラミングで妥当な結果が得られ、プログラム自体も上述の言語と比べて均質的なプログラムに収斂される仕様です。このことに加え、PEP-8^{*6} 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あえすれば、過剰な技巧に走る必要性がなく、文書化も含めて、ある程度の品質を保持したプログラムが得られるという実用本位の言語です。

3.2.3 構文要素としての字下げ

C や Java 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、これらの言語で字下げは任意です。ところが Python の字下げは構文上、必要不可欠な要素で、その結果、Python のプログラムは二次元的な構造を持ち、視覚的にプログラムの構造把握が可能です^{*7}。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。

^{*5} 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。

^{*6} PEP (=Python Enhancement Proposal): Python 改善提案書

^{*7} 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を意図したものではありません。

たとえば、C の if 文は直線的に

```
if (x==0){y=1;} else {y=0;}
```

と記述しても、また、平面的に

```
if (x==0){
    y = 1;
} else {
    y = 0;
}
```

と記述しても空白文字の 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に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もとに戻します。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⁸：

```
if x==0: y = 1 else: y = 0
```

⁸ $y = (1 \text{ if } x==0 \text{ else } 0)$ は可能です。

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。この Python が構文の一要素として字下げを取り入れたことに類似の事例が歴史的な論理式の表記にあります。これはフレーゲの概念記法で、その概念記法で記述した論理式は平面的な構造を持ちます。ただし、フレーゲが活躍した 19 世紀末から 20 世紀初頭の印刷技術で図式の印刷が面倒であったためにその著書「算術の基礎法則」を二巻に分け、二巻目は自費出版する程で、フレーゲの概念記法は一部の記号が現在の論理式の表記で取り入れられているとは言え、ペアノ (Peano) による線的な表記が現在の論理式の表記の源流になっています。この事例を鑑みると、計算機の処理能力とメモリが貧弱であったために、ed や EDLIN といったラインエディタやカード読取機を利用していた環境から、計算機の処理能力とメモリが十分に強化され、エディタもスクリーンエディタが主流でディスプレイも大画面、さらには多画面であることがこの字下げを構文要素の一つとして取り入れる一助になっていること、次に述べる文書文字列というある意味過剰な文書をプログラムの解説文書として包含できること等と Python はさまざまな技術発展による恩恵を受けている言語です。そして、Python の利用もそのような環境に適合させた使い方が行われています。

3.2.4 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、LISP や MATLAB 系言語で見られます。最初に Common LISP *9 の例を示しておきましょう:

```
* (defun add2 (x) を足すよー"2" (+ x 2))
```

ADD2

```
* (documentation #'add2 'function)を足すよー
```

"2"

*9 Common LISP の処理系の一つの SBCL です。

この例では defun 文で函数 add2 の定義を行い、函数に記載した文字列を函数 documentation で表示させています。なお、この例で左端の文字 “*” が SBCL のプロンプトです。このように Common LISP の函数に文字列を入れて、それを参照することが可能になっています。この Python の文書文字列に類似したものに MATLAB 系言語のヘルプがあります。MATLAB は数値行列処理に長じた言語で、この MATLAB に類似した言語を「MATLAB 系言語」^{*10}と呼ぶことにします。この MATLAB 系言語では函数のヘルプをプログラム中の註釈として内包し、函数のヘルプを閲覧するときに函数が記述されたファイルから註釈を検索して表示します。そのために函数単位で一つのファイルの所定の位置にヘルプの内容を註釈としてあらかじめ記述しておく必要があります^{*11}。ここでは MATLAB 言語の中で C に最も類似している Yorick の例を示しておきます：

```
func add2(x)
/* DOCUMENT add2
*
* を足すよー2
*/
{ return x+2;};
```

このように函数定義を行う func 節の直後に註釈としてヘルプの内容を記載し、函数名に対応するファイル名 add2.i で保存しておきます。あとは Yorick 上で函数 help() を使えばファイル add2.i に記載された文書文字列が表示できます。この仕組は他の MATLAB 系の言語でも同様です。以下に Yorick の実例を示しますが、ここで左端の文字 “>” が Yorick のプロンプトです：

```
> include,"add2.i"
> help,add2
/* DOCUMENT add2
*
* を足すよー2
*/
defined at: LINE: 1 FILE: /home/yokota/add2.i
>
```

^{*10} MATLAB クローンの GNU Octave、フランスの INRIA で開発され高機能なツールが揃っている Scilab、MATLAB 風の配列処理が可能な C といえる Yorick があり、これらの言語は行列や配列の処理方法に MATLAB に由来する類似点があります。特に Yorick は多次元配列操作に長けており、多次元配列のゴム添字 (rubber index) は Python の配列の拡張スライス操作に最も類似しています。

^{*11} このファイルのことを MATLAB で M-file と呼びます。その理由はファイルの修飾子が “.m” のためです。たとえば neko() という MATLAB の函数を定義するときは ‘neko.m’ というファイル名に函数一式を記述します。

Yorick の例で示すように MATLAB の系の言語で函数定義文直後の註釈はプログラム中の文書としての性格をより強く持ち, 実際, MATLAB 系の言語ではオンライン・ヘルプの例題を含む文書として記載されています。なお, Python ではわざわざファイルに記載しなくとも, 函数をインタプリタ上で定義する際に文書文字列を所定の位置に書き込みます:

```
>>> def add2(x):
...     u"""
...     を足すよー    2
...
...     """
...     return x+2
...
>>> help(add2)

Help on function add2 in module __main__:

add2(x)を足すよー
    2
>>> add2.__doc__
u'\n    をたすよー    2\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, 函数 add2() の定義が継続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 Python の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 (') で括った文字の列です。ここでは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”を文書文字列に配置しています^{*12}。それからオブジェクト中で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 の文書文字列は前述のように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すが, 再びこれら 3 個の引用符で括られることにならない限り引用符を文書文字列内部に記載することができます。そして, 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。ただし, Python の組込のオブジェクトのものはモジュール 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:

^{*12} 2.x の場合, 3.x では文字コードが UNICODE になります。

```
Help on built-in function open in module __builtin__:
```

```
open(...)
    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__’ で確認できます。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ます。また、この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。また、IPython や Jupyter を Python のシェルとして用いているときは函数 help() の他に記号 “?” が使えます：

```
In [1]: open?
Type: ?      builtin_function_or_method
String Form:<built-in function open>
Namespace: Python builtin
Docstring:
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.
```

これは IPython の例ですが、このように記号 “?” はオブジェクトの名前に空白文字を入れずに続けて入力します^{*13}。

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 は組版指示言語の 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く、さらに Docutils^{*14}で HTML や LaTeX 等のさまざまな組版指示言語への変換が可能なため、reST で記述した文書は高い表現力を持つだけではなく、高い汎用性も持ちます。図 3.1 に SageMathCloud 上で編集した reST の文書を示しますが、左側が reST の記述、右側がそのレンダリング結果です。このようにテキストデータとしても構築し易く、

^{*13} IPython では名前の前に記号 “_” の記載ができるが、SageMath ではエラーになります。

^{*14} Documentation Utilities <http://docutils.sourceforge.net/>

そのデータ自体が既に文書として飾りを入れたもので可読性も高いことが判るでしょう。そして、文書生成ツールの Sphinx^{*15}を使うことで reST 文書を LaTeX, HTML や PDF といった書式の文書に変換することも容易に行えます。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。

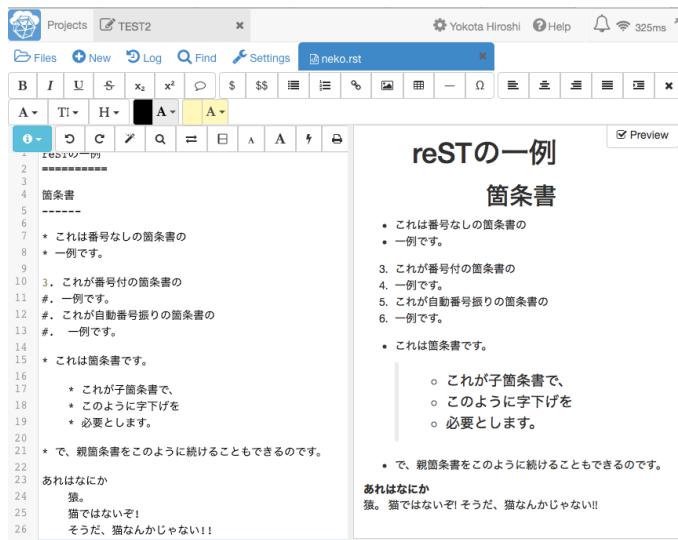


図 3.1 reST の一例

3.2.5 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は扱う対象全てを「**オブジェクト**」として捉えます。これは数値や文字列といったデータはもちろんのこと、それらデータを扱う函数といった諸々がオブジェクトになります。さらに Python は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るために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。そして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であることから「**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ます。このクラスの実体化のことを「**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^{*16}。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**

^{*15} <http://www.sphinx-doc.org/en/stable/index.html> と <http://docs.sphinx-users.jp/> を参照。

^{*16} ただし、クラスが集合になるとは限りません！

「**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。

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.xで古典的クラス(旧スタイル)、Python 3.x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、その差異は露呈しません。

```
class TEST:  
    pass
```

ここで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.weightとa.ageという属性に値を設定するというCの構造体と同様の処理をしています。ここで‘class TEST:’の末尾の記号“:”は文節を示す特殊な記号です。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“.”になります。この例では属性への代入を行っていますが、ここで示すように変数への値の代入は演算子“=”を用います。ここでPythonの代入式では変数リストに対して代入もできます：

```
>>> a = (1,2,3,4)  
>>> x, y, z, w = a  
>>> x, y, z, w
```

```
(1, 2, 3, 4)
```

```
>>>
```

この例では変数リスト ‘x, y, z, w’ に対してタプル ‘(1, 2, 3, 4)’ の内容を代入させています。この代入は演算子 “=” の双方の被演算子の長さが等しいときに可能です。

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 の構造体に類似した書式です：

```
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  
>>> 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 の構造体と同じです。

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、だからといって、犬のボチがそのような仕草をすることは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、通常の函数とは異なり、そのクラスのインスタンスであるか、そのクラスと継承関係にあるクラスのインスタンスでなければ使えません。

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。ここで C++ ではクラス変数とインスタンス変数で宣言が異なりますが、Python にはクラス変数やインスタンス変数といった型の宣言がないために変数の使い方で両者を区別することになります。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 (ディスクリプター)**」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)**」の二種類があり、これらのメソッドの定義で「**デコレータ**」が用いられます。そのためにスタティックメソッドの定義で '@staticmethod'、クラスメソッドの定義で '@classmethod' が用いられます。さらにクラスメソッドの定義で第 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るため、そのクラスの属性の参照が可能ですが、スタティックメソッドは

クラスそれ自体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。そのために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、いわゆる「動的 (dynamic) な参照」^{*17}が行われるのに対し、スタティック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ます。このように「属性の参照が勝手に遡ったりせずに、指定した範囲内で行われる」という「静的 (static) な参照」が行われるという意味です。

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’ いう名前で表現し、必ず第 1 引数に与えます。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。実際にインタプリタで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
>>>
```

^{*17}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 §3.11 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しょう：

```
>>> 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「**参照可能な範囲（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ます。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で用いられます。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の列 __< クラス名 > が先頭に置かれた名前に置換され、そのために本来の属性名で参照できなくとも 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.x 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、削除という「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によって束縛動作での振舞いを利用者の目的に適ったものにすることができます。

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。次に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。

3.3 有理数を構築してみよう

3.3.1 有理数の表現

有理数を定義するにあたって、整数とその算術演算はすでに定義されているため、それらを上手く利用したいものです。しかし、整数からいきなり有理数に到達できません。というのもまず有理数が「何であるか」、「どのようなものであるか」ということが明瞭でないためです。ここで我々は整数を既に手に入れているため、既存の整数と有理数の違いを明瞭にしていかなければなりません。

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。だから有理数を一つ定めるためには二つの整数が必要です。そこで最初に整数対のクラスを定義することにします。ここで整数対は与えられた有理数に対して一つ定まり、有理数全体で一定の値ではありません。したがって、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あり、インスタンス化の時点では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有理数を表示させましょう。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**」と呼ばれるあらかじめ用意されたメソッドがあり、これら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 §3.9 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__(self):
        return '%s/%s' %(str(self.numer), str(self.denom))
```

インスタンスの初期化を行うための特殊メソッド `__init__()` の第一引数の `self` は対象そのもので、この変数は必須です。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、‘`a = PairOfInts(1,2)`’で対象を生成すると `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^{*18}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`' で指定することで '`a = PairOfInts(1,2)`' でインスタンスを生成したときに名前 `a` をインタプリタに入力すると文字列 '`1/2`' を表示す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^{*19}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ます。また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。そして `import` 文によるファイルの読み込みで名前の衝突を避ける方法が「**名前空間**」における工夫です。これは '`import PairOfInts`' で読み込むとファイル内部で定義したオブジェクトの参照を行うと、`PairOfInts.py` ファイルで定義したオブジェクトへの参照で、先頭にモジュール名（ここでは `PairOfInts`）が付きます。ただ、この程度の内容で煩わしくても意味がないために '`from PairOfInts import PairOfInts`' でファイルの読み込みを行うことでモジュール名（=ファイル名）を外して使えるようにできます：

```
>>> from PairOfInts import PairOfInts
>>> a = PairOfInts(1,2)
>>> a
1/2
```

この例ではインタプリタ上で `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、名前 `a` を入力して '`1/2`' という表示を得ています。では、メソッド `__repr__()` を定義してい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]
```

^{*18} Python 2.x では「文」ですが、Python 3.x では「函数」に変更されます。

^{*19} os モジュールの函数 `getcwd()` で確認できます。変更は同様に os モジュールの函数 `chdir()` の引数として経路を与えればできます。

```
>>> 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 クラスをインターフェ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>`' と表示されるだけです。ここで表示されている '`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します。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、名前をそのまま入力しても名前が参照するオブジェクトの番地が返されます。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ることができます。このように特殊メソッド `__repr__()` を定義することでインスタンスの表示を定められます。このようにインスタンスの公式の表示を利用者が決められるということは Python の表示の一貫からオブジェクトの同一性、あるいは同値性が保証できないことを意味します。実際、Python の整数型オブジェクトの 1 と SageMath の Integer 型オブジェクトの 1 が別物であるために SageMath にあらかじめ用意された Integer 型のメソッドが使えません。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)**」と呼びます。また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。

3.3.2 有理数のクラスの構築

さて、この自然数の対はあくまでも自然数の対としての性質の他に何もなく、メソッドにも有理数の表記を使って自然数の対を表示する機能しかありません。ここで我々は有理数を Python 上で表現することを目的にしていますが、現時点では有理数を整数の対として表現することだけで有理数として定義ではありません。つまり、「**有理数とは何なのか**」という問への回答には、二つの整数対が与えられたときにどのような条件を充せば有理数として等しいと判断できるのかという判断基準、互いを比較するという手段がまだ必要です。ここで最も原始的な判断基準は分母と分子がそれぞれ等しければ良いというものです。しかし、これだけでは不十分です。たとえば $1/2 = 2/4$ ですが、有理数として自然数の対 $(1, 2)$ と $(2, 4)$ は等しくなりますが、単なる自然数の対として前述の安易な手法によると別物です。また $(-a, -b)$ と (a, b) は等しいものであるべきです。この「**等しい**」という関係は「**与えられた自然数の対 (a, b) と (c, d) が $'ad - bc = 0'$ を満すとき**」と定めることができます。これで「**等しい**」ことの判断基準が一つ定まりました。そこで有理数は一意に整数対で表現されるべきではないでしょうか？たとえば (a, b) と $(-a, -b)$ は等しい自然数で、このことから分母に相当する `denom` が常に正となるように置き換え、同様

に 0 と異なる整数 c に対して $(a*c, b*c)$ であれば (a, b) で置き換えるべきです。つまり、`RationalNumber` クラスは単なる自然数の対のクラス `PairOfInts` にこの二種類の正規化を加えたクラスであるべきです。では、等しくない場合はどうでしょうか？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入れられます。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るために ‘ $ad > bc$ ’ と ‘ $(a, b) > (c, d)$ ’ が同値になります。これらのこと踏まえて `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.__gcd__()

    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 reexpr(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
```

ここで示すように class 節で [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 RationalNumber 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。

す^{*20}.

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)**」, 繙承する側の RationalNumber クラスを「**派生クラス (derived class)**」と呼びます。また、この継承関係を親子関係にたとえたときに基底クラスを「**親クラス**」, 派生クラスを「**子クラス**」, また、この継承関係を上下関係として捉えるときに基底クラスを「**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、そのまま範疇論の歴史的な議論が活用できます。そして、対象への理解が深まることでより詳細に分類されるということも理解されるでしょう。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^{*21}。

3.3.3 特殊メソッドによる四則演算の導入

さて、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.repr()
    return

def __sub__(self, other):
    """有理数の差を定義するメソッド

    """

```

^{*20} ここでの「等価」はオブジェクトの値について「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。

^{*21} この内包と外延の関係を「**内包外延反比例増減の法則**」と呼びます。

```

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):
    """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)

```

これらの特殊メソッドで、最初のメソッド`__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`__mul__()`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、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っていません。ここで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。そこでSageMathを活用する意味がでてきます。このSageMathにはさまざまな代数的構造を有するクラスが存在しているため、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になります。

簡単な紹介はここまでにして、次の節から Python の構文とオブジェクトについて解説したいところでですが、Python のオブジェクトや構文の解説では「**BNF**」と呼ばれる表記方法が用いられています。この BNF はどのようにオブジェクトや文が記号や文字で記述するかを表記する方法です。Python 言語の簡潔過ぎる BNF の一覧はソースファイルの Grammar/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、与えられた Python のプログラムは構文解析器によって Zephyr ASDL(Abstract Syntax Description Language)^{*22}を使った表現(ASTs, Abstract Syntax Trees)に置き換えます。この ASDL による置き換えの書式はソースファイル Parser/Python.asdl にどのようなものが記載されています。ここで BNF は文字を使った構文の構成方法そのものですが、ASDL は BNF より構文を引数の型宣言を含む函数表現になっており、引数の構成を記載した BNF よりも簡素化した書式になります。そこで、Python の構文を解説する前に、その前提になる BNF について述べます。

3.4 バッカス・ナウア記法(BNF)について

「Python 言語リファレンス」では「**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)**」を用いて Python の構文の解説が行われています。この BNF はジョン・バッカス (John Backus) がプログラム言語 ALGOL^{*23}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：

Backus-Naur 記法の構文規則

非終端記号 ::= 定義₁ | 定義₂ | ... | 定義_n

この式の意味は演算子“::=”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、生成規則が一通りだけであれば右辺は‘定義₁’だけになるため記号“|”が不要になります。逆に‘定義₁| 定義₂’であれば‘定義₁’でなければ‘定義₂’で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**」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“::=”の右辺の各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する記号であるために「**構文変数**」とも呼ばれます。非終端記号があれば終端記号もあり、「**終端記号**」は演算子“::=”の

^{*22} <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。

^{*23} ALOGOrithmic Language: 1950 年代に開発された Pascal の先祖になる言語。

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 では非終端記号を記号 “`()`” を使って「`(非終端記号)`」と括ることで非終端記号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。

さて BNF の簡単な実例を示しておきましょう。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。それでは自然数の 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：

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"
```

になります。この自然数の定義はさきほどの数字の定義と異なり、帰納的な定義を含みます。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です。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ています。と、このように「**自然数が何であるか？**」という問い合わせる定義でありませんが、どのように文字を使って自然数が構築されたものであるかを示す記述になっています。

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。そこで、「**命題論理式の定義**」を示します：

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. なお, 「Python 言語マニュアル」の BNF は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 の具体例を示しましょう. 最初

に整数の構成の BNF を示します:

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 S なものへと置換えられます。もう一つの例を示しておきましょう。これは「Python 言語リファレンス」にある例です²⁴:

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。このように「Python 言語リファレンス」の EBNF では記号 “...” 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の小文字、以後はアルファベットの小文字や記号 “_” で構成された文字列であって、‘_a’ のように先頭がアルファベットでない文字列、具体的には ‘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 に適合しないことが判ります。このように EBNF を利用することで Python の構文が、その働きは別として、具体的にどのような記述がされ得るかより明瞭に表記されます。

²⁴ 紛らわしいのですが 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 を示しているだけです。

3.5 Python の字句解析について

3.5.1 トークン (token) について

Python 処理系の構文解析器 (parser) で入力されたプログラムの字句解析が行われ、プログラムは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になります。Python のトークンで「**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 Python の論理行の区切になるトークン、**INDENT**, **DEDENT** は **INDENT** と **DEDENT** のトークン対で §3.7 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 Python 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^{*25}。また、「**空白文字 (blank character)**」^{*26}とよばれる文字の中で「**欧文間隔 (Space)**」、「**水平タブ (TAB)**」や「**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’ の二つのトークンになります。

3.5.2 行構造について

Python のプログラムは字句解析によってトークンの列に変換され、さらにこのトークンの列は「**NEWLINE**」を区切として分割されます。すなわち、プログラムは複数の論理行に、論理行はさらに物理行に分解されることになります：

■**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)**: 論理行をさらに行末端文字で分割した文字の列です。ここで行末端文字は計算機環境で異なり、UNIX 環境で ASCII 文字の **LF(行送り)**、Windows 環境では **CR+LF**、MacOS で **CR(復帰)** ですが、Python のプログラムに物理行を埋め込むときは計算機環境を問わず C と同様に行末端文字として ASCII 文字“**LF**”に対応す

*25 演算子と区切文字に関しては `Parser/tokenizer.c` にてそれらの名前が定義されています。

*26 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 文字があり、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、空白文字には各言語の間隔文字もありますが、ここでの空白文字と区別します。

るエスケープシーケンス^{*27} ‘\n’^{*28}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^{*29}。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を行うときに注意が必要です。
- **符号化宣言 (エンコード, encoding):** 符号化宣言は、プログラムで用いられる文字がどの文字コードに属するかを明示的に示すための宣言です。Python の既定値の文字コードは 2.7 で ‘ascii’, 3.x では ‘utf_8’ になります。PEP-263 に規定があり、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で、プログラムの先頭の一行目か二行目に置かれ、次の正規表現：

符合化宣言の正規表現

`coding[=:]\s*([-w.]+)`

に適合します。たとえばプログラムの文字コードを UTF-8 に設定するときは

`# coding = UTF-8`

あるいは

`# coding : UTF-8`

といった宣言行が該当します。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^{*30}。この符号化宣言によって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ますが、この宣言がなければプログラムに記載された文字は符号化を指定する接頭辞を持たない限り ASCII 文字として扱われます。

- **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になりますが、それ以上の意味は持ちません。

^{*27} 特殊な作用がある文字列です。§3.5.4 参照。

^{*28} 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“\”が勝手に記号“¥”で置換えられています。そのため多くの書籍でこれらの記号を同一視していますが、UTF-8 等の文字コードでは別の記号です。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 Windows 環境では適宜、記号“\”を記号“¥”で読み直して下さい。

^{*29} 記号“#”は Python のリテラルに含まれません。

^{*30} 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。

■字下げ/インデント (indentation): 論理行の先頭に置かれた Space, TAB の個数で字下げの水準が計算され、こ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 のみで TAB を混在させないことが推奨されています。実際、両者の判別が容易でなく、それらが一貫した順序で並んでいないときに構文解析器から例外 TabError が送出される原因になります。また、ノートブック形式のユーザ・インターフェイスで TAB に入力補完機能を持たせることもあるため、Space のみで間隔を開けるように統一することが現実的です。

■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。物理行の分割を明示的に行う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。このときに Python の構文解析器は継続文字の直後の行末文字を削除して一つの物理行に変換します。ただし、継続文字 “\” に続けて註釈を追記できません。なぜなら、註釈は論理行を終える働きがあり、その結果、註釈の前後で二つの論理行に分割されるためです。また、註釈自体も継続文字を使って分割することができません。ここで改行の例外的な規則として丸括弧“()”，角括弧 “[]”，波括弧 “{ }” の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。

3.5.3 識別子とキーワードについて

「**識別子 (identifier)**」は「**名前 (name)**」に用いられます。そして、オブジェクトへの参照はこの名前を介して行われます。識別子の EBNF を以下に示しておきます：

—— 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"

```

Python の識別子はアルファベットと数字、それと記号 “_” のみで構成された ASCII 文字の列で、日本語の“**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^{*31}。ただし、次に示す「**キーワード**」は式や文の構成要素であるた

^{*31} Python 3.x で文字コードとして UNICODE が採用された結果、漢字、ギリシャ文字、キリル文字等の非 ASCII 文字も 3.x では識別子として使えます。詳細は「PEP-3131 Supporting Non-ASCII Identifiers」を参照。

めに Python の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	

識別子の EBNF からこれらのキーワードを除いたものが「**名前**」として使えます。Python では名前でオブジェクトとの対応付けが行われ、オブジェクトへの参照は名前を介して行われます。この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)**」で表現されます。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。

3.5.4 リテラルについて

「リテラル (literal)」は「文字どおり」、「字義どおり」を意味する言葉です。記号論理学で用いられるリテラルは命題記号（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を指し、プログラミングのリテラル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。ここで Python のリテラルは「**文字列リテラル**」と「**数リテラル**」の二種類に大きく分類できます：

リテラルの分類

リテラル	::=	文字列リテラル 数リテラル
------	-----	-----------------

Python のリテラルは全て「**変更不能なデータ型**」であるためにオブジェクトを生成すると、そのオブジェクトの値を変更することができません。そのためオブジェクトの値が同じリテラルであってもオブジェクトとして一致しないことがあります。

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

このように「**文字列リテラル**」は C 等の言語の文字列を拡張したものと言え、符合化宣言を接頭辞として持てます。また、プログラムに符合化宣言が含まれず、リテラル自体に接頭辞がなければ、文字列リテラルを構成する文字は ASCII 文字として解釈されます。逆に文字列に接頭辞を付与することで、その文字列リテラルの符合化宣言が行えます。たとえば、文字列の先頭に “u” や “U” といった接頭辞を配置すると直後の文字列が UNICODE 文字列型として扱われます。ただし、文字列の符合化宣言をする際に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*32.

それから文字列には引用符の個数による型の区別があります。Python の引用符には单引用符 (') と二重引用符 ("") の二種類があり、文字列はこのどちらかの引用符で対として括られなければなりません。そのため "abc 'def' hij" のように文字列リテラル内部に文字列リテラル全体を括る引用符対と別の引用符対で文字列リテラルを括られます。そして、「**短文字列 (shortstring)**」は一重に引用符で括られた文字の羅列の型、「**長文字列 (longstring)**」は三重に引用符で括られた文字の羅列の型です。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、この文書文字列はオンラインマニュアルとしての性格を持ちます。この文書文字列の中では改行や单引用符 (') や二重引用符 ("") が入れられ、このことを利用して例題等を含む長い文書の記述ができます。このときに長文字列を構成する際に用いた引用符と同じ引用符を三回連続

*32 Python 3.x では既定値の文字列リテラルが UNICODE になったために接頭辞の扱いが Python 2.x と異なります。

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、この例のように単引用符や二重引用符、それと行末端文字を含む文字列は一つの長文字列になります。なお、プログラムで改行の出力を表現するときは C と同様に、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)) を挿入することで出力で文字列の改行が行えます。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 reST 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を格段に向上させることができます。

最後に「エスケープシーケンス (Escape sequence)」は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
\xxxxxx	16ビットの 16進数値 xxxx を持つ Unicode 文字
\xxxxxxxxxx	32ビットの 16進数値 xxxxxxxx を持つ Unicode 文字

ここで接頭辞に ‘r’ や ‘R’ が含まれていなければ Python のエスケープシーケンスは C と同様の表記です。接頭辞 ‘r’, ‘R’ がある文字列はこれらの接頭辞に続く文字列に含まれる記号 “\” を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。また、この一覧の UNICODE 文字名は「**Unicode Character Database**」^{*33}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。

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 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 種類があります。ここで整

^{*33} <http://www.unicode.org/Public/UCD/latest/charts/CodeCharts.pdf> を参照。

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^{*34}、長整数は計算機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^{*35}。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」で表現され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"

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 “I” や “L” といった長整数 (Long Integer) でをやり難くなる一方で、あることを示す文字を配置しますが、小文字 “I” は数字 “1” と紛らわしいために大文字 “L” の使用が推奨されています。また、Python 3.x では長整数に型が統一され、そのときのリテラルが整数のリテラルになります。そのため長整数末尾の “L” は 3.x で不要です。Python では整数型で処理していく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く絶対値が整数型の最大値以下になった時に自動的に整数型へと変換されます。なお、SageMath では後述の GMP を用いて整数が実装されているために、SageMath の数学的対象としての整数のリテラルは Python の整数リテラルと同じです。

ところで Python に標準で実装された長整数による処理は、たとえば乗算でカラツバ法

^{*34} 土214743647 の範囲の数で、Python 2.x では変数 sys.maxint に整数型の上限としてこの値が束縛されています。ただし、Python 3.x 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。

^{*35} 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。

(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。そのため SageMath ではより高速演算が可能な GMP(GNU Multiple Precision Arithmetic Library) を用いた整数クラス Integer^{*36}で整数リテラルが処理されます。次の例^{*37}で SageMath の GMP に基づく整数が 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)
```

それから函数 choose を起動して処理時間の計測を行いますが、ここでの計測は以下の式を評価することで行います：

```
start = time.clock()
print choose(50000, 50)
elapsed = (time.clock()-start)
```

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。この式の評価を SageMath Cloud(SMC) で確認すると、SageMathCloud 上の Python で 1.943, SageMath で 0.594 という結果を得て圧倒的に SageMath の整数処理が高速であることがわかります。ところで、SageMath 上でプログラムを構築するときに注意すべきことがあります。それは明示的に数オブジェクトを生成しなければ、旧来の Python の数オブジェクトとして生成される可能性があります。たとえば式 ‘2^3’ を SageMath で直接入力すると 2³ として処理されるために結果は 8 になります。ところがプログラム内部で安易に ‘2^3’ と記述すると演算子 ‘^’ は Python の整数型に対しては排他的論理和であるために結果は 1 になります。この場合は ‘Integer(2)’, ‘Integer(3)’ と明示的に SageMath の整数

^{*36} 正確には sage.rings.integer.Integer クラスで、Cython を使って GMP の mpz_t integer 型のラッパーとして実装しています。

^{*37} <http://jasonstitt.com/c-extension-n-choose-k> の記事: 「GMP bignums vs. Python bignums: performance and code examples」の例です

クラスのインスタンスとして生成して式の処理を行わなければなりません。

次に浮動小数点数リテラルの EBNF を示します:

—— 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 +

ここで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す。この点は複素数も同様です。この浮動小数点数リテラルで表現される数の型は float 型のみです。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型、すなわち float 型のみを持ちます。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。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「浮動小数から構成されたリテラル」であり、このことが虚数リテラルの性格を決定付けます。実際、SageMath で代数的数の純虚数 $\sqrt{-1}$ に対しては 'I'、あるいは 'i' が定義され、一方の Python の虚数リテラル '1J' は浮動小数点数を基にしているために SageMath の代数的数の 'I' と異なります。このことを SageMath で確認しておきましょう:

```
sage: type(I)
<type 'sage.symbolic.expression.Expression'>
sage: type(1J)
<type 'sage.rings.complex_number.ComplexNumber'>
sage: I^2 is 1J^2
False
sage: I^2
-1
sage: 1J^2
-1.000000000000000
```

函数 type() で I を調べると ‘sage.symbolic.expression.Expression’ と結果が返されることがで I が多項式の仲間の数であることが判ります。ここで純虚数 i が多項式 $x^2 + 1$ の零点となる数であることは良いでしょう。代数学ではこのように整数係数の多項式の零点になる数のことを「**代数的数**」と呼び、その数を零点として持つ最小多項式と呼ばれる最小次数の 1 変数多項式に対応させられ、純虚数は $x^2 + 1$ がその最小多項式になります。それから 1 変数の整数係数の多項式全体、つまり、整数係数の多項式環 $\mathbb{Z}[x]$ に $x^2 + 1 = 0$ という関係を入れてできた世界で変数 x を単純に i と置いてしまえば $\{a + bi : a, b \in \mathbb{Z}\} (\subset \mathbb{C})$ になることが容易に理解できます。これが SageMath の I が多項式の仲間になる理由です。ところでもう一方の 1J を函数 type() で調べると ‘sage.rings.complex_number.ComplexNumber’ であることが判り、先程の I と型が異なります。この虚数リテラルは近似値である浮動小数点数から構成されるために Python の虚数リテラルは近似値としての性格を持ちます。そのために SageMath で近似的な数として虚数リテラルを用いるのか、代数的数である純虚数を用いるかを計算目的に応じて使い分ける必要があります。また、Python や SageMath の数値演算では、整数型から長整数型、各種整数型から浮動小数点数型、浮動小数点数型から複素数型への型の変換は自動的に行われますが、その逆の変換は長整数型から整数型への自動変換を除いて自動的には行われません。また、Boolean の True, False は整数型の 0 と 1 の値をそれぞれ持つために、数オブジェクトとの演算結果は数オブジェクトになります。

3.5.5 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 —————

+	-	*	**	/	//	%	<<	>>	&		^	~
<	>	<=	>=	==	!=	<>						

これらの演算子は算術演算子と論理演算子です。算術演算子は数リテラルを返す演算子で、論理演算子は Boolean を返却する演算子です。なお、演算子 “ $<>$ ” と 演算子 “ $!=$ ” は同じ意味ですが、演算子 “ $<>$ ” は時代遅れの表記のために演算子 “ $!=$ ” 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 “ $|$ ” は二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Pythonのオブジェクトに対しては排他的論理和(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):	<code>+=</code>	<code>-=</code>	<code>*=</code>	<code>/=</code>	<code>//=</code>	<code>%=</code>	
累算算術代入演算記号(2):	<code>&=</code>	<code> =</code>	<code>^=</code>	<code>>>=</code>	<code><<=</code>	<code>**=</code>	

表の上段は括弧の類で、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、单引用符、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。ただし、Pythonのシェルとして使えるIPythonやJupyterでは記号“`?`”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。

3.6 Python の式と文

3.6.1 式、単純文と複合文について

プログラム言語で「式(expression)」は、その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」といった数値、「True」、「False」といった真理値、「1 + 2」、「math.sin(10)」といった数式、「(1, 2, 3)」といったタプルや「['a', 'b', 'c]」といったリスト、「{1, 2, 3}」といった集合、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)」は通常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、Python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。まず、単純文は「一つの論理行に収めることのできる文」であり、名前への拘束を行う代入文、モジュールの読み込むimport文があります。また、「x = 1」のような変数xへのオブジェクトの束縛を行う「代入式」は式ではなく「単純文」になります。これに対して複合文は「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、函数やクラスの定義といったものが相当します。

3.6.2 式の EBNF

最初に Python の式のリストと式の EBNF を次に示します:

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。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場合、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): Python の処理系で評価が行われることで値としてのオブジェクトが返却されるものが式です。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 Python の式で、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。

3.6.3 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 の式の基本単位である原子要素、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)」^{*38}、「識別子 (identifier)」、「リテラル (literal)」と「閉包 (enclosure)」の四種類に分類されます。ここでシングルトンは固有の意味 (値) と型を持ちますが、他はそうではありません。識別子は文字に制限の入ったリテラルで「名前 (Name)」に用いられます。リテラルは「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単純な原子要素になります。これらに対して「閉包」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」や「産出原子式」の七種類に分類されます。この閉包は原子要素を組合わせて構成され、それ自体が Python で意味を持つオブジェクトであるために原子要素に包含されます。

■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 Ω の型は Boolean であり、None の型は None 型になります。また、True には整数の 1, False には整数の 0 が値として与えられています。

■識別子 (identifier): 「名前 (Name)」に用いられます。名前を参照すること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て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’ とするときに名前が ‘_Hom__spam’ に変換されます。このように

^{*38} 「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。

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): 文字列リテラルとさまざまな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.x では長整数型に整数型が統合され, そのリテラルは 2.7 の整数リテラルになります.

■閉包 (enclosure): 識別子とリテラルと異なり, 一定の書式を持つ原子要素で, Python で扱うデータの表現です. 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, これらの書式から得られるオブジェクトは外延表現, つまり, オブジェクトとして生成した時点での成分も定まっています. しかし, 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る点で異なります. なお, ここでの式は評価されることでシングルトンやリテラル等の原子要素を返却するオブジェクトの書式ですが, 値が束縛されていない変数としての識別子は, ここでの式に該当しません.

■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] ")"
```

この EBNF にある括弧 “()” はタプルの領域を明確にするために用いられ, タプル型のオブジェクトの構築で必要であるとは限りません. 実際, 空集合 `{} を表現する空のタプルの書式は ‘()’ と丸括弧を必ず必要としますが, 空の式のリストでなければ ‘1,2,3’ でも ‘(1, 2, 3)’ でも同じタプルが生成されます. ちなみにタプルは後述のリスト表現で生成されるリスト型に類似したオブジェクトですが, リスト型のように融通無碍ではなく, 一旦, 生成するとその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す. 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

の式のリストが成分の具体的な表記、内包表現は、リストの成分を Python で生成するための表記で 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。なお、内包表現については §3.6.3 を参照してください。

■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“{ }”は必須です：

——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。なお、内包表現については §3.6.3 を参照してください。

■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に持つことができます。この添字になる式は鍵データの ENBF 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。なお、内包表現については §3.6.3 を参照してください。

■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 "()"

なお、「式のリスト」として「空白」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 ")"

リストなどの内包表現では内包表現を全て評価した値で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` 節が返す変数を用いた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。なお、内包表現については §3.6.3 を参照してください。

■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。実例を次に示しておきます：

```
>>> 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
>>>

```

yield 文を持つ函数から生成されたジェネレータ型のオブジェクトはメソッド next() を呼び出すことで yield 文に引渡した式を評価した値が返却されます。

Python の内包表現

内包表現は集合の成分を列記せずに、その集合の成分が充すべき命題を記載することで集合を定める方法です。具体的に説明すると、集合を ‘{0, 1, 2, 3, 4, 5}’ と成分を列記する方法、‘{ $x : x \in \mathbb{N} \wedge x \leq 5$ }’ と論理式 ‘ $x \in \mathbb{N} \wedge x \leq 5$ ’ を使って集合の元が充たすべき性質を記載する方法の二通りの集合の表現方法があり、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と呼びます。Python で用いられる内包表現は for 節を主体に if 節を篩として利用する表現です。つまり、for 節で篩にかけるべき集合を生成し、if 節を篩として適切な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

ここで「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いった式の変数として用いない表記も可能です。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 ' $A \vee, \dots, \vee Z$ '^{*39} とするときに

$$((x, y, \dots, z)/\alpha \wedge \alpha \in A) \vee \dots \vee ((x, y, \dots, z)/\omega \wedge \omega \in Z)$$

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 が標的リストに代入可能であり, 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, 逆に最初の式が 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。これらのことから論理式が扱えるように記載がありますが, 実際は論理式である必要性はなくて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__(self, index)` 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]
```

*39 論理式は論理和 \vee の式として変形することができます。

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 節をフィルター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、フィルターとして持つ場合は if 節を for 節のうしろに配置し、if-else を持つ場合は for 節をうしろに配置します。

```
>>> [i if i>3 else -1 for i in [1,2,3,4,5]]
[-1, -1, -1, 4, 5]
>>> [i if i>3 else -1 for i in [1,2,3]and[1,2,3,4,5]]
[-1, -1, -1, 4, 5]
>>> [i if i>3 else -1 for i in [1,2,3]or[1,2,3,4,5]]
[-1, -1, -1]
>>> [i if i>3 else -1 for i in not[1,2,3]or[1,2,3,4,5]]
```

ここでは 4 種類の例を示しています。最初の例が通常の if 節による内包表現です。二番目が演算子 “and” を持つ場合、三番目が演算子 “or” を持つ場合、四番目が演算子 “not” を持つ場合の例になっています。この論理演算子を含む場合は、論理式として全体が True であるか False であるかが判明するまで処理が行われます。演算子 “or” のときはどちらか一方が True であれば良いので True と判断された式の評価で外延が定まり、演算子 “and” のときは双方が True でなければ式が True と判断されないために末尾の式の評価結果で外延が定まります。演算子 “not” は True と False の入替が生じることになりますが、基本的に演算子 “not” のある式の評価が無視されます。

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]”

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ます。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)になります:

```
>>> 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 系の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。ただし、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、モジュール numpy を必要とします。なお、モジュール numpy の配列でもスライス表記が可能で、Matplotlib を組合せることで MATLAB 本体と同等の機能が実現できます。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つています：

```
>>> 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、函数 `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。

呼出

函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」と「キーワード引数」と「式」があります。まず、「定位引数」は通常の函数やメソッドの引数が対応します。ところで「キーワード引数」はオプションの引数になります。つまり、キーワード引数は属性等の既定値を変更させると

に用います。

3.6.4 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になる二項演算子から構成される式、「**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演算式です。なお、ここでの 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、二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です:

二項演算式の EBNF

```

二項演算式 ::= 乗法演算式 | 加法演算式 | 幂演算式 | 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 ")" )
幂演算式 ::= ( 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 “/”, 整数賞演算子 “//” 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。特に Python 2.x で演算子 “/” は被演算子が整数のときは結果も整数で、これは演算子 “//” と同じ結果になります。ところで Python 3.x, C や FORTRAN では、演算子 “/” を使った式は浮動小数点数になるために Python 2.7 の結果と異なります。このことは Python 2.7 で演算子 “/” を使うときは被演算子のどちらかが確実に浮動小数点数でなければ想定した結果が得られないことを意味します。そのために 2.7 では関数 float() で型の変換をあらかじめ行う工夫が必要になるかもしれません。

■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基本的に数オブジェクトか 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。

■幂演算式: Python で幂乗は演算子 “**” を用い、演算子 “^” は幂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幂乗の演算子として演算子 “^” が用いられているため、これらを混同しないように注意する必要があります。

■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、二進数で表示に対して処理を行う演算子です。演算子 “&”, “|” と “^” は二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“>>”は被演算子を二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^{*40}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：

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 文の代用にすることが可能です。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、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が、閉包の論理和式や論理積式を置くことも可能です。ただし、論理積なら最も右端の項、論理和であれば最も左端の項の処理が行われます。この点は Python の内包で説明した状況と異なります：

```

>>> 1 in (a or b)
True
>>> 1 in (b or a)
False
>>> 1 in (a and b)

```

*40 最大の整数型の数です。

```
False
>>> 1 in (b and a)
True
```

これは演算子“and”と“or”的性格によるもので被演算子のどちらか一方が真理値でなかったときに、演算子“and”は右被演算子を、演算子“or”は左被演算子を結果として返すためです。

■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に加え、同値性を示す演算子として演算子“==”，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。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、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 if y 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-式が構成できる式です：

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 "()" でグループ化していれば、改行を入れることも可能ですが、通常は一行で記述します。

```
>>> 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 2.7 にはありません。

3.6.5 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。Python では複数の単純文をセミコロン ";" で区切って続けることができます：

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.x ではそれぞれ函数 print() と函数 exec() で置き換えられています。また、nonlocal 文は Python 3.x 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函数で用いられます：

式文の EBNF

```
式文 ::= 式のリスト
```

■代入文：名前にオブジェクトを束縛す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 output 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 output 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 が生じるというものです。

■pass 文: 「何もしない」文です:

pass 文の EBNF

pass 文 ::= "pass"

この文は構文的に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対して出力する文です.

print 文の EBNF

print 文 ::= "print" [(式 (", 式)* [",,]) | "»" 式 [(", 式)+ [",,]])

なお、この print 文は Python3.x では関数 print() で置換えられます。

■**return 文** 関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]

なお、return 文は yield 文と同一関数内部で混在させることができません。

■**yield 文**: 産出関数の定義に利用されます:

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 文を用いることで定義された関数は産出関数と呼ばれる関数になります、産出関数からは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()
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、`next()` メソッドを使って処理が行われます。

■**raise 文**: 利用者が意図的に例外の送出を行うときに用いる文です:

raise 文の EBNF

```
raise 文 ::= "raise" [式 ["," 式 ["," 式]]]
```

例外そのものについては §3.14.3 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。

■**break 文**: for 文、while 文といった反復処理から抜けるために用いられる文で引数を必要としません。

break 文の EBNF

```
break 文 ::= "break"
```

■**continue 文**: break 文と同様に引数を持たない文で、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**: モジュールの読み込みに用いられる文です:

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 文で読み込まれると、オブジェクトやメソッドはモジュール名を付点名として用いることになりますが、"as"以下に別の名前を指定することで、その名前を付点名することができます。

■**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わされることを指示する文です:

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中では 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はいけません。

■**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)*
```

3.7 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。複合文は複数の行に分かれて記述されますが、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)*[";"]
```

ここで示すように複複合文はクラス定義や関数定義といったオブジェクトの定義、条件分岐、反復、例外処理といった処理で用いられます。

3.7.1 条件分岐に関する複合文

■if 文：条件分岐を行うための文です：

if 文の EBNF

```
if 文 ::= "if" 式 ":" 一揃いの文
          ( "else" 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この if 文のみです。

3.7.2 反復処理に関する複合文

■**while 文**: 後述の 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 のときだけ while 文内部の処理を行います:

—— while 文の EBNF ———

while 文 ::= "while" 式 ":" 一揃いの文

■**for 文**: 前述の 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

Python の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になります。

3.7.3 例外処理に関する複合文

■**try 文**: Python の例外を処理する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 §3.14.3 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 で受けて処理を行うとい手順になります。

3.7.4 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]
```

3.7.5 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 ")"
```

函数定義は Python で実行可能な文ですが、函数定義が函数本体を実行するものではなく、函数が呼び出されたときのみ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 文で行われます。

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
```

クラス定義では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。すると実行フレームは無視されますが、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。

このように Python は非常に見通しの良い構文を持っています。また、反復処理は for 文と while 文しかなく、条件文も if 文しかないといった非常に単純化されたプログラマ言語もあります。次の節では、この Python の式で触れたオブジェクト、名前空間や例外といったことの詳細について述べることにします。

3.8 オブジェクトについて

3.8.1 オブジェクトの概要

ここでは Python のオブジェクトについて概要を述べます。なお、オブジェクトの生成に関する EBNF は §3.6.2 を参照して下さい。まず、Python で扱う対象は全てオブジェクトであり、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。ここで最初の「**識別値**」はオブジェクト生成時に定まり、変更ができないオブジェクト固有の値です。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として表現されます。「**型**」はオブジェクトの生成時に指示され、こちらも変更できません。そして型 (type) 自体もまた型です。オブジェクトはその持ち得る値に関して、その値が「**変更可能 (mutable)**」であるオブジェクトと「**変更不可能 (immutable)**」であるオブジェクトに分類することができます。ただし、この変更不可能なものでも別のオブジェクトへの参照を持つコンテナと呼ばれるオブジェクトであれば、そのオブジェクトの参照先を変更することで実質的に変更可能になりますが、オブジェクトの構造自体を変えるものではありません。そのため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ることができます。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という性質を持ちます。ここでオブジェクトが「**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型の値、すなわち、「**要約値 (hash value)**」を持つときです。この要約値の重要な性質は、二つのオブジェクトが同じ値を持っているときに一致するという性質です。ただし、要約値が一致することとオブジェクトの識別値が一致することは別です。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値で、このオブジェクトの要約値を計算するためのメソッドは `__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。これは `None` の説明でも行いますが、`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符合します。

オブジェクトの参照で「**名前空間**」が使われます。Python 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います。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、オブジェクトの型の参照はその名前を引数にして函数 `type()` を使って調べることができます。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と言え、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は Python の連想配列である辞書と呼ばれる型のオブジェクトが用いられています。

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ます。そして、不要になったオブジェクトを利用者が直接、破棄できません。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す。この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(塵回収, garbage collection)**」によるオブジェクトの破棄が行われます。この GC の実行や停止を利用者が行えますが、通常はオブジェクト総数があらかじめ設定された閾値を越えた時点で GC が自動的に実行されます。

3.8.2 オブジェクトの生成と廃棄を行うメソッド

オブジェクト指向プログラミング言語では、オブジェクトの生成を行う函数やメソッドを「**構築子/コンストラクタ (constructor)**」と呼びます。このオブジェクトの生成ではオブジェクトへの「**メモリの割当 (allocation)**」とオブジェクトの属性の設定等の「**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のことを「**消滅子/デストラクタ (destructor)**」^{*41}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__()` でオブジェクトの生成、メソッド `__init__()` でオブジェクトの初期化を行います。オブジェクトの生成でメソッド `__new__()` が呼び出されると自動的にメソッド `__init__()` が呼び出されます。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ていますが厳密には異なります。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継承によって用いられます。

また、メソッド `__del__()` が Python での消滅子に該当するメソッドになりますが、このメソッドはオブジェクトを直接破棄するメソッドではありません。通常は消滅子で指定された作業の他に、後述のオブジェクトに割り当てられた参照カウントを 1 減じる動作を行います。また、消滅子 `__del__()` を持つオブジェクトの破棄で消去子が使われるとは限らないために消去子にオブジェクトが破棄されたときに行うべき処理を記載しても処理されない可能性があります。これは Python を終了するとき、消去子を持つオブジェクトに巡回参照と呼ばれるオブジェクト間の参照関係があるときです。

3.8.3 GC(garbage collection)について

Python は前述のように「**GC(塵回収, garbage collection)**」がある時点で自動的に実行されて不要になったオブジェクトの破棄とメモリの解放を行います。まず、CPython の GC に「**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、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が 1 増やされます。逆に、その名前に別のオブジェクトが束縛されることで参照関係が解消されると参照カウントが 1 減じられ、最終的に参照カウントが 0 になった時点でオブジェクトの破棄が許可されてメソッド `__del__()` が呼び出されます。なお、明示的な消滅子 (=函数 `del()`) の実行は、それによってオブジェクトが直接破棄されるのではなく、名前との参照関係を外るために参照カウントが 1 減じるだけで、消滅子で指示されている処理は参照カウントが 0 になった時点で実行されます。なお、オブジェクトの参照カウントは sys モジュールの函数 `getrefcount()` で確認することができます。このときに函数 `getrefcount()` で調べた値は本来の参照カウントよりも +1 にな

^{*41} Java ではファイナライザ (finalizer) と呼びます。

ります。これは関数 getrefcount() による参照も追加されるためです。ここで簡単な例で確認をしておきましょう：

```
>>> 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
>>>
```

ここで示すように循環参照がない通常の名前とオブジェクトの間の参照関係だけであれば、名前との参照関係が外れた時点ではオブジェクトの参照カウンタが 0 になり、その時点でメソッド __del__() が呼び出されていることが判ります。

参照カウント方式では参照カウントが 0 になるとオブジェクトの破棄が許可されるだけで、実際にオブジェクトが破棄されるのは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か利用者が gc モジュールの関数 collector() を起動したときです。この閾値は gc モジュールの関数 get_threshold() で確認することができます：

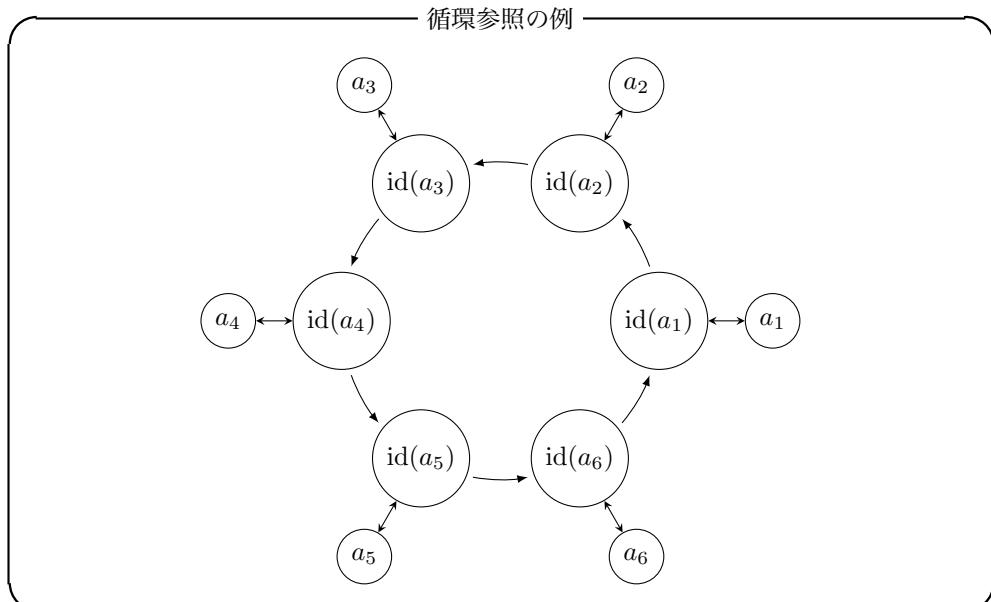
```
>>> import gc
>>> gc.set_debug(gc.DEBUG_STATS)
>>> gc.collect()
gc: collecting generation 2...
gc: objects in each generation: 249 3282 0
gc: done, 0.0017s elapsed.
0
>>> gc.get_threshold()
```

```
(700, 10, 10)
```

```
>>>
```

この例では `gc` モジュールを読み込み、`gc` 向けのデバッグ設定を行って函数 `collect()` でオブジェクトの回収を行っています。函数 `collect()` を実行すると、函数 `set_debug()` で統計量の出力が設定されているためにその情報が表示されています。また、函数 `gc_threshold()` は GC を行う閾値を表示する函数で、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 世代の閾値となっています。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。

ところで、単純な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）で置き換えることで得られるグラフで考えると明瞭になります。まず、得られたグラフに円環（=閉道）がなれば問題ありませんが、グラフに閉道ある場合に問題が生じます。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きの参照関係をグラフ化すると以下のように円環が現れます：



この例では名前 $a_{i,i \in \{1, \dots, 6\}}$ 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 `id()` を用いた理由は函数 `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自己に戻る参照のことを「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ます。したがって、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手段の一つがオブジェクトを世代別に分ける方法です。最初に生成したばかり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 が開始されます。この GC ではオブジェクトが世代毎に到達可能であるかどうかを調べ、このときに第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を行います。その結果、到達不能なオブジェクトだけが残されるため、これらのオブジェクトを回収すればよいことになります。しかし、厄介なのはオブジェクトにメソッド `__del__()` が定義されている場合です。これは循環参照を行っているオブジェクトで、どちらのオブジェクトのメソッド `__del__()` を利用すべきかが判断できないためです。そのために Python は循環参照に陥っている到達不能なオブジェクトでメソッド `__del__()` を持つものを GC で回収しません。

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。ここで設定しているのは `gc.DEBUG_STATS` で、統計情報を出力させるためです。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 す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るで問題がありません。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、不要になったクラス 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 減じられるだけで、循環参照の問題である「**どちらの消滅子を使えばよいのか？**」のために、その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。そのためファイル等の外部リソースを参照する場合はメソッド __del__() で解放するのではなく、適宜、close() 等の解放するためのメソッドを使ってリソースを開放することが妥当です。

3.8.4 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できます。この性質はオブジェクトの型で決定される性質で、あとで変更できません。

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 が追加されています。このことは ‘`a is b`’^{*42}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
```

^{*42} 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という違いがあります。

```
>>> 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ぼ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’ のままであります。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 への参照が途切れますが、オブジェクト 128 への参照が名前 `c` と `d` 以外に存在しないのであれば、このオブジェクトへの参照が途切れた状態になります。この状態を「**到達不能の状態 (unreachable)**」と呼び、オブジェクトが到達不能な状態になると「**GC(塵回収, garbage-collection)**」でオブジェクトの回収処理が自動的に行われます。

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と思えばよく、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ますが、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旧来のものと異なって以前と同様と言えないでしょう。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のためにリストのように成分を直接変更できませんが、コンテナであるために参照しているオブジェクトの値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。

3.8.5 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 組込の定数にこの他にブール型の 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 と前述のブール型を含みます。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^{*43}。数は Scheme の「**数値塔 (Numerical Tower)**」にしたがったもので、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 §3.6.2 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するために用います。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'>
>>> 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

*43 PEP 3141: 「A Type Hierarchy for Numbers」参照。

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が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** にて空集合 \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 と同様の性質を持っています。

■`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` と同値ではありません。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` と同値ではありません。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 系の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 “:” は一次元の添字の省略記号に対し、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^{*44}。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 とは違います。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.x 系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.x 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。ところが Python 3.x 系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。ここでの 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。そして、この型のオブジェクトの構築子は int() で、関数 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

^{*44} Yorick は C に類似した言語で、高次元の配列処理を簡潔に行うために MATLAB 表記を使う軽快な言語です。

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^{*45}。

- 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、SageMath では通常の演算で用いる整数や任意精度の整数として Python 標準の整数型や長整数型ではなく GMP 由来の整数 (`Integer`) 型を用いています。この `Integer` 型では任意桁の整数も整数リテラルで末尾に “L” や “T” を付ける必要がありません。逆に言えば SageMath で整数を扱うときは、それが Python の SageMath の整数型 `Integer` でなくとも良いものか注意を払ってください。少なくとも数論関連の処理であれば SageMath の整数型 `Integer` にしなければ、処理速度や使えるメソッドも含めて無意味なことになります。なお、3.x では整数型は実質的に長整数型に統合されますが、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()`^{*46} 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

^{*45} 数値処理言語によっては補数表現になるものがありますが、Python では自動的に長整数型に切り替わる言語です。なお、Python 3.x で整数型は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。

^{*46} `Bool` 型は 19 世紀の英国の數学者 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。

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 $z.\text{real}$ 、虚部は $z.\text{imag}$ で取り出せますが、複素数型の性質上、これらは実型、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、SageMath で多項式の型を導入しているために代数的数としての純虚数 I, i が虚数リテラルと別にあります。ここで代数的数は代数方程式の厳密解そのもので、浮動小数点数と違って近似の数ではありません。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 の具象化クラスとされ、クラス Complex がクラス Number の子クラスになります。

■列 (sequence): 列は自然数 N からオブジェクトへの対応を持つオブジェクトです。具体的には列 S は上限の自然数 N を持ち、自然数 $i \leq N$ に対してオブジェクト $S(i)$ が対応します。列の濃度を列の長さと呼び、先程の例では $N + 1$ が列 S の長さになります。この列の長さは函数 len() を使って調べられます。Python の列は 0 から開始する整数と結び付けられ、列の成分の指定は 'a[2]' のように列の名前に対して括弧 "["] " に対応する整数を指定することで行えます。

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 系の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 '123456789'[0:8:2] で部分列 '1357' を取り出すことができます。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' になるのです⁴⁷。また、列を逆向きに取り出す場合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 -1 間隔の自然数の列 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

⁴⁷ Python では列の開始は 1 ではなく C と同様に 0 から開始します。

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可能です。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]` と通常は記述しますが、列の端部 0 を省略して `a[:3]` と記述することができます。

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 オブジェクトの列を角括弧 “[]” で括ることで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ることができます。
- 文字列型 (string): 接頭辞に ‘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: 接頭辞に ’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): 任意の Python オブジェクトと記号 “,” を並べたものを一組とし、これらを一列に並べて丸括弧 “()” で括ったものから生成されるオブジェクトです。成分が一つのタプルは「**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。構成するオブジェクト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()` で調べることができます。Python の集合には以下に示す型があります：

- 集合型 (set): 変更可能な型で、組込の構築子 `set()` で生成されます。
- `FrozenSet` 型: 変更不能な型で、組込の構築子 `frozenset()` で生成されます。集合型と違って要約可能 (hashable) である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。通常の列の成分の取り出しあは列 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えましたが、連想配列では添字集合が自然数の部分集合に限定されずに Python の有限個のオブジェクトの集合で与えることができます。そして、連想配列 A の参照は k を添字の集合 I の元とするときに $A[k]$ で行えます。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、連想配列の濃度は列と同様に函数 `len()` で調べることができます。なお、現時点では 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、この辞書型の構築子は `dict()` ここで添字集合のことを「**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): 呼出が可能なオブジェクトの総称です。ここでの呼出とは、形式的に名前等を函数の書式で扱うことによって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操作を行うことです。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。呼出可能なオブジェクトには、その実装方法と挙動が動的から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で生成されるオブジェクトです。ここでの「通常」は `yield` 函数を内部に包含しないという意味で、`yield` 函数を包含する利用者定義の函数は後述の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^{*48}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): クラスやクラス インスタンス、あるいは `None` を「**一次語 (primary)**」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

*48 引数の個数を函数のアリティ (arity) と呼びます。

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)
True
>>>
```

ここで示すように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で記載することで、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): 利用者による函数定義で内部に `yield` 文を持つ函数です。

```
>>> 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
>>>
```

この例で示すように `yield` 文を用いた函数が生成す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): 利用者によるクラス定義で、メソッド `__iter__()` とメソッド `next()` の一対を持つクラスです:

```
>>> 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 の函数のラッパーになります。このような函数の例には組込函数 dir() や函数 math.sin()^{*49}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を生成するために用いられ、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スタイプ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のために注意が必要です。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ます。つまり、Python スクリプトを記した一つ一つのファイルがモジュールになります。なお、Python スクリプトは最初にインタプリタに読み込まれる際にコードオブジェクト翻訳され、コードオブジェクトは.pyc ファイルに蓄えられます。ただし、モジュールにコードオブジェクトは含まれていません。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が「**パッケージ**」です。

■クラス (class): Python 2.x 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.x 系ではクラスオブジェクトが廃止されてクラスタイプ型のみになります。古典的クラス型は Python 2.x 系で基底クラスとして object を継承しないときに生成される型で、クラスタイプ型と違い、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るま

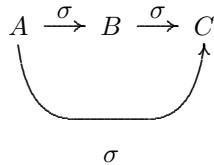
^{*49} 標準モジュール math に包含される函数 sin() を指示する名前になります。

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クラスを生成することもできます。

ここで、継承する側のクラスを「**派生クラス**」と呼び、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なクラスで、その意味で上位のクラスです。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、すなわちクラス C_1 がクラス C_0 の派生クラスであるということを表現します。このときに次の図式が可換になります:



まず、この図式は3個のクラスの派生関係を示すもので、 $A \xrightarrow{\sigma} B$ かつ $B \xrightarrow{\sigma} C$ であれば $A \xrightarrow{\sigma} C$ となること、つまり、この関係 $\xrightarrow{\sigma}$ 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は許容されません^{*50}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は不可でも、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。

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 2.x の対象で、Python 3.x では廃止されています。このクラスインスタンスは古典的クラスを呼出すこと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いるために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、それに伴ってクラスの辞書が更新されることはありません。

■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。ここでは概要を説明しておきます：

*50 参照すべきオブジェクトがなければ例外が発生します。

- **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す。バイトコードはインタプリタ内部のデータ形式で、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、そのバイトコードを Python 仮想マシンで実行します。また、import 文で読み込まれたファイルは ‘.pic’ ファイルがなければモジュールとして解析とバイトコードへ翻訳され、このバイトコードは ‘.pyc’ ファイルに蓄えられます。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か ‘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。なお、このバイトコードは異なる Python 仮想マシンで互換性を保証しません。
- **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び出され、そのオブジェクトの実行が終えた時点で呼び出した側のプログラムで呼び出し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。まず、 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事項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た構文で、配列処理で類似の機能を与えることになります。
- **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、 static-

`icemethod` と同様にインスタンス化しなくても利用可能です。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。また、クラスメソッドは静的メソッドと同様にインスタンス変数の参照ができません。

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。ここでの定義で示すように ‘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。しかし、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置くことで定義ができます。

リスト、集合や辞書の内包表現について

概念にはその外延の成員を列記した外延的表記と、その概念がどのようなものであるかを述べる文による内包的表記の二通りがあります。Python のリスト、集合や辞書には、それらに含まれるオブジェクトを直接、列記する外延表現と、その成員になるオブジェクトを列記するのではなく、Python の文で表記する内包表現の二種類があります。簡単にリストの例で説明すると、外延的表記は '[1, 2, 3, 4]' と具体的に記載する方法、内包的表記は '[i for i in range(1,5)]' といった方法です。このように内包的表現は列を生成する文を利用した記述で、外延的表現は具体的に全ての成員を表記したり、リストであればメソッド append() で成員を追加する表記が対応します。この内包表現については、Python の式の節で詳細を述べていますが、外延的表現は Python 言語のオーバーヘッドが生じ易いこと、一度にデータを生成しておく必要がないことで内包的表現がより高速に処理が行えます。

3.9 特殊メソッド

3.9.1 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 Python で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、定義する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用いられ、定義する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、定義したクラスで基底クラスのものとは別の改変されたメソッドが利用可能になります。たとえば、同じクラスのオブジェクトの比較では「拡張比較」と呼ばれる一群のメソッドがあります。これらのメソッドは、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

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ることができます。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。それから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’ になります。そして、‘cls’、‘self’ と ‘other’ は ‘obj’ と異なり固定です。

3.9.2 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正在すると実行時に `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、いずれ回収されるもので、C++ の消滅子のように不要になった時点でオブジェクトが回収されるものではありません。そのため C++ の消滅子と同様の働きをする消滅子を Python は持たないと言えます。一般的に基

底クラスがメソッド`__del__(self)`を持っているときは派生クラスのメソッド`__del__(self)`で明示的に基底クラスのメソッド`__del__(self)`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。

3.9.3 表示に関するもの

通常、インタプリタでオブジェクトが束縛された名前を入力しても、そのオブジェクトが格納された番地以上の情報が返却されません。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あらかじめ定義しておくメソッドです。そのためこれら のメソッドの引数は`self`のみになります。

■`obj.__repr__(self)` 組込関数`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、このメソッドを用いることで利用者とつ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)` 組込関数`str()`と`print`文^{*51}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Pythonの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__(self)`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、その代わりにメソッド`__str__(self)`が利用されます。

3.9.4 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。したがって、引数は`self`と`other`の二つのみです。

比較の演算子

演算	記号	Python の式	特殊メソッド
小なり	<	<code>a < b</code>	<code>obj.__lt__(self, other)</code>
以下	<code><=</code>	<code>a <= b</code>	<code>obj.__le__(self, other)</code>
等しい	<code>==</code>	<code>a == b</code>	<code>obj.__eq__(self, other)</code>
等しくない	<code>!=</code>	<code>a != b</code>	<code>obj.__ne__(self, other)</code>
大なり	<code>></code>	<code>a > b</code>	<code>obj.__gt__(self, other)</code>
以上	<code>>=</code>	<code>a >= b</code>	<code>obj.__ge__(self, other)</code>

*51 3.x では関数`print()`。

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。ただし、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函数モジュールfunctoolsの函数functools.total.ordering()を使って残りの比較演算子の自動定義を行うことが可能です。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行われます。

3.9.5 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)
幕	**	a ** b	obj.__pow__(self, other)

ここでPython 2.xと3.xでは演算子“/”の挙動が異なります。Python 2.xで演算子“/”の二つの非演算子が整数のときは結果は整数が返却されます。しかし、Python 3.xで整数の除算は演算子“//”で、演算子“/”は割り切れないときに浮動小数点数になります。また、SageMathの数学的オブジェクトの幕演算子として演算子“^”が用いられますが、前述のように演算子“^”を上書きしたものではないためにPython固有のオブジェクトの幕は演算子“**”のままです。

これらの整数演算に加えて整数オブジェクトには、被演算子を二進数として表現したときの二項演算も加わります:

二進数整数演算子(二項演算)

演算	記号	Python の式	特殊メソッド
左シフト	<code><<</code>	<code>a << b</code>	<code>obj.__lshift__(self, other)</code>
右シフト	<code>>></code>	<code>a >> b</code>	<code>obj.__rshift__(self, other)</code>
論理積	<code>&</code>	<code>a & b</code>	<code>obj.__and__(self, other)</code>
排他論理和	<code>^</code>	<code>a ^ b</code>	<code>obj.__xor__(self, other)</code>
論理和	<code> </code>	<code>a b</code>	<code>obj.__or__(self, other)</code>

排他論理和(XOR)に対応する演算子“`^`”はSageMathの多項式や数オブジェクト等で乗算として用いられています。そのためにSageMathでプログラムを構築する際にPythonの数オブジェクトに対して作用させた場合には排他的論理和として動作することに注意が必要です。

3.9.6 浮動小数点数演算子に関連するもの

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	<code>+</code>	<code>a + b</code>	<code>obj.__radd__(self, other)</code>
差	<code>-</code>	<code>a - b</code>	<code>obj.__rsub__(self, other)</code>
積	<code>*</code>	<code>a * b</code>	<code>obj.__rmul__(self, other)</code>
商	<code>/</code>	<code>a / b</code>	<code>obj.__rtruediv__(self, other)</code>
商	<code>//</code>	<code>a // b</code>	<code>obj.__rfloordiv__(self, other)</code>
剰余	<code>mod</code>	<code>a mod b</code>	<code>obj.__rmod__(self, other)</code>
幕	<code>**</code>	<code>a ** b</code>	<code>obj.__rpow__(self, other)</code>

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ものになっています。

3.9.7 累算算術代入演算子に関連するもの

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
$a = (a + b)$	$+=$	$a += b$	<code>obj.__iadd__(self, other)</code>
$a = (a - b)$	$-=$	$a -= b$	<code>obj.__isub__(self, other)</code>
$a = (a * b)$	$*=$	$a *= b$	<code>obj.__imul__(self, other)</code>
$a = (a / b)$	$/=$	$a /= b$	<code>obj.__itruediv__(self, other)</code>
$a = (a // b)$	$//=$	$a //= b$	<code>obj.__ifloordiv__(self, other)</code>
$a = (a \bmod b)$	$\bmod=$	$a \bmod= b$	<code>obj.__imod__(self, other)</code>
$a = (a ** b)$	$**=$	$a **= b$	<code>obj.__ipow__(self, other)</code>

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“=”の左側にあります。そして、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。

二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>

二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。

3.9.8 単項演算子

単項演算子は二項演算子と異なり整数、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 $	$\text{abs}()$	$\text{abs}(a)$	<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がありません。ここで `~a`, すなわち, `~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。

3.9.9 属性値の取得と設定に関連するメソッド

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 `AttributeError` 例外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ときに ‘`self.name = value`’ とした場合は自己参照が生じるために行ってはなりません。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なければなりません。

■`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 `AttributeError` 例外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。

3.9.10 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` でなければなりません。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^{*52}。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^{*53}。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.x 系では `__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()`^{*54}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` になります。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。

3.10 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のことを「規約 (プロトコル, 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

^{*52} <http://effbot.org/zone/python-hash.htm> 参照

^{*53} 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__()` が呼び出せることで行っているためです。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。

^{*54} 何故か `bool` でない!

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。具体的には、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__() の何れかを上書きするメソッドです。ここで記述子はメソッド __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。また、データ記述子で、そのメソッド __set__() の呼出によって AttributeError 例外が送出されるものを「**読み専用データ記述子**」と呼びます。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：

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。

まず、インスタンス束縛における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することができますが、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。ここで組込関数 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可能になっています。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。また属性検索にて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。

3.10.1 記述子 (descriptor) の実装

■`obj.__get__(self, instance, owner)` クラスの属性やインスタンスの属性への参照時に呼出されます。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`onwer`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()`と似た動作になります。

3.11 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たらなければ、より上位にある基底クラスに対して参照が行われます。ここで属性の検索は検索しているクラスに無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**」が関わる検索になります。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、 $\mathcal{L}(C)$ と記述することにします。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) として表現されますが、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 `$\overline{L} \stackrel{\text{def}}{=} \text{head}(L), \underline{L} \stackrel{\text{def}}{=} \text{tail}(L)$` 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 `$C_1 \dots C_n \setminus B_1 \dots B_m$` と表記することにします。たとえば、`abcd \ bd`は`ac`になります。また、検索では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することに

なります。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 ‘ $\cdots W_{(i-1)}W_iW_{(i+1)}\cdots W_{(j-1)}W_iW_{(j+1)}\cdots \searrow \cdots W_{(i-1)}W_iW_{(i+1)}\cdots W_{(j-1)}W_{(j+1)}\cdots$ ’ 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。そして作用素 \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. $\mathcal{L}(C(B_1, \dots, B_n)) = C + \mathcal{M}(\mathcal{L}(B_1), \dots, \mathcal{L}(B_n))$

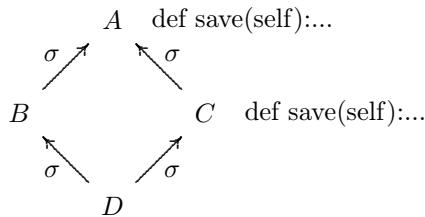
最初の 1. は継承関係を持たないクラス C のときに MRO は C のみであるということを示します。つぎの 2. は $C_0 \xrightarrow{\sigma} C_1$ のとき、つまり、クラス C_0 が C_1 の派生クラスのときに最初にクラス C_0 を検索し、それからクラス C_1 の検索順序に従うということを意味し、つぎの 3. は関係 “ \searrow ” で作用素 $*$ の値は不变であることを示し、最後の 4. が多重継承があるときの処理になります。次に作用素 \mathcal{M} を導入しておきましょう。この作用素 \mathcal{M} の働きは、継承関係を遡る MRO を基本に多重継承のあるクラスにて継承の親達を示すタプルをそのまま用いて基底クラスに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で行われるので「**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. $\mathcal{M}(\cdots, L, \cdots) = \mathcal{M}(\cdots, \underline{L}, \cdots)$
- c. $\mathcal{M}(L_1, L_2, \cdots, L_n) = \underline{L_1} + \mathcal{M}(L_2 \setminus L_1, \cdots, L_n \setminus L_1)$

最初の a. は検索順序を示す語 W 一つが引数であれば、関係 “ \searrow ” の極限 \underline{W} を返すという性質です。そして次の b. は関係 “ \searrow ” で作用素 \mathcal{M} の値は不变であることを示しています。そして最後の 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してゆくという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があるときに有効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 の派生クラスであることと、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 で用いられることがありません。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。ところ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、クラス A の派生クラス C で再定義されたメソッド `save()` が用いられないということになります。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ることになります。そのためクラスタイプでは「**C3 MRO(Method Resolution Order)**」^{*55}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：

^{*55} <https://www.python.org/download/releases/2.3/mro/> や PEP-253 を参照

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'. $\mathcal{L}(C(B_1, \dots, B_n)) = C + \mathcal{M}_{c3}(\mathcal{L}(B_1), \dots, \mathcal{L}(B_n))$

ここで作用素 \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} の場合と同様です。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. $\mathcal{M}_{c3}(W) = \underline{W}$
- 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。

この \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のかを纏めておきましょう：

 \mathcal{M}_{c3} の計算手順

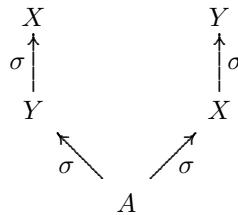
1. $i = 1$ とする。
2. $h_i = \overline{L_i}$ とする。
3. $j \neq i$ に対して $h_i \notin \underline{L_j}$ のときに $k \in (1, \dots, n)$ に対して $\overline{L_k} = h_i$ であれば、 \mathcal{M}_{c3} の引数にある L_k を $\underline{L_k}$ で置換し、 $h_i + \mathcal{M}_{c3}(L_1, \dots, L_n)$ を作用素 \mathcal{M}_{c3} の結果として返却する。
4. $h_i \in \underline{L_j}(i \neq j)$ 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。

ダイアモンド状の継承関係の 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、古典的クラス型の MRO よりも妥当な結果が得られることになります。さらにこの C3 MRO の長所は間違った継承関係を検出することが可能です。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 $Y \xrightarrow{\sigma} A$ 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も C3 MRO では適切な処理が行えることを意味しています。

3.12 名前空間とスコープ

3.12.1 名前と名前空間

「**名前空間 (name space)**」は名前 (name) が不用意に一致することがないように統一的に名前を決定するための手法です。「**名前空間**」という表記は仰々しく思えますが、実際は一般的な概念です。まず、Python は本体を小さくしてライブラリで拡張しています。ここでファイルを読み込んだときにオブジェクトの階層もなしに名前をそのまま展開するものとしましょう。このときに名前には階層構造も何もないでの、あなたがファイル A とファイル B に同名でも個別の場合に対応した函数を定義しているとファイル A の函数とファイル B の函数は名前が一致してしまうので双方が混在することは同じ名前である限りはできない相談です。この別のオブジェクトの名前が一致してしまう現象を「**名前の衝突**」

突」と呼びます。この場合はケース1の場合はファイルAを読み込み、ケース2の場合はファイルBを読み込んで処理を行うといったプログラムが必要になるでしょう。この様な処理もライブラリが巨大になってしまえば非常に煩雑なことになるでしょう。だからと言って、名前の衝突を避けるためにある一定の命名規則を作つて命名するという方法もあるでしょう。ただ、この方法は利用者の意識に依存するもので厳密に守られるとは限りません。それよりもPython上に展開する名前にファイル名に依存する識別子を付けてファイル単位で区別するという至つて機械的な方法です。また、このときにファイル単位で名前を探すようにすれば別ファイルに同名の名前のオブジェクトがあつても検索する際に、同じファイル名のオブジェクトが優先されるようにすればよいでしょう。このように名前の衝突を解決し、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。

名前のBNF

名前 ::= [識別子 分離記号] 識別子

次にオブジェクトの参照を行う際に参照可能な範囲、すなわち、参照の有効範囲が問題となります。この参照の有効範囲のことを「スコープ(scope)」と呼びます。ここで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しかありません。

3.13 名前付けと束縛

■名前(name): オブジェクトの参照で用いられます。名前への束縛(binding)によってオブジェクトと名前が結び付けられ、その結果、その名前を指定することでオブジェクトへの参照が行われるようになります。この名前の参照で問題になるのが参照の範囲、すなわちスコープ(scope)です。ここで参照すべき名前が名前空間に存在しない場合に送出される例外が「**NameError**例外」です。また、名前が名前空間に存在していても値が設定されていない変数を参照したときに送出される例外が「**UnboundLocalError**例外」です。

■ブロック(block): プログラム内の一つの実行単位になる区画で、モジュール、クラスと函数定義はブロックになります。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またブロックになります。

■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の文で構成されたもので、これらコードブロックの実行は「**実行フレーム(execution frame)**」上で実行されます。

■スコープ (scope): 名前の参照可能な範囲/領域のことです。たとえば、局所変数があるブロック内部で定義されているとき、その変数のスコープはその変数に束縛されたオブジェクトが参照可能な範囲であるそのブロックを含みます。したがって函数やクラス内で定義された名前のスコープは、それらのブロック内に制限されます。とは言え、メソッドのコードブロックを含むような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、その名前の参照では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における局所変数となります。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 NameError 例外が出力されます。また、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 UnboundLocalError 例外が出力されます。この UnboundLocalError は NameError のサブクラス (NameError クラスの種の一つ) になっています。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うことになります。

3.14 例外

3.14.1 例外の概要

「例外 (exception)」とはプログラムの処理時に生じた異常のことで、この異常にはプログラムの本質的な欠陥も含まれますが、その他に、0による割算や開くべきファイルがないといった処理の前提から外れた状況から生じた異常、さらには利用者側の理由でプログラムを一時的に停止させるために意図的に異常な状態を発生させることもあります。この例外が発生した状況で、その例外を分析して別の処理につなげる処理を「例外処理」と呼びます。プログラミング言語の多くで例外を処理するために `try` 文があり、Pythonも同様に `try` 文があり、その `try` 節で例外が生じ得る処理を記載し、`except` 節で対応する `try` 節で生じた例外を受け取って処理を行います。ここで利用者が意図的に例外を送出させるために `try` 節に `raise` 文で利用者が指定の例外、「**利用者定義例外 (user defined exception)**」を生じさせることができます。このときに `Exception` クラスを継承することで利用者が独自の例外を構築することもできます。また、`try` 文に複数の `except` 節を記載することで `try` 節に記載した処理から生じる複数の例外を補足して対処することもできます。それから `try` 文は例外処理の後処理 (clean up) を指定することができます。この処理は `try` 文の末端に `finally` 節として処理を記載することで行えます。なお、この `finally` 節は `except` 節と異なり一つだけの `finally` 節が記載可能で、記載した `finally` 節の処理は `try` 文の後処理として必ず実行されます。

この `try` 文、および `raise` 文の構文については、これらの文の EBNF を参照して下さい。ここでは例外に関連するクラスや例外に関する一般的な事項を述べることとします。

3.14.2 BaseException クラスについて

Python で例外に関係するクラスが `BaseException` クラスです。この `BaseException` クラスの組込の派生クラスとして次のクラスがあります：

- SystemExit: フィルス `sys.exit()` で送出される例外です。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ス `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` を持つ、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` の派生クラスになっています。ここでフィルス `sys.exit()` は後処理（`try` 文の `finally` 節）が実行されるようにするために、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、フィルス `os.fork()` を呼んだ後の子プロセス内でフィルス `os._exit()` が実行できます。このクラスは `BaseException` を直接継承することで着実に呼出し元に伝わり、インタプリタを終了させることができます。
- 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。割り込みが起きたかどうかはインタプリタの実行中に定期的に調べられ、組込フィルス `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。この例外は `Exception` クラスの例外を処理するコードに誤って捕捉されてインタプリタの終了が阻害されないように `BaseException` クラスを継承しています。
- GeneratorExit: generator 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」に該当しないために、一般の例外である `Exception` とは別のクラスとしています。
- Exception: 一般的な例外のクラスです。このクラスの詳細は次の小節で述べることにします。

3.14.3 Exception クラスについて

`Exception` クラスはシステム終了以外の全ての組込の例外のクラスです。`Exception` クラスは `BaseException` クラスの派生クラスであり、利用者定義の例外クラスを構築するときは `BaseException` クラスではなく、`Exception` クラスの派生クラスとして構築すべきです。

組込例外としては `Exception` クラスの派生クラスとして `StandardError` があり、この派生クラスに組込例外が含まれます。他の派生クラスとして `StopIteration` と `Warning`

があります。ここでは StandardError に包含される例外について解説します。

Exception クラスの派生クラス			
ArithmetiError	AssertionError	AttributeError	BufferError
EnvironmentError	EOFError	ImportError	LookupError
MemoryError	NameError	ReferenceError	RuntimeError
SyntaxError	SystemError	TypeError	ValueError

■ **ArithmetiError:** 算術演算処理で送出される例学のクラスで、OverflowError, ZeroDivisionError, FloatingPointError をその派生クラスとして持ちます。

- 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。ただし、Python の整数の処理で生じることは殆どなく、むしろ、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。
- ZeroDivisionError: 除算や剰余の計算で零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。
- FloatingPointError: 浮動小数点演算が失敗したときに送出されます。なお、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なります。

■ **BufferError:** バッファ (buffer) に関する処理が行えなかったときに送出されます。組込例外で BufferError クラスの派生クラスはありません。

■ **AssertionError:** プログラムのデバックやテストで用いられる assert 文が失敗したときに送出されます。組込例外で AssertionError クラスの派生クラスはありません。

■ **AttributeError:** 属性参照や代入が失敗したときに送出されます。ただし、オブジェクトが属性の参照や属性の代入を提供していないときは TypeError が送出されます。組込例外で AttributeError の派生クラスはありません。

■ **EnvironmentError:** システム外部の環境で生じる可能性がある例外のクラスで、組込例外の IOError と OSError をその派生クラスとして持ちます。この型の例外はタプルとして生成され、その長さが 2、あるいは 3 のタプルとして得られるとき、タプルの第 1 成分はインスタンスの属性 errno で得られ、この値はエラー番号と見なされます。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 は None になります。

- IOError: ファイルやメソッドによるファイルオブジェクト等への 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引数の型の誤りや他の偶発的な異常を除いて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 の関数 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においてのみ利用可能です。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 の関数 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: input() または raw_input() のいずれかの組込関数で、ただちにファイルの終端 (EOF) に到達したときに送出されます。なお、メソッド file.read() と file.readline() は同じ状況で例外ではなく空の文字列を返却します。なお、EOFError クラスの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、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。ImportError クラスの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である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。なお、LookupError クラスは IndexError と KeyError を組込の派生クラスとして持ちます：

▲ IndexError: 指定した列の添字が列の範囲を超えているときに送出されます。ただし、添字が整数でないときは例外 TypeError が送出されます。また、スライスの添

字については列の範囲内に収まるように自動調整されるために範囲外であっても 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包含されなかったときに送出されます。

■ **MemoryError:** プログラムの処理でメモリが不足し、オブジェクトをいくつか消去することで復旧が可能であるときに送出されます。例外の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。ただし、インタプリタが完璧に復旧できるとは限りません。実際、プログラムの暴走のときでも実行スタックの追跡結果が送出できるようにするために、この例外が送出されます。なお、この例外 MemoryError は組込の派生クラスを持ちません。

■ **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。なお、UnboundLocalError を組込の派生クラスとして持ります：

- ▲ **UnboundLocalError:** 値が束縛されていない函数や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。なお、組込の派生クラスとして `NotImplementedError` があります：

- ▲ **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に遭遇したときに送出されます。この例外は `import` 文、`exec` 文、組込函数 `evel()` 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ることがあります。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。なお、組込の派生

クラスとして IndentationError を持ちます:

▲ IndentationError: 字下の構文エラーに対応する基底クラスです。組込の派生クラスに TabError を持ちます:

♡ TabError: 字下に用いる空白文字として TAB と Space を混在し、それらを一貫した順序で並べていないときに送出されます。組込の派生クラスを持ちません。

■ SystemError: インタプリタが内部エラーを発見したものの、その状況はさほど深刻ではないと思われる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。なお、このクラスに組込の派生クラスはありません。

■ TypeError: 関数やメソッド等で不適切な型オブジェクトが引き渡されたときに送出される例外です。関連値はこの型の不整合について詳細を述べた文字列になります。なお、この方には組込の派生クラスはありません。

■ ValueError: 組込演算や関数で、型は整合していても不適切な値を受け取ったとき、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。この例外は UnicodeError クラスをその派生クラスとして持ります:

- UnicodeError: エンコードまたはデコードで 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: エンコードまたはデコード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.

このクラスには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.

3.14.4 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、通常は異常とみなされないために StandardError ではなく Exception から派生します。

3.14.5 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.

y

第 4 章

数値行列処理即席講座

4.1 はじめに

ここでは MATLAB 系言語の Scilab と Python の多次元配列モジュール Numpy を比較しながら Numpy の基本的な行列操作を解説します。ここで MATLAB は The MathWorks Inc. の商用の数値行列処理向けのシステムです。そして、MATLAB と同様に数値行列処理を行なうアプリケーションで、行列処理や言語仕様で MATLAB の影響を強く受けている言語をまとめて MATLAB 系言語とこの本では呼んでいます。MATLAB の開発目的は FORTRAN で記述された数値行列計算ライブラリ LINPACK や EISPACK^{*1} を学生が容易に扱えることで、開発当初はフリーソフトとして公開されていました。そのために数値行列操作を行うソフトウェアがこの MATLAB と同様の操作になる原因になっています。現在の MATLAB は Toolbox と呼ばれる膨大なライブラリ群を従え、数値行列処理を目的としたソフトウェアの標準的存在です^{*2}。この MATLAB の影響を特に強く受けた「OSS(Open Source Software)」のアプリケーションとして「GNU Octave」と INRIA の「Scilab」、それに加えて C 風の「Yorick」を挙げておきます。GNU Octave は GNU の MATLAB クローンと呼ばれる程の MATLAB との高い互換性を持ちます。とはいえ、全くのクローンでもなく独自の改善点もあります。Scilab は MATLAB に類似したシステムで、MATLAB の GUI 環境の「Simulink」^{*3}に似た「SCICOS」を擁する下手な壳物を凌駕するシステムです。これらに対して Yorick は対話処理可能な C と言える程、C に類似したアプリケーションで、MATLAB 系の各種言語が機能を満載した結果、重量級のアプリケーションになっている現状に対して、こちらは軽量な言語で、MATLAB 風の行列データではなく、Numpy と同様の多次元配列処理を導入し、そのユニークな配

^{*1} LINPACK や EISPACK の後継が後述の LAPACK です。

^{*2} 構成は <http://www.mathworks.co.jp/products/pfo> を参照。

^{*3} 本来は制御系のブロック線図の解析を目的にしていましたが、現在では 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。

列処理が大きな特徴です [30]. この Yorick の配列処理は GNU Octave や Scilab よりも Numpy に類似しています。

SageMath の長所は数式処理だけではなく、実際は数値行列処理についても従来の数式処理システムよりも各段に優れており、MATLAB 系言語と遜色がありません。さらに MATLAB 系の言語と比べ、より高度な数式処理が行える点で優れています。この数値行列処理が MATLAB 系言語と遜色がないという長所を支えているものがモジュール Numpy です。この Numpy は Python に多次元配列とそれらの基本的な数値計算のための演算やメソッドを付与しています。そして、Numpy の数値行列処理は MATLAB の行列処理操作の影響を強く受け、さまざまな類似点を見せる一方で、多次元配列ならではの独特的な配列操作のための流儀があります。これらの理由から、MATLAB 系の言語による基本的な行列処理を知ることで、逆に、Numpy での配列処理の特徴を浮び上らせることを目的にしています。

4.2 Numpyについて

Python 本体にはリスト型や辞書型等のデータを蓄えるためのオブジェクトがありますが、これらは MATLAB 系言語と比較して数値行列処理に優れたものではありません。たとえば、MATLAB 系の言語ではベクトルは 1 次元の数値データの列として角括弧 “[]” で括られたリスト型で表現され、さらに行列も行ベクトルから構成されるものとして容易に構成可能で、それらの演算も数学的な表記に近い式で記述できます。ところで Python 本体のリスト型のオブジェクトの演算にはリストの結合といった処理で、自分で行列演算を行う函数やメソッドを記述する必要があります。この状況を大きく改善するモジュールが Numpy で、この Numpy によって Python に多次元配列 (ndarray 型) が導入され、同時に数値行列ライブラリを利用した演算やパターンマッチング処理、スライス処理による成分の取出といった MATLAB 系の言語と類似の処理を可能にします。さらには Numpy を基盤に MATLAB と同様の 2 次元、3 次元グラフ表示を可能にする matplotlib、科学技術計算ライブラリである SciPy があり、さらにそれらの延長線上に SageMath があります。そのために Numpy の利用方法を理解することは SageMath で数値配列処理を行う上で大きな力になります。

MATLAB の開発目的は、そもそも、学生に FORTRAN で記述された数値行列計算ライブラリ LINPACK を使わせることでした。つまり、学生に LINPACK に用意されたサブルーチンを使った FORTRAN プログラムを作成させるのではなく、LINPACK にアプリケーションを被せて、処理すべき式を本来の式に近い式で対話的処理ができるようにしたもののが本来の MATLAB です。現在の MATLAB 系言語でも行列データは対話

的に定義し、そのデータを BLAS 等の数値計算ライブラリの函数に引き渡して処理し、その計算結果を言語側に引き渡して残りの処理を行うという流れになります。したがって、MATLAB 系言語では、大きな行列であっても実際の計算は数値計算ライブラリの函数を用いるために高速な処理が行われる一方で、反復処理や条件分岐処理は言語側の処理になるために途端に時間がかかるという大きな弱点（処理言語のオーバーヘッドと呼ばれます）があります。また、行列計算の能力は利用する行列計算ライブラリの性能に強く依存します。そのために MATLAB では計算機環境に最適化された行列計算ライブラリ、Intel なら MKL、AMD なら ACML といった商用のライブラリ、あるいは OSS の行列計算ライブラリとして OpenBLAS や ATLAS(Automaticaly Software) が利用できます。この状況は Numpy を導入した Python でも同様で、外部の数値行列計算ライブラリを活用することで Python でも数値行列の高速処理が可能になります。そのために数値行列の処理で MATLAB 系言語ですべことすべきでないことは Numpy でも同様です。さて、Numpy がどのような数値行列計算ライブラリを用いているかは函数 `show_config()` で確認ができます。ここで macOS 上の SageMath の確認の様子を示しておきましょう：

```
sage: import numpy
sage: numpy.show_config()
lapack_opt_info:
    libraries = ['openblas', 'openblas']
    library_dirs = ['/Applications/SageMath-7.6.app/Contents/Resources/sage/local/lib']
    define_macros = [(HAVE_CBLAS, None)]
    language = c
blas_opt_info:
    libraries = ['openblas', 'openblas']
    library_dirs = ['/Applications/SageMath-7.6.app/Contents/Resources/sage/local/lib']
    define_macros = [(HAVE_CBLAS, None)]
    language = c
openblas_info:
    libraries = ['openblas', 'openblas']
    library_dirs = ['/Applications/SageMath-7.6.app/Contents/Resources/sage/local/lib']
    define_macros = [(HAVE_CBLAS, None)]
    language = c
openblas_lapack_info:
    libraries = ['openblas', 'openblas']
    library_dirs = ['/Applications/SageMath-7.6.app/Contents/Resources/sage/local/lib']
    define_macros = [(HAVE_CBLAS, None)]
    language = c
blas_mkl_info:
    NOT AVAILABLE
```

ここで実行していることはモジュール `numpy` を `import` 文で読み込み、`'numpy.show_config()'`

で Numpy で定義された関数 `show_config()` の呼出を行っています。この出力結果から OpenBLAS が数値行列ライブラリとして用いられていることが判ります。これは SageMathCloud でも同様です。SageMath は OpenBLAS が既定値になっていますが、構築時に ATLAS を指定することもできます。ところで、ここで考察すべき対象である数値行列は MATLAB 系言語や Numpy を容器として、実質的な処理はその背後にある数値行列処理ライブラリに強く依存しています。そして、数値行列をその容器にどのように盛り付けるかで処理の速度が大きく異なります。したがって、盛り付けという処理を考える前に容器自体がどのようなものであるかを知っておく必要があります。そこで、MATLAB 系言語や Numpy で数値行列をどのように処理するかを述べる前に、浮動小数点数と数値行列処理ライブラリ BLAS について簡単に解説しておきましょう。

4.3 実数の表現

4.3.1 浮動小数点数の構成

計算機上で実数を表現するために「**浮動小数点数**」と呼ばれる数の表現が使われます。この浮動小数点数は 0 と 1 だけの文字の羅列、たとえば “01111110” のように精度に応じて固定された長さを持つ 2 進数です。この羅列を構成する 0 と 1 の総数を「**bit 長**」と呼び、この bit 長で浮動小数点数が決定されます。たとえば bit 長が 32、つまり、32 文字であれば「**単精度**」、bit 長が 64 であれば「**倍精度**」の浮動小数点数です。浮動小数点数の構成は「**符号部**」、「**仮数（小数）部**」、「**指数部**」の三部構成になっています。ここで符号部を表現する整数を s 、仮数部で構成される有理数を f 、指数部から構成される整数を ε 、さらに基底を正整数 β のときに $(-1)^s \times f \times \beta^{\varepsilon}$ で対応する実数が復元できます。まず、符号部 s が数の正負を示し、仮数部 f は正整数 d_i 、 $0 \leq d_i < b - 1$ 、 $(i = 1, \dots, n)$ を使って $d_1/\beta + \dots + d_n/\beta^n$ という基底のローラン多項式で表現される有理数です。一方で、指数部 ε は $\delta_1 + \delta_2 \times \beta + \dots + \delta_m \times \beta^{m-1} - b$ と基底の多項式で表現される整数です。なお、指数部に現われる定数 b は「**下駄履き値**」と呼ばれる整数です。浮動小数点数は 0 を中心に左右対称に分布することが構成方法からも判りますが、浮動小数点数の濃度（個数）は $2 \times \beta^{m+n} + 1$ で、 m, n の一方を ∞ としても高々 N_0 ^{*4} で、実数の濃度は N_0 よりも大きな N であるために浮動小数点数で実数を網羅できません。そのために浮動小数点数は近似値としての性格を持ちます。

*4 整数の濃度（個数）です。

4.3.2 IEEE 754 による浮動小数点数

浮動小数点数には規格があり、ほとんどの計算機で「**IEEE 754**」が用いられています。この IEEE 754-1985 は 2 進数浮動小数点数のための規格で、「**単精度**」、「**倍精度**」、「**拡張単精度**」と「**拡張倍精度**」の 4 種類の浮動小数点数、および浮動小数点数の「**四則演算**」：(“+”, “-”, “*”, “/”), 「**平方根**」：(“ $\sqrt{}$ ”) と「**剰余**」：(“%”) や「**比較操作**」：(>, ≥, =), 「**整数と浮動小数点数間の変換**」や「**異なる浮動小数点書式間の変換**」、浮動小数点数の基本書式と 10 進数変換、無限大等の「**浮動小数点例外**」と「**非数 (NaN)**」の扱いを定めています。この規格に従っていれば異なった計算機環境でも、これらの演算で同じ結果が期待できますが、三角函数や指数函数といった初等函数が IEEE 754 で規格化されていないために計算機のハードウェア、あるいは数値計算ライブラリの初等函数の実装の違いから計算結果に違いが生じる可能性があります。

ここで IEEE 754 で規定されている 2 を基数とする浮動小数点数の書式には「**32-bit 長の単精度**」、「**64-bit 長の倍精度**」、「**拡張単精度**」と「**拡張倍精度**」の 4 種類があり、計算機に実装すべき浮動小数点数は単精度のみ、拡張単精度と拡張倍精度については充すべき最低限の bit 長と精度が定められています：

IEEE 754 浮動小数点数の言語毎の型

精度	Python(Numpy)	C, C++	FORTRAN
単精度	なし (float16)	float	REAL, REAL*4
倍精度	float(float32)	double	DOUBLE PRECISION, REAL*8
拡張倍精度	なし (float64)	long double	REAL*16

浮動小数点数はその精度に応じた固定長の bit 列で表現されます。具体的には、この bit 列は $a_{(m+n)} \boxed{a_{(m+n-1)} \dots a_{(n)} a_{(n-1)} \dots a_{(0)}}$ で表現されます。ここで $a_{(i)}$ の添字 i が右側から割当られた番地、 m, n が指数部と仮数部の bit 長です。このときに浮動小数点数の番地に対して bit の列を三部に分けることができます： $\boxed{\text{符号部 } s_{[m+n:m+n]}} \boxed{\text{指数部 } e_{[m+n-1:n]}} \boxed{\text{仮数部 } f_{[n-1:0]}}$ 。ここで ‘ $[b : a]$ ’ は番地 a から b までの領域を利用することを意味し、「**符号部**」は 1-bit 長、「**指数部**」が m -bit 長、「**仮数部**」が n -bit 長の 2 進数で、浮動小数点数はこれらの 2 進数を並べた $m + n + 1$ -bit 長の 2 進数 a で表現されます。また、仮数部の bit 長に 1 を加えた $n + 1$ を「**浮動小数点数の精度**」と呼びます。ここで 1 を加える理由は仮数部の構造が関係します。以降、符号部、仮数部と指数部の順番で詳細を解説しましょう。

■**符号部**: 符号部 s は最も大きな番地の bit の “ $a_{(m+n)}$ ” に対応し, 表現する数が正であれば 0, 負であれば 1 です.

■**指数部**: 指数部は 2 進数 a の m 桁の部分 “ $a_{(m+n-1)} \dots a_{(n)}$ ” から構成されます. $\delta_{(i)} = a_{(i+n)}$ ($0 \leq i \leq m-1$) のときに指数部で表現される整数 e は $\delta_{(0)} \times 2^0 + \delta_{(1)} \times 2^1 + \dots + \delta_{(m-1)} \times 2^{m-1}$ で与えられ, $\delta_{(0)} = 0$ と $\delta_{(i)} = 1$ ($1 \leq i \leq m-1$) を充す指数部 e を「**最大許容指数**」と呼び, e_{\max} と表記し, 1 から e_{\max} の指数部 e の範囲を「**正規化浮動小数点数**」と呼び, 通常の実数の表現で用います. また, 全てが 1, すなわち $\delta_{(i)} = 1$ ($0 \leq i \leq m-1$) を充たす指数部 $e_{\max} + 1$ が「**無限大 ∞** 」や「**非数 NaN**」の表現で, 全てが 0, すなわち, $\delta_{(i)} = 0$ ($0 \leq i \leq m-1$) になる指数部 e は浮動小数点数の「**零**」や「**非正規化浮動小数点数**」と呼ばれる 0 周辺の実数を表現する浮動小数点数で用いられます.

なお, IEEE 754 で指数部が表現する整数 e は本来の指数 ε にある定数 b を加えた数を表現し, 指数 ε は $e - b$ で与えられます. この b を履かせた指数部を「**下駄履き表示**」と呼び, 定数 b を「**下駄履き値 (bias)**」と呼びます. この下駄履き値 b は浮動小数点数の精度で異なる値です.

■**仮数部**: n 桁の 2 進数 “ $a_n \dots a_{(1)}$ ” に 1 桁追加して得られる 2 進数 $d = d_nd_{(n-1)} \dots d_{(1)}d_{(0)}$ です. ここで $d_{(i)} = a_{(i-1)}$ ($1 \leq i \leq n$) とし, 残りの $d_{(0)}$ は指数部が表現する整数 e が 0 でなければ $d_{(0)} = 1$, 整数 e が 0 ならば $d_{(0)} = 0$ と定めます. また全てが 1, すなわち $d_{(i)} = 1$ ($0 \leq i \leq n-1$) を充す仮数部を f_{\max} と表記します. この仮数部が表現する有理数 f は $d_{(0)} + d_{(1)}/2 + \dots + d_{(n)}/2^n$ で与えられ, $d_{(0)} = 1$ であれば “1.d₍₁₎d₍₂₎...”, また, $d_{(0)} = 0$ であれば “0.d₍₁₎d₍₂₎...” と 2 進数で表示される数になります. ここで $d_{(0)}$ を「**隠れ bit**」, あるいは「**暗黙の 1**」と呼びます. まず, 仮数部は $n+1$ -bit 長ですが, その表に出ている箇所は隠れ bit を除いた n -bit 長の部分になるためです. そして, 隠れ bit がある浮動小数点数を「**規格化浮動小数点数**」, 略して「**規格化数**」, と呼び, 隠れ bit が 0 の浮動小数点数を「**非規格化浮動小数点数**」, 略して「**非規格化数**」と呼びます. 特に非規格化数は, 指数部を表現する整数 e が 0 のときに非規格化数に切換えられるために 0 の周囲を埋める数としての性格を持ちます. また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. ここで隠れ bit を加えた桁数 $n+1$ が規格化数の有効桁で, これを「**有効桁精度**」と呼びます. ところで非規格化数は先頭が 0 のために実質的に規格化数の有効精度よりも 1 つ低下した桁数 n になります. 同様に指数部の下駄履き値も規格化数と非規格化数で異なります. まず b を正規化数の下駄履き値とすると, 非正規化数で用いられる 下駄履き値 b_v は $b-1$ で与えられます. なぜなら正規化数の絶

対値での最小値と非正規化数の最大値を考えると正規化数の最小値が 2^{1-b} , 非正規数の最大値は $(\sum_{i=1}^n 1/2^i) \times 2^{-b_\nu}$ で, ここで式の総和の部分は $\sum_{i=1}^n 1/2^i = 1 - 1/2^n$ より $(1 - 1/2^n) \times 2^{-b_\nu}$ になります. そして, 最小正規化数と最大非正規化数は隣接したものであるべきです. この要請から $b_\nu = b - 1$ でなければならぬことが判ります.

■0 の表現: IEEE 754 では指数部と仮数部を表現する整数が 0 のとき, すなわち $(-1)^s \times 0$ で浮動小数点数の 0 を定義します. このときに符号 s が 0 と 1 の場合があり, $s = 0$ であれば「**正の零**」と呼び $+0$ と表記し, $s = 1$ であれば「**負の零**」と呼び -0 と表記します. ただし, $+0 = -0$ であることが IEEE 754 で定められています. つまり, 整数 0 の浮動小数点数の表記に正と負の二つの零があっても双方を同じものとみなすために単に 0 と表記できるということです. なお, 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.

■表現可能な領域: 浮動小数点数で表現される領域はどうなるでしょうか? まず整数の表現と異なり, 0 を挟んで左右対称に浮動小数点数が存在することが容易に判ります. それから正規化数の最大値は $x_{\max} = f_{\max} \times 2^{e_{\max}-b}$ で与えられ, 0 と異なる浮動小数点数の絶対値における最小値は, 規格化数であれば $x_{\min} = 2^{-b}$, 非規格化数であれば $x_{\min} = 2^{-n-b_\nu} = 2^{1-n-b}$ で与えられます. ここで b と b_ν をそれぞれ正規化数と非正規化数の下駄履き値とします. 規格化数の最小値 $x_{\min} = 2^{-b}$ を越えて 0 に近づくと非正規化数に切替わって 0 の周囲の浮動小数点数が密になります. ここでは单精度と倍精度について表現可能な領域を纏めておきますが, 小数点部は小数点下 5 桁で四捨五入しています:

单精度と倍精度の表現可能な領域

	单精度	倍精度
指数部の bit 長 (M)	8	11
仮数部の bit 長 (N)	23	52
正規化の下駄履き値 (b)	127	1023
非正規化の下駄履き値 (b_ν)	126	1022
最大正規化数	3.4028×10^{38}	1.7977×10^{308}
最小正規化数	1.1755×10^{-38}	2.2251×10^{-308}
最小非正規化数	1.4013×10^{-45}	4.9407×10^{-324}

ちなみに Python 本来の浮動小数点数は倍精度のみ, モジュール Numpy を導入することで单精度, 拡張倍精度が利用可能になります^{*5}. なお, MATLAB 系の言語では单精度と倍精度の浮動小数点数をサポートしていますが, 拡張倍精度はサポートしないものがほとん

*5 配列生成時にオプションの `dtype` の指定で行います.

どです。

このように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 n -bit 長, 指数部を m -bit 長とする浮動小数点数で表現可能な数の集合を $\mathcal{F}_{m,n}$, 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。

■丸めと切捨: 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, その区間内で連続的な実数とは違って浮動小数点数は離散的です。そのため実数の表現は基本的に近似です。ここで `ulp(Unit in the Last Place)*6` という函数を導入しましょう。この函数 `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 `ulp(x)` を与える函数です。`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} になることから `ulp` は 1 になります。このことを Python 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る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といった工夫をしなくと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 や LAPACK を使った高速な演算が可能です。

さて, ここで実数 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:

*6 「ウルプ」と呼びます

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の中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の中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 に最近の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 \tilde{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.

■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) から丸めの演算子は「**大小関係の順序を保つ**」ことが判ります. ここで比較の演算子 “ \leq ” を “ $<$ ” で置換えることはできません. 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, 数値計算で意味を持つために演算結果も近似値になってい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, 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.

■計算機イプシロン: 微小な数を表現するために用いられ, 浮動小数点数 1.0 の次の浮動小数点数との差で, MATLAB では変数 `eps`, Scilab なら変数 `%eps`, Numpy は ‘`finfo(float).eps`’ の値から判ります. この計算機イプシロンの値は IEEE 754 の倍精度で

$2^{-52} = 2.220446049250313 \times 10^{(-16)}$ です。このことを Numpy を np として読み込んだ Python 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
```

上の二つの確認は丸め誤差を考慮すると大雑把ですが、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、対数函数が負の無限遠に発散することを避けるために用います。

4.4 数値行列ライブラリについて

4.4.1 BLAS の概要

数値行列処理システム MATLAB のそもそもの目的は学生に LINPACK 等の数値行列ライブラリを使わせることでしたが、この LINPACK の後継が LAPACK です。さて、この「**LAPACK**(Linear Algebra PACKage)」は線形代数の求解などの線形代数で用いられる数値行列処理を効率的に、できるだけ高速に処理することが目的のライブラリです。LAPACK は「**BLAS**(Basic Linear Algebra Subprograms)」と呼ばれるベクトルと行列の基本的な計算を効率良く処理することを目的としたルーチン群上で構築されており、LAPACK の性能は BLAS の性能が直接関係します。この BLAS の公式標準実装は **netlib**^{*7}で公開されていますが、各種計算機環境向けに最適化を行った BLAS が幾つか存在します。まず、OSS のものでは「**ATLAS**(Automatically Tuned ALgebra Software)」と「**OpenBLAS**^{*8}」が代表的で、商用のものでは Intel の「**MKL**(Math Kernel Library)」と AMD の「**ACML**(AMD Core Math Library)」が代表的です。ここで、OpenBLAS は高速処理で知っていた GotoBLAS2[11]^{*9} の後継で、SageMath は OpenBLAS を BLAS ライブラリとして利用するように標準で設定されています。なお、行列の大きさや形式、最適化の水準に依存しますが、これらの BLAS の大雑把な処理速度の目安は MKL が全般的に優れ、その次が OpenBLAS、それから ATLAS の順番になります。また、BLAS は数値行列処理を高速に行うことの目的にしていますが、計算量を必要とする大きな行列で効果が出易い一方で、小さな行列では逆に遅くなる傾向があります。

*7 <http://www.netlib.org/blas/>

*8 ATLAS と OpenBLAS は BSDL で配布されています。

*9 後藤和茂氏が開発・管理されていた BLAS ですが、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。

そのために BLAS をブラックボックスとみなして使うべきではなく、どのような原理で利用するルーチンが動作しているか理解しておく必要があります。この章では LAPACK の基礎になる BLAS のルーチンについて軽く触れましょう。

4.4.2 BLAS について

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るようになっています。さて、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

このように三つの水準がある理由は、BLAS 開発の歴史的な経緯があります。1979 年にリリースされた最初の BLAS は 1970 年代に現れたベクトル型計算機で効率的に動作するように構築され、それがベクトル演算を中心とする第一水準のルーチンでした。ところでベクトル演算だけでは不十分であったため行列とベクトルの演算を行う第二水準（1987）と行列同士の演算を行う第三水準（1989）を追加と、10 年かけて拡張されています [36] [38]。また、上記の水準に加えて複素数の 1-ノルムと割算を計算するルーチンを第 0 水準のルーチンが加わっています。

BLAS による処理の効率化では計算機内部の動作を考慮する必要があります。まず、CPU が主メモリにアクセスして処理を行うときの実行性能はデータの転送速度で決定されますが、近年の CPU はコアを増やして並列処理による効率の向上を目指しているために複数のコアがデータ転送の帯域を奪い合うという問題が顕著になります。そこで、キャッシュにデータがある間に並列した演算（「**データの再利用**」）が重要になります。そのためには BLAS のアルゴリズムの改良に留まらず、各種 CPU 専用の最適化が商用、OSS 問わず行われています。

■**BLAS の第一水準**：ベクトル演算を行うルーチンであるため行列を扱う他の水準と比較して演算量が少なく、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。実際、ベクトルの大きさを N とするとデータ量は $O(N)$ 、演算量も $O(N)$ と N に正比例して同程度になります。そして、キャッシュに載っ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では性能向上があまり期待できません。

■BLAS の第二水準: 行列とベクトルの演算が中心のルーチンで、計算量とデータ量共に $O(N^2)$ 、つまりに N^2 の水準に増大するために第一水準よりも効果が期待できます。また、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るために第一水準と異なります。

■BLAS の第三水準: 行列同士の処理が中心のルーチンで、扱うデータ量は $O(N^2)$ ですが、計算量は $O(N^3)$ と最も多くなります。また性能も CPU の理論性能値に依存しますが $O(N)$ 回のデータの再利用性があり、性能向上が望めます。

4.4.3 ベクトルと行列の表記について

BLAS には引数の表記に決まりがあります。まず、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ます。そして、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表記し、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は $\alpha \cdot A$ 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 ${}^H A$ で表記することもあります。

4.4.4 配列への格納方法

BLAS や LAPACK でベクトルは 1 次元配列として扱われています。このベクトルに対して行列は行列の添字をそのまま 2 次元配列で置き換えたり、列や行で繋いで 1 次元配列として格納する方法だけではなく、行列の特性を生かしてメモリを節約すると同時に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ルーチンを用いることで処理効率の向上が狙えます。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に対して、その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」、帶行列向けの「帯格納形式」の三種類があります。

■一般形式: この形式の行列を扱うルーチンは命名規則の“MM”の箇所が“GE”です。この一般形式は行列 A の i 行 j 列成分の A_{ij} をそのまま配列 a の成分 $a[i, j]$ に格納します。また、行列の列を並べて 1 次元配列として扱うこともあります。具体的には $m \times n$ -行列 A の i, j -成分を 1 次元配列 a の $i + m \cdot (j - 1)$ 成分に格納する方法です。この場合は行列を列単位で 1 次元配列に格納する方法です。このように列方向で並べる理由は本来の BLAS が作成された FORTRAN では 2 次元配列の列方向へのアクセスがメモリ上で連続するためにその分、処理を高速に行えるためです。逆に、FORTRAN で配列の行方向へ

のアクセスはメモリ上を飛び飛びにアクセスするために速度向上の面で致命的な弱点になります。この理由から FORTRAN では列が優先されますが、C では逆に 2 次元配列の列ではなく行が優先されます。そのために C から FORTRAN の BLAS ルーチンを呼び出すときは、C の配列として FORTRAN の配列を転置した状態で格納しますが、C 向けの BLAS のインターフェイスの CBLAS を使えば行優先、列優先の指定や引数を番地渡しではなく値渡しといった指定ができます。

■圧縮格納形式: この書式は後述のルーチンの命名規則にて “MM” の箇所の二文字目が “P” のものです。この「**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を 1 次元配列に格納するために用いられ、通常形式より少ない記憶容量で行列を配列に収納できます。この圧縮格納方式で上三角行列であれば $i \leq j$ のときに A_{ij} 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} 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 がどのように格納されるかを確認ましょう：

		行列の配列への格納方法
$U :$	$\begin{pmatrix} A_{11} & A_{12} & A_{13} & A_{14} \\ & A_{22} & A_{23} & A_{24} \\ & & A_{33} & A_{34} \\ 0 & & & A_{44} \end{pmatrix}$	$\Rightarrow A_{11} \underbrace{A_{12} A_{22}} \underbrace{A_{13} A_{23} A_{33}} \underbrace{A_{14} A_{24} A_{34} A_{44}}$
$L :$	$\begin{pmatrix} A_{11} & & & 0 \\ A_{21} & A_{22} & & \\ A_{31} & A_{32} & A_{33} & \\ A_{41} & A_{42} & A_{43} & A_{44} \end{pmatrix}$	$\Rightarrow \underbrace{A_{11} A_{21} A_{31} A_{41}} \underbrace{A_{22} A_{32} A_{42}} \underbrace{A_{33} A_{43} A_{44}}$

これらの例から判るように一般形式で 1 次元配列に変換するときに上三角、あるいは下三角の列のみを並べて 1 次元配列に取り込んで行く様子が分ります。そして、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さえ登録できれば復元可能になるためです。したがって、上三角、あるいは下三角行列の圧縮格納形式で配列に格納し、その行列の種類に応じたルーチンを選択すれば良いことになります。具体的には命名規則の “MM” が対称行列であれば “SP”，エルミート行列であれば “HP” 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。この帯行列 $A \in M(m, n)$ は k_l 行の劣対角成分 (対角成分の下側) と k_u 列の優対角成分 (対角成分の上側) を持つものとします：

$$k_l + 1 \left(\begin{array}{cccccc} & & \overbrace{A_{11} \cdots A_{1(k_u+1)}}^{k_u+1} & & O \\ & \vdots & \ddots & & \ddots \\ & A_{(k_l+1)1} & & \ddots & & \ddots \\ & & \ddots & & \ddots & \\ O & & & \ddots & & \ddots \end{array} \right)$$

この行列 M の成分 M_{ij} を $(k_u + k_l + 1) \times n$ の配列 a に格納します。この収納の方法を netlib で公開されているルーチンのコメント中の記述を見てみましょう：

```

DO 20, J = 1, N
  K = KU + 1 - J
  DO 10, I = MAX( 1, J - KU ), MIN( M, J + KL )
    A( K + I, J ) = matrix( I, J )
10   CONTINUE
20 CONTINUE

```

ここで ‘matrix(I,J)’ が行列 M の i,j 成分の M_{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 M_{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 M_{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 M_{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。LAPACK マニュアルにもある $m = n = 6, k_u = 1, k_l = 2$ 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 ‘*’ の箇所は成分の配置が行われない箇所です。この例からも判るように帯行列の

帶を列単位で対角成分を積み重ねるように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。この帶格納形式は一般形式と同様に列で並べた配列に最終的に変換できます。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、これらのときの格納方法は通常形式と一致します。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、あるいは下三角成分のみを帶格納形式で格納しておくことができます。

4.4.5 ルーチンの命名規則

BLAS や LAPACK のルーチンの名前は命名規則によって「**PMMAAA**」の書式に限定されます。名前の文字数は FORTRAN の函数名の制約に由来し、次の意味があります：

$\underbrace{P}_{\text{精度}}$	$\underbrace{MM}_{\text{行列の型}}$	$\underbrace{AAA}_{\text{処理内容}}$
------------------------------	---------------------------------	----------------------------------

では、これら **P**, **MM** と **AAA** の内容を解説をしましょう。

■**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

	单精度 (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を指示します。行列の型が対角行列なら「**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、他の一般の行列ならば「**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**」、帶格納形式なら「**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	帶行列
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	エルミート帶行列	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	正値対称/エルミート帶行列	PO	正値対称/エルミート行列
PP	正値対称/エルミート行列(圧縮格納形式)	PT	正値対称三重対角行列/エルミート三重対角行列	SB	対称帶行列
SP	対称行列(圧縮格納形式)	SY	対称行列	TB	三重対角行列/帶行列
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。この文字数は2文字から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」が逆行列の計算という意味があり、これらを組合せた名前になります

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

4.4.6 ルーチンの引数について

ここではルーチンの引数の決まりごとを述べます。まず、行列 A を格納する配列は a と小文字で、行列の行数を m 、列数を n と表記します。なお、正方行列は ‘ $m = n$ ’ であるため行数と列数を区別せずに用います。また、ベクトル v, u を格納する配列を x, y と表

記します。ここでベクトル v, u を格納する配列 x, y に対してそれぞれ 引数「**INCX**」と引数「**INCY**」があり、これらの引数は各配列の添字の増分を指示する引数で、ここでは ' d_x ', ' d_y ' と表記します。この INCX と INCY で指示される添字の増分は 0 より大でなければなりません。なお、GotoBLAS2[11] の解説で INCX, INCY がともに 1 ときに最適化されていても、1 の他で最適化がれておらず、メモリを飛び飛びにアクセスするために処理速度が致命的に低下するとあります。そして、行列 A を格納する配列 a には「**Leading Dimension**」と呼ばれる引数 **LDA** があり、ここでは ' L ' と表記します。この Leading Dimension の由来は、FORTRAN で列優先のために行列 A を格納した配列 a の列を繋いだ 1 次元配列に対して行列 A の (i, j) 成分へのアクセスが ' $a[i + j*LDA]$ ' で行われます。のことから多くの BLAS ルーチンで $\max(1, m)$ と等しいという制約があり、「**行列の行数を指定すれば良い**」と Leading Dimension の解説で書かれていることもあります。LDA は行列 A を「**帶行列格納**」するときに行列の行数と違う値を設定しなければなりません。具体的には対角成分数 k_u 、劣対角成分数 k_l の $n \times n$ 行列のときの格納先の配列の大きさが $(k_u + k_l + 1) \times n$ になるために引数 LDA に $k_u + k_l + 1$ を指定します。

BLAS のルーチンの引数には行列変換のフラグ、行列が上三角か下三角のどちらかを指示するフラグ、そして、行列が対角成分が全て 1 になる三角行列、すなわち単位三角行列であるかを指示するフラグの三種類のフラグがあります。最初の行列変換に関するフラグは行列の転置や転置共役を指示します。ルーチンの引数として **TRANS** と記述されていますが、ここは f と表記します。次に f で定まる行列の作用素 op_f を示します：

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	tA	転置
C, c	$t\bar{A}$	転置 + 共役

フラグ f の値は二重引用符 ("") で括った文字列です。次に、三角行列が上三角行列か下三角行列を指示するフラグは、ルーチンの引数として **UPL0** で与えられます。ここでは 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

4.4.7 BLAS のルーチン

BLAS のルーチンは計算する行列やベクトル等を含む式の形式から三つの水準に分類されます。第一水準が最も基本的な処理で、行列の複製や回転、ベクトルの和やノルムの計算です。第二水準は行列とベクトルの演算です。そして、第三水準が行列同士の演算です。ここで第一水準のルーチンで直接、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の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ておきます。そして、型以外の部分^{*10}を「ルーチン名」と表示します。

4.4.8 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(x) _1 + \Im(x) _1$
s d c z	adiv	$x/y \quad x, y \in \mathbb{C}$

■cabs1: 構文は $\text{cabs1}(x)$ で複素数 x の 1-ノルムを計算します:

$$\text{cabs1}(x) = |\Re(x)|_1 + |\Im(x)|_1$$

ここで $\Re(x)$ と $\Im(x)$ は複素数 x の実部と虚部を返す函数です。

■ladv: 複素数 $x = (a + i b)$ と $y = (c + i d)$ の割算 x/y の実部 p と虚部 q を返却します。構文は実数引数のときは $\text{dladv}(a, b, c, d, p, q)$ と 6 引数、複素数引数のときは $\text{zladiv}(x, y)$ と 2 引数で次の計算が行なわれます:

^{*10} 要するに語幹に相当する箇所。

$$\begin{cases} \frac{ac - bd}{c^2 + d^2} \longmapsto p \\ \frac{ad + bc}{c^2 + d^2} \longmapsto q \end{cases}$$

実数のときに実部と虚部の p, q が、複素数のときは複素数 $p + i q$ が返却されます。

4.4.9 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。ここでの処理でデータの再利用性はなく、計算量も少なく、性能は CPU の理論性能値、あるいは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

■axpy：構文は $\text{axpy}(n, \alpha, x, d_x, y, d_y)$ で次の結果を 1 次元配列 y に代入します：

$$\alpha \cdot x_{1+d_x \cdot (i-1)} + y_{1+d_y \cdot (i-1)} \longmapsto y_{1+d_y \cdot (i-1)}$$

ここで $d_x, d_y \in \mathbb{N}$ ，スカラー α と 1 次元配列 x, y は型で指定された数です。

■copy：構文は $\text{copy}(n, x, d_x, y, d_y)$ で次の結果を 1 次元配列 y に代入します：

$$x_{1+d_x \cdot (i-1)} \longmapsto y_{1+d_y \cdot (i-1)}$$

ここで x, y は 1 次元配列, $n, d_x, d_y \in \mathbb{N}$ です.

■swap: 構文は $\text{swap}(n, x, d_x, y, d_y)$ で, 指定した 1 次元配列 x, y に対して次の入換処理を行います:

$$x_{1+d_x \cdot (i-1)} \leftrightarrow y_{1+d_y \cdot (i-1)}$$

■dot: 構文は $\text{dot}(n, x, d_x, y, d_y)$ で, 1 次元配列 x, y の内積を計算します. 特に配列 x が複素数値であればその複素共役を計算して dot 積(配列の成分単位の積の総和を行う演算)を計算します:

$$\sum_{i=1}^n \overline{x_{1+d_x \cdot (i-1)}} \cdot y_{1+d_y \cdot (i-1)}$$

このルーチンは数値を返却するために配列の書換が生じません.

■dotu: 構文は $\text{dotu}(n, x, d_x, y, d_y)$ で, 複素数 1 次元配列 x, y の dot 積の計算を行います. dotc との違いは複素共役を計算しない点です:

$$\sum_{i=1}^n x_{1+d_x \cdot (i-1)} \cdot y_{1+d_y \cdot (i-1)}$$

このルーチンも数値を返却するために配列の書換が生じません.

■sdot: 構文は $\text{dotu}(n, x, d_x, y, d_y)$ で, 実数 1 次元配列 x, y の dot 積の計算を行います:

$$\sum_{i=1}^n x_{1+d_x \cdot (i-1)} \cdot y_{1+d_y \cdot (i-1)}$$

このルーチンも数値を返却するために配列の書換が生じません.

■scal: 構文は $\text{scal}(n, \alpha, x, d_x)$ で, 1 次元配列 x とスカラー α の積を計算します:

$$\alpha x_{1+d_x \cdot (i-1)} \mapsto x_{1+d_x \cdot (i-1)}$$

このルーチンでは配列 x の書換が生じます.

■**rot**: 構文は $\text{rot}(n, x, d_x, y, d_y, c, s)$ で, c, s は必ず実数値です. 以下の処理を $i \in \{1, \dots, n\}$ に対して行います:

$$\begin{cases} cx_{1+d_x \cdot (i-1)} + sy_{1+d_y \cdot (i-1)} & \mapsto x_{1+d_x \cdot (i-1)} \\ cy_{1+d_y \cdot (i-1)} - sx_{1+d_x \cdot (i-1)} & \mapsto y_{1+d_y \cdot (i-1)} \end{cases}$$

配列 x と y の双方で置換が生じます.

■**rotg**: 構文は $\text{rotg}(a, b, c, s)$ で, Givens 回転を行ったときに Y 座標が 0 になる点 $P(r, z)$ の座標と Givens 回転行列のパラメータ c, s の値を返却します. ここで rotg の引数 c, s は必ず実数です:

$$\begin{cases} \text{sgn}(a)\sqrt{a^2 + b^2} & \|a\| > \|b\| \\ \text{sgn}(b)\sqrt{a^2 + b^2} & \text{その他} \\ b/r & \|a\| > \|b\| \\ r/a & \|b\| \geq \|a\| \wedge c \neq 0 \\ 1 & \text{その他} \\ a/r & \mapsto c \\ b/r & \mapsto s \end{cases} \longmapsto \begin{cases} r \\ z \end{cases}$$

このルーチンで配列の書換が生じません.

■**rotm**: 構文は $\text{rotm}(n, x, d_x, y, d_y, p)$ で, 修正 Givens 回転を適用します. ここで x, y は 1 次元の実数配列, p は 5 成分の配列です. このルーチンでは配列 x, y の双方に計算結果による値の入換が生じます.

■**rotmg**: 構文は $\text{rotmg}(d_1, d_2, d_x, d_y, p)$ で, 修正 Givens 回転行列を生成します. ここで p 以外の引数は全てスカラ値で p のみが 5 成分の配列です.

■**nrm2**: 構文は $\text{nrm2}(n, x, d_x)$ で, ベクトル x の 2-ノルムを計算します:

$$\sqrt{\sum_{i=1}^n (\Re(x_{1+d_x \cdot (i-1)}))^2 + (\Im(x_{1+d_x \cdot (i-1)}))^2}$$

■**asum**: 構文は $\text{asum}(n, x, d_x)$ で, ベクトル x の 1-ノルムを計算します:

$$\sum_{i=1}^n (|\Re(x_{1+d_x \cdot (i-1)})| + |\Im(x_{1+d_x \cdot (i-1)})|)$$

1-ノルムを数値で返却するために引数の添字の置換を行いません.

■**amax**: 構文は $\text{amax}(n, x, d_x)$ で, ベクトル x の 1-ノルムで最大値を取る成分の添字を返却します.

4.4.10 BLAS の第二水準のルーチン

第二水準に含まれるルーチンは行列とベクトルの積、行列同士の和の処理です。ベクトルに関してデータの再利用性があり、計算量もそれなりにあります。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 $\text{op}_f(A)x$ や $\text{op}_f(A)^{-1}x$ を計算するルーチンを紹介します：

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	tbsmv	帯行列
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	帯行列
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計算します。ここでの三角形行列は対角成分の下側、あるいは上側が全て 0 になる正方形で、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帯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) かを指定
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) かを指定

ここで行列 A が「**単位三角行列 (unit triangular matrix)**」であるとは、その対角成分が全て 1 の三角行列です。

■**trmv:** 構文は $\text{trmv}(f_U, f, f_D, n, a, L, x, d_x)$ で, $n \times n$ の三角行列 A でその Leading dimension を L とするときに n 次元のベクトル v の $\text{op}_f(A)v \mapsto x$ を処理します. 結果は配列 x に格納されます.

■**tpmv:** 構文は $\text{tpmv}(f_U, f, f_D, n, a, x, d_x)$ で, $n \times n$ の帶三角行列 A に対して $\text{op}_f(A)x \mapsto x$ を処理します. 行列 A は圧縮形式で配列 a に, 処理結果は配列 x に格納されます.

■**trsv:** 構文は $\text{trsv}(f_U, f, f_D, n, a, L, x, d_x)$ で, $n \times n$ の帶三角行列 A でその Leading Dimension を L とするときに $\text{op}_f(A)^{-1}x \mapsto x$ を処理し, 結果を配列 x に格納します.

■**tbsv:** 構文は $\text{tbsv}(f_U, f, f_D, n, k, a, L, x, d_x)$ で, $n \times n$ の帶三角行列 A に対して $\text{op}_f(A)^{-1}x \mapsto x$ を処理します. ここで配列 a は行列 A の帶行列格納形式, k はその帶の幅に対応します. この処理結果は配列 x に格納されます.

■**tpsv:** 構文は $\text{tpsv}(f_U, f_t, f_D, n, a, x, d_x)$ で, $n \times n$ の三角行列 A について $\text{op}_f(A)^{-1}x \mapsto x$ を処理します. ここで配列 a は行列 A の圧縮格納形式になります. このルーチンの処理結果は配列 x に格納されます.

$\alpha \cdot \text{op}_f(A)v + \beta \cdot u$ の計算を行うルーチン

第二水準のルーチンに $\alpha \cdot \text{op}_f(A)v + \beta \cdot u$ の処理を行うものがあります. ここで各ルーチンの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 , スカラ α, β があります. また ‘ $L = \max(1, m)$ ’ を充す引数 L もあります. 以下に, この処理に関連するルーチンの名前を挙げます:

$\alpha \cdot \text{op}_f(A)v + \beta \cdot u$ 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	帶行列	
c z	hemv	エルミート行列	
c z	hbmv	エルミート帶行列	
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	対称帶行列	
s d	spmv	圧縮格納形式の対称行列	

■gemv: 構文は $\text{gemv}(f, m, n, \alpha, a, L, x, d_x, \beta, y, d_y)$ で, $m \times n$ の一般行列 A で Leading dimension を L とし, n 次元のベクトル v , m 次元のベクトル v に対して処理し, 結果を配列 y に格納します.

■gbmv: 構文は $\text{gbmv}(f, m, n, k_l, k_u, \alpha, a, L, x, d_x, \beta, y, d_y)$ で, $m \times n$ の帶行列 A を格納した配列 a と Leading dimension を L とし, ベクトル v, u を格納した配列 x, y に対して処理します. ここで引数の k_l, k_u はそれぞれ対角成分よりも下側の帶の幅, 上側の帶の幅を指定する正整数です. このルーチンの処理結果は配列 y に格納されます.

■hemv: 構文は $\text{hemv}(f_U, n, \alpha, a, L, x, d_x, \beta, y, d_y)$ で, $n \times n$ のエルミート行列 A を格納した配列 a で Leading dimension を L とし, ベクトル v, u を格納した配列 x, y に対して処理します. なお, フラグ f_U は上三角 ("u", "U") か下三角 ("l", "L") の何れかを用いて計算するかを指定するフラグです. このルーチンの処理結果は配列 y に格納します.

■hbmv: 構文は $\text{hbmv}(f_U, n, k, \alpha, a, L, x, d_x, \beta, y, d_y)$ で, $n \times n$ のエルミート行列で対角成分からの幅が k の帶行列 A を帶行列形式で格納した配列 a , その Leading dimension を L としてベクトル v, u を格納した配列 x, y に対して処理します. なお, フラグ f_U は上三角 ("u", "U") か下三角 ("l", "L") の何れかを用いて計算するかを指定するフラグです. このルーチンの処理結果は配列 y に格納されます.

■hpmv: 構文は $\text{hpmv}(f_U, n, \alpha, a, x, d_x, \beta, y, d_y)$ で, $n \times n$ のエルミート行列 A の圧縮格納した配列 a , ベクトル v, u を格納した配列 x, y を用いて処理します. なお, フラグ f_U は上三角 ("u", "U") か下三角 ("l", "L") の何れかを用いて計算するかを指定するフラグです. このルーチンの処理結果は配列 y に格納されます.

■symv: 構文は $\text{symv}(f_U, n, \alpha, a, L, x, d_x, \beta, y, d_y)$ で, $n \times n$ の対称行列 A を格納した配列 a , その Leading dimension を L とし, ベクトル v, u を格納した x, y を用いて処理します. なお, フラグ f_U は上三角 ("u", "U") か下三角 ("l", "L") のいずれかを用いて計算するかを指定するフラグです. このルーチンの処理結果は配列 y に格納されます.

■sbmv: 構文は $\text{sbmv}(f_U, n, k, \alpha, a, L, x, d_x, \beta, y, d_y)$ で, 対称行列で帶行列でもある行列 A を幅 k の帶形式で格納した配列 a , その Leading Dimension を L とし, ベクトル v, u を格納した x, y を用いて処理します. なお, フラグ f_U は上三角 ("u", "U") か下三角 ("l", "L") の何れかを用いて計算するかを指定するフラグです. このルーチンの処理結果は配列 y に格納されます.

■spmv: 構文は $\text{spmv}(f_U, n, \alpha, a, x, d_x, \beta, y, d_y)$ で, 対称行列 A を圧縮して格納した配列 a , ベクトル v, u を格納した x, y を用いて処理します. なお, フラグ f_U は上三角 ("u", "U") か下三角 ("l", "L") の何れかを用いて計算するかを指定するフラグです. こ

のルーチンの処理結果は配列 y に格納されます。

行列を生成するルーチン

第二水準の最後のルーチンは、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	$\alpha \cdot v^t v + A$	$A = {}^t A$
s d	spr	$\alpha \cdot v^t v + A$	$A = {}^t A, A:$ 圧縮格納形式
s d	ger	$\alpha \cdot v^t u + A$	$A \in M(m, n)$
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	$\alpha \cdot v^t u + \alpha \cdot u^t v + A$	$A = {}^t A$
s d	spr2	$\alpha \cdot v^t u + \alpha \cdot u^t v + A$	$A = {}^t A, A:$ 圧縮格納形式
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。

■her: 構文は $\text{her}(f_U, n, \alpha, x, d_x, a, L)$ で、階数 1 の更新を行うルーチンです。エルミート行列 A を一般形式で収納した配列 a , Leading dimension を L とし、ベクトル v を収納した配列 x に対して $\alpha \cdot v^t \bar{v} + A \mapsto A$ を処理します。

■hpr: 構文は $\text{hpr}(f_U, n, \alpha, x, d_x, a)$ で、階数 1 の更新を行うルーチンです。エルミート行列 A を圧縮格納形式で収納した配列 a , ベクトル v を収納した配列 x に対して $\alpha \cdot v^t \bar{v} + A \mapsto A$ を処理します。

■**syr:** 構文は $\text{syr}(f_U, n, \alpha, x, d_x, a, L)$ で, 階数 1 の更新を行うルーチンです. 行列 A は実数対称行列で, 一般形式で配列 a に格納され, Leading dimension を L とし, ベクトル v を収納した配列 x に対して $\alpha \cdot v^t v + A \mapsto A$ を処理します.

■**spr:** 構文は $\text{spr}(f_U, n, \alpha, x, d_x, a,)$ で, 階数 1 の更新を行うルーチンです. 行列 A は実数対称行列で, 圧縮格納形式で配列 a に格納され, ベクトル v を収納した配列 x に対して $\alpha \cdot v^t v + A \mapsto A$ を処理します.

■**ger:** 構文は $\text{ger}(f_U, n, \alpha, x, d_x, y, d_y, a, L)$ で, 階数 1 の更新を行うルーチンです. 行列 A は実数行列で, 一般形式で配列 a に格納され, Leading dimension を L とし, ベクトル v と v を収納した配列 x と y に対して $\alpha \cdot v^t u + A \mapsto A$ を処理します. gerc との違いは配列 y の共役を計算する点です.

■**gerc:** 構文は $\text{gerc}(f_U, n, \alpha, x, d_x, y, d_y, a, L)$ で, 階数 1 の更新を行うルーチンです. 行列 A は一般の複素数行列で, 一般形式で配列 a に格納され, Leading dimension を L とし, ベクトル v と v を収納した配列 x と y に対して $\alpha \cdot v^t \bar{u} + A \mapsto A$ を処理します. geru との違いはベクトル u の共役を取ることです.

■**geru:** 構文は $\text{geru}(f_U, n, \alpha, x, d_x, y, d_y, a, L)$ で, 階数 1 の更新を行うルーチンです. 行列 A が実数行列であれば A は対称行列, 行列 A が複素数行列であれば A はエルミート行列で, 圧縮格納形式で配列 a に格納され, Leading dimension を L とし, ベクトル v と v を収納した配列 x と y に対して $\alpha \cdot v^t u + A \mapsto A$ を処理します. gerc との違いは geru ではベクトル u の共役を計算しないことです.

■**syr2:** 構文は $\text{syr2}(f_U, n, \alpha, x, d_x, y, d_y, a, L)$ で, 階数 1 の更新を行うルーチンです. 行列 A が実数行列であれば A は対称行列, 行列 A が複素数行列であれば A はエルミート行列で, 圧縮格納形式で配列 a に格納され, Leading dimension を L とし, ベクトル v と v を収納した配列 x と y に対して $\alpha \cdot v^t u + \alpha \cdot u^t v + A \mapsto A$ を処理します.

■**spr2:** 構文は $\text{spr2}(f_U, n, \alpha, x, d_x, a)$ で, 階数 1 の更新を行うルーチンです. 行列 A が実数行列であれば, A は対称行列, 行列 A が複素数行列であれば A はエルミート行列で, 圧縮格納形式で配列 a に格納され, ベクトル v を収納した配列 x に対して $\alpha \cdot v^t u + \alpha \cdot u^t v + A \mapsto A$ を処理します.

■**her2:** 構文は $\text{her2}(f_U, n, \alpha, x, d_x, a, L)$ で, 階数 1 の更新を行うルーチンです. 行列 A が実数行列であれば A は対称行列, 行列 A が複素数行列であれば A はエルミート行列で, 圧縮格納形式で配列 a に格納され, Leading dimension を L とし, ベクトル v を収納した配列 x に対して $\alpha \cdot v^t \bar{u} + \overline{\alpha} \cdot u^t \bar{v} + A \mapsto A$ を処理します.

■**hpr2:** 構文は $\text{hpr2}(f_U, n, \alpha, x, d_x, a)$ で、階数 1 の更新を行うルーチンです。行列 A が実数行列であれば A は対称行列、行列 A が複素数行列であれば A はエルミート行列で、圧縮格納形式で配列 a に格納され、ベクトル v を収納した配列 x に対して $\alpha \cdot v^T \bar{u} + \bar{\alpha} \cdot u^T \bar{v} + A \mapsto A$ を処理します。

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	$\alpha \cdot \text{op}_{f_1}(A) \text{ op}_{f_2}(B) + \beta \cdot C$	
	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	$\alpha \cdot A B + \beta \cdot C$ $\alpha \cdot B A + \beta \cdot C$	$s \in \{"L", "l"\}$ $s \in \{"R", "r"\}$
	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	$\text{op}_{f_1}(A) X = \alpha \cdot B$ $X \text{ op}_{f_1}(A) = \alpha \cdot B$	$s \in \{"L", "l"\}$ $s \in \{"R", "r"\}$
	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	$\alpha \cdot A^T A + \beta \cdot C$ $\alpha \cdot {}^T A A + \beta \cdot C$	$t \in \{"N", "n"\}$ $t \in \{"C", "c"\}$
	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	$\alpha \cdot A^T B + \alpha \cdot B^T A + \beta \cdot C$ $\alpha \cdot {}^T A B + \alpha \cdot {}^T B A + \beta \cdot C$	$t \in \{"N", "n"\}$ $t \in \{"C", "c"\}$

■**gemm:** 構文は $\text{gemm}(f_1, f_2, m, n, k, \alpha, a, L_a, b, L_b, \beta, c, L_c)$ です。このルーチンは $\alpha \cdot \text{op}(A) \text{ op}(B) + \beta C \mapsto C$ を処理します。ここで f_1, f_2 は作用素 op が行列の転置を行うかどうかを指示するフラグで、 $\text{op}_{f_1}(A) \in M(m, k)$, $\text{op}_{f_2}(B) \in M(k, n)$ を充します。行列 A, B, C の Leading dimension を L_a, L_b, L_c とし、計算結果は配列 c に格納されます。

■hemm: 構文は $\text{hemm}(s, f_U, m, n, \alpha, a, L_a, b, L_b, \beta, c, L_c)$ です。ここで行列 A は $m \times m$ のエルミート行列、行列 B, C は $m \times n$ の行列です。このルーチンは s の値で処理する式の切替があります。 s の値が "L", あるいは "l" のときに $\alpha \cdot A B + \beta \cdot C \mapsto C$, 値が "R", あるいは "r" のときに $\alpha \cdot B A + \beta \cdot C \mapsto C$ の処理を行います。また、 f_U は行列 A が上三角行列か下三角行列であるかを指示するフラグです。処理結果は配列 c に格納されます。

■symm: 構文は $\text{symm}(s, f_U, m, n, \alpha, a, L_a, b, L_b, \beta, c, L_c)$ です。行列 A は $m \times m$ の対称行列で、行列 B, C は $m \times n$ の行列です。このルーチンは s の値で処理する式の切替があります。 s の値が "L", あるいは "l" のときに $\alpha \cdot A B + \beta \cdot C \mapsto C$, 値が "R", あるいは "r" のときに $\alpha \cdot B A + \beta \cdot C \mapsto C$ の処理を行います。また、 f_U は行列 A が上三角行列か下三角行列であるかを指示するフラグです。処理結果は配列 c に格納されます。

■trmm: 構文は $\text{trmm}(s, f_U, f, d, m, n, \alpha, a, L_a, b, L_b)$ です。行列 A は $m \times m$ の三角行列で、行列 B は $m \times n$ の行列です。このルーチンは s の値で処理する式の切替があります。 s の値が "L", あるいは "l" のときに $\alpha \cdot \text{op}_{f_U} A B \mapsto B$, 値が "R", あるいは "r" のときに $\alpha \cdot B \text{ op}_{f_U} A \mapsto C$ の処理を行います。また、 f_U は行列 A が上三角行列か下三角行列であるかを指示するフラグです。処理結果は配列 b に格納されます。

■trsm: 構文は $\text{trsm}(s, f_U, f, d, m, n, \alpha, a, L_a, b, L_b)$ です。行列 A は $m \times m$ の三角行列で、行列 X, B は $m \times n$ の行列です。このルーチンは s の値で処理する式の切替があります。 s の値が "L", あるいは "l" のときに $\text{op}_f(A) X = \alpha \cdot B$, 値が "R", あるいは "r" のときに $X \text{ op}_f(A) = \alpha \cdot B$ について X を計算します。ここで、 f_U は行列 A が上三角行列か下三角行列であるかを指示するフラグで、結果は配列 b に格納されます。

■herk: 構文は $\text{herk}(f_U, t, n, k, \alpha, a, L_a, \beta, c, L_c)$ です。行列 C は $n \times n$ のエルミート行列、行列 A は $n \times k$, あるいは $k \times n$ の行列です。このルーチンは t の値で処理する式の切替があります。 t が "N", あるいは "n" のときに $\alpha \cdot A {}^t \bar{A} + \beta \cdot C \mapsto C$, 値が "C", あるいは "c" のときに $\alpha \cdot {}^t \bar{A} A + \beta \cdot C \mapsto C$ を処理します。ここで、 f_U は行列 A が上三角行列か下三角行列であるかを指示するフラグで、結果は配列 c に格納されます。

■syrk: 構文は $\text{syrk}(f_U, t, n, k, \alpha, a, L_a, \beta, c, L_c)$ です。行列 C は $n \times n$ の対称行列、行列 A は $n \times k$, あるいは $k \times n$ の行列です。このルーチンは t の値で処理する式の切替があります。 t が "N", あるいは "n" のときに $\alpha \cdot A {}^t A + \beta \cdot C \mapsto C$, 値が "C", あるいは "c" のときに $\alpha \cdot {}^t A A + \beta \cdot C \mapsto C$ を処理します。ここで、 f_U は行列 A が上三角行列か下三角行列であるかを指示するフラグで、結果は配列 c に格納されます

■her2k: 構文は $\text{her2k}(f_U, t, n, k, \alpha, a, L_a, \beta, c, L_c)$ です。行列 C は $n \times n$ のエルミート行列、行列 A は $n \times k$ 、あるいは $k \times n$ の行列です。このルーチンは t の値で処理する式の切替があります。 t が"N", あるいは "n" のときに $\alpha \cdot A {}^t \bar{B} + \bar{\alpha} \cdot B {}^t \bar{A} + \beta \cdot C$, 値が"C", あるいは "c" のときに $\alpha \cdot {}^t \bar{A} B + \bar{\alpha} \cdot {}^t \bar{B} A + \beta \cdot C$, を処理します。ここで、 f_U は行列 A が上三角行列か下三角行列であるかを指示するフラグで、結果は配列 c に格納されます。

■syr2k: 構文は $\text{syr2k}(f_U, t, n, k, \alpha, a, L_a, \beta, c, L_c)$ です。行列 C は $n \times n$ の対称行列、行列 A は $n \times k$ 、あるいは $k \times n$ の行列です。このルーチンは t の値で処理する式の切替があります。 t が"N", あるいは "n" のときに $\alpha \cdot A {}^t B + \alpha \cdot B {}^t A + \beta \cdot C$, 値が"C", あるいは "c" のときに $\alpha \cdot {}^t A B + \alpha \cdot {}^t B A + \beta \cdot C$, を処理します。ここで、 f_U は行列 A が上三角行列か下三角行列であるかを指示するフラグで、結果は配列 c に格納されます。

4.4.11 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といったルーチンです。

- 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

ここで「方程式系の均衡化/平衡化」とは、行列 A を相似変換することで行列の成分の絶対値の差を減らす処理です。この処理を行う理由は、行列を構成する浮動小数点数はあくまでも数の近似値なので、行列成分の絶対値でその最大値と最小値の差が大きければそれだけ計算精度を保つ上で不利になります。そこで差を小さくすることで精度の問題を低減することを目的とします。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に二倍程度の記憶容量を必要とします。

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くことになります。

補助ルーチン

補助的に用いられるルーチンです。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、この函数の実装方法が BLAS の計算処理速度を大きく左右します。

4.5 MATLAB 系言語と Numpy の速習

4.5.1 定数の扱い

MATLAB 系言語では円周率 π 、ネイピア数 e や純虚数 i 等の数学的定数は初期化ファイルの読み込みで設定が行われます。Python では純虚数は定数ではなく数リテラルとして現れ、Python 本体に数学的定数は含まれていません。この数学的定数は Python 本体には含まれておらず、Numpy などのモジュールで実装されています。ただし、これらの定数は保護されているとは言い難く、書き換えが可能です。とくに MATLAB の円周率、ネイピア数と純虚数は ‘pi’、‘e’、‘i’ で、これらのリテラルは変数名としても利用可能です。実際、for 文で i を局所変数として利用が可能で、このときに i の値の書き換えが生じてしまします。このことは Python でも基本的に属性の保護が行われないこともあって同様に注意が必要です。ただし、Python の特性上、Numpy のネイピア数と円周率が import 文の読み込みによって ‘numpy.e’ や ‘numpy.pi’ のようにモジュール名込みのリテラルで参照が行われ、この場合は通常の変数のリテラルと異なっていることが MATLAB や Octave で生じ得る偶発的な上書きの抑止策になっています。なお、Scilab ではこのような数学定数に対しては変数名先頭に文字 “%” を付けて変数と区別しています。

4.5.2 MATLAB 系言語でのベクトルと行列の定義

ベクトルと行列の基本定義

MATLAB 系言語で 1 次元配列は角括弧 “[]” で括ったリスト表記で表現されます。Python も同様ですが、Python のリスト型のオブジェクトで MATLAB 系言語と同等のベクトル・行列演算ができません。そのためにモジュール Numpy で定義される ndarray 型の配列で対処します。この ndarray 型の配列は基本的に多次元配列で、MATLAB 系言語のベクトル・行列と性格がやや異なりますが、リストを基盤に配列を構築することができます。ここでは MATLAB 系言語を代表して Scilab で行列を生成する例を示しておきます：

```
-->[1, 2, 3; 4, 5, 6]
```

```
ans =
```

```
1.    2.    3.  
4.    5.    6.
```

```
-->[1;2;3]
```

```
ans =
```

```
1.  
2.  
3.
```

```
—>a=[1 2 3]
```

```
a =
```

```
1.    2.    3.
```

```
—> a'
```

```
ans =
```

```
1.  
2.  
3.
```

MATLAB 系言語では行列やベクトルは角括弧 “[]” で数値の列を括ったリスト書式で生成できます。まず、ベクトルは数値の列の区切文字をカンマ “,”、あるいは Space とするリストから行ベクトルが生成され、同じ長さのリストで、セミコロン “;” を各リストの区切文字とすることで数値行列が生成されます。このときに各リストが行列の行を構成することになります。ここでの例で ‘[1, 2, 3; 4, 5, 6]’ と入力していますが、この列には二種類の区

切記号があります。まず、セミコロンが行列の行を区分する区切です。この例では一行目が‘1, 2, 3’、二行目が‘4, 5, 6’になります。すると、各行はカンマ“,”で区切った数の列になります。このように MATLAB 系の言語ではベクトルの成分の区切記号をカンマ“,”, 行の区切記号をセミコロン“;”とします。つまり、

$$[1, 2, 3; 4, 5, 6] \Leftrightarrow \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

という対応があります。このように MATLAB 系言語で行列の入力は行を区切記号“;”を使って 1 次元的に繋ぐ書式ですが、非常に直接的な入力になっています。また、MATLAB 系の言語は最近、オブジェクト指向プログラミング言語としての味付けされたものもありますが、基本的に構造化 Basic 的な処理言語で、行列操作も函数や演算子を使い、その他は後述の配列操作でさまざまな計算処理を行う言語です。そして、行列演算は式に近い表記です。ここで演算子(‘)を行列の右側に配置していますが、これによって転置行列が生成されます。この表記も数学上の行列 A の転置行列 A' に合せたものです。

ベクトルや行列の成分指定

MATLAB 系の言語ではベクトルや行列の成分の指示は、Python の添字表記の角括弧 “[]” を用いずに、函数と同様の丸括弧“()”を用い、添字は FORTRAN 同様に 1 から開始します。たとえば、行列 a の i 行 j 列の成分は ‘ $a(i,j)$ ’ で指示します。そして、ベクトルや行列の元の設定はリスト書式で一度に設定する方法と、‘ $a(1,2)=1$ ’ のように成分を一々指定して設定する方法もあります。また、変数 a に何も束縛されていないときに ‘ $a(1,2)=1$ ’ を入力すると変数 a に 1 行 2 列の行列で $a(1,2)$ のみが 1 で他の 0 の行列が束縛されます。

このときに続けて ‘ $a(3,2)=10$ ’ と入力すると a は 3 行 2 列の行列に自動的に拡大され、その値が代入された個所 ($a(1,2)$ と $a(3,2)$) を除いて 0 です。この方法以外に行列をベクトルとみなして要素を指定できます。さらに配列 a が m 行 n 列の行列のときに $a(i,j)$ は $a(i+(j-1)*m)$ として解釈されますが、これは BLAS の一般行列の 1 次元配列への収納の添字と一致します。なお、このように解釈されるのは行列であって、行や列ベクトルが成分数が等しい行列として解釈し直されるという意味ではありません。この解釈が行われるのは行列の場合だけです。つぎに MATLAB 系言語では配列の大きさは函数 size() で、1 次元配列としての長さは函数 length() で調べられます。ここで Scilab による幾つかの操作例を示しておきます:

```
—>a=[1 2 3;4 5 6]
```

```
a =
```

```
1.    2.    3.  
4.    5.    6.
```

```
—>a(1,2)
ans =
2.

—>a(3)
ans =
2.

—>size(a)
ans =
2.    3.

—>length(a)
ans =
6.
```

スライス表記による生成と成分の取り出し

MATLAB 系言語では Python のスライス操作に類似した方法でベクトルの生成が可能です。まず、等間隔のベクトルを入力するときに、これらの数値を全て直接入力する必要はありません。Python のスライス表記と比較を容易にするために BNF を示しておきます：

MATLAB 系言語のスライス表記

```
スライス表記      ::= 配列名 "[" 1 次元スライス表記 |
                           2 次元スライス表記 "]"
2 次元スライス表記 ::= (式 | 1 次元スライス表記) ","
                           (式 | 1 次元スライス表記)
1 次元スライス表記 ::= スライス本体 | スライスリスト
スライスリスト   ::= 式 ("," 斜線リスト)
スライス本体     ::= [始点] ":" [(刻幅 ":")] [終点]
始点             ::= 式
終点             ::= 式
刻幅             ::= 式
```

Python との違いは添字が 1 から開始することと Python の長スライスに相当するここでのスライス本体で、刻幅が末尾ではなく中間に配置される点です。さらに MATLAB 系言

語ではスライス本体で列の生成が可能で、本質的に生成的な表記ですが、Python のスライス表記は既存の成分の抜き出しに関わる参照的な表記です。スライス本体を使った行ベクトルの生成を次に示しておきましょう：

```
-->[0:0.1:0.5]
ans =
0.      0.1      0.2      0.3      0.4      0.5

-->[0:3:10]
ans =
0.      3.      6.      9.
```

最初の例では始点から刻幅を 0.1 でベクトルの生成を行っています。このときに終点は刻幅の関係でちょうど一致しますが、二番目の例では終点の 10 は 0 から開始する間隔の 3 の数列に含まれません。この場合は始点から開始し、終点に向かう数列で終点を越えない最大の数を数列の末端にします。そのために 9 でこの列が停止することになります。

このことをまとめておきましょう。MATLAB 系の言語では、定数 $i, j (i \geq j)$ に対し、 $i : j$ と記述することで、先頭が i 、末端を j とする 1 刻みの数列が得られます。この刻幅の指定も、刻幅 d であれば $i : d : j$ とします。これは Numpy のスライス表記と異なる点で、Numpy の場合は $i : j : d$ と添字 d を末尾に配置します。ただし、MATLAB 系言語と違い生成的な機能を持たないために、あくまでも添字でしか使えません。また、 $i < j$ のときに $j : i$ とすると、増分 1 で j から i に到達できないために空行が返されます。このときは刻幅を負の数にします。それから ‘1:2:6’ のように大小関係に問題はなくとも刻幅で i から j に直接到達できないときは j を越えない最大の数で数列が止まります。これらの例を以下に示しておきます：

```
-->1:5
ans =
1.      2.      3.      4.      5.

-->1:2:5
ans =
1.      3.      5.

-->1.1:0.1:1.5
ans =
```

```
1.1      1.2      1.3      1.4      1.5  
-->5:1  
ans  =  
[]  
-->5:-2:1  
ans  =  
5.      3.      1.  
-->1:2:6  
ans  =  
1.      3.      5.  
-->1:0.3:2  
ans  =  
1.      1.3     1.6      1.9
```

もちろん、配列の生成は for 文を用いて行えます：

```
-->for i=1:5  
-->bb(1,i)=0.1*i;  
-->end;  
  
-->bb  
bb  =  
!    0.1      0.2      0.3      0.4      0.5 !
```

ただし、MATLAB 系言語ではスライス表記で数列がより簡易なかたちで生成されるため、この目的のためだけに for 文を使うことはまずないでしょう。

MATLAB 系言語では、ベクトルや行列の成分だけではなく部分の取り出しある可能です。この取り出しあはスライス表記を使って行えます。まず、行ベクトルの i 番目の成分から j 番目の成分を持つベクトルは ‘ $a(i:j)$ ’ で得られます。また、記号 “ $:$ ” の両端に数字を入れないときは、記号 “ $:$ ” が置かれた箇所の添字全てという意味になります。たとえば、行列 A の i 行目で構成される行ベクトルと j 列目で構成される列ベクトルはそれぞれ ‘ $A(i,:)$ ’ と ‘ $A(:,j)$ ’ で得られます：

```

—>a=[1:10]
a =
1.    2.    3.    4.    5.    6.    7.    8.    9.    10.

—>a(4:6)
ans =
4.    5.    6.

—>A=[1:4; 5:8; 8:11]
A =
1.    2.    3.    4.
5.    6.    7.    8.
8.    9.    10.   11.

—>b=A(2, :)
b =
5.    6.    7.    8.

—>c=A(:, 2)
c =
2.
6.
9.

```

MATLAB 系の言語で添字としての記号 “:” は、その添字に対する次元の全てを意味する省略記号です。この省略方法は Python の配列でも見られる表記ですが、Python の Ellipsis“...” は残りの次元も含めた省略になっています。

特定の行列を生成する函数

MATLAB 系の言語で行列処理に for 文を用いると処理速度の大幅な低下を招くことに注意が必要です。MATLAB 系の言語では特定の形式の行列を素早く生成するための幾つかの函数があります。その中で一般的なものを述べておきましょう。行列の対角成分に決まった数値を設定する函数 diag() があります。この函数 diag() は行列の対角成分に設定する数値をベクトルで与えます。対角成分を対角線上から指定した数値分移動させることもできます。また、逆に与えられた行列の対角成分を抜き出すこともできます。対角成分が全て 1 で他の全て 0 の単位行列の生成は函数 eye() があります。そして、成分全てが 0

の行列を生成する函数 zero() もあります。これらの函数を Scilab で使ってみましょう：

```
—> diag([1,2],1)
```

```
ans =
```

```
0. 1. 0.  
0. 0. 2.  
0. 0. 0.
```

```
—> diag([1,2],3)
```

```
ans =
```

```
0. 0. 0. 1. 0.  
0. 0. 0. 0. 2.  
0. 0. 0. 0. 0.  
0. 0. 0. 0. 0.  
0. 0. 0. 0. 0.
```

```
—> diag([1,2],-2)
```

```
ans =
```

```
0. 0. 0. 0.  
0. 0. 0. 0.  
1. 0. 0. 0.  
0. 2. 0. 0.
```

```
—> eye(3,2)
```

```
ans =
```

```
1. 0.  
0. 1.  
0. 0.
```

```
—> A=rand(3,3)
```

```
A =
```

```
0.6678341 0.4273231 0.3739742  
0.0256438 0.1086697 0.1518774  
0.0315958 0.7679413 0.7219999
```

```
—> diag(A)
ans =
0.6678341
0.1086697
0.7219999

—> zeros(2,3)
ans =
0.    0.    0.
0.    0.    0.
```

```
—> ones(2,3)
ans =
1.    1.    1.
1.    1.    1.
```

このように MATLAB 系の言語では容易にベクトルや行列の生成が行えます。

4.5.3 Numpy の場合

Numpy を使った配列の定義

Python のモジュール Numpy で定義される ndarray 型の配列は構築子 array() を使って生成します。このときに配列データは角括弧 “[]” のリスト型や括弧 “()” で括ったタプル型のオブジェクトとして与えます。 ndarray の次元は括弧の深さとして表現されます。そのため括弧でリストを括ったものは 1 次元配列で、行列は行を括弧で括ったリストとして表現されます：

```
>>> 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
```

この例では import 文でモジュール numpy を読み込む際に np と名前を変えています。そのためにモジュール Numpy で定義された函数や定数への参照は先頭に “np.” を付ける必要があります。最初に配列の生成を Python のリストを利用して行っていますが、これはリスト形式である必要性はなく、括弧 “()” で括ったタプルでも構いません。ただし、‘1,2,3’ のような括弧のない「式のリスト」はエラーになります。いずれにせよ行列やベクトルの定義で、行は角括弧 “[]” や丸括弧 “()” で括られていることが必要です。また、Numpy ではベクトルや行列は 2 次元配列で扱うことになります。そのため ndarray オブジェクトの表示で括弧 “[]” の深さが 2 層になっています。この配列の次元はメソッド ndim で調べることができます。また、配列を 1 次元配列とみなしたときの長さはメソッド size、配列の具体的な大きさはメソッド shape で調べられます。

Numpy では別に matrix 型もあります。後述の配列の演算では ndarray 型の配列は基本的に成分単位の演算になりますが、matrix 型では行列演算が容易に行えます。Numpy での定義の様子を示しておきます：

```
>>> 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]])

```

構築子 `matrix()` による行列の定義も構築子 `array()` と同様に行えます。ただし、構築子 `matrix()` で生成したオブジェクトと構築子 `array()` で生成したオブジェクトの型は当然異なります。また、これらの構築子 `matrix()` と `array()` で互いの型に変換できます。これらの変換は

Numpy の配列生成では、1次元配列として生成し、それから配列としての形式をメソッド `reshape()` で変更するという方法があります。なお、MATLAB 系言語と違い、スライス表記で1次元配列の生成はできません。その代わりに MATLAB 系言語ではできない内包表現による配列の生成ができます。また、`ndarray` は多次元配列であるために MATLAB 系言語のように行列 `a` を1次元の配列とみなして成分を取り出すことはできません：

```

>>> import numpy as np
>>> a = np.array([i for i in range(0,10)])
>>> a = a.reshape((5,2))
array([[0, 1],
       [2, 3],
       [4, 5],
       [6, 7],
       [8, 9]])

```

```
[6, 7,
 [8, 9]])
>>> b = np.array([i for i in range(0,8)])
>>> a.reshape((2,2,2))
array([[[0, 1],
       [2, 3]],

      [[4, 5],
       [6, 7]]])
>>> b = np.array([1:10])
File "<stdin>", line 1
    b = np.array([1:10])
^
SyntaxError: invalid syntax
```

この例ではメソッド `reshape()` で 1 次元配列を 2 次元配列や 3 次元配列に変換していますが、これらの変換は配列 `a` を入れ替えるものではないために配列 `a` に結果を入れるようにしています。また、1 次元配列から 2 次元配列へのメソッド `reshape()` による変換は配列の大きさが $m \times n$ のときに $a[i, j] \leftrightarrow a[j + n \cdot i]$ で与えられ、MATLAB 系の言語で見られた列優先ではなく行優先になります。なお、Numpy の配列の添字は 0 から開始するために、ここでの解説では 0 から開始の場合に合わせて書き直しています。このように MATLAB 系言語に似たことができますが、スライス表記で配列の生成はできません。

このように Numpy での配列/行列の生成は構築子を用いるために MATLAB 系言語のような直接入力する感覚が乏しくなりますが、ほとんど同じ処理ができます。むしろ、配列の生成も内包表現による生成が可能なだけ、より、細かな指示が行えます。

スライス表記による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ことから、配列の生成でスライス表示は直接的に関係することがなく、また、配列の添字にリストを与えることもできません：

```
>>> a = np.array([1:10])
File "<stdin>", line 1
    a = np.array([1:10])
^
SyntaxError: invalid syntax
>>> np.array([i*0.1 for i in range(0,6)])
array([ 0.,  0.1,  0.2,  0.3,  0.4,  0.5])
>>> np.array([range(0,10,3)])
```

```
array([[0, 3, 6, 9]])
```

最初の例は MATLAB 式に生成できないことを示す例です。そのために函数 range() を使った内包表現等で対処することになります。まず、函数 range() は単体では三番目の例のように始点と終着点と必要なら刻幅を指定します。この函数 range() の引数は全て整数で、返却値も Python の整数を成分とするリスト型です。そのため、刻幅が 0.1 といった浮動小数点数を利用する場合は二番目の例のように内包表現を利用します。この状況が Python のスライス表記が生成的でないという意味です。ただし、配列成分参照のスライス式は MATLAB 系言語と同等であり、Numpy の ndarray 型配列が多次元配列であることから、Ellipsis 型の添字もあり、MATLAB 系言語以上の機能を持ちます。ここで 1 次元配列の例を示しておきます：

```
>>> 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[-2]
1.8
>>> 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. ])
```

配列の添字は 0 から開始します。配列の末尾は -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になります。また、スライス表記で配列の一部を取り出すこともできます。これらの点は MATLAB 系言語と同様です。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 系言語のもの

とずれることです。次に行列のスライス表記を見てみましょう：

```
>>> 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つたときと同様の「省略」の意味もありますが、この Ellipsis には「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 $3 \times 2 \times 4$ とっています。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。Ellipsis“...”はこのように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、Yorick はさらに高次元配列操作に特徴があり、配列の次元を自由自在に拡張・縮小ができます。また、添字に和、平均値、最大値、最小値といった指示を入れ、それらに相当する値を持った配列を返すこともできます。

特定の行列を生成する函数

Numpy にも MATLAB 系言語と同様に特定の行列を生成する函数が揃っています。ここでは MATLAB 系言語の紹介でも挙げた良く利用される配列生成の函数の実例を示しておきます：

```
>> 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])  
>>> np.ones([2,3])  
  
array([[ 1.,  1.,  1.],  
       [ 1.,  1.,  1.]])  
>>> np.zeros([2,3])  
  
array([[ 0.,  0.,  0.],  
       [ 0.,  0.,  0.]])
```

このように MATLAB 系の言語にあった、指定した対角成分を持つ配列、対角成分が 1 のみの配列、全ての成分が 0 や 1 の配列を生成する函数が、MATLAB 系言語と同じ名前で、同じ使い方ができます。

4.5.4 行列処理について

はじめに

ここでは添字を使った行列処理について述べます。添字を使った成分の取出はスライス処理で行えます。ここで述べる処理は添字を使った代入処理、ある条件を充す成分の取出といった操作です。

4.5.5 MATLAB 系言語での添字を使った代入操作

記号 ":"を利用して行列を列や行ベクトルの集まりとみなして抽出だけではなく代入もできます:

—>A=rand(4,4)

A =

0.2113249	0.6653811	0.8782165	0.7263507
0.7560439	0.6283918	0.0683740	0.1985144
0.0002211	0.8497452	0.5608486	0.5442573
0.3303271	0.6857310	0.6623569	0.2320748

—>B=zeros(4,4)

B =

0.	0.	0.	0.
0.	0.	0.	0.
0.	0.	0.	0.
0.	0.	0.	0.

—>B(:,2)=A(:,4)

B =

0.	0.7263507	0.	0.
0.	0.1985144	0.	0.
0.	0.5442573	0.	0.
0.	0.2320748	0.	0.

—>

a(:,1)に代入を行うときは代入される側の大きさが記号 ":"で省略されているときでも、当然、両方とも同じ大きさでなければなりません:

—> A=rand(4,4)

A =

0.3840511	0.1684622	0.2354968	0.8804158
0.2286183	0.9912121	0.0611997	0.525798
0.6369692	0.8467154	0.1432106	0.7148241
0.355703	0.0738953	0.3392998	0.3894592

```
—> c=zeros(2,2)
```

```
c =
```

```
0. 0.  
0. 0.
```

```
—> c=zeros(2,4)
```

```
c =
```

```
0. 0. 0. 0.  
0. 0. 0. 0.
```

```
—> c(:,2)=A(:,3) 誤ったサブ行列の定義が行われました
```

```
.
```

```
—> c(1,:)=A(:,3)
```

```
c =
```

```
0.2354968 0.0611997 0.1432106 0.3392998  
0. 0. 0. 0.
```

このように記号“:”で示した行列が右辺と左辺の大きさが違うときにエラーになります。ただし、ここでの大きさとは左辺の代入される配列の成分の総数と右辺の代入で用いられる配列の成分の総数が一致するかどうかという意味です。この点は行列が1次元配列としての表現を持つために可能です。逆に言えば、代入先の添字指定が適切でなければ間違った大きさで代入が行われる可能性があります。つまり、行ベクトルとして代入したつもりが列ベクトルに代入されるといった塩梅です。

Numpy での添字を使った代入操作

Numpy でも添字を使った代入操作は、添字が0から開始することと、スライス表示の部分の位置が MATLAB 系言語と異なる点に注意する必要がありますが、同様のことができます。

4.6 配列の演算について

MATLAB 系言語での加減算、積、商、冪といった演算は数式に近い表記で、行列やベクトルを強く意識したといった仕様になっています。この点は Numpy を取り込んだ

Python では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は基本的に同じ大きさの対象同士での成分単位の演算であること、さらに冪乗の演算子が MATLAB 系の言語で用いられる演算子 “ \wedge ” ではなく演算子 “ $\star\star$ ” を使うことです。ただし、Numpy の配列クラス ndarray を継承したクラス matrix では MATLAB 系言語と同様に、その四則演算は行列演算を主体にした演算になります。そのため Numpy であれば、その配列の成分単位の演算が中心であれば ndarray 型、行列演算が中心であれば matrix 型と使い分ける必要があります。ここでは成分単位の演算と、通常の線形代数での行列演算処理と分けて解説します。

4.6.1 MATLAB 系言語での四則演算

MATLAB 系言語ではベクトルや行列演算は線形代数の式に近い書式で四則演算が行えます。ここで注意することは積演算子 “ $*$ ”，商演算子として “ $/$ ” と “ \backslash ”，冪演算子 “ \wedge ” に対しては記号 “ $.$ ” が付随したものがあって意味が異なることです。具体的には積演算に関しては “ $*$ ” と “ $.*$ ”，商演算に関しては “ $/$ ” と “ \backslash ” に対して “ $./$ ” と “ $.\backslash$ ”，冪演算に関しては “ \wedge ” と “ $.\wedge$ ” です。これらの記号 “ $.$ ” が付随した演算子、つまり、dot 演算子はベクトルや行列の成分単位の演算を行うことを意味します。なお、二つの商演算子 “ $/$ ”，“ \backslash ” に対応する演算子 “ $./$ ”，“ $.\backslash$ ” は次の意味になります：

$$\begin{aligned} A \backslash B &\Leftrightarrow A^{-1} * B \\ A / B &\Leftrightarrow A * B^{-1} \\ A.\backslash B &\Leftrightarrow A^{-1} .* B \\ A./B &\Leftrightarrow A.*B^{-1} \end{aligned}$$

なお、行列 A の逆行列は函数 inv() を使って ‘inv(A)’ でも計算できます。ただし、逆行列は正則行列のみ計算可能です。では、これらの演算の実例を示しておきましょう：

—> a=[1,2;3,1]

a =

1. 2.
3. 1.

—> b=[1;0]

b =

1.
0.

—> a*b

ans =

1.

3.

—> c = [2, 3; 1, 2]

c =

2. 3.

1. 2.

—> d = a/c

d =

0. 1.

5. -7.

—> d*c - a

ans =

0. 0.

8.882D-16 0.

—> d1 = c\ a —> d1 = c\ a

d1 =

-7. 1.

5. 0.

—> c*d1 - a

ans =

0. 0.

0. 0.

このように積演算子 “*” と商演算子 “/” 等の通常の行列やベクトルの演算子として使えます。次に成分単位の積、商や冪乗を行う “.*”, “./” と “.^” の実例を示しておきましょう：

-->[1 2 3].*[3,2,1]

```

ans  =
3.      4.      3.
-->[1 2 3 ]./[3 ,2 ,1]
ans  =
0.3333333      1.      3.

-->[1,2,3;3 2 1]./[1 ,2 ,3;3 ,2 ,1]
ans  =
1.      1.      1.
1.      1.      1.

-->[1,2,3;3 2 1].*[1 ,2 ,3;3 ,2 ,1]
ans  =
1.      4.      9.
9.      4.      1.

-->[1,2,3;3 2 1].^ [4 ,5 ,6;3 ,2 ,1]
ans  =
1.      32.      729.
27.      4.      1.

-->[1,2,3;3 2 1].^2
ans  =
1.      4.      9.
9.      4.      1.

```

上の二つの計算では行ベクトル同士、下の二つの計算では行列同士の成分単位の演算を行っています。このような成分単位の演算では、二つの被演算子の大きさが一致しなければなりません。これは成分単位の演算になる和演算子“+”や差演算子“-”と同様です。ところで、この成分単位の演算やベクトル/行列演算で、これらの演算に適合する被演算子になるように成分の抜き出しを行うことは一向に構いません。以下にその例を示しておきましょう：

```

-->a=[1:10];
-->b=[1:10;10:-1:1];

```

```

-->a(4:8).*b(1,1:5)
ans =
4.      10.      18.      28.      40.

-->a(4:8)./b(1,1:5)
ans =
4.      2.5      2.      1.75      1.6

-->a(4:8).^b(1,1:5)
ans =
4.      25.      216.      2401.      32768.

-->a(4:8).^2
ans =
16.      25.      36.      49.      64.

-->1./b(1,1:5)
ans =
0.0181818
0.0363636
0.0545455
0.0727273
0.0909091

```

この例では行列 a から 4 列目から 8 列目の 5 成分の行ベクトル ‘[4,5,6,7,8]’ と行列 b から 1 行目の 1 列目から 5 列目の成分で構成される行ベクトル ‘[1,2,3,4,5]’ を取り出して、それらの成分毎の積を計算しています。このように成分単位の演算 “.*”, “./”, “.\”, “.^” はどちらか一方が定数の場合か、両方とも同じ大きさの行列（部分行列も含む）に対してのみ行えます。

4.6.2 成分単位の演算

MATLAB 系言語で成分単位の演算を行う演算子として積 “.*”, 幂 “.^” と商 “./” があります。Numpy の ndarray 型の配列では成分単位であるため、このような演算子の区別はありません。通常、計算機では和演算が最も高速で、それから減算、積、商、幂の順番になります。このことを Scilab で確認しておきましょう：

```
→ timer(); z=a.^12; timer()
ans =
0.00009

→ timer(); z=exp(12*log(a)); timer()
ans =
0.000091
```

ここでの結果はおよその傾向を示すものと考えてください。これは MATLAB 系の言語は函数を必要に応じて読み込む性格があり、また、ベンチマークの性質として、キャッシュなどに残ったデータの影響があるためです。しかし、これらの結果から処理の早い順に並べると、積 “*”，幕 “^”，log や exp を含む函数による処理の順になり、幕が大きくなると幕演算と函数による処理の差が縮まります。つまり、MATLAB 系言語では積で記述可能な箇所は積で記述すべきであり、場合によっては函数をうまく使うことも考えるべきと言えます。

Python の Numpy での演算は二種類あります。より正確には、構築子 array() で生成した ndarray 型の配列での演算が成分単位で、構築子 matrix() で配列を初期化したもののが行列演算になります。具体的な例として SageMath 上で Numpy の数値配列と SageMath の行列で同様の計算をさせてみましょう：

```
sage: eps=np.finfo(float).eps
sage: b=10*np.abs(np.random.randn(1000,1))+eps
sage: M=[]
....: t1=time.clock();z=b*b;t2=time.clock()-t1;M.append(t2)
....: t1=time.clock();z=b**2;t2=time.clock()-t1;M.append(t2)
....: t1=time.clock();z=np.exp(2*np.log(b));t2=time.clock()-t1;M.append(t2)
....: t1=time.clock();z=b*b*b;t2=time.clock()-t1;M.append(t2)
....: t1=time.clock();z=b**3;t2=time.clock()-t1;M.append(t2)
....: t1=time.clock();z=np.exp(3*np.log(b));t2=time.clock()-t1;M.append(t2)
....: t1=time.clock();z=b*b*b*b*b*b*b*b*b*b;b=t2=time.clock()-t1;M.append(t2)
....: t1=time.clock();z=b**12;t2=time.clock()-t1;M.append(t2)
....: t1=time.clock();z=np.exp(12*np.log(b));t2=time.clock()-t1;M.append(t2)
....:
sage: M

sage: type(b)
<type 'numpy.ndarray'>
```

この例では正規分布にしたがう 1000 成分の乱数配列を `random.randn()` で生成していますが、この構築子で生成される数値配列は `ndarray` 型です。そのために四則演算や函数演算が成分単位の計算になっています。ここでの結果はおおよそ MATLAB 系言語と似た傾向がじょうじています。つまり、積演算子 “`”` と比較して幕演算は格段に遅く、3 乗以上になると函数計算と大差なく、このことから加減算と乗算を主体にした計算式で計算を行うべきであり、幕演算は使わない方が良いことが判ります。

MATLAB 系の言語であれば演算子で行列演算と成分単位の演算に切り替えられますが、Python の Numpy では ndarray は成分単位、matrix は行列演算と切り分けられています。では、SageMath で行列を生成し、その行列に対して成分単位の演算処理を行うとどうなるでしょうか？SageMathCloud の行列で計算した結果を示しておきましょう：

成分単位の演算で Numpy を使った処理と比較すると劇的に遅っています。これの配列単位の計算がライブラリを直に使った処理であるのに對して，SageMath

成分単位の演算で Numpy を使った処理と比較すると劇的に遅っています。これは Numpy の配列単位の計算がライブラリを直に使った処理であるのに対して、SageMath の行列で

成分単位の演算をメソッド経由で行っているためです。ただし、函数 `random_matrix()` で生成した行列は実数環 \mathbb{R} の行列です。したがって、`ndarray` 型の配列を構築子 `matrix()` で変換したものの方が処理速度が良いかもしれません。実際に試して見ましょう：

多少の差はあるにしても, ndarray 型の処理のほうが格段に早いことに違いはありません。このように Python では見てくれが同じでもオブジェクトとして異なってしまえば、その処理の過程も異なるために処理速度にも大きな違いが生じます。このようなこと態は数値行列データが主体の MATLAB 系の言語では生じません。逆に言えば、Python では計算目的と計算コストの双方を考慮した上で最適なオブジェクトの型を選択することができるだけの自由があり、その自由を享受するためにはそれ相応の知識が必要になることを意味します。

4.6.3 配列のパターンマッチング処理

MATLAB 系言語での処理について

MATLAB 系の言語ではベクトルや行列の成分に対して大小関係の判断が容易に行えます。つまり、あるスカラー値に対して各成分の大小関係の判断を for 文等の反復処理を行わずに ‘ $A > 0$ ’, ‘ $A == 0$ ’ や ‘ $A < 0$ ’ のようにベクトルや行列単位で行うことができます。この処理を上手く使うことで、たとえば行列 A の成分が 0 より大であれば 128 を設定し、0 よりも小さいときに -3 を割当てる処理も ‘ $(A > 0) * 128 + (A < 0) * (-3)$ ’ と行列の式に置換えられ、和と積と判断のみの式であるために MATLAB 系言語の行列処理の高速性が損なわれません。この点は Numpy でも同様で、行列処理が必要でなければ ndarray 型で処理を進めた方が処理速度で有利です。

この機能を利用すれば、与えられたベクトルから適合するものがあるかどうか検証することもできます。以下の例では与えられたベクトルから 2 に等しいものがあるか検証し、その場所を函数 find() を用いて探す処理を行っている。Scilab では以下に示す様に、‘ $x == 2$ ’ の結果として表われる Boolean は T と F の値を持っている。Boolean を数値に変換すると T が 1, F に 0 が各々対応します。Octave と MATLAB では True の意味で 1, False の意味で 0 と直接数値が返されます。

```
—>x=[1:5,5:-1:1]
x =
1.    2.    3.    4.    5.    4.    3.    2.    1.

—>x==2
ans =
F T F F F F F F T F

—>y=find(x==2)
y =
2.    9.

—>x(y)
ans =
2.    2.

—>
```

上記の例では 2 と等しいものを函数 find() を使って検出していますが、C と比べても非常に簡単な処理で 2 と等しい成分の位置を見付け出しています。この検出は等号だけではなく、不等号に対しても用いることができます：

```
—>x=[1:5,5:-1:1]
x =
1.    2.    3.    4.    5.    4.    3.    2.    1.
```

```
—>x>3
ans =
F F F T T T T F F F
```

```
—>y=find(x>3)
y =
4.    5.    6.    7.
```

```
—>x(y)
ans =
4.    5.    5.    4.
```

```
—>y=x(x>3)
y =
4.    5.    5.    4.
```

```
—>x.*y
ans =
0.    0.    0.    4.    5.    5.    4.    0.    0.    0.
```

この函数 find() は与えられた行列で 0 と異なる成分位置を返す函数です。

```
—>aa=rand(5,5);
```

```
—>bb=aa>0.5
bb =

```

```
F T T F F
T T T F T
```

```

F T T F F
F T F T F
T F T T F

—>aa(bb)
ans  =

0.7560439
0.6653811
0.6283918
0.8497452
0.6857310
0.8782165
0.5608486
0.6623569
0.7263507
0.5442573
0.8833888
0.6525135
0.9329616

—>aa
aa  =

0.2113249   0.6283918   0.5608486   0.2320748   0.3076091
0.7560439   0.8497452   0.6623569   0.2312237   0.9329616
0.0002211   0.6857310   0.7263507   0.2164633   0.2146008
0.3303271   0.8782165   0.1985144   0.8833888   0.312642
0.6653811   0.0683740   0.5442573   0.6525135   0.3616361

```

MATLAB クローンでは ‘ $y=x(x > 3)$ ’ のように函数 `find()` を利用せずに処理することもできます。このパターンマッチングを適用して処理の簡略化が行えます。たとえば、上記の与えられたベクトルから 3 よりも大きな数値に対してのみ 2 倍するといった計算が次で行えます：

```

—>x=[1:5,5:-1:1]
x  =

1.      2.      3.      4.      5.      4.      3.      2.      1.

—>y=find(x>3)
y  =

4.      5.      6.      7.

```

```
—>for i=x(y)  
--->2*i  
—>end  
ans =
```

```
8.  
ans =  
  
10.  
ans =  
  
10.  
ans =  
  
8.
```

ところで MATLAB 系言語で for 文といった処理言語側で反復処理等を利用することはあまり薦められることではありません。むしろ、次の方法で処理する方が美しくて処理も速くなります：

```
—>z=2*x(find(x>3))  
z =  
  
8.      10.      10.      8.  
—>2*x.* (x>3)  
ans =  
  
0.      0.      0.      8.      10.      10.      8.      0.      0.      0.  
  
—>z = 0;  
  
—>tmp=find(x>3);  
  
—>z(tmp)=2*x(tmp)  
z =  
  
0.      0.      0.      8.      10.      10.      8.  
  
—>z(tmp)=2*x(x>3)  
  
0.      0.      0.      8.      10.      10.      8.
```

```
—>z(x>3)=2*x(x>3)
z =
0.      0.      0.      8.      10.     10.     8.
```

この例では同じ処理を行っているが、最初の例では3以上の成分のみを2倍にした計算であるのに対し、下の3個の例では配列全体の処理となっています。ここでのパターンマッチングではT, Fを成分とする行列が得られます。このBooleanはそれぞれ整数値の1と0と等価であるために‘ $2*x.(x>3)$ ’のような計算も行えます。次に、‘ $2*x(x>3)$ ’では成分の指定を行っていないませんが、一番下の例では成分の指定も入れています。この一番下の例を応用すると、より複雑な入換え処理も非常に容易に行えます。

Numpyでの処理

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.          ,  0.          ,
        0.          ]])
```

```
[ 0.          ,  0.          ,  0.          ,  13.88220513, -3.  
])
```

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ありません。だからといって SageMath を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単純な Python の拡張であるとは言い切れません。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 の両者に微妙な違いが生じます。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、FORTRAN 風の演算子 “**” が冪乗の演算子として用いられています。ところで数式を扱う分野で幅広く利用されている組版ソフト TeX では演算子 “ \wedge ” が冪演算子として用いられ、たとえば、多項式 x^2 は TeX で ‘ x^2 ’ と記述されます。そのこともあって、多くの数式処理では冪乗として演算子 “ \wedge ” が用いられ、SageMath でも数式全般の冪乗の演算子として演算子 “ \wedge ” が用いられています。だからといって SageMath で演算子 “ \wedge ” が冪乗の演算子として全面的に置き換えられているという訳ではありません。実際、SageMath 上で int 型や float 型の数に演算子 “ \wedge ” を使うと従来の論理排他和になるためです。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 のそれとは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 がどのような意味付けをされた対象であるかを調べてみましょう^{*1}。そのためには函数 type() の引数にオブジェクトと参照関係にある名前を指定すれば十分です:

```
sage: type(1)
<type 'sage.rings.integer.Integer'>
sage: type(1.0)
<type 'sage.rings.real_mpfr.RealLiteral'>
```

このように SageMath で対象 1 は sage.rings.integer.Integer, 1.0 は sage.rings.real_mpfr.RealLiteral のインスタンスで、名前から判断できるように共に SageMath のクラスであることが判ります。また、“rings” との言葉から判るように、数学的構造に対応するオブジェクトになっています。ちなみに IPython で同じことを実行した結果を参考までに示しておきましょう:

```
In [1]: type(1)
```

^{*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。

```
Out[1]: <type 'int'>  
  
In [2]: type(1.0)  
Out[2]: <type 'float'>
```

今度は対象 1 は int 型で対象 1.0 は float 型であると返しており、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おり、ここに数学的構造は特に見られません。この観察から数を表現するオブジェクトが SageMath では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について、それぞれが属するクラスが異なることは同一の処理を行うメソッドを双方に用意しなければ同等の処理が行えないことを意味します。このように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意味不明な現象に陥ります。つまり、SageMath で入力した数リテラル ‘8’ が代数的数の 8 であるのに対し、プログラム内に記載した数リテラル ‘8’ が Python の int 型のオブジェクトであるために代数的数 8 に用意された SageMath のメソッドが使えないという現象です。したがって、処理すべきオブジェクトが「何であるか」、そして「何であるべきか」を常に意識していかなければなりません。

さて、ここで気になることは 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 での数の構成を含めて 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目的にしています。

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

凡そ計算機の CPU には整数や実数といった数の CPU 内部での表現と算術演算は最初から実装されています^{*2}。したがって、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ゲの「算術の基礎」や集合論のように全てを構成する必要がなく、ハードウェアに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

^{*2} 実数を近似する浮動小数点数はその演算も含めて IEEE-754 で規格化され、それらの数の表現に対する基本的な算術演算は CPU の命令セット (instruction set) に包含されています。

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成員、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。そうとなれば、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、すなわち、これらの数オブジェクトの間に継承関係を導入することで、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」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、Python の函数 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

*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>

```
/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 での函数 help() の結果を示していますが、モジュール 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 (親と子) 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。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。だから、モジュール numbers がどのように定義されているかを知りたければ、ここに記載されたファイル numbers.py の中身を見てしまえば良いのです。では、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

```

このクラス 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を持つものであるかを指定することではありません。そこで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

@property
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 例外を送出する 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に記載されます。

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。そこで、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。まず、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」の上側から順に定義し、演算も順次拡張することで数値塔の継承順序と逆の構成順序になります。しかし、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ればどうでしょうか？まず、複素数であれば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行えますが、実数はどうでしょうか？複素数を継承していれば演算は可能ですが、双方が実数の場合、虚部が $0 \times i$ の形式で現れる可能性があります。その意味で、双方が実数であれば実部のみを返すと演算を書き直すことになるでしょう。以降、同様のことが有理数と整数でも生じ、同じクラスのオブジェクトの処理を再定義することになります。このように数値塔は土台になる数と演算の存在を前提にした数の実装方法です。

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るといえます。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発揮します。

5.3 SageMath の数の構成

5.3.1 SageMath の数

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継承しません。この理由として、

- 整数が 32bit 整数、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。
- 算術演算の実装がそもそも効率が良くない。

が挙げられます。これらの問題に対処するために SageMath では GMP(GNU MP, GNU Multi Precision Arithmetic Library) で整数、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}。そのために SageMath で式を構成する数は SageMath の数オブジェクトが用いられます。特に整数と有理数に関して 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 for 文や while 文でカウンターとして使っている整数が Python の数オブジェクトになることを意味します。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、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っ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 が提供する精度以上の必要とされる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 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

^{*5} GMP を利用しているものに商用の数式処理 *Mathematica* があります。

リコードが含まれています。

SageMath の数の構成で利用される GMP のクラスを以下に示します:

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。名前が <code>mpz_</code> で始まる 150 程度の函数を含む。
<code>mpq_t</code>	有理数とその演算。名前が <code>mpq_</code> で始まる 35 程度の函数を含む。
<code>mpf_t</code>	浮動種数点数とその演算。名前が <code>mpf_</code> で始まる 70 程度の函数を含む

なお、有理数は整数の対として表現されるために、ここで挙げた 35 程度の函数に加えて整数演算の函数が必要になります。

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。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に用いられます。これらのファイルを基に Cython のコンパイラが C のコードに変換^{*6}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ることになります。

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

^{*6} ソースの構文解析は Pyrex で行います。

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:  
    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することが可能です。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

有理数の実装も、GMP の mpq_t 型を Cython でクラス Rational として取り込んだものになっています。この Raional には 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

実数は計算機で近似される数です。この近似では浮動小数点数と呼ばれる書式で表現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浮動小数点数が用意され

ていますが、Python に用意されている浮動小数点数は倍精度のみです。SageMath では GMP 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することができる。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である一方で、倍精度の浮動小数点数であれば CPU で直接処理が可能で、単純な計算速度の面では任意精度よりも倍精度の浮動小数点数の処理が有利です。そのために倍精度以上の精度で浮動小数点数を扱うことは、対象の特性を考慮した上で判断する必要があります。また、MATLAB 等の数値行列処理系も倍精度の浮動小数点数を扱うことから、SageMath では線形代数、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。SageMath の倍精度浮動小数点数クラスは `RealDoubleField_class`、略して `RDF` です。また、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。

5.3.7 複素数の実装

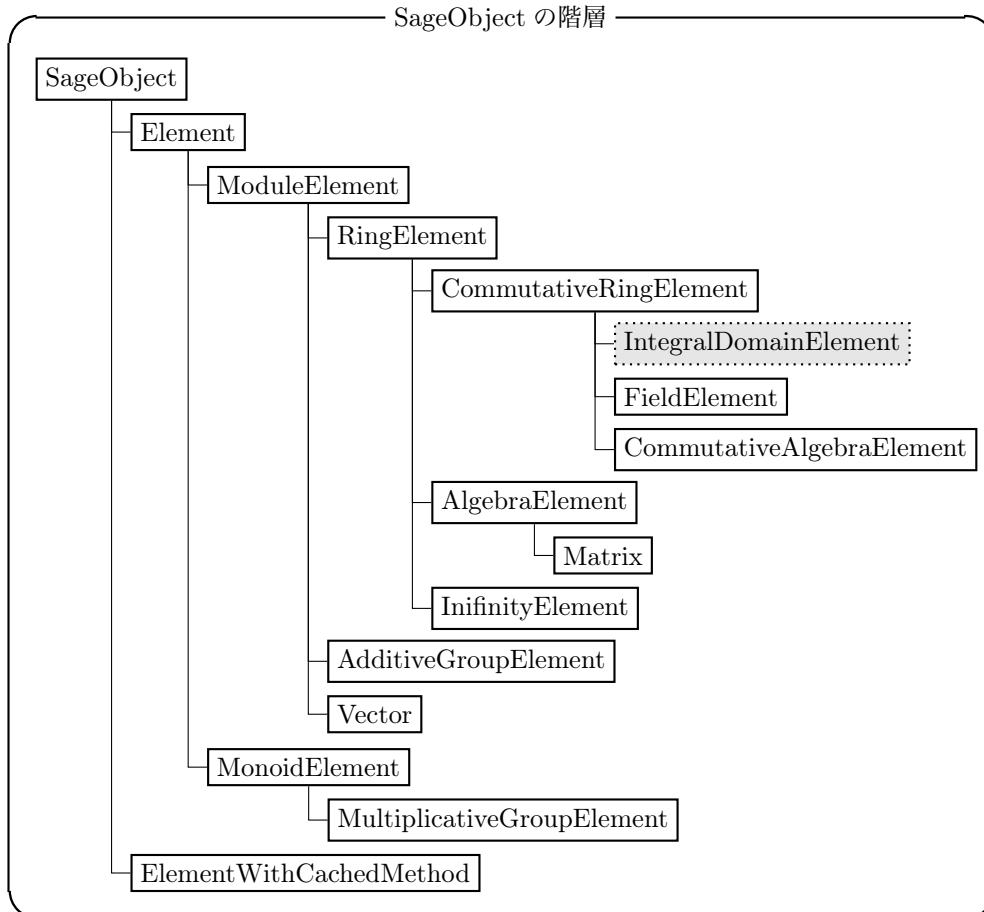
複素数 \mathbb{C} は実部と虚部に別れることから実数対 $\mathbb{R} \times \mathbb{R}$ と一対一対応があることが別ります。ここで SageMath で実数は浮動小数点数で近似されることから、複素数の実装も同様に倍精度と任意精度の浮動小数点数の二種類の対象で近似されることが判ります。ここで倍精度浮動小数点数で近似した複素数が `ComplexDoubleField_class` であり、任意精度の浮動小数点数で近似した複素数がクラス `ComplexNumber` になります。

5.4 SageMath のオブジェクト

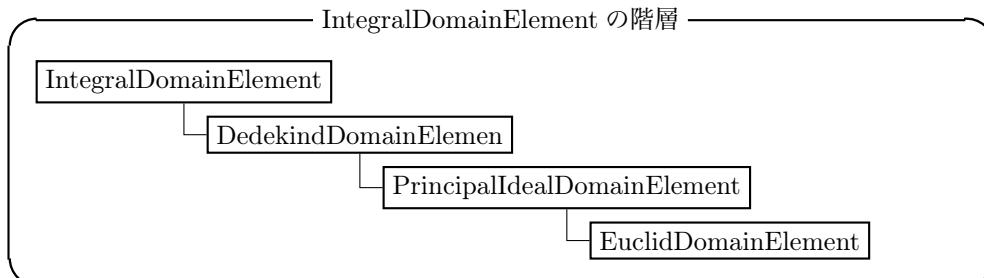
5.4.1 SageMath の抽象基底クラス

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 “anstract 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:



次に煩雑になるために省略したクラス 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

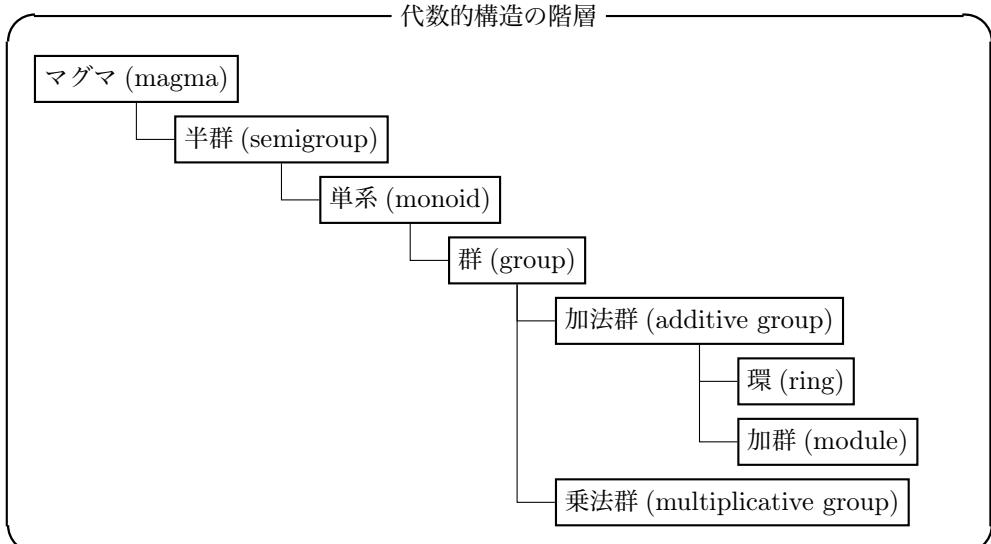
ありません。あくまでも抽象基底クラスの継承関係を図示したものです。そしてクラス Element の継承関係は数学的な構造が反映されたものになっていますが、数学的な厳密さ以上に実装が容易になるように継承関係が導入されています。

5.5 SageObject の各階層について

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場合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 の子クラスとして定義された特殊メソッドを用いています。

ここで代数的構造の階層を以下に示しておきましょう：



SageMath ではこの代数的構造がそのまま SageObject の階層構造に対応しません。むしろ、SageObject の階層はどちらかと言えば定義のし易さが影響します。そのため Element の直下にマグマに対応する抽象基底クラスがある訳ではなく、抽象基底クラス Element の直下に環上の加群に対応する ModulElement と单系に対応する MonoidElement が配置されていることから明瞭です。なぜなら、加群は演算子 “+” と演算子 “*” の二つの演算子を必要としますが、单系では一つだけの演算子、この場合は積演算子 “*” の存在を前提にしているためです。その結果、ModuleElement にメソッドとし

て積のメソッドと和のメソッドの二つがあらかじめ準備され, MonoidElement に積のメソッドのみが準備され, その結果, 乗法群の抽象基底クラス MultiplicativeGroupElement が MonoidElement の直下に置かれ, 和演算子 “+” を必要とする加法群の抽象基底クラス AdditiveGroupElement が ModuleElement の直下に置かれることになります.

このように SageMath の抽象基底クラスと通常の数学的構造の階層との違いについて述べましたが, 結局のところ, SageObject ではメソッドに重点が置かれ, 数学的構造では数学的対象に重点が置かれたために生じたことです. 結局のところ, オブジェクト指向プログラミングは対象の間との対応関係こそが主軸であり, このことは圏論で対象そのものよりも, 対象間に存在する矢, すなわち, 関係を重視する姿勢に対応します.

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になります. このクラスのメソッドに計算結果をアスキ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.

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) と单系 (モノイド, Monoid) に対応する抽象基底クラスになります.

■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.

■MonoidElement: Element を継承する单系 (モノイド, Monoid) の抽象基底クラスです. ここで单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つ数学的対象です。ここで集合 R が環であるとは次の条件を充たす場合です:

■**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、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 “*” の可換性があります。

■**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ということではなく、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可換になります。

■**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のことです。

■**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 $f(r) < f(a)$ 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

SQLiteについて

6.1 SQLite 速習

6.1.1 SQLite 概要

SQLite は「リレーションナルデータベース管理システム (RDBMS)」の一つで、その特徴として C で記述されてサイズも軽量で動作も軽快であること、ソフトウェアパッケージとしても他のパッケージに無依存であること、利用する際には専用のサーバーを必要とせず、前もってさまざまな設定を行わずに手軽に使えること、これらの手軽さに加えて言語的に「ACID」^{*1}対応であることと SQL92 版 (SQL2) に準拠しているという特徴があります。SQLite はその名前からも判るように PostgreSQL や MySQL で構築するような大規模なシステムを組むことに不向きですが、軽量で、事前設定（サーバーの設定、ユーザの設定等々）が不要でただちに利用できるという性質から小規模なシステムに向いています。この章の目的は SQL について軽くおさらいを行い、SQLlite の Sage での簡単な利用についてその概要を述べることで、より具体的な事例はのちの章で述べることにします。

6.1.2 SQLについて

データベースシステムはデータを計算機に蓄え、必要に応じてデータを引っ張り出したり更新したりすることができるシステムのことです。だから Excel シートにデータを記載してどこかのフォルダに保存しておくのも^{*2}データベース (DB) と言えなくもないでしょうが、システムと呼ぶには人手に頼り過ぎています。ここで解説する SQLite は RDBMS と呼ばれるデータベースシステム (DBS) ですが、ここでの RDB、すなわちリレーションナルデータベース (Relational Database System) とは、先頭の「リレーションナル」という言葉から見当がつくようにデータに関する「関係 (relation)」を考慮した DB です。より正確には「関係モデル」に基づく DB で、あらゆるデータは n 項の関係として表現されます。このときに「表 (テーブル, table)」は関係を 2 次元で可視化したもので、見方を変えると x と y の関係は x が y で述語付けられていると解釈できます。このことから集合に対する演算によって適合したデータを取り出す処理が行えます。また DB の検索、データの追加・削除や更新は「SQL」と呼ばれる言語が用いられます。この SQL という名前は IBM が開発した RDBM の実証機 System R の操作言語「SEQUEL(Structured English Query Language)」に由来し、その名前が示すように DB に対して「問い合わせ」を行い、それに応じた処理を行うための構造化された言語です。この DB への問い合わせでは、述語からそれに適合する外延を抽出するもので、

^{*1} Atomicity(不可分性), Consistency(整合性), Isolation(隔離性), Durability(持続性) の略語で、トランザクションシステムが持つべき性質としてジム・グレイが定義したものです。

^{*2} どこかの出来の悪い自称 IT 企業のように！

この抽出では「**関係代数 (relational algebra)**」や「**関係論理 (relational calculus)**」に基づいて処理が行われます。なお、SQLについては完全な形で関係論理を実装していないという批判もありますが、このSQLはDBへの問い合わせを行うだけの言語ではなく、「**表の構造 (schema)**」の決定や検索結果の表示方法に至るまでのDBの管理が行えるようになっています。また、計算機言語としてのSQLは各RDBMS毎に拡張が行われていましたが、近年はANSIやISOで言語仕様の規格化が行われており、SQLiteは1992年の規格化であるSQL92(SQL2)に準拠しています。

SQLでは命令は大文字でも小文字で記載しても構いませんが、通常は読み易さのためにSQLの構文は大文字を用いて記載されています。この章でも構文として記載するときは大文字で記載し、他と区別し易いようにしておきます。

6.1.3 SQL の文法について

SQLの構文はデータ定義言語 (DDL: Data Definition Language), データ操作言語 (DML: Data Manipulation Language), データ制御言語 (DCL: Data Control Language) の三種類に大きく分けられます。

■DDL: DDLの文はRDBの構造、具体的には表の組(行), 属性(列), 関係(表), 索引といったDB固有の構造を定義、あるいは表の削除を行います:

データ定義言語 (DDL) の主要な命令

CREATE	DBの対象(表, 索引等)の定義
DROP	既存のDBの対象の削除
ALTER	既存のDBの対象の定義変更
TRUNCATE	表からの不可逆的な削除

■DML: DMLの文はRDBに対して登録された対象の検索、削除、更新や対象のDBへの登録を行います:

データ操作言語 (DML) の主な命令

INSERT	データの挿入
UPDATE	表の更新
DELETE	表から指定した行を削除
SELECT	表データから指定した条件に合致するものの抽出

■DCL: DCLの文は利用者のDBに対するアクセス制御を行うことを目的とします:

 データ制御言語 (DCL) の主な命令

GRANT	指定した利用者に DB への特定の作業権限を与える
REVOKE	指定した利用者から特定の作業権限の剥奪を行う
BEGIN	トランザクションの開始
COMMIT	トランザクションの確定
ROLLBACK	トランザクションの取消
SAVEPOINT	ロールバック地点の設定
LOCK	表等の資源を占有

■カーソル (cursor): DB 検索結果の集合は表として表現することができます。その表に対して表の行位置 (ポインタ) を指示するものがカーソルになります。また、機能としてはカーソルは検索条件とその結果集合に対応する表の行位置を保持するものと言えます。SQL にはカーソルに対して次の命令があります:

 カーソルの主な命令

DECLARE CURSOR	カーソル定義
OPEN	カーソルのオープン
FETCH	行データを取得
UPDATE	行データの更新
DELETE	行データの削除
CLOSE	カーソルのクローズ

ここで FETCH, UPDATE, DELETE はその時点の行位置に対して処理を行い、行位置の変更はありませんが、FETCH の場合は行位置からデータを取り出すと行位置を一つ進める処理を行います。このように処理自体は読み書きを伴うファイル処理に類似しています。使い方は

1. 検索条件を指定してカーソルを定義。
2. カーソルを開く。
3. 検索結果に対する処理。
4. 項目の追加等の表の変更処理。
5. 表に変更を行った場合は確定処理。
6. カーソルを閉じる。

の手順になります。

6.1.4 スキーマ (schema)

DB の構造は「**スキーマ (schema)**」で決定されます。RDB の場合は表(関係)と表内部の属性等の定義です。このスキーマを次の概念、論理、物理の三層に分けて説明することができます：

概念-論理-物理によるスキーマの分類

-
- ・ 概念スキーマ 概念間の関係を定義
 - ・ 論理スキーマ 実体とその属性、および実体間の関係を定義
 - ・ 物理スキーマ 論理スキーマの実装
-

ちなみに SQLite3 では DB のスキーマは `sqlite_master` という特殊な表にスキーマが記録されています。この `sqlite_master` は DB 単位で一つだけ生成され、表、索引、ビューとトリガーの情報が記録されます。この表の構造は次の通りです：

SQLITE_MASTER の構造

type	オブジェクトの型、table, index, view, trigger の何れか
name	オブジェクトの名前
tbl_name	
rootpage	
sql	オブジェクト生成で用いられた SQL 文

ここで `table`, `index`, `view` と `trigger` は何れも CREATE 文で生成されるオブジェクトです。この表の `sql` にはそのオブジェクトの生成で実行された SQL 文が保存されているため、DB に含まれる表の情報を入手したければ

```
SELECT name FROM sqlite\_{}master WHERE type='table';
```

によって表の名前が入手可能で、表の構造は

```
SELECT name FROM sqlite\_{}master WHERE type='sql';
```

によって入手できます*3。

*3 SQLite3 の命令に `.table` や `.schema` があり、これらで内容を調べることも可能ですが、Python から SQLite3 を使うときにこれらの命令が使えないため、この `sql_master` を検索しなければなりません。

6.1.5 DB の有り難味

基本的な使い方は、最初に表 (TABLE) を生成し、以降は INSERT でデータを指定した表に追加、SELECT で条件に合致するデータを指定した表から取り出すといった処理です。と書いてしまうと Excel のような表計算ソフトウェアの表を構成してシートを管理すれば良いのではないかと思われるかもしれません。それも少々のデータで稀に個人が書き直す程度であればそれで十分でしょう。しかし、それなりの規模のデータでありながら参照する事項が僅かなことであったり、複数のシートを横断的に参照する必要がある場合、さらには複数の人間が表の管理に関わるのであれば Excel を利用することは効率が良くて安全な方法ではありません。無理に Excel シートで管理する暇があれば DB を活用すべきです^{*4}。そして Sage には SQLite が付属しているので、ちょっとした計算結果の保持で、メモを取る暇があるのであればそれを使わない手はありません。

6.2 SQLite のキーワード

SQLite では表、データベースの名前に何でも使えるとは限りません。実際、SQL 文で用いられる言葉（キーワード、予約語）は使えません^{*5} とは言え、SQL で用いられている言葉は SQL だけの特殊な言語ではないために項目名等に使わなければならぬこともあります。そのときは次の表記にしなければなりません：

キーワード

单引用符 ‘ ’で括る	'keyword'
二重引用符 " "で括る	"keyword"
括弧 [] で括る	[keyword]
引用符 ‘ ’で括る	'keyword'

ここで括弧 [] で括るのは ACCESS や SQL Server の流儀、引用符 ‘ ’で括る構文は MySQL の流儀で、SQLite では各 DB との互換性のためにこれらの流儀が使えるようになっています。

^{*4} 急いでいるときにシートを開いて「読み取り専用で開きます」で悩まされた人、表が大きくなりすぎて開くのに時間がかかり過ぎることで困った人は手を上げて！

^{*5} キーワードの詳細については SQLite Keywords: https://www.sqlite.org/lang_keywords.html を参照のこと。

6.3 SageMath から SQLite を使う

6.3.1 sqlite3 について

ここでは SageMath から SQLite を扱う実例を挙げます。SQLite 単体を立ち上げて利用する実例ではありません。Sage は Python 環境であるために SageMath から SQLite を使うというよりは Python から SQLite を扱う状況ですが、ここでは SageMath から使うと安易に記載します。なお、Python から SQLite を扱う話の詳細については ‘<http://docs.python.jp/2/library/sqlite3.html>’ も参考にして下さい。

まず Python から RDB である SQLite を利用するためには Python Database API(DB-API) インターフェイスである sqlite3 を用います。ここで Python Database API とは Python から RDB への共通のインターフェイスを策定したもので、現行の DB-API 2.0 は PEP-249 で定められています^{*6}。このように DB-API で Python からの RDB の操作手順が定められているために他の RDB についても共通する操作は同じ作法で行えます。この PEP-249 によると RDB への接続は connect(), SQL 文の実行は execute() で行ないます。より具体的には connect() が構築子であり、Connection オブジェクトを返します。この Connection オブジェクトに対しては次のメソッドがあります：

Connection オブジェクトに対するメソッド

close()	RDB への接続を切断します。
commit()	トランザクションの確定
rollback()	トランザクションの取消
cursor()	cursor オブジェクトを生成します。

RDB への処理は Connection オブジェクトを構築したのちに Cursor オブジェクトを生成し、その Cursor オブジェクトのメソッドで行います。そこで Cursor オブジェクトに対する主要なメソッドを次に示しておきます：

^{*6} PEP-248 は DB-API 1.0 と古い版

Cursor オブジェクトに対するメソッド

close()	Cursor を閉じます.
execute()	一つの SQL 文を実行します.
executemany()	
fetchone()	SELECT 等による問合の結果を一件取り出します.
fetchmany()	SELECT 等による問合の結果を取り出します.
fetchall()	SELECT 等による問合の結果を全て取り出します.
arraysize()	fetchmany() で取得する行数の指定を行います.
setinputsizes()	execute() の実行前に操作のパラメータの記憶領域の大きさを設定します.
setoutputsizes()	列のバッファの大きさを指定します.

ここで重要なメソッドは execute() と fetchall() です. メソッド execute() で SELECT 文等の問合を発行し, メソッド fetchone() やメソッド fetchall() 等でその結果を取り出す処理になります. 次に簡単な実例を見ることにしましょう.

6.3.2 sqlite3 の利用例

SQLite の利用は幾つかの方法があります. まず一つが SageMath に含まれる SQLite を直接使う方法です. もう一つは Sage を起動して SageMath から SQLite を利用する方法です. 前者の場合はあらかじめ環境設定ができていなければなりません. ただし, OSX 上で Sage.App を起動している場合は SageMath メニュから「Terminal Session」の「Misc.」から「sh」を選択することで環境設定が行われたシェルが立ち上がり, そこで‘sqlite3’と入力するだけで利用を開始することができます. このことは後述の Cloud SageMath でも同様です.

もう一つの方法は Python 向けの DB-API 2.0 インターフェイスである sqlite3 ライブライアリを SageMath 上で読み込んで, そこから利用する方法です. この場合は SageMath を立ち上げてから通常のライブラリと同様に sqlite3 ライブライアリの読み込みを行い, それから SQLite が使えます. この様子を以下に示しておきましょう:

```
sage: import sqlite3
sage: conn=sqlite3.connect("/home/yokota/TEST.db")
sage: conn.execute("create table mycat (name text, age int, weight int)")
<sqlite3.Cursor at 0x4f56180>
sage: conn.execute("insert into mycat values (?,?,?)", ('tama',int(4),int(15)))
<sqlite3.Cursor at 0x4f56420>
sage: x1 = conn.execute("select name from mycat")
```

```
sage: x1.fetchall()
[(u'tama',)]
sage: x1 = conn.execute("select * from mycat")
sage: x1.fetchall()
[(u'tama', 4, 15)]
sage:
```

まず最初に import 文で sqlite3 の読み込みを行ってい、それから関数 connect() によってディレクトリ '/home/yokota/' 上に生成したデータベース 'TEST.db' に接続するためのオブジェクト conn を生成します。ここで指定した TEST.db が指定したディレクトリ上に存在しなければ新規に作成されます。ここまで処理は PostgreSQL や MySQL といったより本格的な RDBS であればユーザーの設定等が必要になりますが、SQLite であればそういうことにお構いなしにデータベースに接続することができます。それからメソッド execute() を使って接続先のデータベースに対して SQL 文を実行させています。ここで最初の SQL 文 'create table ...' は mycat という表を生成します。この表には text 型の name, 整数型の age と weight を項目として持つことが指示されています。それから insert 文で表 mycat に値の書き込みを行いますが、このときはメソッド execute() に SQL 文を文字列で与えてしまうと引数の型が全て text 型になってしまふので引数の部分を ‘u’ で置換え、メソッド execute() の第二引数にタプル型として引き渡します。それからあとは select 文で検索を行っています。ここで文字コードが UTF-8 であるために文字列の先頭に UNICODE 文字列であることを示す記号 ‘u’ が付いていることに注目して下さい。なお、文字列として UTF-8 であることを指定せずに ASCII 文字以外の文字（たとえば日本語の文字）を入力するとエラーが発生するので注意が必要です。

このように前処理を行なうこともなく簡単に RDB の表を生成して、それを活用することが可能であるために大量の計算結果を配列等に記憶させるだけではなく、RDB の表に登録しておいて、それらの処理を行うといったことも容易に行えます。

第 7 章

結び目理論への適用

7.1 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。そのため結び目/絡み目を組紐で表現して組紐群として処理することもできますが、それでは面白くないためにガウス・コードと呼ばれるリストで結び目/絡み目を表現し、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も入れてみましょう。それから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比例してデータが増加し、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ってみましょう。

7.2 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。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)」と呼ばれる図形で, 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のない状態で 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図形です。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, 絡み目や絡み目には「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, それらに反時計回りや時計回りの二種類の向きを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が, 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。

7.3 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, -10) ]
Kb = parametric_plot(sb,(t,0,2*pi),color='blue',
    thickness=20,plot_points=200)
(T + K + Kb).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（ロンジチュード, 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（ロンジチュード）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 $Z=-10$ 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 7.1 に示すグラフです：

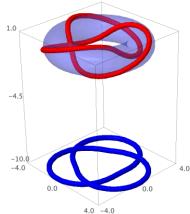


図 7.1 3 次元空間内の三葉結び目

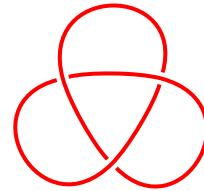


図 7.2 結び目の射影図

ところで図 7.1 のように結び目を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ような平面への射影図にすると、平面上の图形として扱えるためにより簡便になります。しかし、この射影図にはどちら側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、その各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ってしまう性質、つまり、「横断的」と呼ばれる性質が射影図の各交差点になければいけません。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があります。これら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の射影図を「正則射影図」と呼びます。一例として図 7.2 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/絡み目に制約を入れましょう。つまり、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。

7.4 結び目/絡み目の同値性

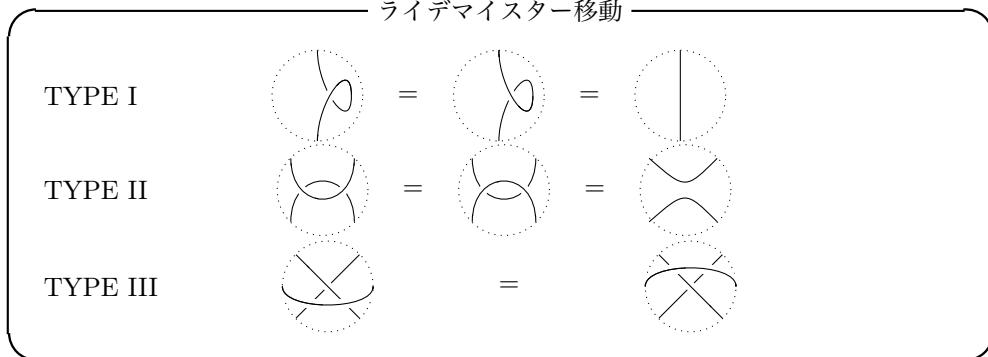
では、与えられた二つの結び目/絡み目が「同じである」とはどのような状態でしょうか？三角形であればそれが重なり合うことを示せれば十分ですが、結び目/絡み目はそんなに簡単ではありません。なお、結び目は1成分の絡み目であるために絡み目で話を進めることにします。まず第一に、一方の絡み目の成分の総数が n であればもう一方の成分の総数も n でなければなりません。それから絡み目をゴム紐でできたものと考えて3次元球面 S^3 内部で紐を切ったり、紐を交差させずに変形することで互いの絡み目に移り合えるときに同じ絡み目と呼ぶことにしましょう。このことは二つの同じ成分数 n の絡み目 L_1 と L_2 の間に「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 $F_1(L_1) = L_2$

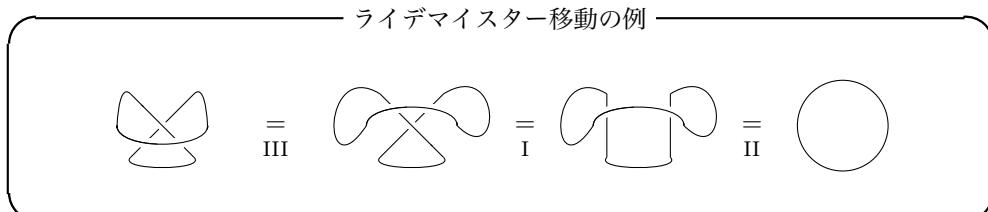
二つの同じ成分数 n の絡み目 L_1 と L_2 の間にアンビエント・イソトピーが存在するときに、これらの絡み目が「同値」と呼び $L_1 \Leftrightarrow L_2$ と表記します。このアンビエント・イソトピーが存在するということは閉区間 $[0, 1]$ を時間、時間0を現在、時間1を未来とするときに、現在 $t = 0$ で絡み目 L_1 、その絡み目 L_1 を3次元球面 S^3 内部で連続的に変形して未来 $t = 1$ で絡み目 L_2 にする映画が存在することと言い換えられます。と言ったものの、このアンビエント・イソトピーを式をいじくり倒すことで簡単に構築できるようなものではありません。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TYPE IIを複雑にしたもので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おり、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。

たとえば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)**」と呼びますが、次にライデマイスター移動で自明な結び目になる例を示しておきます:



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」ということと「ライデマイスター移動が存在しない」という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、機械的な操作で求まるものでなければなりません。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。

7.5 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。

群 $(A, *)$ の持っている「良い性質」を列記しておきましょう。まず、集合 A の任意の二つの元 a, b に対して $a * b$ は必ず集合 A の元になります。これは演算子 “ $*$ ” が集合 A の新しい元を生成する能力を持つことを示しています。この演算子の能力を「集合 A は演算 “ $*$ ” で閉じている」と呼びます。たとえば、整数の集合 \mathbb{Z} に対して演算を和 “ $+$ ” とするときに和 “ $+$ ” は \mathbb{Z} で閉じていますが、演算が商 “ $/$ ” であれば $\frac{1}{2}$ が整数にならないために演算 “ $/$ ” は閉じていません。このように二項演算は何らかの対象を表記上は生成しますが、演算が閉じていなければ生成した対象が集合 A に含まれる保証がありません。また、表記上の例として $\frac{0}{0}$ を挙げておきましょう。これと $\frac{1}{2}$ は意味が異なります。まず、 $\frac{1}{2}$ は集合を整数 \mathbb{Z} から有理数 \mathbb{Q} に拡張しさえすれば実在する数ですが、一方の $\frac{0}{0}$ は不定値と呼ばれ、一定の値を持たない表記上の存在でしかありません。

つぎの良い性質ですが、この演算 “ $*$ ” が閉じているときに集合 A の任意の 3 個の元 a, b, c に対して $(a * b) * c$ と $a * (b * c)$ は同じ集合 A の元になるでしょうか? 残念ながらこの保証もありません。この ‘ $(a * b) * c = a * (b * c)$ ’ という関係は「結合律」と呼ばれます。結合律を充てよいことは最初に $a * b$ を計算して c との演算を計算する方法である $(a * b) * c$ と $b * c$ を計算して a との演算を計算する方法である $a * (b * c)$ の両者が一致することが保証されるためです。日常の言葉では括弧 “()” 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)」[25]と解釈するかで意味が異なります^{*1}。群であればこのような式の構造上の曖昧さがなくなるために $(a * b) * c$ か $a * (b * c)$ かを悩まずに $a * b * c$ と括弧を外して表記できます。ここで集合 A が演算 “ $*$ ” に対して閉じていて結合律を充すときに「半群」、あるいは「準群」と呼びます。この「半」から予測されるように集合 A が群になるためには閉じた演算 $*$ が結合律を充たさなければなりません。では残りの半分は何でしょうか？それは集合 A の元に関わることになります。集合 A が群であれば演算 “ $*$ ” に対して殊な元があります。その一つが「単位元」で、 $u \in A$ が単位元であれば集合 A の任意の元 a に対して ' $a * u = u * a = a$ ' を充します。そして単位元は存在すれば一つだけです。なぜなら単位元として u の他に v も存在すれば $u * v$ は u が単位元であるため ' $u * v = v$ '。ところで v も単位元であるために ' $u * v = u$ '、両方を併せて ' $u = v$ ' が得られるためです。通常、単位元は 1、あるいは e や u と表記されます。

半群や群といった数学的構造を持つものは整数や行列といった数から構成された対象とは限りません。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/結び目の集合には「連結和」と呼ばれる操作があります。この連結和という操作は二つの結び目(絡み目であればその一つの成分)の紐でまっすぐな箇所を選び、その箇所を取り外して二つの絡み目/絡み目を繋ぎ合わせて新しい絡み目/結び目を作るという操作です。このときの演算子は“#”を用います。

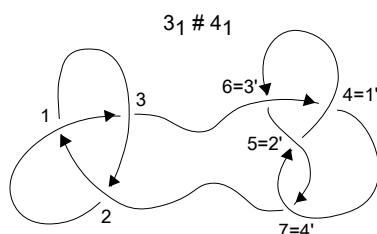


図 7.3 連結和の例

図 7.3 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。また、自明な結び目が連結和の単位元 0 になります。このことから(向き付けられた絡み目/結び目, #)は単位元 0 を持つ半群になることが判ります。また、「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)」と呼びますが、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

^{*1} 「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ぬ文の構造のためです。

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 のときに限られます. なお, 連結和で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ることが知られており, 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.

このように半群に漂う半端感の原因の「半」を外して群になるために必要な条件は前述の単位元の存在に加えて逆元の存在が必要になります. ここで集合 A の元 b が集合 A の元 a の「**逆元**」になるのは半群 $(A, *)$ の単位元 u に対して ' $a * b = b * a = u$ ' を充すときです. この元 b を a^{-1} と表記^{*2}し, 逆元を持つ元 a のことを「**正則元**」と呼びます. 単位元を持つ半群 $(A, *)$ が晴れて群になるためには集合 A の全ての元が演算 “ $*$ ” に対して逆元を持つこと, すなわち, 集合 A の元が全て正則元でなければなりません. さきほど結び目の連結和 “ $\#$ ” は全ての結び目が正則元にならないために結び目は演算が連結和 “ $\#$ ” のときに単位元を持つ半群であっても群になりません.

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:

任意の $a \in A$ に対し, $a * b = b * a = 1$ を充す $b \in A$ が存在する.

ここで最初の二つの条件が半群の条件です. 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. ここで (結び目, $\#$) は单系の例になります. 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

^{*2} 演算子 “+” のときは $-a$ と表記します.

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.

これで群がどのようなものであるか定義ができました. では群がどのようなものであるかを具体的に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**」というものがあります. 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 \emptyset 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 $W_1 W_2 W_3$ 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^{*3}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に $\langle a, \dots, z \rangle$ 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, ここでの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同様に群になるので, 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, その結果, もとの語とは別の語ができるため自由群は通常は非可換群です. 実際, 最初の語の例で ‘inu’ と ‘uni’ は別物です. ただし, 生成元が一つだけのときのみ可換群になります. 実際, 一成分 a だけであれば, 結合律から $a^m * a^n = \underbrace{a * \dots * a}_m * \underbrace{a * \dots * a}_n$, 結合律によって括弧 “()” の付け替え操作が可能であるために $\underbrace{(a * \dots * a)}_n * \underbrace{(a * \dots * a)}_m$ であることが判ります. このことで可換性: $a^m * a^n = a^n * a^m$ が証明できます.

^{*3} 「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.

なお、自由群は繰々とその元を演算“*”を使って生成できる群ですが、全ての群がそのようなものではありません。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」があることが判ります。このように群の持つ規則はその群の元で構成される式、すなわち、「関係式」で表現されます。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：

$$\boxed{\begin{array}{c} \text{群の表示} \\ \hline \langle a_1, \dots, a_n | r_1, \dots, r_m \rangle \end{array}}$$

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。スイッチの例では関係式が $a^2 = 1$ のために $\langle a | a^2 \rangle$ という群の表示にします。そのような 1 に等しくなる式 r_1, \dots, r_m のことを「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：

$$\begin{array}{ccc} f & : & \langle a_1, \dots, a_m \rangle \rightarrow \langle a_1, \dots, a_m | r_1, \dots, r_n \rangle \\ & & \Downarrow \qquad \qquad \Downarrow \\ a_i & \mapsto & a_i \qquad \qquad i \in \{1, \dots, m\} \end{array}$$

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。

7.6 結び目/絡み目を表現する群

7.6.1 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。

7.6.2 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。

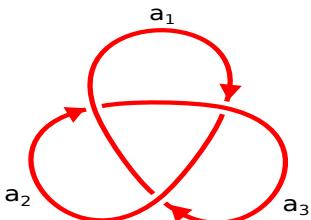


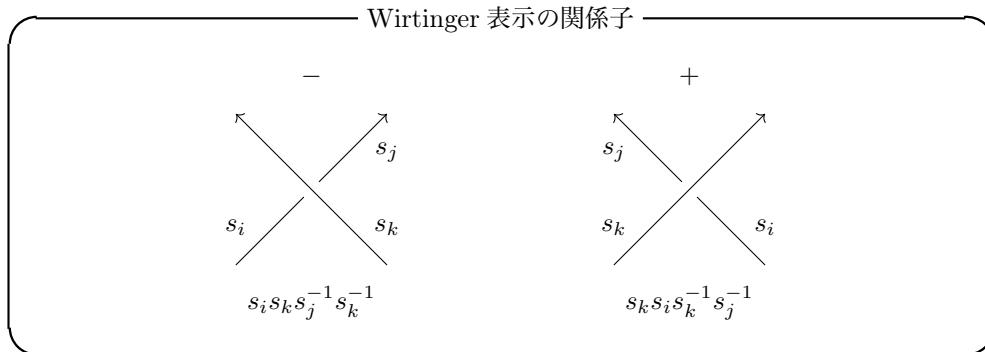
図 7.4 向き付けられた正則射影図

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 7.4 に示しておきます。ここで各交差点での交差の状況が基本群の関係子を決定しますが、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**」の二種類の群の表示方法に関連した手順があり、ここでは Wirtinger 表記による群の構成を解説します。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：

—— 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。

7.6.3 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：

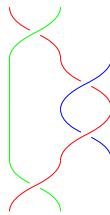
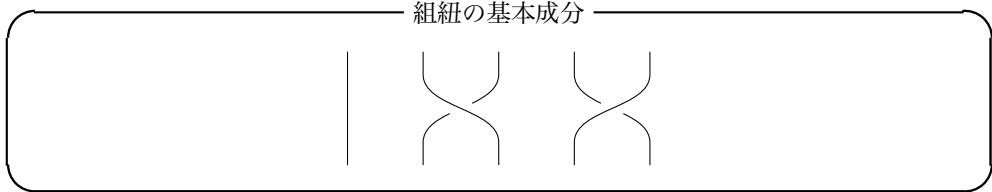
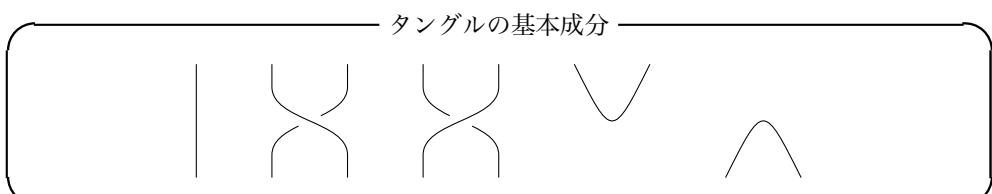


図 7.5 組紐の例

組紐の一例として図 7.5 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 [33]。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：



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。

7.6.4 組紐群

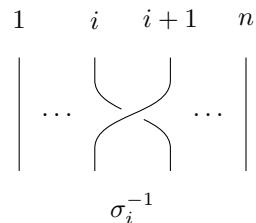
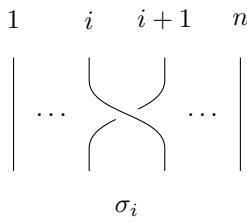
組紐からは「**組紐群 (Braid group)**」と呼ばれる群が構成されます。この群は紐が n 本の組紐、すなわち n 糸の組紐であれば次の群の表示になります：

— n 糸の組紐群 —

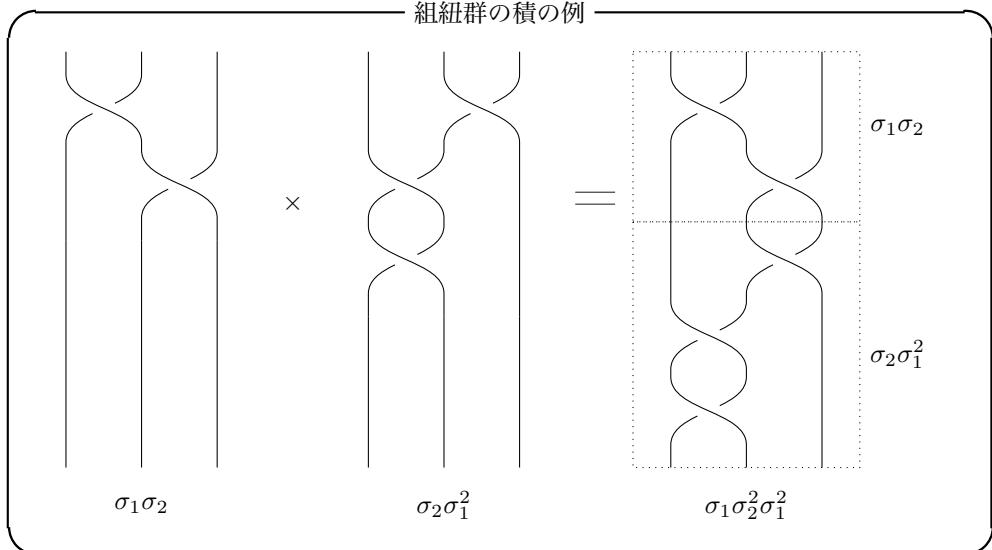
$$\left\langle \sigma_1, \dots, \sigma_{n-1} \mid \begin{array}{l} \sigma_i \sigma_k = \sigma_k \sigma_i, (|i - k| \geq 2, i, k \in [1, n-1]), \\ \sigma_i \sigma_{i+1} \sigma_i = \sigma_{i+1} \sigma_i \sigma_{i+1}, (i \in [1, n-2]) \end{array} \right\rangle$$

では生成元 σ_i は具体的にどのようなものでしょうか？ 次に σ_i と σ_i^{-1} と対応する組紐を示しておきましょう：

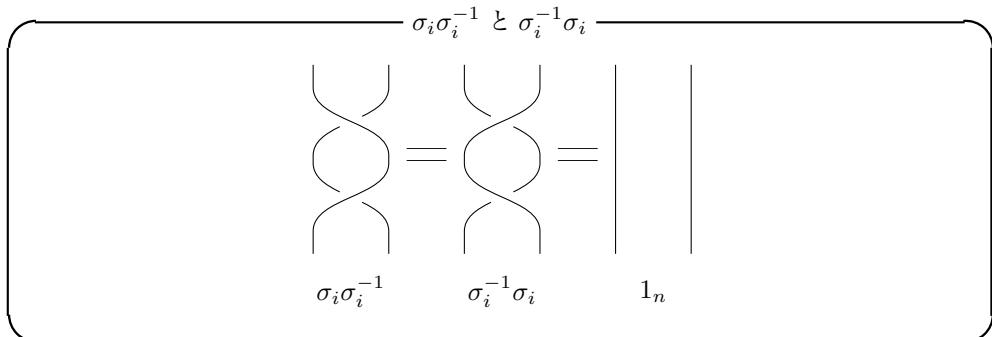
— n 糸の組紐群の生成元 —



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 度回すという操作です。それに対して 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：



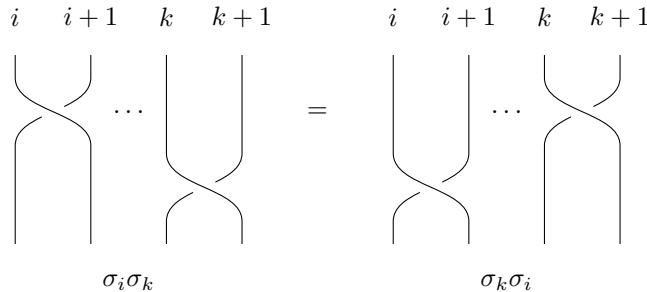
この例では $\sigma_1\sigma_2$ と $\sigma_2\sigma_1^2$ の積 $\sigma_1\sigma_2^2\sigma_1^2$ を示しており、右に示すように $\sigma_1\sigma_2$ を $\sigma_2\sigma_1^2$ の上に積み上げる形になります。また積の順番から言えば、左側から右にかけて対応する組紐を上から下に繋ぐ恰好になります。さて、ここで n 本の紐がまっすぐに垂れた組紐を n 糸の組紐の上に付けても下に付けても各紐の長さを調整してしまえば元の n 糸の組紐に戻せます。つまり、このことはまっすぐに垂れた n 本の組紐が n 糸の組紐群の単位元 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：



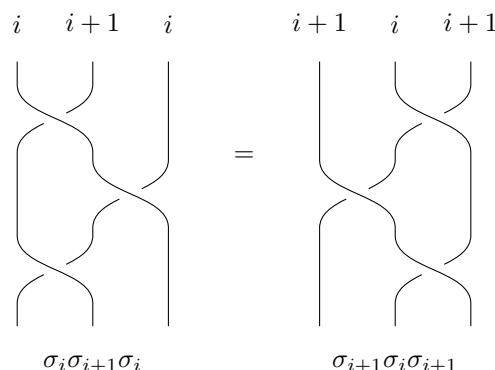
実際、左側の状態の紐を引っ張って長さを調整すれば 2 本のまっすぐな紐になります。このことから σ_i^{-1} は表記どおりに σ_i の逆元です。また $a_1a_2 \cdots a_n$ が与えられたとき、その逆元は $a_n^{-1} \cdots a_2^{-1}a_1^{-1}$ と語の順序を逆にして幕の正負を逆にしたもので与えられることも判ります。

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：

 $\sigma_i\sigma_k = \sigma_k\sigma_i$ の図示

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ら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：

 $\sigma_i\sigma_{i+1}\sigma_i = \sigma_{i+1}\sigma_i\sigma_{i+1}$ の図示

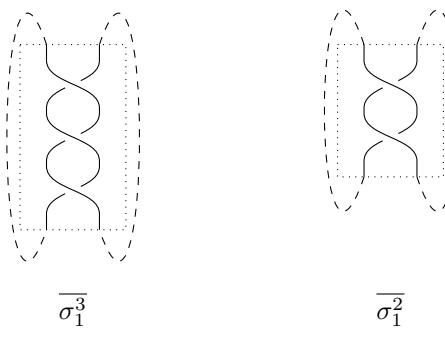
この関係式も図示してしまえば明確です。実際、この関係は i 番目の紐を上下に動かして相互の組紐に変形できることを意味します。この操作は同値な結び目/絡み目が得られるライデマイスター移動の TYPE III に対応する操作です。このように関係式は項の同値性を与える式で、この式を利用して項をより簡単なものにまとめられますが、この関係式をこのように可視化することで、その図形的な解釈が明瞭になります。

7.6.5 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：

閉じた組紐の例



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 [10]。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 を許容するときには限ることが知られています:

———— マルコフ移動 ————

$$\begin{array}{rcl} \text{M I.} & \overline{ab} & = \quad \overline{ba} \\ \text{M II.} & \overline{a\sigma_n} & = \quad \overline{a\sigma_n^{-1}} = \quad \overline{a} \end{array}$$

これも図示すると明快になります。移動 M I は a を構成する生成元を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 と同じ状況になっていることに対応しますが、ただ、この場合は紐の数に変動が生じます。

7.7 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 の時と同様に置換 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ると置換 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.

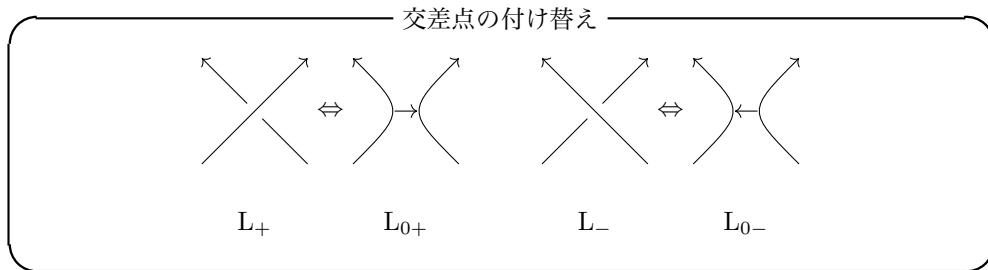
7.7.1 ザイフェルト曲面

任意の結び目と絡み目には組紐としての表現を持つと述べました. この組紐への変換を機械的に行う処理がありますが, この処理は後述の結び目/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)」と呼ばれる重要な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一緒に解説します. 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, 「向き付け可能」という意味は曲面上に「裏と表がある」ということです⁴. 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/絡み目にしてから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:

⁴ 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ぐという処理です。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」と呼び、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の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 7.6 に示します：

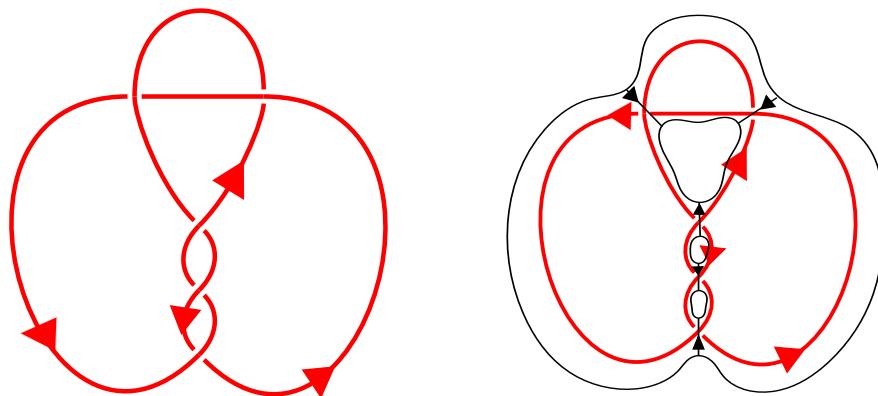


図 7.6 結び目と交差点の変換

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帶で置き換えることで得られる曲

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/絡み目です。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。なお、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ることがあります、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7.7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：

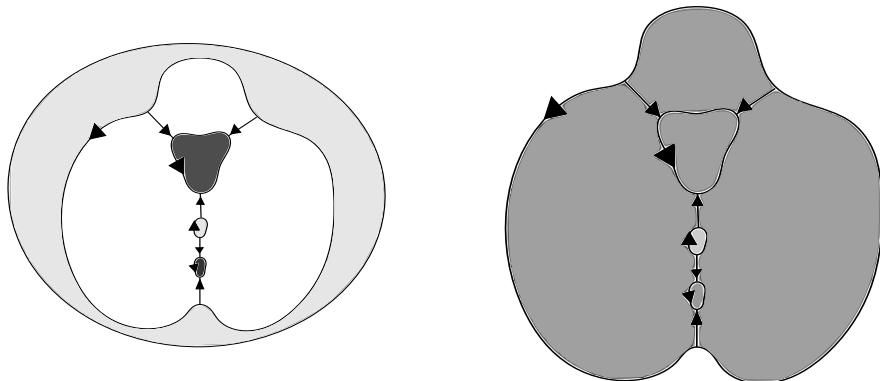
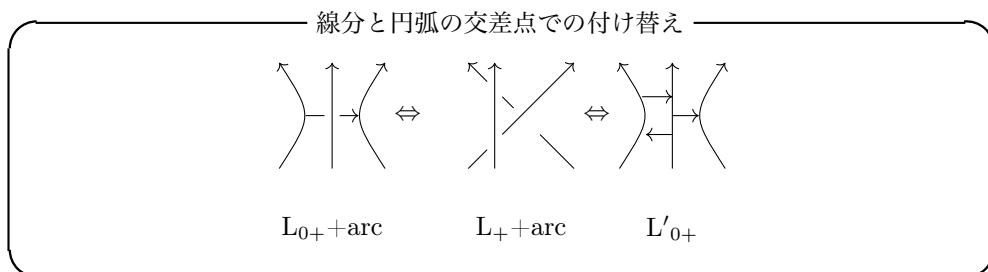


図 7.7 結び目の円盤

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。ところで、ここで得られたザイフェルト系に手を加えれば組紐表現に持ち込むことができます。そのため同心円状にザイフェルト円周を配置する必要があります。

■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の左側に平行移動します。円の中心と向きが揃っていない場合は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. 基準となる円を一つ定める。作業を楽にするために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を基準にする。
2. 基準になる円に線分でつながった円で、その向きが一致した円は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にした円を内部に含むことがないために、その円から出ている線分を適宜、平行移動させて、線分と交差点を持たない側の円弧を動かし、基準になる円を包含するようにできる。このときに基準にした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回の操作でこの処理を終えられる。ここで基準にした円と線分で繋がっていない円があるときは基準にした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. の円弧の移動によって交差点が増え、これらの交差点では線分への置き換え操作を行います。たとえば L_+ の線分については次の置き換えが生じます。このことは L_- の交差点でも同様で、この場合は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、ザイフェルト系に対して基準になる円を選んだ様子を以下に示します：

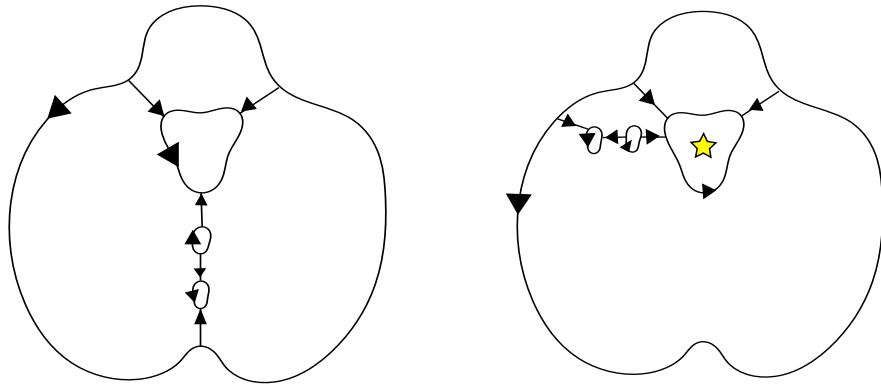


図 7.8 結び目のザイフェルト系と基準円の選定

この図で左側が結び目のザイフェルト系で、そこから基準になる円を選んで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ですが、この円を包含する円は最も外側の円のみで残る二つの円は違います。だから、これらの二つの円に対して変形操作を行わなければなりません:

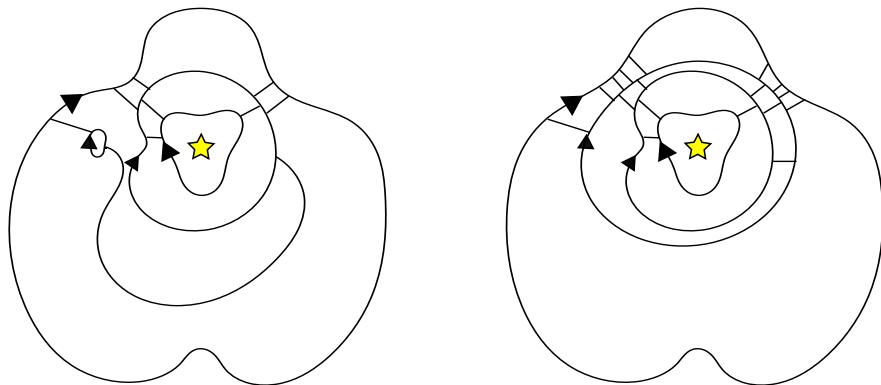


図 7.9 結び目のザイフェルト円周の変形操作

ここでの左図で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対しても置き換え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えます。

■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に沿って円の左側に水平移動させて纏め、それから線分を本来の交差に戻す。これらの処理で閉じた組紐の表現が得られる。ただし、この変換は交差数が最小になるような「綺麗な組紐」を与えるものではない。

この様子を先程の例に対して行ったものが図 7.10 です：

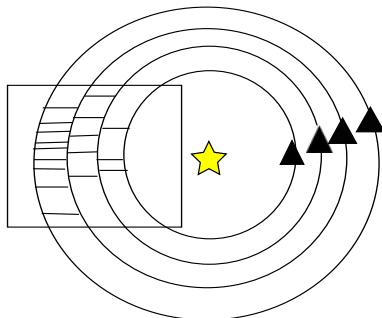


図 7.10 結び目の閉じた組紐としての表現

閉じた組紐表現が得られた時点で各円に円盤を貼り、線分を帯で置き換えることで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 糸の組紐が得られたのであれば n 段重ねのホット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。

7.8 ガウス・コード

絡み目/結び目の正則射影図をリストで表現する一つの方法として「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ります。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。この様子を三葉結び目で確認しましょう。

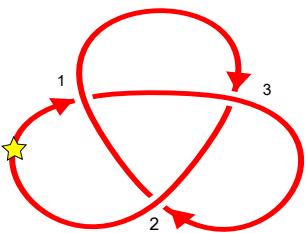


図 7.11 三葉結び目でのガウス・コードの生成

この三葉結び目の基点を☆で与えます。さて、この基点から開始するときに、基点の乗った道が交差点 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、こうして

得られた数列を「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カウス・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点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り、計算機の処理は俄然、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 に現れなかったときに p が含まれる成分 K_j を捜し、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 7.11 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しまっても、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・コードの元に入れ替えます。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。そのためには $i \in L_c$ が正規ガウス・コード G_c でどのように現われるかで分類して考察しましょう：

- $-i$ の場合：

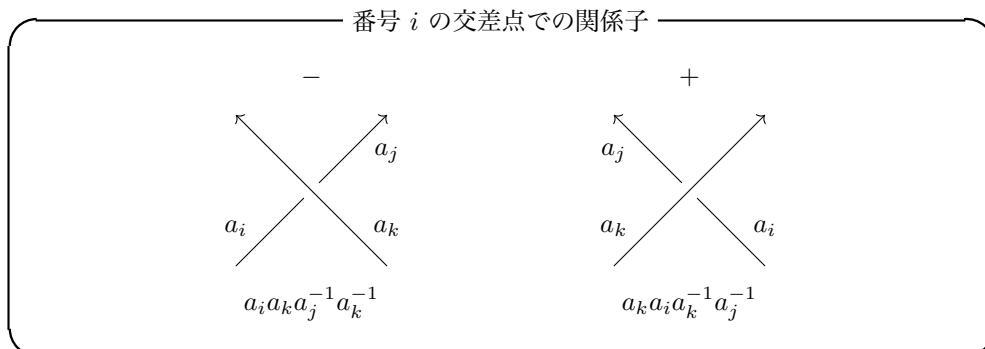
下道 a_i が交差点 i に入るため、ここでは交差点 i から出る下道を探します。まず、 G_c の $-i$ を含む成分リストを円環にして、数 $-i$ を基点にして順方向で隣の数から順番に調べます。ここで j を検証すべき成分リストの数とします。このときに数 j が負の数であれば求める a_{-j} が交差点 i から出る下道になりますが、 j が正の数であれば順方向で j の隣の数を調べます。この操作を繰り返して成分リストを一周して $-i$ に戻ったときは a_i が交差点 i から出る下道になります。このようにして交差点 i に関する二つの下道が求められます。次に交差点 i を通過する上道の探

素では、 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判りません。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 とするとときに、 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 7.11 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 $\llbracket -1, 3, -2, -1, -3, -2 \rrbracket$ 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 $G_c = \llbracket -1, 3, -2, 1, -3, 2 \rrbracket$ 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^{*5}。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 の符号を求ることで符号リスト L_s を生成しています。函数 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 = []
```

^{*5} 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。

```

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/結び目群のリストを返却する函数ができました。

7.9 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とためです。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 __repr__() 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。

7.9.1 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しかありませんが、ガウス・コードの一貫性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。

7.9.2 導入すべき代数的構造

この `LinkClass` に入れる「**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對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次の正方形行列の集合 $M(2)$ 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次の正方形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次の正方形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^{*6}。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發揮します。

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

^{*6} 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。

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はありません。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。

7.9.3 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。そのために 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、絡み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、絡み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なります。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、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。これらのこととは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労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

```

7.9.4 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))
        chck = map(lambda (x):axgssc.count(x)==2, axgssc)
        for i in chck:
            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)
    chck = map(lambda (x):aexgssc.count(x)==1, aexgssc)
    k = 0
    for i in chck:
        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])
        xcrrsns = map(lambda(x):abs(x), xgausscode)
        crsns = list(set(xcrrsns))
        cnt = map(lambda(z): xcrrsns.count(z)==1, crsns)
        w = range(0,len(cnt))
        if sum(cnt)>0:
            crps = list(set(map(lambda(z):cnt[z]*crsns[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 が指示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いはありません。

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。のちの不变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。

7.9.5 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い、以降、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 7.12 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 7.13 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帶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：

```
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]]
```

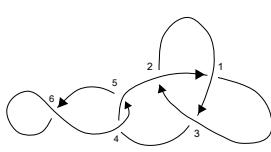


図 7.12 扱っている絡み目

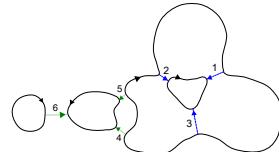


図 7.13 手書きのザイフェルト系

```
Components: 2
sage: g2 = d2.draw_seifert_circles(file="d2_test.png", layout=spring)
```

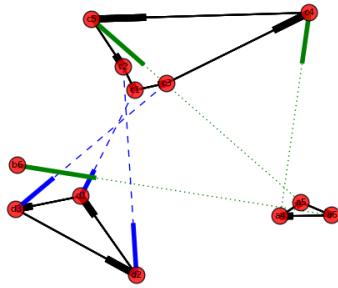


図 7.14 layout=spring による描画

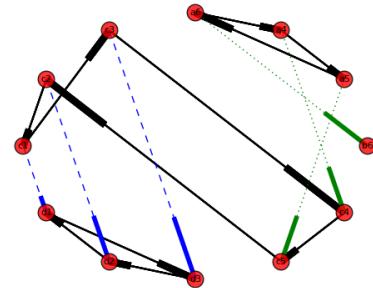


図 7.15 layout=circular による描画

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 7.14 と図 7.15 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 7.14 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 7.15 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.

7.9.6 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, それらが同じものかどうかの判別が難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 [19] 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

$$\begin{array}{ccc}
 \langle x, y, z \mid xyz(yzx)^{-1} \rangle & \xrightarrow{\text{II}} & \langle x, y, z, a \mid xyz(yzx)^{-1}, a(yz)^{-1} \rangle \\
 & \xrightarrow{\text{I}} & \langle x, y, z, a \mid xa(ax)^{-1}, a(yz)^{-1}, xyz(yzx)^{-1} \rangle \\
 & \xrightarrow{\text{I}'} & \langle x, y, z, a \mid xa(ax)^{-1}, a(yz)^{-1} \rangle \\
 & \xrightarrow{\text{I}} & \langle x, y, z, a \mid xa(ax)^{-1}, z(y^{-1}a)^{-1}, a(yz)^{-1} \rangle \\
 & \xrightarrow{\text{I}'} & \langle x, y, z, a \mid xa(ax)^{-1}, z(y^{-1}a)^{-1} \rangle \\
 & \xrightarrow{\text{II}'} & \langle x, y, a \mid xa(ax)^{-1} \rangle
 \end{array}$$

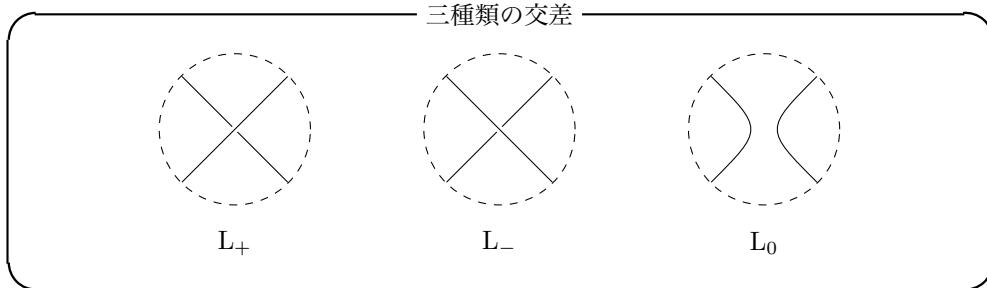
このように有限回の Tietze 変換の列によって二つの群の表示が同値であることが示せますが、この手法の難点は Tietze 変換を見つけなければ二つの群の同値性が主張できず、そのような Tietze 変換の列を見付けられないことと Tietze 変換が構築できないことが別問題であるため、Tietze 変換が判らないことが別の群の表示であると結論付けられない点です。この点はライデマイスター移動と似ていて、いずれにせよ計算と比較が容易な不变量が望まれる理由になります。

7.10 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。

7.10.1 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)**」^{*7}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): $\nabla_{L_+}(z) + \nabla_{L_-}(z) = z\nabla_{L_0}(z)$

■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変量です。

7.10.2 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

^{*7} skein は縫糸の意味です。

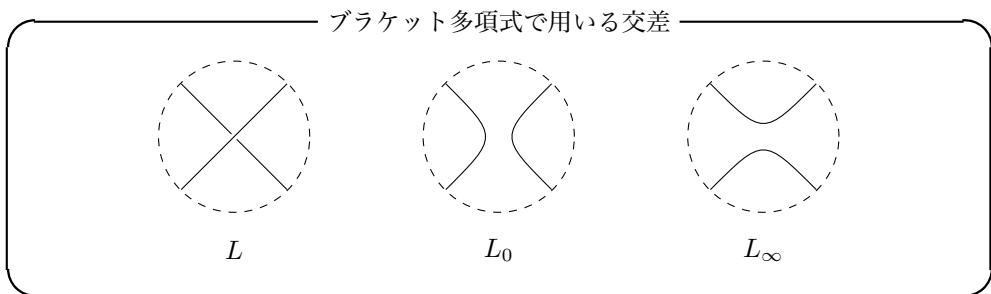
るもののが発見されていません。なお、SageMath の BraidGroup パッケージは、その関係からオプションの指定がなければカウフマンのブラケット多項式の流儀で表示され、オプションの指定があればスケイン形式から得られる多項式を出力します。

7.10.3 アレキサンダー多項式

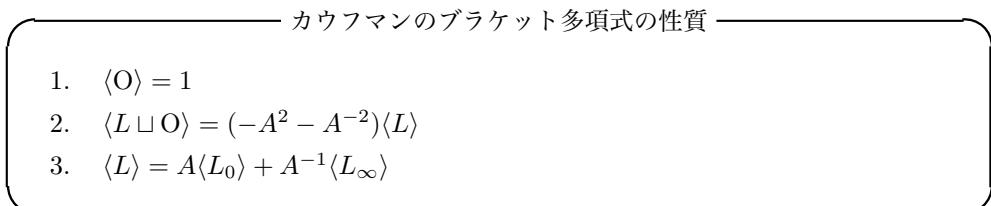
アレキサンダー多項式は古典的な結び目多項式です。スケイン多項式として認知されるまでは結び目群の表示から計算したり、ザイフェルト曲面を使って求めていました。この求め方からも判るように基本群と密接な関連を持っていますが、機械的に求められる手法が、結び目群の表示から求める方法で、結び目群の n 個の関係子をフォックス微分と呼ばれる特殊な「微分」を使って $n \times n + 1$ の大きさのフォックス微分によるヤコビ行列を計算し、このヤコビ行列から $n \times n$ の行列を取り出して、それらの行列式の最小公約数として(1次の)アレキサンダー多項式が得られます。この多項式の幾何学的解釈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。

7.10.4 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:



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はライデマイスター移動の TYPE II と TYPE III に關

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 ○ の交差点の符号が 1, ○ 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 の不変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 の置換によってでジョーンズ多項式 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：

$$J(L) = V_L(A^4)$$

7.11 カウフマンのブラケット多項式を計算するプログラム

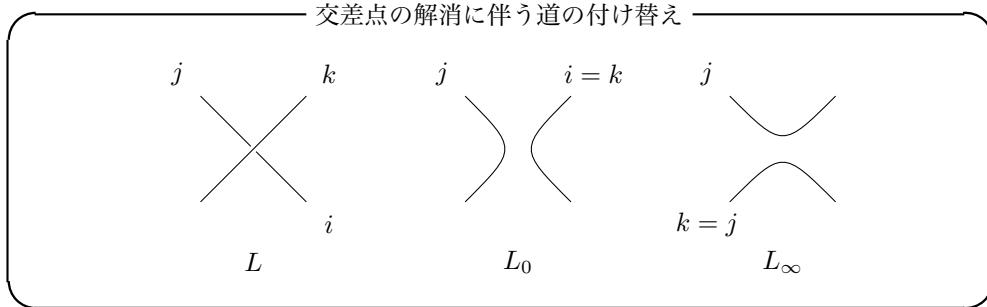
カウフマンの多項式の計算は非常に機械的に行えます。実際、 L から L_o, L_∞ への移行で交差が一つ解消しており、この交差一つの解消で二つの正則射影図が得られます。この操作を次の交差点で行うと 2×2 で 4 個の正則射影図が、さらに次の段階では 4×2 、すなわち 8 個の正則射影図が得られます。そして n 段目では 2^n 個の正則射影図が得られます。その一方で、各段で交差点が消去されるので交差点が n 個の正則射影図であれば最終的には n 段目で 2^n 個の互いに交差しない自明な結び目を成分とする絡み目の正則射影図が得られます。ここで自明な結び目を $-A^2 - A^{-2}$ で置き換え、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。

次に交差点の次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 `LinkDi-`

agram 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します:

```
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う函数ができました。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産物であるためです。

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_change_a() と crossing_change_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
        KBIK.append((-A**2-A**(-2))**j)
    n = len(KBIK)
    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 + KBIK[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、ここでの関数も局所変数に一時的に蓄える以上のこととはしていません。そこで SageMath に最初から含まれている 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レ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 にはパッケージに含まれているので import 文で読み込むだけです。

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。この処理は函数 `kauffman_bracket()` で用いています。それから函数 `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、函数 `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()

```

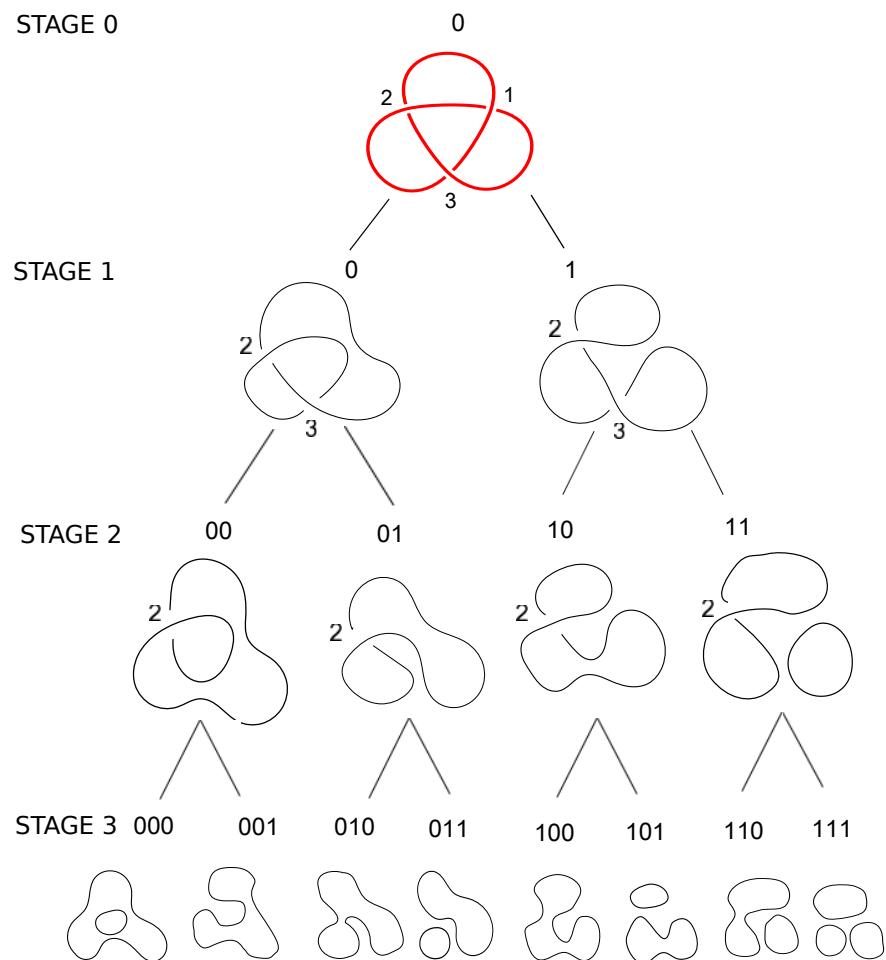


図 7.16 三葉結び目での処理

```

sage: bf1 = curl.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 = curl.execute("select * from diagrams where Stage==2")
sage: ans=bf1.fetchall()
sage: for i in ans:
....:     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 = curl.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なく、繰り返し書

き込むだけです。

第8章

CoCalcについて

8.1 CoCalc とは

CoCalc^{*1}は SageMath を基に Linux のディストリビューションの一つの Ubuntu 上で構築されたクラウド計算機環境の商用サービスであり、正式名称 Colaborative Calculation in Cloud が示すようにノートブックを複数の人々と共有することができます。また、商用サービスであるとは言え、利用に関しては無償も入れて四種類の料金カテゴリーがあります。



図 8.1 CoCalc の起動画面

TeX や reST の編集環境、ウェブ・ブラウザ上の仮想端末までもが組込まれています。それから、CoCalc のフロントエンドである Jupyter notebook は 2017/05 の時点で 40 以上のプログラム言語のフロントエンドとして利用可能^{*2}。で、CoCalc でもこれらのプログラム言語の編集が可能です。そして、CoCalc の利用は PC だけではなく、iPhone や iPad といった iOS 機器、Android 携帯でも 3 次元グラフ表示も含めて利用が可能です。このように CoCalc は SageMath 以上の統合化された数学環境になっていますが、CoCalc はそのサイトにウェブ・ブラウザにアクセスして利用するだけではなく、Docker をインストールした環境で利用できるように Docker のイメージとしても提供されています。この CoCalC の Docker イメージは SageMath、TeX と reST 形式の文書編集環境が付随する程度とは言え、7GB 程度のディスク容量を必要とします。

この CoCalc は 2017/5 に従来の SageMathCloud から改名したもので、旧名称の SageMathCloud との違いは名前と URL、初期画面が変更になった程度で、むしろ、CoCalc の Docker コンテナが “smc” と名付けられたままと、SageMathCloud に由来する名称は今後、しばらくは継続するでしょう。

^{*1} <https://cocalc.com>

^{*2} <https://github.com/ipython/ipython/wiki/IPython-kernels-for-other-languages>

8.2 利用者登録について

SageMath を利用するためにはあらかじめ利用者登録を行う必要がありますが、この登録は無料です。図 8.2 に示すログイン画面から Google+, Facebook, GitHub, Twitter のアカウントを使ってログインすることもできます：



図 8.2 ログイン画面

ここでの登録は無料、指定を行わない限り無課金です。無課金で利用するときに一つのプロジェクトに割り当てられるディスク容量は 3GB、メモリは 1GB、CPU の 1 つの core を有する形になります。またタイムアウトする時間が定められており、1 時間、無操作であればタイムアウトになります。無課金で利用する場合は、CoCalc のシステムが高負荷になったときに利用できなくなるといった不便さもありますが、軽く使う分には全く問題がありません。課金制でこの制約が緩和されます。この課金では 3 種類のプランが選べ、それぞれ月単位、あるいは年単位の課金が必要になります。なお、処理速度については無課金のものでも大雑把ですが MacBookAir 2013 と同程度で、そこそこ本格的な処理が可能です。だから無課金でもちょっとした計算や文書の作成、それらの共有といったことが十分に実用的な範囲で行えます。だからウェブベースであることも含め、教育や試計算には適しているのではないかと思います。実際に 400+ の学生の学生の教育で用いている例もあるようで、詳細は <https://github.com/sagemath/cloud/wiki/Teaching> を参照して下さい。また、CoCalc の FAQ が <https://github.com/sagemath/cloud/wiki/FAQ> に纏められていますが、このような教育事例が挙げられています。

ここでは CoCalc を使う上で、どのようなアプリケーションがあって、どのように使え

るのかといった簡単な事例を幾つか紹介することにします。

8.3 基本設定

CoCalc を利用するうえで基本的な設定は CoCalc のサイトに接続した状態で、ブラウザ右上側に表示されている利用者の名前を押すと基本設定のページに遷移します。CoCalc コンテナを利用しているときには「Settings」しか現われませんが、cocalc.com に接続しているときは「Settings」、「Billing」と「Upgrades」という名前のタグがあります。通常の利用に関連することは「Settings」で行います。この「Settings」設定可能なことを以下に纏めておきます：

- 利用者情報の設定
- 仮想端末のフォントと画面と文字色の設定
- キーボードショートカットの設定
- エディタの設定
- その他の設定
- UI の外観の設定

図 8.3 に利用者情報とエディタの設定の一部を示しておきます：

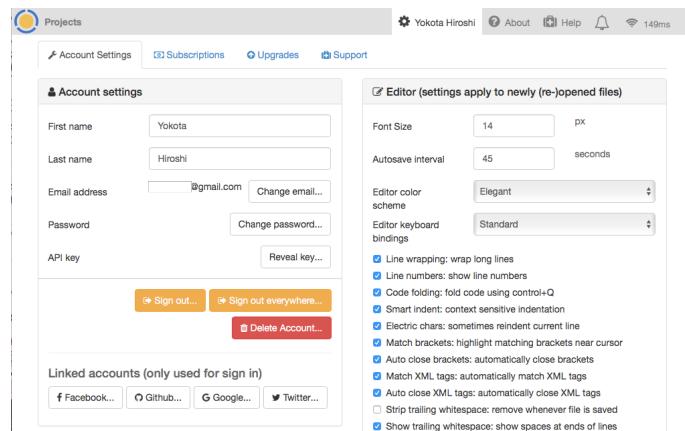


図 8.3 利用者情報と端末の設定

「Billing」と「Upgrades」は有償で利用するときに請求先のクレジットカードの指定、それとどのような条件で利用するかといったことを設定する箇所で、後述のプロジェクト側からはここで設定した有償利用の設定に基づいた設定以外のことはできません。現時点

では有料利用のプログラムとして 3 通りの設定があり、支払いは月単位か年単位の双方が選べます。図 8.4 に Upgrades のページの一部を示しておきます：

The screenshot shows the 'Upgrades' section of the CoCalc interface. At the top, there's a navigation bar with a user icon, the name 'Yokota Hiroshi', and various status indicators like 'About', 'Help', 'Bell', and network connectivity. Below the navigation is a section titled 'Personal subscriptions' with a brief description. Three plans are listed in boxes:

- Standard plan**: Includes 2 upgrades for Member hosting, 20 upgrades for Internet access, 1 day idle timeout, 3 GB Memory, 5 GB Disk space, and 0.5 shares CPU shares. Priced at \$7/month or \$79/year.
- Premium plan**: Includes 16 upgrades for Member hosting, 80 upgrades for Internet access, 8 days idle timeout, 24 GB Memory, 40 GB Disk space, 4 shares CPU shares, and 2 cores CPU cores. Priced at \$49/month or \$499/year.
- Professional plan**: Includes 40 upgrades for Member hosting, 200 upgrades for Internet access, 20 days idle timeout, 60 GB Memory, 100 GB Disk space, 10 shares CPU shares, and 5 cores CPU cores. Priced at \$99/month or \$999/year.

図 8.4 Upgrades のページ

8.4 プロジェクト

CoCalc ではプロジェクト単位でジョブやファイルの管理を行います。そのために利用者として登録したての場合は最初にプロジェクトを生成します。このプロジェクトの生成は「New Project...」とあるボタンを押すと「Create a New Project」というページに遷移します。ここでは表題と概要を「Title」と「Description」欄にそれぞれ記載して左下側の「Create project」ボタンを押せばプロジェクトが新規に生成され、最初の Projects のページに戻ります。プロジェクトでさまざまな作業を行うためには、この Projects ページに表示されたプロジェクトの名前を選択すればよいのです。プロジェクトの設定もプロジェクト単位で行えます。このときにプロジェクトを選択し、そのプロジェクト画面の右上にある「Project settings and controls」から行うことができます。このボタンを押すと図 8.5 に示す「Settings and Configuration」ページに遷移します：

ここで設定できる項目として、もしも有償利用の設定をあらかじめ行っていれば、それを適用するための設定が可能です。それ以外にできることを以下に挙げておきます：

- プロジェクトの隠蔽と削除
- プロジェクトの制御
- プロジェクトの共有設定

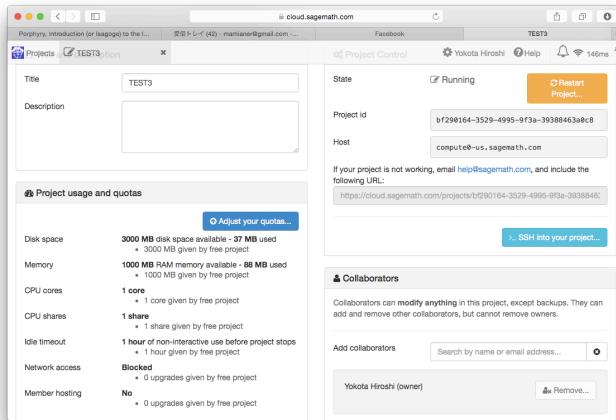


図 8.5 Project settings and controls のページ

- Sage ワークシートサーバの再起動

ここでプロジェクトの制御は基本的にプロジェクトの再起動を行う程度で、その他はプロジェクトの ID やそのプロジェクトが起動しているホスト名の情報や SSH の公開鍵の情報があります。

次に CoCalc ではプロジェクト単位で共有、つまり、CoCalc の利用者で一つのプロジェクトを共有することで互いに閲覧や編集が自由に行えます。この設定を行うためには「Search by name or email address...」と表示された欄に名前か E-mail アドレスを記入して CoCalc の登録者の検索を行います。ここで条件に合致する利用者が複数存在すれば表示されたリストから相手を選択して招待するためのメールを送付します。それでプロジェクトの共有が行えるようになります。

8.5 ファイルのアップロードとダウンロード

まず PC 等の端末上のファイルを SageMath 側にアップロードすることも、逆にファイルをダウンロードすることも可能です。まずアップロードのときはファイルの送り先のプロジェクトで右上の「New」メニューを押して図 8.6 に示すファイル生成に移ります：このページで「Upload files from your computer」とある項目で大きく「Drop files to upload」と表示された箇所にローカルにあるファイルをドラッグしてドロップするか、その箇所をクリックします。もしもクリックしたときはファイルのセレクターが現われるるので、そこでアップロードするファイルを選択します。またファイルをドロップしたのであ

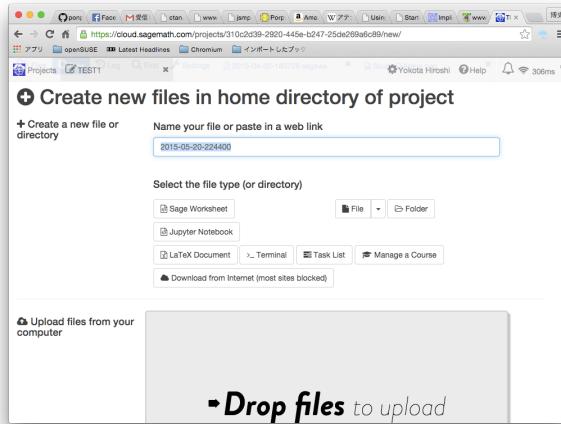


図 8.6 ファイル生成ページ

れば、しばらく待つと図 8.7 に示すようにサムネイルがやがて表示されます。当然のことですが大きなファイルだと転送に時間がかかります：

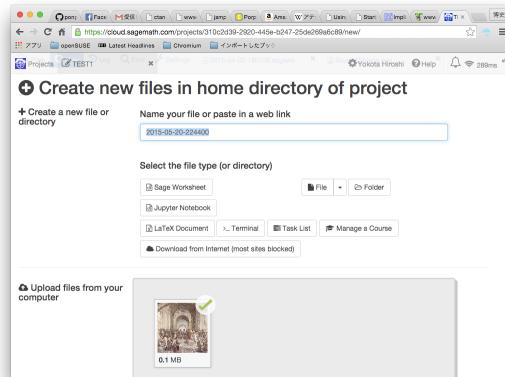


図 8.7 ファイルのドロップ後

なお、アップロードしたファイルや、プロジェクト内で生成したファイルの一覧は「File」メニューを押すことで表示される図 8.8 に示すページで確認することができます：

この図の右上に「Terminal command..」と記載のある入力欄があります。ここでは通常の UNIX コマンドが入力可能です。このページではファイルやディレクトリ操作（ファイル/ディレクトリの生成/削除、移動、名前変更等）が行えます。

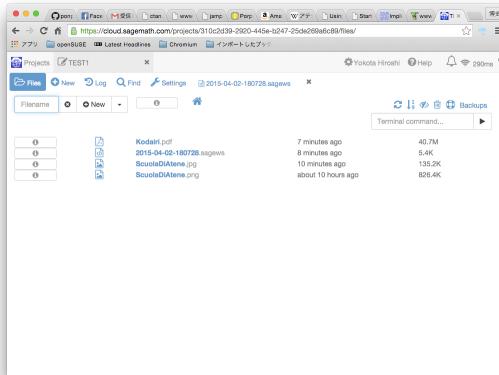


図 8.8 プロジェクト内のファイルの一覧

CoCalc ではプロジェクト単位にファイルが保管され、プロジェクト間のファイルの移動も複製することで行えます。また、同一プロジェクトであれば CoCalc が包含するアプリケーションからも通常の Linux 環境と同様に扱うことができます。たとえば CoCalc 上にアップロードしたファイルの読み込みの例を図 8.9 に示しておきます アップロードした画像を CoCalc 上に読み込んで、関数 `matrix_plot()` で表示した例を示しておきます：

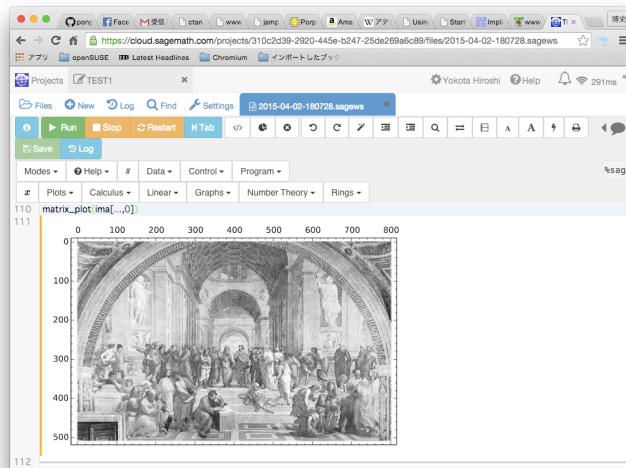


図 8.9 アップロードした画像の表示

8.6 端末, Jupiter, L^AT_EX の利用について

CoCalc は単に数式処理システム SageMath を単にクラウド環境で実現するだけではなく, SageMath が包含するさまざまなアプリケーションが利用できる環境です。アプリケーションの起動には二つの方法があります。一つは先程のファイルのアップロードで用いた New メニューから図 8.6 に示すファイル生成に移行し、そこで「Select the file type(or directory)」の項目以下に並んだ「Sage Worksheet」, 「Jupyter Notebook」, 「LaTeX Document」や「Terminal」を選ぶか、あるいは「File」メニューを押してプロジェクトのファイル一覧のページに移行し、そこで「New」メニューを押して図 8.10 に示すように「Sage Worksheet」等のメニューを出して選択する方法があります:

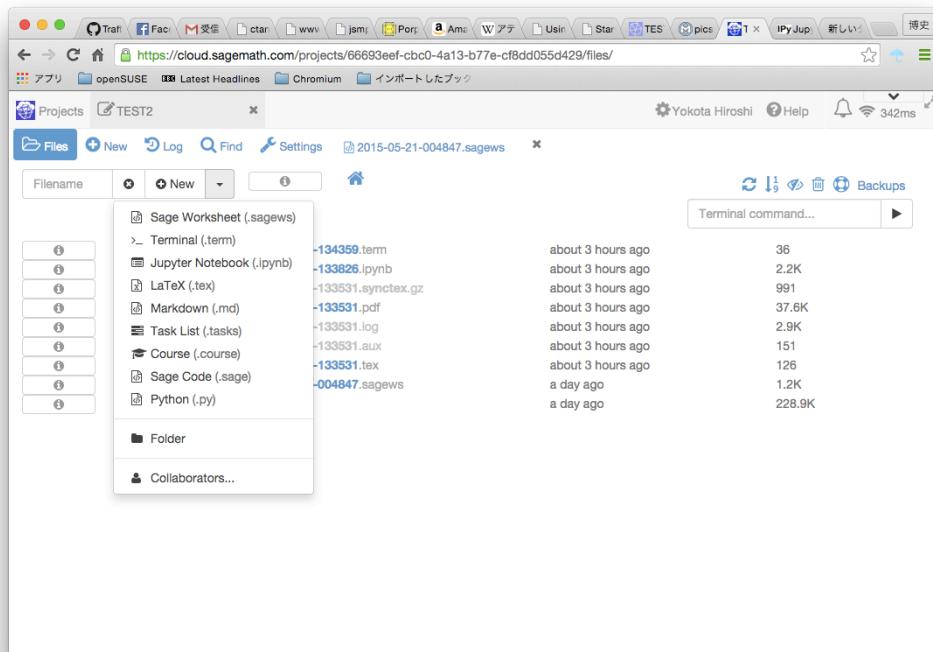


図 8.10 New メニューの一覧

ここで「Sage Worksheet」は説明するまでもなく SageMath のワークシートですが、通常の SageMath のワークシートよりも洗練されています。それから「Terminal」は起動するとウェブ・ブラウザを通常のテキスト端末として利用できます。ここで CoCalc というサービスを Ubuntu Linux 上で実現していることから、Terminal を起動するということ

は Ubuntu という Linux 環境が直接利用できることを意味します。また、SageMath というシステムは雑多なアプリケーションやライブラリの集合体であるために、仮想端末を使うことで SageMath を構成するアプリケーションやライブラリを SageMath から必要に応じて呼び出すだけではなく、そのアプリケーションやライブラリを直接活用することが可能になります。なお、端末で処理した結果はそのままプロジェクト単位でディスク上に残すことができます。

次の「Jupyter」は Jupyter notebook と呼ばれるノートブック形式のシェルを起動します。この Jupyter^{*3}は IPython の後継のプロジェクトで、Jupyter notebook は Python 専用のノートブック形式のシェルの IPython notebook の中核を他の言語のシェルとして利用できるように抽出したものです。そのためには他の言語、たとえば統計処理システムの GNU R、近年注目されている数値計算言語 Julia 等に対応するカーネルを利用することで、ウェブ・ブラウザ上でそれらの言語のノートブック形式のシェルとして使えます。ちなみに <https://try.jupyter.org/> に Jupyter をシェルとして利用する GNU R、Julia と IPython のデモがあります^{*4}。なお、CoCalc では Jupyter Notebook で利用できる kernel として Python2、Python3、GNU R と Julia があらかじめ用意されており、カーネルの切替はノートブックのセル単位で行えます。つまり、Jupyter では一つのノートブックにこれらの言語の結果が混在可能です。この kernel の切替はメニューの「Kernel」から使いたい言語に対応する kernel を選択することで行いますが、この選択を行った時点でアクティブなセルから次に kernel を切替えるまでが指定した kernel に対応する言語環境が利用できることになります。ただし、kernel を切替えて再び戻したときに以前の処理結果はノートブックには記載されても言語環境が再起動されているために、それ以後の入力セルに以前の処理結果が引継がれません。なお、kernel の切替を行わなければ kernel を切替えてからの処理結果はそのまま後続のセルに引継がれます。

Jupyter+Python でノートブックにグラフや絵を表示させることができます。この場合はあらかじめおまじないとして `%matplotlib inline` を入力しておきます。すると Matplotlib に含まれる函数で画像の表示を行うと図 8.11 に示すようにそのまま Jupyter 側のノートブックに画像が表示されます。このことは IPython Notebook も同様です：

それから LaTeX は幅広く利用されている組版処理環境です。こちらは図 8.12 に示すように、ブラウザ画面を左右に二分して LATEX ファイルのエディタ画面が左半分、レンダリングした結果が右半分に表示されます。左半分に表示された LATEX ファイルを直接編集す

^{*3} <https://jupyter.org>

^{*4} <https://github.com/ipython/ipython/wiki/IPython-kernels-for-other-languages> には Jupyter で利用可能な kernel の一覧があります。

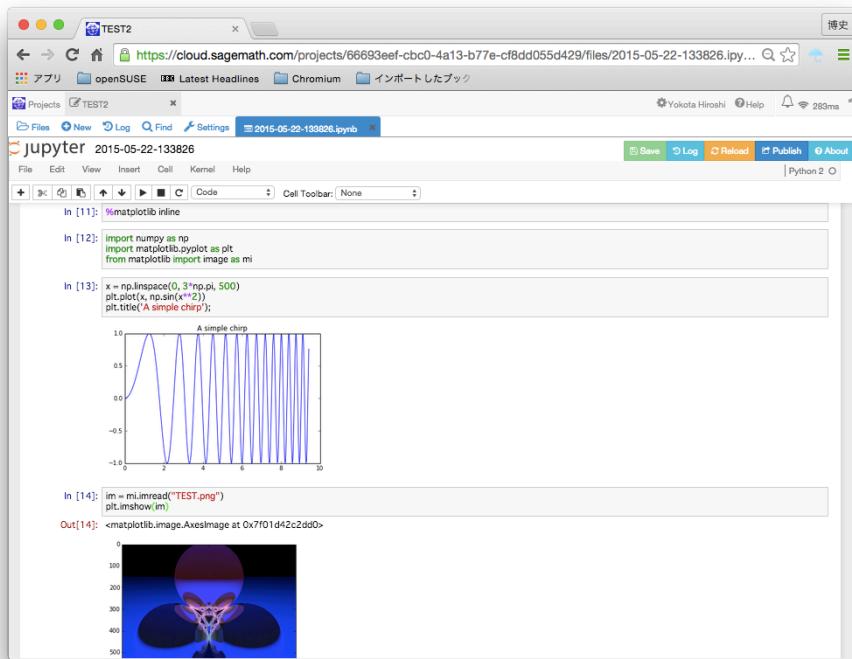


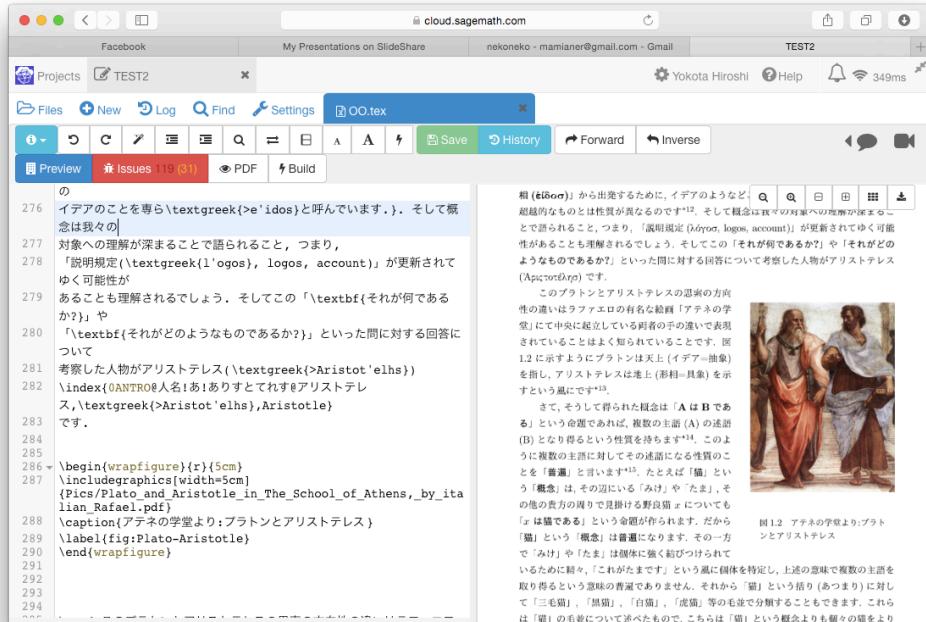
図 8.11 Jupyter+Python

ると早速コンパイルが行われて結果が右側に表示されるというもので、作成した文書のプレビューを行い、最終的に仕上げた文書を PDF に変換してダウンロードできます：

なお、クラウド上で L^AT_EX のサービスを行うものに Cloud LaTex ⁵があります。この Cloud LaTex は CoCalc が提供する L^AT_EX 環境よりも本格的なサービスで、原稿ファイルのドラッグ・ドロップによるアップロードが容易に行えること、日本語、中国語等のマルチバイト言語にも標準で対応しています。また、CoCalc 上の LaTeX も TeXLive であるために一通りのスタイルファイルが準備されていますが、日本語文書の作成では幾らかの工夫が必要になります。まず、CoCalc は通常、PDFLaTeX が呼び出されます。このときに CoCalc 側であらかじめ用意されたテンプレートでは日本語を扱うことができません。そのためにはクラスファイルやパッケージの設定を行う必要があります。また、PDFLaTeX ではなく pLaTeX や LuaTeX-ja を呼び出して利用することも可能です。ただし、CoCalc で利用できないクラスファイルも一部あるようです⁶。この CoCalc の L^AT_EX 環境で、こ

⁵ <https://cloudlatex.io/ja>

⁶ ここでの情報は Mathlibre の WIKI⁷を参照しています。

図 8.12 CoCalc 上の L^AT_EX

の本の §2 の文書のように図や図式が多くなるとやや苦しくなりますが、一寸した論文の著作には問題にはならないでしょう。

■PDFLaTeXで日本語文書を作成する場合: PDFLaTeX が CoCalc の L^AT_EX 環境で既定値として使われています。そのため日本語の利用では次の設定で日本語文書の作成が可能となります：

```
\documentclass{scrartcl}
\usepackage[whole]{bxjxkjkjatype}
```

ただしスタイルファイルの `bxjsarticle` には対応していないとのことです。

■LuaLaTeX(LuaLaTeX-ja)を利用する場合: PDFLaTeX が既定値であるために、LuaLaTeX を利用するためには PDFLaTeX の代りに LuaLaTeX が動作するように指定する必要があります。そのために CoCalc の環境変数 `latex_command` に `lualatex` のオプションとレンダリングすべき L^AT_EX 文書も含めて記載する必要があります：

```
%sagemathcloud={"latex_command ":" lualatex -synctex=1 -interact nonstopm
```

```
\documentclass{ltjsarticle}
```

ここでの例では L^AT_EX ファイルの名前が “sample.tex” のときのヘッダ部分を示しています。

このように CoCalc は Cloud 環境における Sage の動作に限定されるものでなく数学の研究に必要とされる一切合切を統合した、より大規模なシステムを目指しており、同列に論じられることの多い *Mathematica*, MATLAB や Maple といった数式処理や数値行列処理システムとの際立った違いを見せてています。SageMath の核となる Python は Python(ニシキヘビ)⁸ 」というだけあって一切合切を丸呑みしている傾向がありますが、Python 言語を中心としたさまざまなアプリケーションを統合した環境を構築しているという面で注目すべきことです。

8.7 Docker コンテナとしての CoCalc

CoCalc は Docker のイメージとしても提供されています。Docker が導入された環境では、単に

```
docker run --name=cocalc -d -v ~/cocalc:/projects -p 443:443 sagemathinc/sagemathcloud
```

と入力するだけで CoCalc のイメージのインストールができます。コンテナの起動は `docker start cocalc`、コンテナの終了は `docker stop cocalc` で行います。実際の利用はウェブ・ブラウザから <https://localhost> にアクセスすることで行えます。このときに証明書のことで通信がブロックされることもあるため、その場合は利用ブラウザで sageMath の証明書を信頼するように設定する必要があります。また、macOS(OSX) 環境のノートブックではサスペンド/レジュームで時間が狂う問題があります。そのため `docker-time-sync-agent`^{*8} を導入する必要があります。このインストールは

```
curl https://raw.githubusercontent.com/arunvelsriram/docker-time-sync-agent/master/install.sh | sh
```

とするだけでできますが、ここで用いられているスクリプト `install.sh` が Python 2.x 向けで、Python 3.x ではエラーになります。そのため Python 2.x で使えるように環境変数 PATH の設定が必要になります。macOS で Python 3.x を用いる Anaconda をインストールしている場合は、Anaconda のインストーラで同梱の Python 3.x を用いるように PATH を設定してあるために注意が必要です。

つぎに Docker イメージの更新は

^{*8} <https://github.com/arunvelsriram/docker-time-sync-agent/>

```
docker pull sagemathinc/sagemathcloud
```

で行います。この箇所は cocalc ではなく sagemathcloud であることに注意が必要です。
また、CoCalc の Docker イメージが不要になったときは

```
docker stop cocalc  
docker rm cocalc
```

で削除できます。なお、プロジェクトはディレクトリ ‘~/cocalc’ に保存されたままで、削除による影響を受けません。

第 9 章

SageMath の拡張

9.1 sagemath からのパッケージ入手

SageMath にはさまざまなパッケージが存在し、それで十分に思えるかもしれません。しかし、Python の GUI ライブライアリの wxPython や PyQt4 といったライブライアリ、数学関連のデータベースや CLISP 等のアプリケーションといったものを追加することも可能です。このようなパッケージは ‘<http://www.sagemath.org/download-packages.html>’ で公開されています。現在、Standard, Optional, Huge, Experimental の四種の範疇に分類されて FTP や BitTorrent 等の P2P からの入手が可能となっています。

そして入手したパッケージは SageMath をインストールした user-id で、仮想端末上で
`sage -i <パッケージファイル>` と入力することで SageMath にインストールすることができます。

9.2 一般の Python パッケージのインストール

SageMath は Python 上で構築されたシステムであるため、SageMath 側の Python にライブラリやパッケージをインストールすることが可能です。この場合は sagemath.org からのパッケージを入手してインストールする方法と異なり、あらかじめ環境変数の設定を行う必要があります。なぜなら SageMath 側の Python やライブラリ一式を利用しなければならないので、環境変数 PATH と LD_LIBRARY_PATH をインストールしてある SageMath の状況に合せておく必要があります。たとえば UNIX 環境で Sage が /usr/local/sage にインストールされ、仮想端末で Bash をシェルとして用いている環境なら `export PATH=/usr/local/sage/local/bin:$PATH` と
`export LD_LIBRARY_PATH=/usr/local/sage/local/lib64:/usr/local/sage/local/lib:$LD_LIBRARY_PATH` と設定しておきます。あとは Sage に easy_install があるので、それを用いたり、setup.py 等を用いたりと、インストールしようとする Python パッケージ別の対処になります。

9.3 GNU R のパッケージのインストール

上記の設定を行っていれば、SageMath に付属する GNU R に CRAN で公開されているパッケージの導入も行えます。

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] 飯田隆, 言語哲学大全 I 論理と言語, 効草書房, 1987.
- [5] 井筒俊彦, イスラーム思想史, 中公文庫, 中央公論社, 1991.
- [6] 今道友信, アリストテレス, 講談社学術文庫, 2004.
- [7] 大石進一, 精度保証付き数値計算, コロナ社, 2000.
- [8] 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.
- [9] 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/
- [10] 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.
- [11] 後藤和茂, BLAS の概要 (http://jasp.ism.ac.jp/kinou2sg/contents/RTutorial_Goto1211.pdf), 2006.
- [12] 柴田有, グノーシスと古代宇宙論, 効草書房, 1982.
- [13] 清水哲郎, オッカムの言語哲学, 効草書房, 1990.
- [14] 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.
- [15] 藤野登, 論理学 -伝統的形式論理学-, 内田老鶴園, 2003.
- [16] 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>
- [17] 田中尚夫, 選択公理と数学, 星雲社, 1987.
- [18] 聖トマス, 形而上学叙説, 岩波書店, 1935.
- [19] 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.
- [20] プラトン(著), 藤沢令夫(訳), 国家, 岩波文庫, 岩波書店, 1976.
- [21] フレーゲ, フレーゲ著作集 1 概念記法, 効草書房, 1999.
- [22] フレーゲ, フレーゲ著作集 3 算術の基本法則, 効草書房, 2000.

- [23] 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.
- [24] 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
- [25] M.Lynne Murphy, Ane. Koskela, 意味論キーターム辞典, 開拓社, 2015
- [26] 皆本晃弥, IEEE754 と数値計算,
<http://www.ma.is.saga-u.ac.jp/minamoto/doc/kyudai.pdf>
- [27] 山内志朗, 普遍論争, 平凡社ライブラリー, 2008
- [28] 横田博史, はじめての Maxima, I/O Books, 工学社, 2006.
- [29] 横田博史, はじめての Maxima 改訂 α 版 (MathLibre に収録)
- [30] 横田博史, 数値計算・可視化ツール Yorick, I/O Books, 工学社, 2010.
- [31] 吉田光邦, 鍊金術 - 仙術と科学の間 -, 中央公論新社, 2014.
- [32] J.Barnes, PORPHYRY INTRODUCTION, Oxford University Press, 2006.
- [33] Birman, Braid groups and mapping class groups, Ann. Math, Princeton University Press, 1963.
- [34] Boethius, Isagoge, <http://www.forumromanum.org/literature/boethius/isag.html>
- [35] G. J. Brose, MATLAB 数値解析, Ohmsha, 1998.
- [36] Haigh, An interview with Jack J. Dongarra,
http://history.siam.org/pdfs2/Dongarra_%20returned_SIAM_copy.pdf, 2004.
- [37] Goodger, reStructuredText ディレクティブ,
<http://docutils.sphinx-users.jp/docutils/docs/ref/rst/directives.html>
- [38] An interview with Charles L. Lawson,
http://history.siam.org/pdfs2/Lawson_final.pdf, 2004
- [39] Mac Lane, The Category theory for working Mathematician, Springer
- [40] Mac Lane, Moerdijk, Sheaves in Geometry and Logic, A First Introduction to Topos Theory, Springer Verlag, 1992.
- [41] Porphyry, Introduction(Iagoge) to the logical Categories of Aristotle,
http://www.ccel.org/ccel/pearse/morefathers/files/po.../porphyry_isagogue_01_intro.htm
- [42] Porphyry, Letter to Marcella,
http://www.tertullian.org/fathers/po.../marcella_02_text.htm
- [43] B.Russell, The Principles of Mathematics, W.W.Norton & Company, Inc., 1996.
- [44] B.Russell & A.N.Whitehead, Principia Mathematica to *56, Cambridge Mathematical Library, Cambridge University Press, 1997.
- [45] Diving into Python: <http://www.diveintopython.net/toc/index.html>
- [46] MathWorks 日本: <http://www.mathworks.co.jp/>
- [47] アテナイの学堂 <http://ja.wikipedia.org/wiki/アテナイの学堂>