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 -

USO667 版

横田博史

2025 年 04 月 12 日 (土)

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みながら仮想環境やクラウド環境へと柔軟に対応しつつ, その機能を拡充して行くことに顕著に現われています. 多様なものを組み込んだ SageMath は下手をすれば鵺やキメラのような継ぎ接ぎだらけのシステムになりかねませんが, Python がこれらの機能を上手く繋ぎ合せることでシステムとしての統一感を与えています. 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です. だから, この SageMath の「**車輪の再発明をしない**」という開発手法は非常に興味深い手段であり, その方針で構築したシステムはそれだけでも注目に値するシステムです.

この文書は SageMath の解説書と銘打っているものの, 読んで頂ければ判りますが, 古くは古代ギリシャ, 中世のスコラ学, それに集合論や圏論等と, 必要と思われる哲学的, 数学的なことがらや Python 自体のことを適宜, 私の好き嫌いにしたがって鬼火がフラフラと漂うように記述しています. 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が, その浮遊具合を色々と楽しんで頂ければと思う次第です.

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

令和 7 年 01 月 24 日 (金)　　　　狸穴主人 横田博史

目次

第 1 章

SageMath について

πάντες ἄνθρωποι τοῦ εἰδέναι ὀρέγονται 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. 開発者のモットーである「**車輪の再発明をしない**」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しています. この開発手法の背景について簡単に説明しておきましょう.

OSS のアプリケーションには大学等の公的な研究機関での研究成果として一般に公開されたものが多くあります. しかし, その多くが専門分野で非常に優れた性能を持っていても用途と利用者が限定され, GUI を持たずにコマンドライン入力だけで, 処理言語やデータ構造が独特な仕様, それに加えてマニュアル等の文書が不十分であったりと誰もが簡単に使える状況でないことがあります. ここで入出力を抽象化して汎用性を持たせ, 処理言語やデータ構造も Python で統一するとどうなるでしょうか? このときに利用者に見えるものは処理言語の Python と, その Python を使って定義したデータでしかなく, 新たに定義したデータも専門家の観点のみで定義したものから, より広範囲な観点を加えた普遍的なものとして Python で再定義された環境です. その結果, どのように対象が Python で表現されているかということを理解していれば門外漢でも高度な処理が行え,Python を基盤にすることで統一的な操作環境も得られて以前は利用できなかった別分野のアプリケーションとの連携を視野に入れた基盤も整備できます.

このように良いことづくめですが, この既存のアプリケーションを連携して使うという手法が十分に実用的になった背景に近年の計算機環境が非常に贅沢な環境であるという事実があります. 実際, 携帯電話の CPU でさえも 1GHz 以上の動作周波数で複数のコアが動作し, 3GB 以上のメモリ, 最低でも 64GB 程度の記憶媒体を持ち, 高速ネットワークに当然のように接続可能になっています. このような「**贅沢な環境**」では既存のアプリケーションを Python 言語のような比較的低速な対話処理言語で繋ぎ合せたシステムで, 常識的なプログラミングで実用的な処理速度で動作し, 職人技で最適化したシステムよりもリリースの期間や作業量等を含めて全体的なコストが安く上がるという長所まであります[*1].

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

1.1.2 SageMath の中核としての Python

SageMath は Python で記述され, Python をその処理言語として利用します. この Python はオブジェクト指向プログラミングの考えが取り入れられて生産性の向上が図られています. たとえば,「**継承**」と呼ばれる機能は新しいクラスを構築する際に既存のクラスを雛形として使うことで, その既存のクラスに付随する属性やメソッドを新しいクラスのメソッドとして流用し, その書き換えや, 新しいクラスへの追加分のみのの追加ができる機能です. 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実装されていなかったとします. ところで, この言語で複素数を扱う必要が生じたときはどうすれば良いでしょうか. まず, 複素数が実数の対として表現できることから, 複素数という数がその言語で既に実装されている実数を使って定義できます. それから複素数が持っている四則演算を実装することになるでしょう. しかし, この四則演算の定義では実数の四則演算を利用して定義することになります. これがオブジェクト指向プログラミング言語ならば実数のクラスを雛型に複素数のクラスが構築できます. このときに実数のクラスにはメソッドとして実数に付随する四則演算が定義されていますが, これらのメソッドが複素数クラスにそのまま継承され, 複素数上の四則演算を頭から構築する必要がなく, 実数の四則演算を基に足りない純虚数に対する処理を追加することで複素数の四則演算が定められます. このことはライブラリを構築したときに, それを基礎とするさまざまなライブラリが容易に構築できることを意味し, ソフトウェア資産の効率的な利用ができます. そのために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 では, 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おり, 処理すべき問題に適合したクラスを選択してメソッドの修正や追加, あるいは類似のクラスを継承して新たなクラスを自らが定義しさえすればよいのです. このように数学の諸問題への日々の対処が将来への資産として容易に生かせ, また, SageMath は OSS の各種アプリケーションやライブラリを組み込んでいるために, その動作や実装方法に疑問があれば何時でも自分でソースコードレベルから調べられ, 必要に応じてカスタマイズも可能な点が SageMath の強みです. もちろん, オブジェクト指向であれば全ての問題が解決されるというものではありません. オブジェクト指向の場合は, 逆に言えばオブジェクトが存在しなければメソッドも何もないために何もできないという側面があります. つまり, 多項式 $(x-1)^2$ の展開を行いたければ, それを展開するメソッドや函数以前に, 多項式のクラスがなければ意味のあるオブジェクトとして入力さえもできないということです. ただし, SageMath の場合は変数 x が定義されている多項式環があらかじめ設定されているために, このことをあまり意識しなくても使えるように工夫はされています.

1.1.3　電池込みだよ

SageMath は OSS のさまざまな分野の成果を統合したソフトウェアです．ノートブック形式のユーザ・インターフェイスを Jupyter Notebook で, 仮想端末上では IPython を用いています．ここで, Jupyter は IPython から Python 以外のアプリケーションに対してもウェブ・ブラウザ上でノートブック形式のユーザ・インターフェイスが使えるように IPython から分岐したもので, 利用可能なアプリケーションに対応するカーネル (kernel) をノートブックのセル単位で切り替えることで一つのノートブックに複数のアプリケーションのスクリプトとその結果が混在できます．SageMath や各種アプリケーションで作成した計算結果やグラフィックスをノートブックに表示させ, 組版指示言語の Markdown で記述した文書をレンダリングして表示ができるために報告書の作成どころか本の著述もできます．このように SageMath は様々なアプリケーションやライブラリが有機的に取り込まれて SageMath という一つのアプリケーションとしての外観を持たせることに成功したシステムです．この SageMath を数式処理システムとして使うも良し, 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)**」[*2]を彷彿させるものです．

この SageMath に類似したものに Anaconda Inc. の Anaconda があります．これは GNU Linux, macOS と MS-Windows の主要な計算機環境に対応したデータ・サイエンティスト向けの Python のディストービューションで, 関連するライブラリや OSS のアプリケーションを纏めたものです．SageMath にも Anaconda にも共にクラウド環境 (CoCalc と AnacondaCloud) がありますが, システムとしての統合度は大きく異なります．たとえば, Python で記述された数式処理パッケージ SymPy が SageMath と Anaconda の双方に含まれていますが, Anaconda で SymPy の扱いは NumPy や Matplotlib 同様に Python の一つのパッケージでしかありません．つまり, Anaconda は必要とされるパッケージやライブラリを正常に動作するように揃えた Python 環境です．ところが SageMath は統合化された環境を提供します．実際, SageMath で SymPy は数式の表現を受け持つ不可分の部品の一つであり, 数式全般的の処理は Common Lisp で記述された汎用の数式処理 **Maxima** が中心に据えられています．この Maxima は MIT で人工知能の研究と並行して開発が行われた最古参の数式処理 Macsyma の OSS 版で, 文脈 (Context) 等の興味深い機能を持っていますが, 近年, 急速に発展した計算機代数の結果が十分に取り入れられたものではなく, 古風な数式処理であることは否定できません．そこで, SageMath は可換環の処理を **Singular**, 数論を **PARI/GP**, 有限群論を **GAP**

[*2] 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．

といった OSS の専門の数式処理に任せています．さらに SageMath では数値行列の処理とプログラミング機能は Python に, そして, フロントエンドは IPython や Jupyter に任せて弱点を克服しています．このように様々なシステムを組み込んでいても利用者には Python をその処理言語として用いる一つの大きなアプリケーションとしか見えません．このように SageMath は多様なアプリケーションを一纏めにした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．なお, SageMath も Anaconda の試験的なパッケージとして conda-forge に登録されています．これも一つのパッケージとしてではなく, アプリケーションとしての扱いです．

1.1.4 数値計算と SageMath

多くの数式処理システムでは数値行列処理が不得手です．数値計算で任意精度演算を採用していることもありますが, それ以上に数値行列計算ライブラリの最適化が不十分であること, さらには数値行列を処理するための構文や函数が貧弱なことが拍車をかけています．SageMath の任意精度数値計算では GMP 等の C ライブラリの利用だけではなく, 浮動小数点数の数値行列処理で NumPy を利用します．この NumPy は効率的な数値行列の処理が行えるように数値行列処理向けのライブラリ「**BLAS(Basic Linear Algebra Subprograms)**」[*3] として OSS の「**OpenBLAS**」が組込まれており, 高速な数値行列処理が可能になっています．また, 任意精度による数値計算についても GMP や GNU MPFR ライブラリを用いた演算が行われるために必要な精度を保った高速な処理ができます．さらに商用の数値行列処理システムである MATLAB 本体と同等の描画機能を実現するためのパッケージ Matplotlib も SageMath には組込まれており, 商用の数式処理システム *Mathematica* に匹敵する数式処理能力と業界標準の数値行列処理システム MATLAB 本体に匹敵する数値行列処理能力の双方を SageMath は兼ね備えています．ただし, SageMath はこれらの商用のアプリケーションに取って代ることを目的にしたシステムではありません．商用のアプリケーションはより洗練されたアルゴリズムや GUI, さらには徹底した最適化が行われていることに加えて, アプリケーションやライブラリを含めたファミリー展開も行っており, さらに過去の蓄積を越えることは容易ではありません．あくまでも商用のものに対する別の選択肢であることを目指しています．また, 後述のように Anaconda のパッケージとしても存在しているために, Anaconda の上位版のように思われるかもしれませんがそうではありません．たとえば, SageMath には標準的に用いられる Pandas パッケージが包含されていません．このパッケージは別途, pip を用いてインストールする必要もあります[*4]．このように SageMath は独自の数学環境です．

[*3] 公式標準実装は netlib(http://www.netlib.org/blas/) で公開されています．
[*4] ‘%pip install pandas’ で Pandas をインストールし, それから SageMath を再起動．

1.1.5　SageMath が使える環境は？

SageMath は各 Linux ディストリビューションや macOS 上で主に動作し, MS-Windows 上の場合は WSL 上で動作します．ここで SageMath には通常のコマンドライン・インターフェイスと Jupyter ノートブック形式のインターフェイスの二つの UI があります．ノートブック形式利用するためには Chrome 等のウエブ・ブラウザ, もしくは nteract[*5]等の Juputer ノートブック形式に対応したアプリケーションが必要です．

Linux ディストリビューションによってはパッケージとしても配布されており, ソースファイルから構築する必要がない場合もあります．また, macOS の場合は通常のバイナリによる配布や Homebrew を使ってインストールも行えます．さらに Anaconda のパッケージと仮想計算機版もあります．まず, Anaconda 版では conda を使ってインストールを行うことになります．ここで Anaconda パッケージリポジトリの利用は商用目的であれば有償ですが, miniconda+conda-forge として Python 環境を構築していれば, この有償化の問題には抵触しません．幸いにして SagaMath も conda-forge に収録されいるため, 商用目的の利用であっても問題なく使えます．したがって, miniconda+conda-forge で Python 環境が構築されているのであれば, この環境下で SageMath を導入するのが良いでしょう．

最後の SageMath の仮想計算機版は VirtualBox を利用したものであるため, 処理速度はネイティブの SageMath と比較して格段に劣ります．こちらはあくまでも SageMath のインストールが何らかの理由で出来ず, さらに Docker が導入できない場合に限定されるでしょう．

なお, SageMath のパッケージは 4G 程度, Docker イメージの CoCalc になると 15GB 程のディスク容量を必要とする大きなシステムです．そこでネットには繋っていてもこのような大きなパッケージを格納するだけの余裕がない場合の SageMath の導入方法もあります．それが SageMath の GUI を改良した商用の CoCalc [*6]を利用することです．この CoCalc はクラウドベースの SageMath で, ちょっとした計算であれば無課金で利用可能ですが, SageMath と比較して利用可能なアプリケーションを満載した圧倒的な数学環境になっています．この CoCalc はサポート無しの Docker のイメージとしても配布されており, 商用の CoCalc 程はアプリケーションが収録されていないものの通常の SageMath よりも一段優れた GUI 環境が使えます．この Docker イメージの利用は, Docker の起動後に ‘docker start cocalc’ でイメージを起動させ, ウエブ・ブラウザから ‘https://localhost/’ にアクセスすると CoCalc に接続され, そこから新規ならは利用者登録を行ない, そうでなければ Sign-in すれば CoCalc が使えます．

[*5] http://nteract.io

[*6] https://cocalc.com

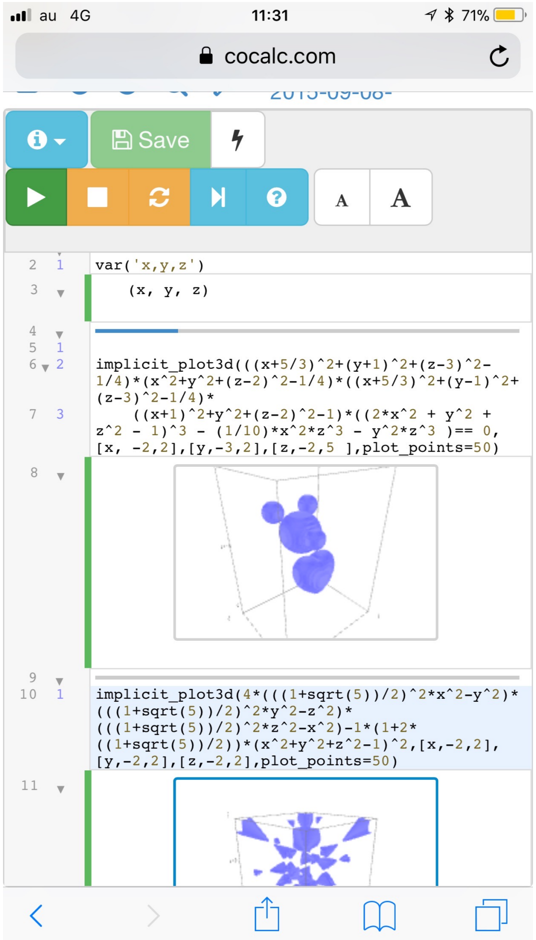


図 1.1　スマートフォンで利用

図 1.1 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．もちろん，Blutooth に対応したキーボードとマウスがあればより快適な操作環境が得られます．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．また，この他に MathLibre を利用する方法もあります．この MathLibre は Debian Light ベースの 1-DVD で立ち上がるシステムで，その ISO イメージは 4G 程度ですが，SageMath の他にも多種多様な数学アプケーションや TeX 環境，さらには数学アプリケーションに関する日本語文書を包含しています．MathLibre では SageMath を含めてさまざまなアプリケーションで遊べ，これらのアプリケーションの中から貴方にとって必要なものを見付けられるでしょう！ また，SageMath のためのディスクの領域を割けないのであれば USB メモリに MathLibre をインストールして，そこから起動する手段もあります．

1.2　SageMath の簡単な使い方

1.2.1　SageMath のユーザ・インターフェイス

SageMath は Python をその基盤としていますが, Python の言語的な側面は §3 で詳細を述べることとし, ここでは数学に関連する話に限定して幾つかの例題を示します. SageMath があらかじめ貴方の計算機に導入されていると仮定して解説しますが, そうでなければ CoCalc を試すと良いでしょう.

SageMath のフロントエンドには仮想端末上で動作する IPython とノートブック形式の Jupyter notebook と CoCalc のフロントエンド, それと ntract があります. IPython は標準の Python のシェルよりも履歴機能, ログ出力や GUI 等の機能が強化されていますが, 処理内容の保存はできません. Jupyter notebook のようなノートブック形式のフロントエンドではセル単位で式の入力と評価を行い, グラフや数式の計算結果も混在できます. そして, これらの処理結果を保存できます. なお, CoCalc で用いられているフロントエンドは Jupyter notebook を基盤としていますが, CoCalc 専用で SageMath には提供されていません. また, Jupyter では Chrome 等の WEB ブラウザをフロントエンドに利用しますが,nteract はノートブック形式のフロントエンドを提供する独立したアプリケーションのために, 立ち上げたのちにカーネルを選択するだけで使えます.

まず, SageMath の起動は UNIX 系の OS であれば仮想端末上で $\boxed{\text{sage}}$ と入力します. これで IPython をフロントエンドにして SageMath が立ち上がります. LibreMath, macOS や MS-Windows でラウンチャから立ち上げるとノートブック形式のフロントエンドが立ち上がります. 仮想計算機版の場合は最初に VirtualBox を立ち上げてから SageMath の仮想計算機を起動させますが, 立ち上げてしまえば仮想計算機のウィンドウいっぱいにノートブック形式のフロントエンドが立ち上がるために SageMath というアプリケーションが立ち上がったように見えます.

この本では対話的な処理では IPython をフロントエンドに用い, プログラムの記述や複数行にわたる処理の表示では Jupyter ノートブックのセルに記載したスクリプトやノートブックの様子を示します. まず, 仮想端末上で対話処理を行ったときの様子を示します:

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

ここで ‘sage:’ は仮想端末で利用するときの SageMath のプロンプトで, 式の入力は Python と同様にプロンプトに続いて入力します. 入力式や文は Python の書式で, Python のシェルと違って数式の先頭に空白文字が入っていてもインデント関連のエラーが出ません. とはいえ, 条件分岐, 反復文等の Python の文でブロック単位でインデント

を行わなければエラーになります．入力式の評価は仮想端末であれば Enter キー，ノートブック形式であれば式を入力したセル内で *Mathematica* と同様に Shift+Enter で行います．ここでの例では最初の行の処理が変数 p1 への 1 + 1 の計算結果の割当を行っていますが，Python/SageMath では割当で評価のエコーバックが行なわれず，名前 p1 を入力すると変数 p1 に割り当てられた値が表示され，Python/SageMath では入力行末尾に記号 “;” を置くと評価のエコーバックを行ないません．

1.2.2　ノートブック形式のフロントエンド

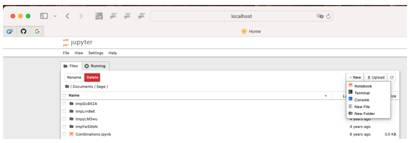


図 1.2　jupyter の Home

SageMath には標準で Jupyter notebook が利用可能です． このノートブック形式のフロントエンドで Chrome 等のウェブ・ブラウザを利用し，仮想端末から起動するときは ‘sage –notebook=jupyter’ で Jupyter,‘sage –notebook’ で Jupyter 以外のフロントエンドを指定していなければ既定値の Jupyter が立ち上がります．ここで Jupyter 以外に利用可能なものは Jupyter-Lab 程度です．図 1.2 に示すように Jupyter notebook の起動時に「Home」という名前のタブにホームディレクトリ上のノートやファイルの一覧が表示され，そこから既存のノートをクリックして開くか，図 1.2 に示すように右上の New を押して「Notebook」を選択して新規ノートを生成します．このときに「Select Kernel」と表示された小窓が開きますが，ここではプルダウンからインストールしている「SageMath」を選択します．

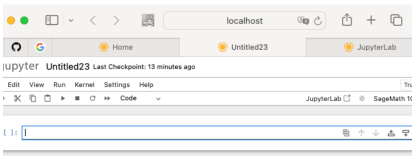


図 1.3　選択したカーネルの表示

この選択したカーネルは図 1.3 に示すようにノートブックの右上に表示されているので，処理を始める前にここを見て確認すると良いでしょう．

Jupyter ではセルが入力と評価の単位で，その評価は SHIFT+Enter で行い，結果は入力セルの直下に出力されます．なお，出力式はキャラクタを用いたアスキーアート風の表示が既定値で，図 1.4 に示すように ‘pretty_print_default(True)’ の評価後は評価結果が MathJax [*7] を用いてレンダリングされ，‘pretty_print_default(False)’ の評価後は既定値のキャラクタ式で表示されます．

[*7] 数式をウェブ・ブラウザ上で表示するための JavaScript ライブラリ．

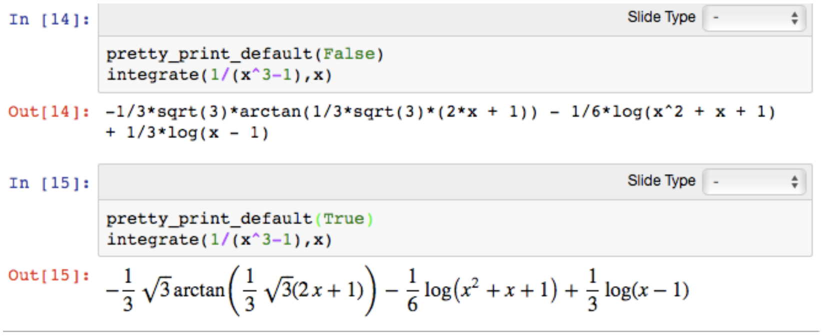


図 1.4　ノートブックでの数式の表示

また, セル単位で SageMath 以外のアプリケーションのフロントエンドに切替られます. 標準の SageMath では Python と仮想端末程度しか選択できませんが, 自分で Jupyter 向けのカーネルをインストールしていればインストールしたもの, CoCalc を利用する場合は数多くのアプリケーションが選択できます. さらにセルのモードを切替えて数式が扱える簡易的なワープロとしても使えます. そのためにはセルを既定値の Coding から Markdown に切替えてから Markdown 文書を書き込んで評価すれば文書がレンダリングされて表示されます. この Markdown は組版言語と呼ばれる文書の組版を指定できる言語で, Markdown に似たものに reSTructuredText(reST) があります. ちなみに Markdown はウエブ・ブラウザで表示するための比較的簡易な文書の作成, reST は Sphinx と組合せて, より一般的な文書作成を目的にしていますが, Jupyter で Markdown を使って記述された書籍も存在します[*8]. Markdown と reST では組版指示をアスキーアート風に行うために, 文書自体が一種の WYSIWYG を実現しています. 実際, 見出は行の先頭に記号 “#” を置き, “#” の数が見出のレベルに対応し, reST 風に行直下に文字数だけ記号 “=”, 行直下に文字数だけ記号 “-” を配置しても “#” と “##” を配置したときと同じ効果があります. そして, 本文の改行は行末に空白文字を二文字入れ, 箇条書は先頭に記号 “-”, “+” か “*” を入れて, 箇条書の入れ子の構造はインデントで表現します. また, 箇条書の先頭を数字にするときは数字のうしろに記号 “.” を入れますが, “1.” とするとレンダリング時に番号が自動で割り当てられます. 文字の強調は “**”, イタリックは記号 “*” で強調すべき箇所を括り, リンク先は見出を “[ ]” で括り, その直後に URL を記載します. 表もアスキーアート風に記号 “|” を縦線, 記号 “-” を横線として記述できます.

この他に独立したノートブック形式のフロントエンドを提供するアプリケーションの nteract も利用可能です. こちらは nteract を起動すると独自のノートブック形式のフロントエンドが立ち上がり, そこからメニューの「Runtime」からカーネルとして SageMath

[*8] 「Python で体験するベイズ推定」のように Git で公開されています.

を選択すると SageMath が直ちに使えます．使い勝手は Jupyter notebook と大差はなく, WEB ブラウザを用いないために簡易性は高いでしょう．ただ, Jupyter notebook と多少異なる挙動を示すこともあるので, ipywidgets を使う場合は Jupyter を用いる方が無難です.

この Markdown 文書の実例を挙げておきましょう．Jupyter で入力を行うセルでファイルメニューの「Cell」にて「Cell Type」を既定値の「Code」から「Markdown」に切替えて以下の内容をそのまま入力してみましょう:

```
# Markdownの例

## はじめに

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



1. 多項式：
   1. 一変数: $x^2+1$
1. 積分:
   1.  $\displaystyle{\int^1_0 x^2 = \frac{1}{3}}$
1. 微分：
   1. $\displaystyle{\frac{d \sin~x}{dx} = \cos~x}$
   1. $\Large{\frac{d^2 f(x)}{dx^2}}$

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

表
--

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

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

数式と同様に Shift+Enter で評価すると図 1.5 に示すレンダリング結果が得られます:

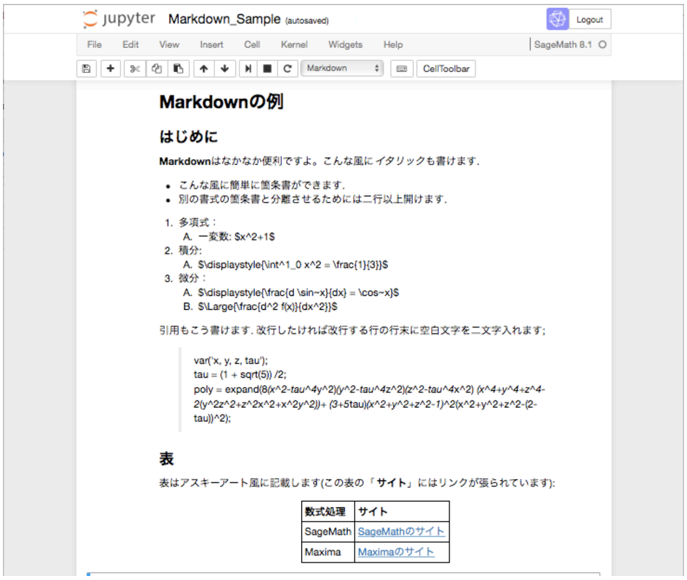


図 1.5　Markdown の例

レンダリングされた文書が表示されているセルをクリックすれば編集モードなって文書の編集ができます．このように Markdown は HTML や LaTeX のようなタグや命令を使わずに組版を持った文書を容易に作成できます．

SageMath のフロントエンドには命令入力の補完機能があります．これは入力中に TAB キーを押せば入力中の文節に合致する函数，メソッドや属性の候補の一覧が表示され，それに合致するものが選択できます．この機能は *Mathematica* と同様のもので，この点からも商用のものと比べてさほど劣るものではありません．この命令の補完機能に加えて，SageMath にはヘルプ機能があります．これは幾つかの函数や演算子の呼出で利用できます．まず，最も代表的な方法が Python と同様に函数 help() を使う方法で，たとえば，函数 expand() を調べたければ ‘help(expand)’ と入力すれば函数 expand() のヘルプ文書が読めます．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．Python 組込函数の函数 help() とは別に，Jupyter では記号 “?” もオンラインヘルプとして使えます．たとえば，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，さらに “??” でソースファイルを確認できます．たとえば expand?? と入力すると函数 expand() のソースファイルが確認できます．

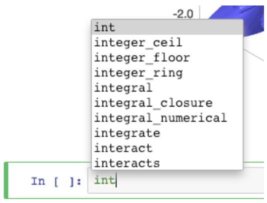


図 1.6　補完の様子

1.2.3 SageMath での数の表現

つぎに SageMath で扱う数について簡単に説明しましょう．高校までに扱う数に自然数, 整数, 有理数, 実数と複素数があり, これらの数は本質的に無限個ですが, 計算機はメモリ容量の上限があるために数の表現もメモリ容量が許す範囲内に限定されます．まず, 複素数は実部と虚部の対として表現されます．そこで, 実数をどのように計算機で扱うかが問題になりますが, ここで自然数, 整数は計算機内部で 2 進数として扱われます．そして有理数は整数対で表現すればこれも 2 進数を使って表現できることになります．それに対して 2 進数で表現できないような数の表現には工夫が必要になります．たとえば, 円周率 π は有理数で表現できない無理数ですが, 実際の生活では 3.14 と近似して利用しているように計算機でも同様にある桁数に制限した近似の数として表現するか, $2\cos^{-1}0$ と等しい数を意味する記号として表現します．数値計算を主に行うアプリケーションでは前者の方法である桁数で近似した数, つまり, 浮動小数点数と呼ばれる数として扱い, 数式処理であれば浮動小数点数による表現に加えて, ある方程式の解や函数の値であることを意味する式や記号として扱います．

SageMath では円周率 π, ネイピア数, 黄金率のように重要な数は定数としてあらかじめ登録されています．これらの重要な数には名前があり, 実際の処理で記号的な数として SageMath で扱われます．たとえば円周率は pi, ネイピア数は e, そして黄金率は golden_ratio, $\log 2$ は log2 といった名前を持ち, これらの数はその名前を SageMath に評価させても名前がそのまま返却され, メソッド n(), あるいは函数 N() を使って具体的な数値が表示されます:

```
sage: pi
pi
sage: N(pi, digits=3)
3.14
sage: pi.n(digits=6)
3.14159
sage: log2
log2
sage: log2.n(16)
0.6931
sage: log2.n(prec=16)
0.6931
sage: log2.n(digits=16)
0.6931471805599453
```

函数 N() は引数で指定した整数に対応する 10 進数桁数の精度で数を表示する函数で, メソッド n() がこの函数 N() に対応します．これらの函数とメソッドはオブジェクトの型

(クラス) を変換せずに指定された桁数の数を表示するために表示された数は同じでも等しくないという意味不明な事態が生じます．このことをオブジェクトの所属するクラス (型) を返却する函数 type() を使って確認してみましょう:

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

ここでの例では ‘N(pi, digits=3)’ で円周率 π を 3 桁表示させたものと ‘3.14’ という (リテラルと呼ばれる) キーボードから入力した数字を比較していますが同値ではありません．そこで，函数 type() で参照先のオブジェクトが何であるかを調べています．まず，‘N(pi,digits=3)’ が所属するクラスの sage.rings.real_mpfr.RealNumber は SageMath の任意精度の数のクラスで，リテラル ‘3.14’ で指示される数が所属する sage.rings.real_mpfr.RealLiteral は入力された数値が小数点 “.” を持つときに暫定的に所属するクラスで，任意精度の数 (real_mpfr.RealNumber) に変換できます．そして，これらの型の名前に現われる ‘real_mpfr’ は SageMath が用いている多倍長浮動小数点数ライブラリ MPFR に由来します．ところで，‘N(pi,digits=3)==3.14’ が False ですが，函数 N() やメソッド n() は指定した精度に対応する数として表示を行っているだけで，その他の桁数の部位もあるためです．また，実数を小数点付きの数字列として入力した時点ではリテラルであり，その型について指定がないときに SageMath で計算処理されると自動的に多倍長浮動小数点数の数として表現されます．ただし，多倍長浮動小数点数ではデータとしても大きく，その上，処理時間もかかるために CPU に実装されている倍精度浮動小数点数で処理することが望ましい場合があります．SageMath では倍精度浮動小数点数のクラスもあり，そのインスタンス化で構築子 RDF() を使います:

```
sage: a = 1.3
```

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

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

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

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

SageMath の自然数 $\mathbf{N}$, 整数 $\mathbf{Z}$, 有理数 $\mathbf{Q}$, 実数 $\mathbf{R}$ と複素数 $\mathbf{C}$ の数を初期化する構築子を以下にまとめておきます:

数の構築子 (コンストラクタ)

数	表記	事例
自然数 $\mathbf{N}$:	NN()	NN(1.0) $\mapsto$ 1
整数 $\mathbf{Z}$:	ZZ()	ZZ(1.0) $\mapsto$ 1
有理数 $\mathbf{N}$:	QQ()	QQ(1.4) $\mapsto$ 7/5
実数 $\mathbf{R}$(任意精度):	RR()	RR(7/5) $\mapsto$ 1.40000000000000
実数 $\mathbf{R}$(倍精度):	RDF()	RDF(7/5) $\mapsto$ 1.4
複素数 $\mathbf{C}$(任意精度):	CC()	CC(7/5+I) $\mapsto$ 1.40000000000000 + 1.00000000000000*I
複素数 $\mathbf{C}$(倍精度):	CDF()	RDF(7/5+I) $\mapsto$ 1.4+1.0*1

数の初期化では, 包含関係 $\mathbf{N} \subset \mathbf{Z} \subset \mathbf{Q} \subset \mathbf{R}$ で包含される数が対応する構築子を使って初期化され, これらの構築子の名前は環 (ring) と呼ばれる数学的構造と密接に関連します. また, 整数への変換では ZZ() よりも明示的な Integer() が使われ, 実数と複素数はその計算機上の表現で任意精度の浮動小数点数と倍精度浮動小数点数に分けられます. 数学

的構造と SageMath での実装と表現については §5 でその詳細を述べます.

1.2.4　算術演算

複素数は実数を使った式であるため, ここでは整数, 有理数と実数の算術演算式について述べます:

SageMath の算術演算子

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

SageMath の算術演算式は他の計算機言語とほぼ同様の演算子と書式です. 商の演算子 “/” では被演算子に浮動小数点数が含まれると浮動小数点数を返し, 整数同士で割り切れれば整数, そうでなければ有理数を返却する計算処理ですが, 被演算子が π のような SageMath の数学的定数, $\sqrt{2}$ のような代数的数のときは無理に近似の数として返却せずに, これらの記号や式をそのまま含む簡略化した式を返します. また, 被演算子が整数のときの演算子 “//” は整数の商を返します. それから演算子 “^” は Python では XOR 演算子ですが, この記号は TeX, 多くの数式処理システムや MATLAB 系言語で冪演算子として扱われているために SageMath で定義されたオブジェクトに対しては冪演算子として作用します.

SageMath の算術演算では, その式に複数の型の数オブジェクトが混在していても, 型の変換が自動的に行われます. この型変換は数の包含関係 $\mathbf{N}\subset\mathbf{Z}\subset\mathbf{Q}\subset\mathbf{R}\subset\mathbf{C}$ で結果が包含される最小の数のクラスのオブジェクトになるように処理されます. たとえば, 被演算子が自然数, 整数, 有理数のインスタンスのみで, その演算結果が整数になるときは整数の型で返却されます:

```
sage: 10 + 1/2
21/2
sage: 10 + 0.5
10.5000000000000
sage: 10 * 0.5
5.00000000000000
```

```
sage: 10 * 1/2
5
sage: (10 + I) * (10 - I)
101
```

ただし, 浮動小数点数の書式を持った被演算子を含む算術演算の結果は浮動小数点数を含む書式になります. これは浮動小数点数が近似の数であるためです.

整数, 有理数や実数では大小関係があります. Python では同じクラスのオブジェクトの同値性は演算子 "==" で調べられ*9, その値の大小関係は ">=", ">", "<=", "<" といった比較の演算子で調べられます. また, これらの演算子は和演算子 "+" と同様に被演算子の型が異っていても, 数学的対象として大小関係で比較可能であれば比較の演算子が使えます. ただし, 浮動小数点数は丸めによって異なる数が同士が同じ浮動小数点数として扱われることもあります. その程度は計算機イプシロンと呼ばれる 0 付近の浮動小数点数で与えられます. ただし, 浮動小数点数による実数の表現は, 0 付近が隣合う浮動小数点数の間隔 (ulp) が小さく, 絶対値が大ききなる程, その間隔は大きくなります. ちなみに数の絶対値が 2^{53} よりも大きくなると, 浮動小数点数の間隔は 1 を越えるために整数を一意に浮動小数点数で表現できなくなります.

算術演算子は C と同様ですが, 他の数式処理システム, たとえば Maxima や *Mathematica* にある階乗の演算子 "!" は SageMath になく, 函数 factorial() を使います. これは Python を用いていることに一因があります. この Maxima では利用者による演算子の定義も非常に簡単です. 実際, 演算子をラプラシアン Δf のように被演算子の前に演算子記号を配置する「前置演算子」であるか, 和 $1+2$ のような被演算子の間に演算子記号を配置する「中置演算子」であるか, 階乗 $n!$ のように被演算子の後に演算子記号を配置する「後置演算子」であるか, あるいは絶対値 "| |" のような被演算子を内包する演算子であるかを形式的にでも宣言するだけで目的の演算子が定義できます. しかし, SageMath は Python 言語の制約上, そこまで自由な演算子の定義が行えず, 記述子 infix_operator を使って中置演算子の定義が何とかできる程度です:

```
sage: def dao(x, y): return x * y + 1
sage: dao = infix_operator('multiply')(dao)
sage: 3 *dao* 4
13
```

*9 そのクラスで比較の特殊メソッドが定義されている場合で, それ以外はオブジェクトとして同一であるかが判断されます.

1.2.5　名前と変数

SageMath の基盤である Python が扱うデータは全てオブジェクトで, その値などの参照は「**名前 (name)**」を用います. この名前はオブジェクトの名札にたとえられ, この名前とオブジェクトの対応関係を「**名前空間 (name space)**」と呼び, この対応付けは<「**名前への束縛**」と呼ばれる演算子 “=” を使った操作で行われます. たとえば, ‘a = 1’ という Python の式は, 文字列 a が 1 というオブジェクトの名前であるという対応関係を与える式で, C や FORTRAN で説明される変数 a という箱にオブジェクト 1 を入れるような操作ではありません. したがって, ‘a = b = [1]’ という式の意味は, [1] という一つのオブジェクトに対して a という名前と b という名前の双方で参照できるという関係を入れるということで, そのために, a.append(2) を実行するとオブジェクト [1] に 2 が追加されることになります. その結果, 変数 a と b で参照しているオブジェクトは [1, 2] になります. この点は C や FORTRAN での演算子 “=” と大きく異なる点で, 注意が必要な点です. そして, 名前空間に含まれている名前の一覧は函数 dir() で確認できます.

数式処理では多項式や函数式に値が束縛されていない自由変数が用いられます. 名前とは別に自由変数として SageMath では “x” があらかじめ登録されています. そのために変数 x を使って ‘2*x’ や ‘cos(x)’ といった変数 x の多項式や函数式を直接, SageMath に入力できます. そこで, 函数 type() で x に対応しているオブジェクトが何であるかを確認しておきましょう:

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

この変数 x という名前に対応するオブジェクトは ‘sage.symbolic.expression.Expression’ という多項式のオブジェクトです. なお, x 以外に自由変数が登録されていないために, ‘y + x’ 等の x 以外の自由変数を持つ式を入力したければ, それら変数を函数 var() [*10]を使って変数宣言をしなければなりません:

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

[*10] 函数 var() は SymPy 由来の函数で, 自由変数 x, y の宣言は var('x, y') で行います.

1.2.6 多項式の扱い

前述のように変数 x があかじめ登録されているため, ‘x^2 - 1’ のような x の多項式や ‘sin(x)’ のような x を変項に持つ函数項を含む数式の処理が可能で, x 以外の変数が必要であれば函数 var() で宣言して使います. 入力された式は, 和, 差, 項に対する自明な積や商の結果を整理して, ある順序で並び替えを行った式を返却します:

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

この項や変数の並び変えは「**項順序**」に従って処理され, 特に指定がなければ「**辞書式順序**」と呼ばれる通常の英語の辞書と同様の並びで変数や項を並べる項順序が用いられ, その結果, 多項式は一意にその表示が定まります. この多項式を数式処理システムで効率良く処理するために, 単なる文字列ではなく, 演算子をその頂点とするグラフが用いられます. このグラフの一次元的な表示は Lisp の S 式に類似した演算子を先頭に置いたリストとして表現され, また, 二次元的に演算子を幹, 被演算子を葉に対応させた樹形図としても表現されます. そして, これらの表現を拡張すること通常の数式も表現できます. そのために数式の項の置き換えや冪の次数の取出といった処理は文字列の正規表現ではなく, 樹形図表現を使って処理を行うことになります. この数式の内部表現に関しては §6 にて詳細を述べます.

なお, 多項式同士の積や商, 式の展開は自動で行わないために, 式の展開は函数 expand(), あるいはメソッド expand() を用い, 式の因子分解は函数 factor(), あるいはメソッド factor() を用いますが, 式の冪乗の次数が整数でないときは式の展開ができません:

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

展開ができている $(x+1)^3$ と展開ができていない $(x+1)^{3.0}$ を函数 type() で調べてみても ‘symbolic.expression.Expression’ というクラスのオブジェクトです．式で異なるのは次数の ‘3’ と ‘3.0’ だけですが, ‘3.0’ に対応する SageMath のオブジェクトは浮動小数点数と呼ばれる書式の数であり, この数は本質的に近似値になります．多項式の展開のような代数的な処理を行うためには対象になる式を構成する各成分も基本的に整数, 有理数や代数的数に対応する SageMath の対象でなければなりません.

1.2.7　リストや集合の扱い

LISP や MATLAB のリストに対応する SageMath/Python のリスト型は演算子 “[ ]” を使ってオブジェクトや名前の列を ‘[1, 2, 2*x+1]’ のように括った書式で生成されます. なお, SageMath のリストの生成は Python そのものよりも拡張されており, たとえば, 1 から 10 までの自然数のリストの生成は ‘[1..10]’ でできます:

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

この例では自然数のリストを生成し, それからリストの成分を取出しています．この演算子 “..” は数式処理システム Maple にもある演算子ですが, SageMath で使えても Python では使えません．また, Python では配列, リスト等の添字は Maxima や MATLAB のように 1 からではなく, C と同様に 0 から開始します．Python では MATLAB 風の代表的なリスト処理である**スライス処理**と呼ばれる処理ができますが, 添字が 0 から開始するために添字が 1 から開始する MATLAB との違いに注意が必要です．たとえば, 5 成分のリスト L に対して先頭から 4 成分取り出すとき, Python では ‘L[0:5]’ , MATLAB で ‘L[1:4]’ とします, このように先頭が 0 で開始することに加えて, 演算子 “:” の右被演算子の整数に対応する添字の成分は Python では含まれず, MATLAB では含まれるといった違いがあります．添字を負の整数にするとリストの末尾, つまり, 右端からの成分が返却できます．たとえば, ‘L[-4:-1]’ で添字が-4, -3, -2 の L の成分, つまり L の右側から 2 番目, 3 番目と 4 番目のリストを返却し, ‘L[-1:-4:-1]’ で初期値が-1, 増分-1 で-1 から-4 までの

添字の成分リストを返却します．数値ベクトルと行列は, Python の数値計算向けの基本パッケージである NumPy の数値配列に変換することでより効率的に処理できます．この NumPy を使った数値計算に関しては §5.5 を参照して下さい.

SageMath の集合型は Python の集合型と同じで, SageMath の対象の列を記号 “{ }” で括って生成します．集合型はタプル型, リスト型と違ってその成分は一つだけに集約され, 明示的な順序がないために, その成分を添字を使って取り出せません．とは言え, 内部の成分に順序があるためにリスト型やタプル型と同様に for 文で利用できます.

1.2.8 線形代数

SageMath は NumPy を使うことで MATLAB 風の数値行列処理や倍精度浮動小数点数の多次元配列の迅速な処理ができます．ここでは多項式等を含む SageMath の対象で構成された行列の処理について述べ, NumPy の数値配列処理の詳細は §5.5 で解説します．まず, ベクトルの定義は函数 vector(), 行列の定義は函数 matrix() を使い, その際に成分の型が指定できます．この型の指定は数と同様で, 多項式や函数式を含むときは “SR” を指定し, その省略も可能です:

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

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


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

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

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

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

行列の転置はメソッド transpose(), あるいは函数 transpose() を使いますが, 行列の転置はいわゆるビューの切替で, 行列そのものの変換ではありません. そのために a.transpose() や transpose(a) の影響を行列 a が受けること, つまり, a.transpose() によって名前 a で指示されるオブジェクト自体は転置されません. 逆行列計算はメソッド inverse() や函数 inverse(), 数と同様に冪演算子 “^” が使えます. これらのメソッドや函数も行列オブジェクトそのものを変換するものではありません. 行列式の計算はメソッド det(), determinant(), あるいはこれらのメソッドの函数型からも行え, 固有多項式はメソッド charpoly() で計算できます:

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

[ 1,  2, 3 ]
[ 0, -2, 1 ]
[ 0,  0, 1 ]
sage: A.transpose()

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

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

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

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

そして, ベクトルの和, 差は演算子 “+” と “-”, スカラー積や行列同士, 行列とベクトルの積は演算子 “*” を用います. ここで構築子 vector() で生成したベクトルは縦ベクトルでも横ベクトルでもない単なるベクトルとして扱われ, 行列との積で行列の右側にあれば縦ベクトルとして, 左にあれば横ベクトルとして処理されます. ベクトルと行列の加法で ‘0’ は零ベクトル, あるいは零行列として作用し, 正方行列の加法で ‘1’ は単位行列として作用します. また, 函数 exp() を使って指数行列も計算できます:

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

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

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

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

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

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

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

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

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

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

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

1.2.9　解析

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

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

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

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

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

微分を行う函数 diff() には第 1 引数に函数, 第 2 引数に変数を記載します．ここで微分の階数が 2 以上であれば, 第 3 引数に階数を記載します．多変数の函数の微分では第 1 変数の微分の記載を終えると次に第 2 変数以降の変数と部分の階数を記載しますが, 階数が 1 のときは省略可能です．函数の積分では第 1 引数に積分すべき函数, 第 2 引数に積分変数を指示します．そして, 函数 integrate() の不定積分では積分定数の既定値が 0 です．なお, 記号積分は数値積分よりも難度が高く, 初等函数を項に持つ多項式の計算になると間違った結果を返すこともままあるために, 重要な計算であれば結果の妥当性を確認しましょう．返却された函数を描画するだけでも間違いが判ることもあります．たとえば, $\sqrt{x+1/x-2}$ の積分を行ってみましょう:

```
sage: f = integrate(sqrt(x + 1/x - 2), x)
sage: f
2/3*(x^2 - 3*x)/sqrt(x)
sage: diff(f, x).factor()
(x - 1)/sqrt(x)
sage: f1 = integrate(abs(x-1)/sqrt(x),x)
sage: f1
2/3*(x^(3/2) - 3*sqrt(x) + 2)*sgn(x - 1)
```

この結果は正しそうですが, 積分の結果を微分してみると安易に $\sqrt{(x-1)^2/x}$ を $(x-1)/\sqrt{x}$ で置き換えて積分した結果であることが判ります．実際, $\sqrt{x+1/x-2} = \sqrt{(x-1)^2/x} = |x-1|/\sqrt{x}$ であるために $x \geq 1$ のときは正解でも, $1 > x > 0$ と $x < 0$ のときに分子の符号が反転さていないと間違いです．この場合は, 与えられた式を $|x-1|/\sqrt{x}$ として積分すれば正しい答が得られます．次に $3/(5-4\cos x)$ を積分してみましょう．この式には極もないために十分に滑らかで連続な式が結果になる筈です．

```
sage: f = integrate(3/(5-4*cos(x)),x)
sage: f
2*arctan(3*sin(x)/(cos(x) + 1))
```

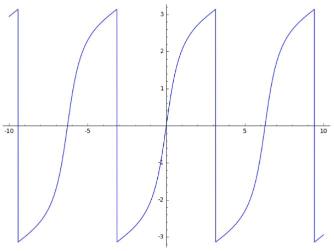


図 1.7 積分結果のグラフ

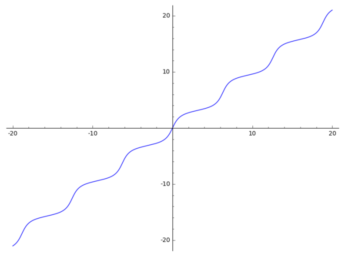


図 1.8 あるべき姿

結果を微分するともとの 3/(5-4*cos(x)) が返却されるために問題がなさそうですが, 結果のグラフは図 1.7 に示すように鋸の歯のようなグラフで, 図 1.8 のような階段状の滑らかなグラフではありません. この積分結果は $n \in \mathbf{Z}$ のときに $[\pi(n-1), \pi n]$ の範囲であれば間違っていませんが, これらの範囲を繋げると πn の前後の繋ぎ合せの処理が行われていないために結果として間違っています. この処理を行う函数を以下に示しておきましょう:

```
def correction1(x):
    ax = abs(x+pi)
    fl = sign(x+pi)
    n = fl*floor(ax/(2*pi))
    return 2*pi*(n-(1-fl)/2)*(heaviside(ax-n))
```

この函数は Heaviside 函数を用いており, 積分結果と合せることで全体で滑らかな函数が得られます. このように局所的に結果がもっともらしいときでもグラフを描くと大域的な間違いが判ります. これらの例から判るように記号積分は簡単ではありませんが, 適切な処理を加えれば正しい結果が得られます. このことは積分処理に限った話ではなく, 複雑な式はできるだけ単純で明瞭な式, より正確には計算機が解き易い式に変形すれば正しい結果により確実に到達できるようになります. そして, 計算結果も複雑な式になればなるほど鵜呑みをせずに, なんらかの確認手段でその妥当性を確認するべきです. このことは数式処理に限ったことではなく, 計算機に全てを任せるのではなく, 利用者が処理すべき問題を適切に分析し, その問題の特性から必要な処理を判断し, 適宜, その処理を手助けをしながら使いこなすことが重要であり, それを行うに十分な知識が必要です.

SageMath には無限大 inf があり, その無限大の表記としては ‘infinity’ と ‘oo’ の二通りです. この無限大を用いた実例を幾つか示しておきましょう:

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

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

1.2.10　方程式

ここではいろいろな方程式を解いてみましょう．まず，代数方程式は函数 solve() で解けます:

```
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．また, 連立方程式を解くときは式と求めるべき変数をリストで与えれば, 函数 solve() は連立方程式を解きます．ここで函数 solve() は可能であれば厳密解を求めますが, 代数的に解けなければ浮動小数点数で近似解を返却します．

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

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

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

1.2.11 代入

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

“==” の式として与えます.

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

1.2.12　グラフ表示

函数 plot() で 2 次元グラフ, 函数 plot3d() で 3 次元グラフが描画できます. そして, $y^2 - x^3 + 4xy + 1 = 0$ のような零点集合を描くための函数に implicit_plpt() と implicit_plot3d() があります. より高度な描画を行いたければ Matplotlib の函数が利用できます.

■函数 plot() と plot3d():　通常はこれらの函数が使われるでしょう. これらの函数は, 表示すべき函数や式を第 1 引数に, 以降, 変数の領域を指示するタプルやリストが並びます:

```
sage: var('x, y');
(x, y)
sage: plot(x^2 - x + 1, (x, -2, 2))
sage: plot3d(x * y, (x, -2, 2), (y, -2, 2))
```

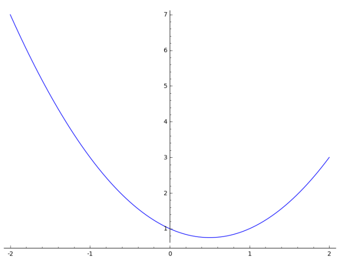


図 1.9　$x^2 - x + 1$ のグラフ

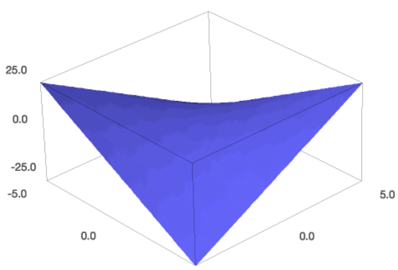


図 1.10　xy のグラフ

なお, 3 次元グラフはマウスを使って回転や拡大も行えます．この 3 次元グラフの描画では JMol が用いられていおり, 仮想端末から 3 次元グラフの表示を行うときは JMol そのものが立ち上がります.

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

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

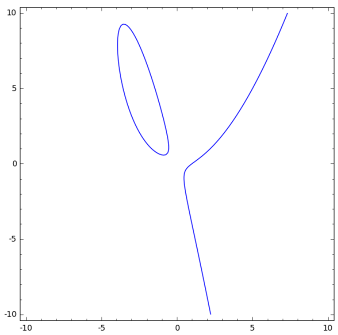


図 1.11　implicit_plot の例

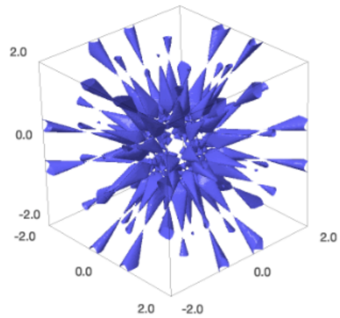


図 1.12　implicit_plot3d の例

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

　また, CSV データをファイルから読み込んで表示できます．ここでは NumPy の函数 loadtxt() を使って CSV 形式のファイルを読込み, 函数 list_plot() で表示します:

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

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

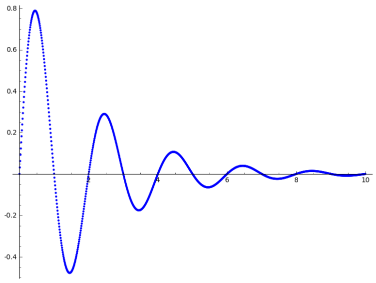


図 1.13　list_plot の例

グラフィックス・オブジェクトに関しては和を処理できます. つまり, 複数のグラフィックスオブジェクトを足し合せて同一グラフ上に表示できます:

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

この例はさきほど読み込んだ CSV データのリスト L を再度, 函数 list_plot() で描画し, そのときに生成されるオブジェクトを P に保存します. 同様に函数 plot() で生成した描画オブジェクトを Q と R に保存し, これらのオブジェクトの和 P+Q+R をメソッド show() で表示すると次の重ね合せたグラフが得られます:

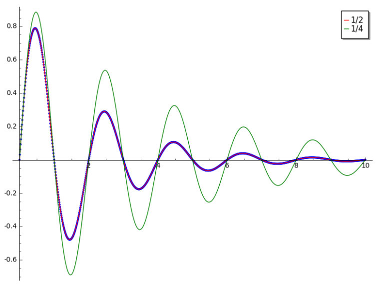


図 1.14　グラフィックス・オブジェクトの和の例

1.2.13　微分方程式の数値的解法

被食者・捕食者の個体数のモデルで有名な「**Volterra-Lotokka の連立微分方程式**」で遊んでみましょう．この連立微分方程式は被食者 (たとえばシマウマ) の個体数を x, 捕食者 (たとえばライオン) の個体数を y としたときに次で与えられます:

Volterra-Lotka の被食者・捕食者モデル

$$\begin{aligned} \frac{dx}{dt} &= (A - k_1 y)x \\ \frac{dy}{dt} &= (k_2 x - B)y \end{aligned} \tag{1.1}$$

まず, 微分方程式のベクトル場と解曲線を描いてみましょう．SageMath でベクトル場の表示は函数 plot_vector_field() で描けます:

```
sage: var('x, y')
sage: f=[(10 - 2 * y) * x, (3 * x - 10) * y]
sage: plot_vector_field(f, (x, 0, 20), (y, 0, 20), color="red")
```

図 1.15 に示すように $(5, 3\frac{1}{3})$ を中心とする周期性が見えます．このベクトル場に解曲線を追加しましょう．この解曲線の計算では函数 desolve_odeint() を使います．まず, 第 1 引数に微分方程式, 第 2 引数にその初期値, 第 3 引数に出力数値解の範囲, 第 4 引数に解くべき変数をリストで与え, 以降, 精度の指定などの解析のための変数指定を行うだけで数値解を計算します:

```
sage: ci1=[0.2,0.4]
sage: ci2=[4,3]
sage: t=srange(0,10,0.01)
sage: v=[x,y]
sage: sol1=desolve_odeint(f,ci1,t,v,rtol=1e-3,atol=1e-4,h0=0.1,hmax
    =1,
....:                    hmin=1e-4,mxstep=1000,mxords=17)
sage: sol2=desolve_odeint(f,ci2,t,v,rtol=1e-3,atol=1e-4,h0=0.1,hmax
    =1,
....:                    hmin=1e-4,mxstep=1000,mxords=17)
sage: dx=plot_vector_field(f, (x,0,25), (y,0,35), color="red")
sage: d0=list_plot(sol1,color="blue")
sage: d1=list_plot(sol2,color="green")
sage: (dx+d1+d0).show()
```

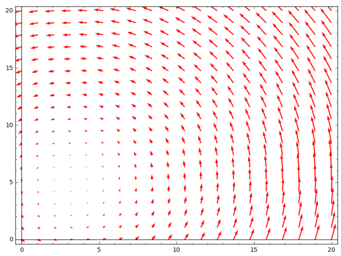


図 1.15　plot_vector_field の例

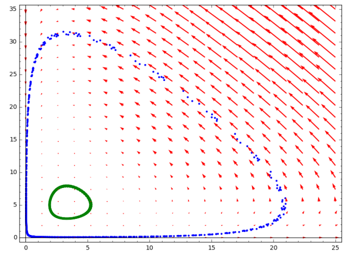


図 1.16　Volterra-Lotka 方程式の数値解とベクトル場

この程度のことは Python+NumPy+SciPy+Matplotlib でもできますが, ベクトル場の表示のためにメッシュの生成を行ったりと, SageMath のように二つの函数だけでは済みません.

ちなみに Jupyter notebook では ipywidgets を使って簡易的な GUI が導入できます. ここではチェックボックスとスライトバーを入れてみましょう:

```
from ipywidgets import interact
var('x, y')
@interact(x0=(0, 25, 1), y0=(0, 35, 1), contourf=False)
def VL(x0, y0, contourf):
    f=[(10 - 2 * y) * x, (3 * x - 10) * y]
    ci=[x0, y0]
    t=srange(0, 10, 0.01)
    v=[x, y]
    if contourf:
        d0 = plot_vector_field(f, (x, 0, 25), (y, 0, 35), color="
            yellow")
        dx = contour_plot(sqrt(f[0]^2 + f[1]^2),(x, 0, 25),(y, 0,
            35),
                    cmap='hsv',contours=128)
    else:
        d0 = plot_vector_field(f, (x, 0, 25), (y, 0, 35), color="
            red")
        sol = desolve_odeint(f, ci, t, v, rtol=1e-3, atol=1e-4,
                    h0=0.1, hmax=1, hmin=1e-4, mxstep=1000, mxords
                        =17)
        dx = list_plot(sol, color="blue")
    (dx+d0).show()
```

この例ではチェックボックスにチェックを入れると図 1.17 に示すようにベクトル場のベクトルの大きさに合せたコンター図を表示し, チェックを外すと図 1.18 に示すようにベクトル場と解曲線の双方を描きます:

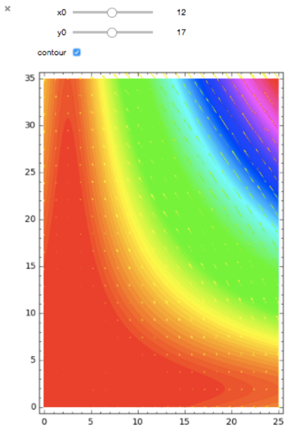


図 1.17 コンター図とベクトル場

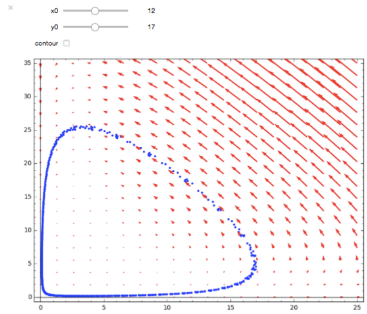


図 1.18 数値解とベクトル場

このように Jupyter notebook の利用では簡易的なウィジェットを使ってセレクタやチェックボックス等を扱えるため，これらを用いて簡易的な GUI を持ったノートブックを作成できます．

1.2.14 画像の簡単な処理

ここでは簡単な画像処理の例を示しておきましょう:

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

最初にパッケージ Matplotlib のモジュール pyplot を ‘plt’, モジュール image を ‘mi’ と読み替えて SageMath に読込みます．pyplot と image の読込み方に違いがありますが，これらはパッケージを構成するモジュールの呼び出し方の例で，パッケージ/モジュールの階層構造に依存するものです．まず，Python ではパッケージ，モジュール等の階層構造を表現するために区切文字として記号 “.” を用い，左側に上層，下側に下層を記載します．最初の読込では，pyplot の上位が matplotlib であるためにパッケージ内部の階層を含めた表記は matplotlib.pyplot で，これで pyplot の所在が分かります．次の文は matplotlib の配下からモジュールを指定して取り出す文で，はじめに from 節を使って ‘from matolotlib’ とし，パッケージの階層構造でのその下にあるモジュール image を読込むという意味で

す．ここでの例では共に as 節を用いてモジュールの読み替えを行っています．こうすることで本来なら ‘matplotlib.image.imread()’ とモジュールに含まれる函数等を呼出さなければならないところを ‘mi.imread()’ と簡易に済ませられます。

この函数 imread() で読込まれた画像データは ‘numpy.ndarray’ 型, つまり, NumPy の多次元配列です．この数値配列の大きさは属性 shape で調べられますが, ここでは ‘(509, 800, 3)’ と返却されています．この結果の最初の二つの整数値が画像の縦と横の画素数で, 最後の ‘3’ が Red, Green, Blue の RGB に対応します．ここでは SageMath の函数 matrix_plot() で画像の表示を行っています．この表示で SageMath を仮想端末で利用していれば画像表示のためのウィンドウが開かれ, Jupyter を利用していれば図 1.19 のようにノートブック側に表示されます:

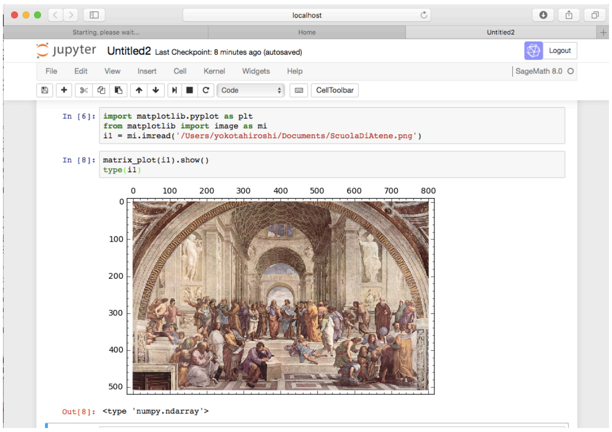


図 1.19　読込画像の表示例

次に RGB 画像の輝度のヒストグラムを赤, 緑と青で出力してみましょう．ヒストグラムのグラフ出力は SageMath の函数 histogram() でできます．この函数 histogram() は 1 次元数値配列のヒストグラムを描きますが, 画像は 2 次元です．そこで画像データを NumPy の多次元配列のメソッド reshape() を使って 1 次元化し, ヒストグラムの棒を 16 本にして描きます:

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

```
sage: (hR + hG + hB).show()
```

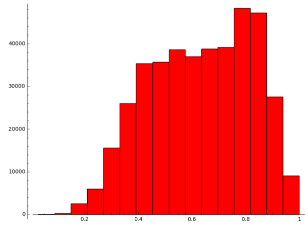


図 1.20　赤のヒストグラム

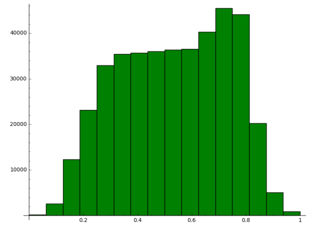


図 1.21　緑のヒストグラム

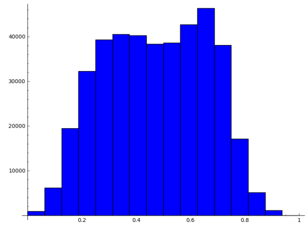


図 1.22　青のヒストグラム

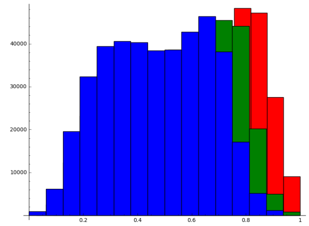


図 1.23　RGB のヒストグラム

ここでは函数 histogram() が出力するグラフィックス・オブジェクトを hR, hG, hB に割り当て，それらをメソッド show() を使って表示させています．このヒストグラムの例で imat[:,:,0], imat[:,:,1], imat[:,:,2] という表記があります．添字の 0, 1, 2 は赤 (R), 緑 (G) と青 (B) の輝度に対応する配列であることを指示し，他の添字記号 ‘:’ は，その添字記号が置かれた場所で添字が取り得る値の全てを意味する MATLAB 系言語を特徴づける構文で，Python でスライス・オブジェクトと呼ばれる文を構成するオブジェクトです．この構文は配列の i から j までの $j-i$ 個の添字で指示される成分を取り出す構文です．なお，MATLAB では添字が 1 から開始するため，Python の ‘a[i:j]’ と同値な MATLAB の表記は ‘a(i+1:j)’ です．また，添字の増分が 1 に固定されていますが，増分を 1 以外に設定するとき，Python では ‘10 : 0 : −1’ のように ⟨始点⟩ : ⟨終点 + 1⟩ : ⟨増分⟩ と表記し，MATLAB 系の言語では ⟨始点⟩ : ⟨増分⟩ : ⟨終点⟩ と始点と終点の間に増分を挟みます．

　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 の赤を指示するために図 1.24 に示す赤の輝度を示す画像です:

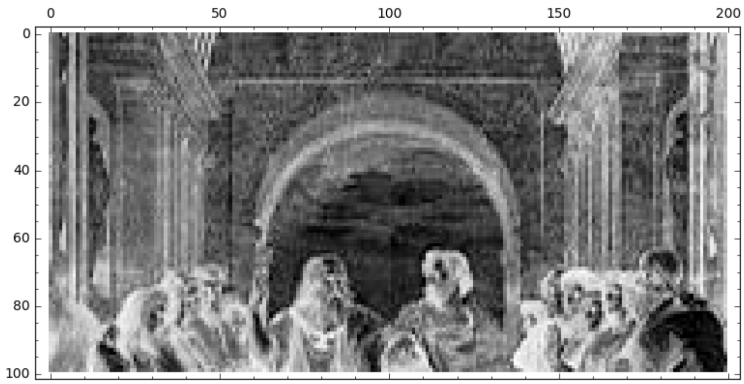


図 1.24　画像の一部切取り

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

いかがでしょうか? SageMath を導入すれば, まっさらな Python 環境にさまざまなパッケージを取捨選択しながら導入する必要もなしに, 本格的な数式処理, 数値計算や統計処理をノートブック形式フロントエンドで利用できことが理解されたでしょうか.

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られた数学のための統合環境ですが, この Python は各アプリケーションのインターフェイスとして入出力の面倒を見たり, その処理言語として用いられているだけではありません. 実際, SageMath に実装された数学的対象自体も Python のクラスとして表現されており, だからこそ SageMath を使いこなすためには 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「**Python はどのような言語なのか?**」という問いかけに対し, その手始めとして Google 等の検索エンジンや Wikipedia が参考になるでしょう. そこで Wikipedia で Python の項目を見ると, Python の作者はオランダ人の**グイド・ヴァンロッサム (Guido van Rossum)**[*1] が作った OSS (Open Source Software) であり, オブジェクト指向プログラミングに対応した言語であること, その名前の由来として 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, 核になる構文や文法を必要最小限に抑えていること, そして, さまざまなライブラリがあることが書いてあります. この他のに Python それ自体の開発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るでしょう[*2].

では,「**オブジェクト指向プログラミング (Object Oriented Programing)**」はどのようなものでしょうか? 再度, Wikipedia を見ると「**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**」とありますが, これだけでは「**メッセージとは何?**」,「**オブジェクトって何?**」,「**どのような技法?**」といった疑問が湧いてきます. ここでは「 **オブジェクト**」を「**計算機上で扱う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「**抽象化**」と「**関係**」といった用語の説明をあとまわしにして暫定的に定義すると,「**どのような技法?**」が残ります. 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. ここで最初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, 実際に処理すべきデータを対応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表

[*1] カレル・チャペックの戯曲「ロボット (R.U.R)」はRossum's Universal Robot(ロッサムの汎用ロボット) で, 発明家のロッサムが創ったロボットをベースに製品化したロボットを作成する会社の話ですが, 何となくそれを思い出してしまいます.

[*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したが, PEP-572 の議論が原因になって 2018 年 7 月に引退されました. https://mail.python.org/pipermail/python-committers/2018-July/005664.html

現します．ここでクラスには属性値と呼ばれる値とメソッドと呼ばれるインスタンスに作用できる手続があり, これらでオブジェクト間の関係が与えられ, さらにクラスには下位のクラスで上位の属性やメソッドを利用する継承と呼ばれる機能と, それを実現させる親子関係に類似した階層構造があり, このクラスに基く言語に Python の他に Java や Ruby があります．そして, プロトタイプに基く言語ではクラスを構築せずに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．これらの技法を実例でたとえるなら, ある証明書を作成するときに, その証明書がどのようなもので, しかも, その書式がどのようなものであるかを明らかにした上で, 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, 適宜修正して再利用する方法がプロトタイプに基く方法とたとえられるでしょうか. では, 一時的に保留した「**抽象化**」と「**関係**」はどのようなもので, 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ることを目的とします.

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

クラスに基くオブジェクト指向プログラミングの説明で, プラトン (Πλάτων, Plato) [*3] の「**イデア論 (Theory of Forms)**」がしばしば引き合いに出されます．このとき,「**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, そのイデアの現世での像が「**みけ**」です．この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ちます．このようにイデアは現実にある対象を「**理想化**」し, ちょうど,「**現実のモノ**」の「**鋳型**」に対応します. さらに, プラトンはイデア界こそが真実の世界であり, 現世はイデアが投影された「**影の世界**」,「**模倣物 (εἰκών) の世界**」とさえ言っています (c.f.「洞窟の比喩」[30])[*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, それに対応するクラスが計算機内部で「**実体化したもの**」と説明されす. ちなみにクラスがプログラムで処理すべき対象として「**実体化**」することを「**インスタンス化 (instantiation)**」, そして,「**実体化したクラス**」を「**インスタンス (instance)**」と

[*3] 体格が良くて肩幅が広かったこと (πλατύσ) 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.

呼びますが,「**イデアの現世における実体化**」も英語では同じ「**instantiation**」であり,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.こうなると誰がどのような理由でイデアを実体化するのか気になりますが,プラトンは「**デーミウールゴス (δημιουργόσ, demiurge)**」がイデアを実体化した張本人であり,この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述べています,ところで,このイデアは美や善に関わるものであるために,醜いものや悪にイデアは存在しないともプラトンは述べていますが,それならば「**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もなるでしょう.

図 2.1 ヘルメス・トリスメギストス

このような「**機械仕掛けの神 (Deus ex machina)**」[*5]を持ち出されて信じるしかない点は哲学よりも宗教であり,実際,イデア論はヘレニズム世界の宗教に大きな影響を及ぼします.まず,プラトンのイデア論を基に超越的な「**一者 (το ἕν, to hen)**」からの流出による世界の創造というプロティノスの「**流出説**」を取り入れた「**新プラトン主義**」[*6]からはデー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て星辰の支配を受ける人間が死後に神への魂の帰一することを柱にする「**グノーシス主義 (Γνωσισ)**」に繋がります.たとえば,ヘルメス・トリスメギストス (三重に偉大なヘルメス, Hermes Trismegistus, Ἑρμῆς ὁ Τρισμέγιστοσ) が記したとされる「**ヘルメス文書**」と呼ばれる一群のグノーシス主義の文書があります.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 (Θώϑ)[*7]がヘレニズム時代にエジプトで融合したと考えられており,錬金術では「**賢者の石**」[*8]を実際に手にした人物[*9] とされています.そのヘルメス文書の一つの「ポイマンドレース (Poimandres)」[16] には,人間は美しい神の似姿として創られた神の子でしたが,あるとき彼は天上界から地上へと降下します.その

[*5] 古代ギリシャ・ローマ悲劇で収拾がつかなくな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,たとえば,ソフォクレース (Σοφοκλήσ) の「ピロクテーテース (Φιλοκτήτησ)」では終盤にヘーラクレース (Ἡρακλησ) が現れて事態を一刀両断で解決してしまいます.とは言え,この物語の結末は当時の観客にとって既知のことで,水戸黄門の「葵の印籠」を待つような心情だったのでしょう.

[*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霊薬で,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**」であり,「*A* **ニシテ** Ω」です!

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, 最後にたどり着いた地上[*10]にてフュシス (φύσισ, 物質) 内に写った自分の姿に恋した結果, 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**」ために人間はフュシスに捕えられ, その肉体を牢獄とする存在になったとあります. この伝説[*11] は人間が神の似姿のために本質的に不死である一方で, 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とイデア論を中核に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」等の対話で人々を正しさへと導こうとするソクラテスから「ポイマンドレース」の自説への反駁を一切許さない高圧的なヘルメスとの違いにも伺えます. 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全知全能の神が半端なことをする筈がないと反駁されます. それどころか黙示的な宗教であったキリスト教は, 新プラトン主義の「一者」を取り込んで徐々に合理的な宗教へと変貌します[*12]. 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[*13]であった教父アウグスティヌス (Augustinus Hipponensis, Augustine of Hippo) が新プラトン主義をキリスト教神学の理論付けに用いたことが大きく影響しています. このようにキリスト教と古代の間には大きな断絶がある一方で, 地球中心の同心円による階層的な宇宙観, 星辰信仰やイシス信仰をマリア崇敬として引継ぐ等, 直接, あるいはイスラム文化を介した間接的な方法等でヘレニズム文明の遺産を引き継いでいます.

さて, 本筋に話を戻すとプラトンのイデア論はオブジェクト指向プログラミングのクラスとインスタンスの双方の関係に類似がみられる程度です. 実際, 真っ新なシステムで「**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, そもそも, クラスは「**神聖ニシテ侵スヘキアラス**」な超越的な代物であるよりも, 現実の対象から抽出され, さらに「**どのようなものであるかを語られるもの**」であるべきです. そして, このようなものに「**概念**」があります.

[*10] この宇宙観は同心円状の階層構造を有する天動説です.

[*11] 宗教・宗教的な代物は「**伝説**」を続々と生成します. 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聖者とそれにまつわる伝説を生産するという有様で, 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] その際にグノーシスの影響にあった教義の排除が行われています. たとえば,「ユダの福音書」等を含むナグ・ハマディ文書は瓶に入れて洞窟に埋められています. とは言え, 表面からは消えてもその痕跡は伏流として残り, 10 世紀のカタリ派のように時々, 表舞台に現れます.

[*13] 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. ちなみに胡瓜は光を含んでいて良い食べ物だそうです.

2.2.2 概念について

「**概念 (concept)**」は「**それが何であるか?**」や「**それがどのようなものであるか?**」という問に対する回答で, そのものを特徴付ける形や色や機能等をまとめたもの, つまり, 概念は対象を特徴付ける「**徴表**」から「**属性**」を抽出し, これらの属性を共通性で纏めたもので, 概念はイデアのように超越的ではなく, 人間が認知し得る対象の形や色といった具体的な特徴から出発し, 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, ロゴス, account)**」です.

概念は「**名辞 (term)**」としても現れますが, 名辞は概念が載る器であって概念そのものではありません. また, 概念は超越的なイデアと異なって人間が認知し得る具体的な事物から出発し, それら対象を我々がどのように語るかという説明規定であるため, 対象への理解が深まれば語られる内容も深まるという特性を持ちます. この「**それが何であるか**」と「**それがどのようなものであるか**」といった問いかけにより深く考察した人物がプラトンの弟子のアリストテレス (Ἀριστοτέλησ, Aristotle) です.

図 2.2 アテネの学堂より: プラトンとアリストテレス

図 2.2 に示すラファエロ・サンティ (Raffaello Santi) の「**アテナイの学堂**」[64][*14]はアリストテレスとプラトンの方向性の違いを判り易く図解したものとして有名で, プラトンは天上 (イデア=抽象) を, アリストテレスは地上 (形相=具象) を指で示すことで両者の思索の方向性の違いが表現されています[*15]. だからと言って, イデアが天界に安住する一方で概念が地べたを這い回る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りうるという性質をアリストテレスの論理学に由来する伝統的論理学では「**普遍**」と呼びます[*16]. 具体的に

---

*14 幾つか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て「**古代神学**」と呼ばれます. これを契機に新プラトン主義は大きな影響をイタリアの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.

*15 ここで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. そもそも, 古代人にとって永遠の運動は天空を移動する太陽や月であり, 直線運動は始点と終点持つために永遠ではありません. そして, 最も地球の近くを円運動する月から永遠の世界が開始すると考えています.

*16 いろいろ取り替えて使える道具の名前で「**ユニバーサル**」を冠する理由は, このように主語を取り替えられる性質に擬したものです.

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ます. このように「猫」は「普遍」ですが, もう一方の「**みけ**」や「**たま**」は個体に強く結びついていて,「**猫はたまです**」とは主張できずに「**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で普遍でありません. 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. ところで「猫」には「三毛猫」,「黒猫」,「白猫」,「虎猫」といった「猫」があり, これらは「猫」の毛並から, より詳細に「猫」を説明しようとする意図があります. このように概念には「**類似する個体とまとめてより包括的に説明しようとする概念**」, すなわち,「**個体から離れた側の概念**」, それから逆に「**個体をより詳しく説明しようとする概念**」, すなわち,「**個体に近い側の概念**」の二種類があり,「**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. このように包含する側の概念を「**上位概念**」, もう一方の包含される側の概念を「**下位概念**」と呼びます. 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**」が下位概念です. そして, 上位概念を「**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*17. 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」です. そして種の違いを示す徴表 (特徴) を「**差異 (種差)**」と呼びます*18. 先程の「三毛猫」,「虎猫」, ... の例では「毛並」の違いが差異です. そして, 概念にはその上限と下限があり, 上限になる最上位の概念を「**範疇 (カテゴリー, Category)**」, 下限になる最下位の概念を「**単独概念**」, あるいは「**個体概念**」と呼びます. この個体概念は個体を直接指示する個体に最も近い概念で, 範疇は個体を含む概念の中で最も普遍的な概念です*19.

このように概念は対象を説明する一方で普遍性も目指しており, その点ではイデアに似ています. そのこともあって, イデア論が秘儀と化しはじめた新プラトン主義の哲学者達は, 何かとイデア論に批判的なアリストテレスを「**師の思想を秘匿するために批判していた**」と捉えはじめます. ここで幾度も言及している「**手引 (エイサゴーゲー,Εἰσαγωγη′,Isagoge[45])**」[56][45] は, 三世紀に新プラトン主義の創始者であるプロティノスの弟子であるポルフュリオス (Πορφύριοσ, Porphyry of Tyre) が弟子にカテゴリー論の解説書を書いて欲しいと請われてギリシャ語で著述した小冊子で, 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た劃期的な著作でもありま

*17 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.

*18 「**種差**」とも呼びますが,「カテゴリー論」[2] では「**差異特性**」と訳しています. また, ポリピュリオスの「手引」[36] では種を構成するような差異を「**種成的差異**」とさらに差異を分類しています. ここでの説明では種成的差異のことであるために種差でも良いでしょうが, διαφορά が二つの訳語を持つのもどうかと思うために「**差異**」としています.

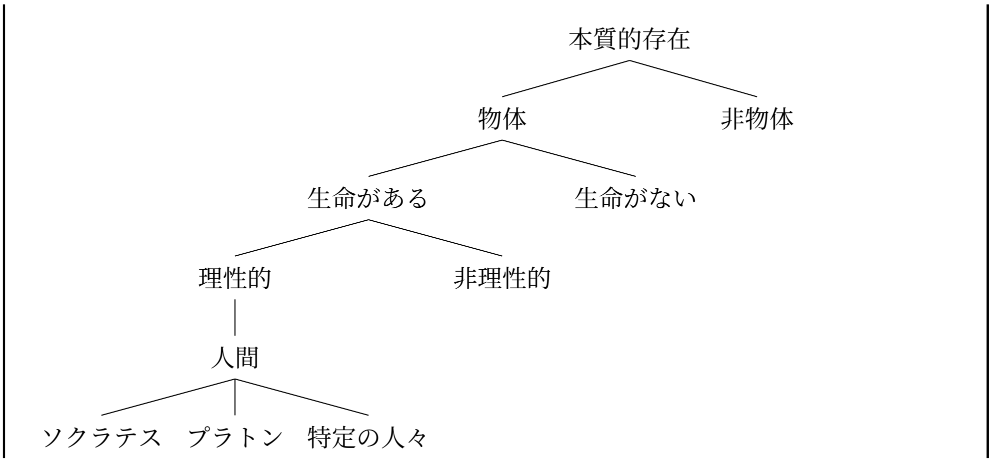
*19 アプリケーションのメニューで最上段が「**カテゴリー**」という名称で分類されている理由です.

す．この小冊子は (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, その後も長く哲学の入門書として使われ, 特に西ヨーロッパでは西ローマ帝国崩壊後に僅かに残った哲学書の一つです.

このポルフュリオスの「手引」から幾つか説明をしておきましょう．まず,「**ものごとを語る**」ことに「**類**」,「**種**」と「**差異**」に加えて「**特有性**」と「**偶有性**」があるとポルフィリオスは述べています．ここで「**類 (genus)**」と「**種 (species)**」はそれぞれ「genos, γένος」,「eidos, εἶδος」とアリストテレスは呼んでおり, これらの語源に「**形**」という意味があることから概念が対象の形から抽出されるものであったことが伺えます．そして, 類が種の上位概念, 種は類の下位概念に対応し, 共に「**それが何であるか?**」という問への回答になります．つぎに「**差異**」,「**特有性**」と「**偶有性**」は「**それがどのようなものであるか?**」という問への回答です．まず,「**差異**」が「**ものを特徴付ける属性**」です．たとえば,「動物」を類とする「人間」という種が他の「猫」等の動物との相違点として「理性を有すること」が挙げられ, この「理性を有すること」が差異になります．ちなみにアリストテレスは「類-差異」, ボエティウスは「類-種」で述べており, 類と種の階層構造にも言及しています．それから「**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**」であり, たとえば,「下町のナポレオン」や「貴志駅の猫駅長」といった個体が持つ属性です．そして, 最後の「**偶有性**」は「**その程度を語られるもの**」であり, 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」のように, その程度が表現可能であり, さらには「日焼けしていない」といった属性を持たない状況も考えられるように, 他の属性と違って形や模様, あるいは量の把握が可能な「**感覚的に把握できる属性**」です．さらに感覚され得るものは変化するものであることから偶有性は永遠不滅な属性ではなく, 都度, 変化する属性です．このように偶有性は対象を特徴付けるものですが, その対象の本質ではありません.

ポルフュリオスによる述語の「**類**」,「**種**」,「**差異**」,「**特有性**」と「**偶有性**」を表現したものが「**ポルフュリオスの樹 (Arbor Porphyrianae)**」です．ただし, 実際に図示したのはポルフュリオス自身ではなく後世の注釈者で,「**ポルフュリオスの樹**」という言葉自体は 6 世紀の Sergius of Reshana の著作まで遡れ, 古代の註釈者は「鎖」,「線分」等と呼んでいます.[45]．ここで人間に関するポリフュリオスの樹の一例を示しておきましょう:

人間に関するポルフュリオスの樹



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, 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．たとえば, リンネ (Carl von Linné) による学名の命名方法は「**二名法**」と呼ばれ, 動物/植物が属する種の名前とその種を包含する属の名前を使います．これらの名前の配置は, 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します．実例として人類の学名 ‘Homo sapiens’ では属が Homo, 種が sapiens 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も用いられて, 全体を俯瞰するときには樹形図が用いられています．

ここで, アリストテレスの西ヨーロッパでの受容について簡単にその歴史を述べておきましょう．まず, 五賢帝のハドリアヌスの頃のローマ帝国においては知識人はギリシャ語を当たり前のように使っており, この傾向は後の東ローマ帝国の公用語がギリシャ語になることにも繋がります．ところが, 西ローマ帝国滅後のイタリア半島では状況が異なり, 以前と異なってギリシャ語が判る人は逆に少ない状況になっています．前述の手引をラテン語に翻訳したボエティウス (Boetius) は「**最後のローマ人**」と呼ばれる人物ですが, その一方では「**最初のスコラ学者**」とも言われます [27]．彼はプラトンの開設したアテネのアカデメイア (Ἀκαδεμηια) で哲学を学んでおり, ギリシャ語も堪能で, 彼は東ゴート王国のテオドリック大王に高官として仕えます．彼は高官としての勤めを果す一方で, アリストテレスの著作のラテン語への翻訳や後に「**普遍論争**」知られる西ヨーロッパの哲学上の論争の発端となる哲学論文を書いたことでも知られています．彼がテオドリック大王の宮廷における陰謀に巻き込まれて刑死することがなければ, より体系的なアリストテレスのラテン語文献が西ヨーロッパ残されことになったでしょうが, 残念なことに彼の死によってギリシャ語文献のラテン語への翻訳は中断されることになり, 結果として, 西ローマ帝国滅亡後の西ヨーロッパに残された哲学書はボエティウスがラテン語に翻訳した「手引」,

アリストテレスの「オルガノン」,「範疇論」,「命題論」といった論理学関係の文献の他には, プラトンの「テマイオス」[20] 程度しかないといった状況に陥いります. さらにはボエティウスが刑死からしばらくした 529 年には東ローマ皇帝ユスティニアス一世の非キリスト教学校閉鎖の勅令が出され, そのような背景もあってプラトンが開設したアカデメイアも閉鎖されることとなり, その結果, 西ヨーロッパでの学問の担い手は聖職者の養成を目的とした司教座聖堂附属学校や修道院学校, 医者や官吏養成目的の個人の学校, 君主の趣味による宮廷のサークル程度になります.

この状況は 11 世紀末になると大きく変化します. まず, 修道院学校が衰退し, 代って司教座聖堂附属学校が急増しています. さらには教師が生徒を募った私立の学校も多く出来ます. このような学校には法律学校や医術学校,「**リベラル・アーツ**」と総称される文法学, 修辞学, 論理学, 算術, 天文学と音楽の「**自由七科**」を基礎科目とし, 神学を上位科目とする学校です.

このように学校が増加した理由として, 10 世紀から 14 世紀にかけてヨーロッパは温暖期で, 社会が安定してたことによって人口増加と経済発展があり, それらの結果, 都市も発達します. それを背景として聖職者, 官僚や医者等としての教養人が求められ, そのような教養人が目覚しい社会進出をしたという背景があります.

また. 交易の拡大とともに西ヨーロッパで失われた古代ギリシアやローマの哲学・科学文献やそれらの注釈書がイスラム文化圏から大量に伝播し, 盛んにラテン語への翻訳が行われます. その文献の中に西ヨーロッパでは失なわれたアリストテレスの多くの著作があり, 百科事典的なアリストテレスの全貌が明かになり,「中世の覚醒」[44] で「**アリストテレス革命**」と表現されているように西ヨーロッパの知識人に大きな衝撃を与えることとなります. ところが, アリストテレスの哲学にはキリスト教と相容れることのない側面もあることとに加え, 急増する学校に対して, 従来から教育を独占していた司教団は脅威を覚えることになり, 教師が学校を開く為に司教が発行する教育許可証を必要とする教育資格制度を打ち出します. これはリベラル・アーツの教師の多くは聖職者の資格を持っていたためです.

また, 当初の学校は教師の周りに集う結社のような状況でしたが, イタリアでは 1190 年頃になると学生達が出身地別で同郷団を組織するようになります. この同郷団は学生同士の対立を解決し, 学校が所在する都市の住民から身を守るといった生活面の必要性に加え, 教師との契約を結ぶといったことを行うもので, このような学生の同郷団が「同業組合 (ギルド)」として互いに集結し, これが大学の起源の一つになります. たとえば, イタリアで最古の大学であるボローニャ大学は学生の同業組合を起源としており, このような学生達の動きに対して自治都市側はギルドの組織化を阻止しようとします. この状況に対して教皇側がボローニャ大学に「教育資格制度」を導入させ,1270 年以降になってようやく自治

[20] アトランティス伝説でも 有名で,「アテナイの学堂」で プラトンが手にしている書籍でもあります.

都市側も大学の存在を公式に認め, さらには学生の特権 (家賃の価格統制, 課税免除) が認められます.

この動きはアルプス北側では様相がやや異なります. まず, パリやオックスフォードでは自由七科の教師達が 1200 年頃に結集し始め, やがて教師の同業組合を結成します. このように教師が結集した理由には, 学生が急増した結果, 学生数の急増に対処できなくなる学校もあり, その学生達の集団が社会的な脅威と見做されるようになったこと, また, 教師同士の競争も激化し, 教師の質の問題も出てきます. つまり, 教師が新しい文献を適当に撮み食いして好き勝手な授業を行うことや, 科目を混ぜ合して教えるといった混沌とした状況があり, 教師の組合を結成することで学校の乱立を防ぎ, 読むべき文献を系統的に整理して危険な文献を禁書にすることや, 研究を行う際に各人が従うべき規定を定めることになります. パリでの教師達のこのような動きに対してフランス国王からの反対は無いものの, パリ司祭と司教区尚書担当司祭が教師達の一連の動きを押さえ込もうとします. やがて, 1215 年に法王庁からの規約が「教師と学生の組合」に対して付与され, この規約によって司教区尚書担当司祭の支配は制限され, 教師達の推す学士号志願者に教育許可証を無償で与えることが義務付けられ, 大学の自治・自律の権利も保証されます. 同時期にオックスフォードでは 1214 年に教皇から特許, 国王の認可によって, 大学の自治・自律が認められます.

このように学生や教員が同業組合を結成した理由は 中世西ヨーロッパの都市は「**都市の空気は自由にする**」の文言で知られるように「**自由と自治**」が享受できる場所でしたが, この権利は個々人に保証されたものではなく, 都市の共同体に対して認められた特権であったためです. 教師は都市に対して服従することで市民として認められていますが, 余所者の学生は市民でないためにそのような権利がありません. 都市での権利が無いということは権利を持つ有力者やギルド等に庇護されなければ身の安全が確保できるか怪しいことになります*21. このことが学生が同郷団を組織した理由です. また, 教師が同業組合を組織するに至った理由も同様で, 同業組合を組織することで, 司教や国王といった権威者や, 都市の住民からの脅威から身を守ることが権威者に頼らずとも可能になります. これらの同業組合はラテン語で「**ウニベルシタス (universitas)**」といい, これが大学の呼び名になります. この同業組合 (ギルド) では他のギルドと同様に構成員である教師, あるいは学生が自分達で規約を決め, その代表者を選出し, 地域住民や権威者から構成員を庇護する等の相互扶助のシステムを確立することに加え, ギルドの営利目的として, 研究と教育が自律的に行えるように法規を定めています.

そのような社会状況において, 体系的で百科事典的なアリストテレスの著作は当時の知識人を熱狂させるだけではなく, その一方で従来のキリスト教と相容れない箇所について困惑や反発も招くことになります. 前述のように, アリストテレスの哲学はプロティノス

*21 このことについては「刑吏の社会史」[1] の第三章「都市の成立」がその背景を知る上で参考になります.

の著書「エンネアデス」*22の影響で新プラトン主義化したものが西ヨーロッパに導入さています．これはヘレニズム文化を受け入れて咀嚼したイスラム文化圏の事情も関わっています．まず，ヘレニズム文化のイスラム文化への導入では，シリアのキリスト教徒の医者が大きく関係しています．実際，イスラム文化への受容では，シリアのキリスト教徒が様々な文献，特に医学書をアラビア語に翻訳して行きます．これがアッバス朝が成立し，マアムーンの時代になると，マアムーンが建てた「智の家 (Bait al-Hikmah)」で組織的な翻訳と研究が行われるようになります．この時点でカリフのマアムーンが支持していたイスラム神学は「神も理性に従うべき」と主張する非常に自由主義的な神学派である「ムアタズィラ派」でした．イスラムの宗教的主題も関係して，それまでの純粋なヘレニズム文化のものから，より折衷的なものへと変化します．つまり，最高善の

この解釈で代表的な哲学者が，スコラ哲学でアヴィケンナ (Avicenna) として知られるイブン・スィーナ (ibn Sīnā) で，東方のイスラム文化を代表する 11 世紀の哲学者・医学者です．アヴィケンナによって東方のイスラム哲学が体系化されることになります．このアヴィケンナに対して新プラトン主義的な夾雑物を排した「**純正アリストテレス**」を標榜する，スコラ哲学でアヴェロエス (Averroes) と呼ばれるイブン・ルシュド (ibn Rušd) が現れます．アヴェロエスが活躍した 12 世紀では，ほぼ同時代のアルガゼル (Algazer) として知られるアル・ガザーリ (al-Ghazali) による哲学批判 (「哲学者の矛盾」等) の影響もあり，イスラム文化圏で宗教と相容れない哲学が問題視されるようになる時代に突入します．この結果，アヴェロエスのアラビア語の著作は断片を除いて残っていない有様になりますが，一方で，コルドバやモロッコで活躍したアヴェロエスの著作の方がペルシャで活躍したアヴィケンナよりもやや早く西ヨーロッパへ伝来し，パリ大学でアヴェロエス学派と呼ばれる集団が現われる等の大きな影響を与えることになります．これはコルドバ等に在住のユダヤ人哲学者達がアヴェロエスの強く影響を受け，その著作物をヘブライ語に翻訳したものが，イタリアでラテン語に翻訳されたり，コルドバのレコンキスタ後により組織的にラテン語に翻訳されたことが関係しています．

アヴィケンナやアヴェロエスといったイスラム哲学者の著作は 12 世紀から 13 世紀にかけて成立した西ヨーロッパの大学に受け入れられ*23，それらの大学の発展に伴い，西ヨーロッパのスコラ哲学は新しい局面を迎えます．ただし，このアリストテレスの西ヨーロッパのキリスト教社会における受容は緊張を孕んだものでした．なぜならアリストテレスの宇宙観には「**天地創造**」がないどころか永続性があり，おまけに神も何かと奇跡で介入したがる存在ではなく，むしろ，その運行を見守る存在です．その上，人間それ自体もプラトンやグノーシス主義の「**魂の牢獄**」でもなければキリスト教の「**原罪**」を持つ存在でもなく，それどころか，それ自身で幸せにもなれる存在であり，註釈者アベロエスの

*22 「手引」の著者ポルフュリオスがプロティノスの遺稿を編集したものです．

*23 トマス・アクィナスの「有と本質について」[26] では「哲学者」はアリストテレス,「註釈者」はアヴェロエスを指しています．

「**知性単一論**」に従うとキリスト教の「**個人**」とも矛盾するといった有様でした. このこともあって, アヴェロエスの影響を強く受けたパリ大学のアヴェロエス派と呼ばれた急進的な神学派に対して教皇庁は異端としてアヴェロエス主義を禁止しています. しかし, アリストテレスの哲学の受容を止めた訳ではなく, キリスト教の神学者達は用心深くアリストテレスの哲学を取り込んで行き, 最終的にトマス・アクイナスの「**神学大全 (Summa Theologiae)**」でこの取組は一応の完成を見ることになります. そして, この神学大全がカトリック教会正式の教義となるに従い, アリストテレスが西ヨーロッパで権威として君臨することになります. このことは神秘思想 (スーフィズム) に沈む一方で, その反発から過激に原理化した宗派を生み出したイスラム神学との違いを決定付けていると言えるでしょう.

2.2.3 定義すること

さて, 我々は事物を抽象することで概念に辿りつきましたが, 名辞 X について「**Y を充すものが X である**」とも言える筈です. この操作を「**(X を) 定義付ける**」と言います. 具体的には「**定義付ける**」ということには「**タマは猫である**」のように類や種で定義付ける「**実体的定義**」, あるいは「**分析的定義**」と呼ばれる方法,「**点は平面上の平行でない二直線の交わりとして構成される**」という点の定義のように対象がどのような条件で発生, あるいは成立するかを記述する「**発生的定義**」, または「**綜合的定義**」と呼ばれる内包的な定義と外延を用いる「**実例, または代表・典型を用いた定義**」があります. ちなみにアリストテレスが創始者である逍遥学派の「**定義**」は類と種や差異を用いてその「**説明規定**」(λόγοσ, logos, account) を与えることになります. ここで, 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, その一方で「**一つの主語は一つの述語あるのみ**」[3] 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. このこと弟子のディオゲネス (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毟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なくもないでしょう. この立場に立脚すれば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.

なお, 定義については次のことに注意する必要があると「哲学者の意図」[12] で知られる 12 世紀の神学者ガザーリは述べています:

1. あるものをそれ自体で定義すること
2. 曖昧さの点で同程度のもので定義すること
3. より曖昧なもので定義すること
4. それ自身でしか知り得ないもので定義すること

ここで第 1 のものを「**循環的定義**」と呼びます. たとえば、「経済学は経済に関する学問である」のようなものです. 第 2 のものは「白」を定義しようとして「白は黒の反対である」とした場合,「黒」自体が理解されない限り, その反対のものも理解できないことになります. つまり,「黒」が曖昧なままでは「白」も同様に曖昧のままになります. 第 3 のものは,「絵画は無声の詩である」のように比喩的表現であるために幾らでも解釈が可能で誤解が生じ易い定義になっています. そして最後のものとしては「太陽は昼間空にあって光を放つ天体である」を挙げておきましょう. これは一見問題なさそうですが,「昼間」という言葉は「太陽」についての認識がなければ理解できない言葉です. なぜなら「昼間は地面を太陽が照す時間」であるためです.

2.2.4 形相 (Eῖδοσ)

プラトンの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, 人間としての類似を示す尺度としての「**人間のイデア**」が必要で, これを「**第三の人間**」と呼びます. この第三の人間を認めると今度は「**第三の人間**」とその他のイデアに対しても類似の尺度になるイデアが存在しなければならなくなり, 以降, 同様に第四, 第五, 第六... の人間が存在することになって議論が収拾しないという反論です. また人間には乳児, 少年, 青年, 壮年... といった成長の過程を辿りますが, 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話がますます複雑になります. さらに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り, 生殖の理由を説明したところで何気ない現象の説明には無理があります. この有様にアリストテレスも「形而上学」にて「**物を数えようとする場合に, 数が少なくては数えられないと思って, その数を増やして数えようとする者の ごときである**」*24とイデア論を批判しています.

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, アリストテレスの「**形相 (εῖδοσ, eidos)**」はプラトンの「**イデア**

*24 形而上学 [3] 第一巻九章

(ἰδεα)」のような「**個体から離れた存在 (χοριστά)**」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 (ὕλη)**」との「**結合体 (σύνολον)**」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が, 現代の科学でも対象である個体が何であるか, どのような理由でその個体がそれ自体であるかを説明しようとするもので, この流儀はアリストテレスの考察に源流があることが判ります. それゆえにアリストテレスが「**万学の祖**」と呼ばれる所以です. そして, 金属の金が金としての, 鉄が鉄としての形相を持つのであれば, それらの形相に直接働きかけて鉄という卑金属を金という貴金属できるという考えが錬金術に繋り, 錬金術究極の目的がそれを可能にする「**賢者の石**」の生成です. 錬金術が非常に長い時代にわたって研究されていたこと, さらには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ってのことです.

さて, この形相と質料を計算機上で考えるとそれなりに面白いことが判ります. まず, 質料それ自体は何らの特性を持ちませんが, これをビットの列に, それから形相をデータ構造等の意味付けに対応付けられるでしょう. 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収拾がつきませんが, 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.

2.2.5　普遍の存在

概念/イデアの存在は物理学の原理や数学の定理が先に存在し, 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ものと考えるかといった議論にも繋がります. ここで事物に先行して概念があると考える立場を「**プラトニズム (Platonism)**」, あるいはプラトンの「**実在論 (Realism)**」と呼びます.

では, 概念やイデアは実在するものでしょうか? 最初に概念は実在の「みけ」から「猫」や「三毛猫」といった概念に到達するために「**事物のあとの普遍**」と呼ばれる普遍です. 後者のイデアは事物に先立って存在するもので, 事物は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. この事物の前の普遍はアリストテレスのイデア論への批判もあって否定的です. そして, 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, これらが実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. そのこともあってポルフュリオスの「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, それらが存するのであれば, それらは物体 (σώμα, body) を持つものなのか, それとも非物体のもの (ἀσώματοσ, incorporeal) なのか, そして, それらは離在可能 (χωριστόσ, separable) なものなのか, あるいは明瞭に知覚できるものの中にあって, それらに関わって存在するものなのか - こういったことの議論を私は避けようと思います. というのも, このような事象は非常に深淵で, 他の物事やより広範囲の探求を必要とするためです...

と述べています. なお,「種や類が存在するのか, あるいは単に理解のうちにあるだけなのか」,「物体として存在するのか, あるいは非物体として存在するのか」,「知覚対象から離在したものなのか, それとも知覚できるものの中にあるものか」という三つの問に対してボエティウスは「手引」の「**第二註解**」で答えていますが, この註解が後世のスコラ哲学における「**普遍論争**」の導火線になります.

ちなみに, アリストテレスが創始し, 発展した伝統的形式論理学で扱う命題には「**存在含意 (external import)**」と呼ばれる命題が付随しています. この存在含意は命題の主語が存在しているという暗黙の条件であり, 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 ‘A = B’ という意味の命題で, その主語 A と述語 B を繋げる働きをする「**繋辞 (copula)**」に主語 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も関係しているでしょう. たとえば, 日本語の「**A は B である**」[*25] を印欧語族の一つである英語で「A is B」と置換したときに日本語の「は」は A と B が一致すること意味する以上の意味を明示的に持ちませんが, 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,「**A が存在し, かつ, A = B である**」という命題でもあります. そのためにトマス・アクィナスは「有と本質について」[26] にて

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

と, ある命題が真であれば, その命題に存在含意も真であるため, その主語も存在すると述べています. ちなみにアリストテレスが整備した三段論法も本質的に命題の外延についてその包含関係 (周延) を前提にした推論であるために命題の主語の存在が怪しければ推論が行えません. この存在含意は現代の論理学の創始者のフレーゲ (Frege) の概念記法で除外され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**」あるいは「**神は正しい**」といった命題が真であれば自動的に神の存在が保障された伝統的形式論理学と異なり, 命題の正しさと主語の存在性を別問題とする現代の形式論理学はある意味, 千年にも

[*25] 「A は B である」という命題に「**ある**」が何気に含まれていることに, このような用語を作り定着させた人々の何気ない凄さを私は感じます. 徒然草の聖海上人と同様かもしれませんが...

及ぶ神の実在から神の存在の不確実さを招来し, 悪く言えば「**神を殺した**」状況に陥ったとも言えなくもないでしょう. とは言え, その存在がどのようなものであったかはポリピュリオスが述べたように容易に答えられるものではありません. なお, この存在含意があることで後述の「ラッセルの逆理」のような逆理を伝統的形式論理学が排除できていますが, フレーゲの論理主義では排除できずに彼の論理主義は破綻しています. そして, 集合論では存在含意は導入しないものの集合自体に制約を加えた「**公理的集合論**」で, このような逆理が体系から排除されます. このことは §2.4 で解説します.

なお,「**普遍論争**」での争点の「**普遍**」は注意が必要です. 普遍論争の争点は「**存在における普遍**」, あるいは「**形而上学上の普遍**」と呼ばれる普遍です. たとえば,「馬」という概念の実体は牧場や動物園に個々の馬として存在しますが, これらの馬に対する「馬の概念」は馬という存在に共通する「何か」であり, この「何か」を「**馬性**」と呼ぶときに「**馬性**」は「**馬**」という概念とは異なった振る舞いをします. 実際,「**A は馬である**」と主張できても「**A は馬性である**」と主張できないために馬性は論理学上の普遍ではありませんが馬であるためには個々の馬に共通する特徴である馬性がなければなりません. そのために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. この点で馬性は類, 差異よりも偶有性 これはアヴィケンナが述べたこと[*26]です. この議論を計算機言語の「**型 (type)**」を使って「**馬性**」を「**馬という型**」で置換えると, この「**馬という型**」は「**馬という概念**」そのものとは異なって個々の馬に付属する型以上のものではないために, 結局,「馬性は馬という型に他ならない」と言い換えられるでしょう. この型を入れるよりも馬の「**双対**」を考えるともっと具体的に馬性を説明できます. まず, 集合 X から実数 $\mathbf{R}$ への函数で対象 x に対してのみ 1 を返し, 他は 0 を返す函数のことを x の双対と呼びます. ここで馬の双対として「**馬性**」という函数を考えます. この函数は引数が馬の類に属する対象に対してのみ真 (=1) を返し, その他は偽 (=0) を返します. したがって,「**馬であるということ**」はこの函数が真になることと言い代えられます. このように馬性は一般の馬と同列のものではなく, その双対, つまり, 述語函数になります. このように対象とその双対といったさまざまなものを一纏めに同列の対象とみなしたために生じた混乱と私には思えます.

それから, 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) [17] は

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

と概念について述べており, アリストテレスや手引でほとんど触れられなかった概念を認識する心の働きを加えた考察になっています. この概念の「**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」[23] で数概念を各個人の心理的な

[*26] ラテン語訳:Equinitas est Equinitas tantum.

動機を含めて説明していることにも繋がるでしょう. この数概念で現れる「**心理主義**」, それと数を単なる記号と考えるだけの素朴 (粗雑) な「**形式主義**」をフレーゲは「**あなたの心の中の '1' と私の心の中の '1' は同値なのか?**」と批判し, 数を純粋に論理学だけで構築しようとします. そのために「**概念記法**」と名付けた言語を定め, その言語を用いて著書「概念記法」[31] で純粋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」[32] で論理学から数学を構築するという「**論理主義**」を遂行しています.

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たかということに対して, アリストテレスは「**A は B である**」という命題の述語 B を「**範疇 (カテゴリー)**」[2] で分類してます. ここで「**範疇**」に対応するギリシャ語のカテゴーリアー (κατηγοριά) 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 (κατηγορῖεσται) に由来し, 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:

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. どのようか (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, 個体により近く, 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「**ウーシアー (οὐσία)**」と呼ばれ,「**存在**」を意味する動詞 εἶναι 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述語の分類に対応するように, カント (Kant) は命題 (判断) を量, 質, 関係と様相の 4 綱目に分け, さらに各自を 3 項目に分けて 12 の範疇に分類しています:

カントによる判断の分類

量	単一性	関係	属性と実体性
	数多性		因果性 (原因と結果)
	全体性		交互性
質	実在性	様相	可能性 (不可能性)
	否定性		現実性 (非現実性)
	制限性		必然性 (偶然性)

述語の範疇で重要なことは,「**それが何であるか?**」や「**それがどのようなものであるか?**」という問に対する答は範疇に分類されます. このように概念には類と種 (差異) による階層が入り, 語られる内容も範疇で分類されます. そして, これらは我々がこれから考察しようとするオブジェクト指向プログラミングのクラスとその構造に深く関わります.

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二つの方法があり, 前者を「**内包**」, 後者を「**外延**」と呼びます. 最初の「**内包**」は概念が持つ徴表/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**」,「**ニャオと鳴く**」等の属性 (性質) から構成され, 外延なら「ペルシャ猫」,「シャム猫」といった猫の種,「黒猫」,「白猫」,「虎猫」,「三毛猫」といった毛並で分類する方法, あるいは「**栗根さんのペットのタマ**」のように個体を列記する方法になるでしょう. このように内包は概念を説明する述語から, 外延は概念に対応する具体的な個体や下位概念の列記から構成されます. そして, 内包と外延には「**内包外延反比例増減の法則**」と呼ばれる関係があります. これは内包が増大すれば外延が減少し, 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」が「猫」と「茶、黒、白の三色の毛並」の外延から消えますが, 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て個体に近付く結果, 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普遍的な事柄を抽出するために内包が減少する関係です.

外延で表現された概念は内包で説明規定できますが, 逆に内包で説明規定された概念が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も限りません. 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髭を剃らない人の髭だけを剃ると言っています. では, その床屋の主人の髭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.

この命題を主張したのがエピメニデス (Ἐπιμενιδεσ, Epimenides) で, 彼が紀元前 6 世紀頃のクレタのクノッソスの哲学者であっ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あるものの, 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」[58] とフレーゲが独自の論理式のために嫌がる出版社を説得して二部に分けて出版した「算術の基本法則」[32] といった著作の成果を葬り去るに十分でした*27。

ラッセルやフレーゲの論理主義*28とカントール (Cantor) の (素朴) 集合論*29に批判的であったポアンカレ (Poincaré) は彼のエッセイ「科学と方法」[33] 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ています. このような定義方法を「**可述的**」と呼びますが, 床屋の逆理のような循環論法に訴えなければ自分自身を定義できない定義を「**非可述的**」と呼び, ポアンカレは非可述的な定義に問題があると述べています ([33],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 $\mathbf{R}$ の有限部分集合はその最小上界を持つ」はある対象を含む上界全体に言及していながら, 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**」を導入して病的な非可述的な命題の排除に成功したものの, 今度は数学的帰納

*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」[59] 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.

法が使えないという重大な副作用が生じます．すると，今度は「**還元可能性公理**」[*30]を導入して解決を図ったものの，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，その結果，「ラッセルの逆理」は集合論の体系から排除されています．

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

今迄の考察を基にオブジェクト指向プログラミングを再度，吟味してみましょう．まず，扱うべきデータが個体と考えるなら，データを抽象することで得られる概念に対応するクラスがあり，データはそのクラスが実体化したものとして捉えられます．ここでクラスは，「**それがどのようなものなのか**」という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」，「雨の前に顔を洗うような仕草」，「ネズミを掴まえる」等々) をメソッドとして列記します．そして，「みけ」は「猫」というクラスが実体化したもの，すなわち，インスタンスになります．このときにクラス同士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**」という機能があります．

2.2.9 継承

概念には，より普遍的な上位概念と逆に個体に近い下位概念があります．これらの概念を内包として書換える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追加したものとして表現できます．このことはある概念に新しい「属性」を与えることでその概念の「下位概念」が構築できることを意味し，この操作がオブジェクト指向プログラミングでの「**継承**」に該当します．この継承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，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

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

う珍しいことではなく, この細分化では上位のクラスにない値やメソッドの追加で行われます. 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せずに自然に拡張が行えることを意味し, この継承を上手く行うためには系統立った分析が必要なことは言うまでもありません. さらに継承関係が一子相伝的な継承であれば継承関係が直線的な関係のために属性やメソッドがどこから引き継がれたかを容易に探せますが, 無駄に複雑な継承関係を持てばメソッドや属性の検索で不利になります. 実際,「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」と階層を細かくしていた場合で, 飼い猫の「みけ」が持つ「猫の属性」や「猫の習性」を知りたくなったときに最初の継承関係で「三毛猫」を間に一つ挟む程度で済んだことが, 後者の継承関係では「アジアの猫」,「東アジアの猫」と「日本猫」の三つのクラスが増えた結果, 検索の手間が増えて不経済です. そして, 実際の継承は複数のクラスからの継承があってさらに複雑です. そのために属性やメソッドの検索順位の定め方次第で新しいクラスの属性やメソッドが反映されなくなる恐れもあります. この問題については §3.10 にて解説する「**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重要で, それも根源的であることは言うまでもありません.

2.3　判断と推論

2.3.1　判断

アリストテレスに始まる伝統的論理学は主語と述語の関係の考察がその中心にあります. 伝統的論理学の命題は「**主語**」, すなわち「**主辞**」と「**述語**」, すなわち「**賓辞**」, あるいは「**客語**」, そして, これらを結びつける「**繋辞**」の三つで構成されるために「**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. この本では命題の主辞を S, 賓辞を P, 命題を $S-P$ と表記します. ただし, 表記 $S-P$ の記号 “$-$” には後述の判断の種類が記入されます. ここで, 判断の種類はカントによって量, 質, 関係, 様相の 4 つの範疇のグループに分類されて計 12 個の判断があります.

■**量**:　主辞の外延の大きさに関する判断です.

[*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.

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

単称判断は全称判断の特殊な例であるために実質的に全称判断と特称判断の二つです.

■**質**:　肯定と否定に関する判断です.

質に関する判断

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

ここでの「非 P」について説明しておきましょう. ある概念「A」に対して概念「非 A」を概念「A」の「**矛盾概念**」と呼びます. たとえば,「動物」という類概念の中の種概念「猫」に対して「非猫」が猫以外の動物の種概念を指し, このときに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て「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 と共通の個体を一切持たない概念が矛盾概念「非 A」で, $\overline{A}$ とも表記します. なお, 質に関する判断の無限判断は肯定判断の特殊な例として考えられるために肯定判断と否定判断の二種類が質の判断の代表として挙げられます.

■**関係**:　主辞と賓辞の関係に関する判断です.

関係に関する判断

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

関係の判断は基本形として「A ならば B」, つまり,「A は B である」の断言判断の形式になります. そのため, この関係では断言判断が代表になります.

■**様相**:　命題の確実性に関する判断です.

様相に関する判断

実然判断	:	S は P である
蓋然判断	:	S は P に違いない
必然判断	:	S は P でないはずがない

実然判断の「S は P である」を判断の骨子として考えられるために実然判断を代表にできます.

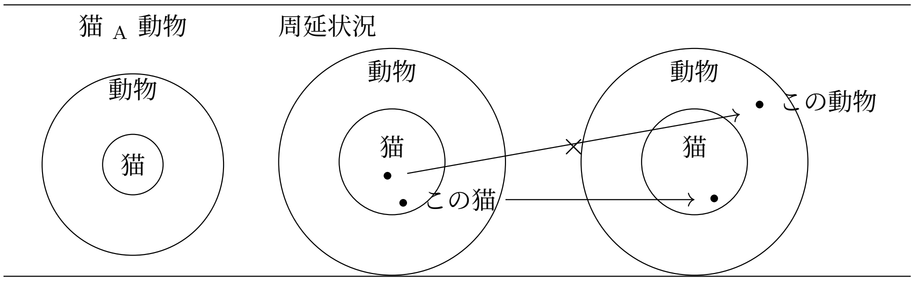
以上から判断の代表を纏めると「**A: 全称肯定判断 (Universal Affirmative)**」,「**E: 全称否定判断 (Universal Negative)**」,「**I: 特称肯定判断 (Particular Affirmative)**」,「**O: 特称否定判断 (Particular Negative)**」に判断を纏められます. このことを以下にまとめておきましょう:

AEIO

(A)	全称肯定判断：	「すべての S は P である」	$S\ _{\mathrm{A}}\ P$
(E)	全称否定判断：	「すべての S は P ではない」	$S\ _{\mathrm{E}}\ P$
(I)	特称肯定判断：	「ある S は P である」	$S\ _{\mathrm{I}}\ P$
(O)	特称否定判断：	「ある S は P ではない」	$S\ _{\mathrm{O}}\ P$

これら「**A**」,「**E**」,「**I**」,「**O**」はラテン語の動詞<u>A</u>FF<u>I</u>RMO (我は肯定する) と N<u>E</u>G<u>O</u>(我は否定する) に由来する中世の論理学者が付けた略称で, $S\ _A\ P$ 等の表記は, これら AEIO を表記する式の主辞と賓辞の周延の状況を示す表記です. この「**周延 (distribution)**」は中世のスコラ哲学で発展した「**代表 (suppositio)**」に由来し, 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状況を示します. つまり, ある概念が「**周延**」されている状況は命題に起因する指示関係がその概念に属する全ての個体で成立, すなわち, 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で成立しないとき, すなわち, 分配できないときです. このことは命題 $S - P$ を $\mathrm{Pred}(S, P)$ と表記して各概念に属する個体全てに函数 $\mathrm{Pred}(x, P)$ と $\mathrm{Pred}(S, x)$ を分配したときに, これらを評価した値が常に真であれば周延されている状況, 真であるとは限らならなければ不周延の状況です. 簡単な見分け方は主辞「S」と賓辞「P」に「**この**」という言葉を付加すれば判断できます. たとえば,「すべての猫は動物である」ならば,「**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」とは主張できないため,「動物」は不周延です. また, 判断 $S - P$ において概念 S が周延されていれば S^d, そうでなければ S^u と表記します. この周延関係は「**オイラー図**」で図示されます. オイラー図はベン図と同様に概念の外延を閉じた円領域で表現し, 概念間の関係を円同士の交差状況で表現します. つまり, 共通元が存在すれば領域を交差させ, そうでなければ交差させません. 以下に「すべての猫と動物である」という命題の周延関係を図示しておきましょう:

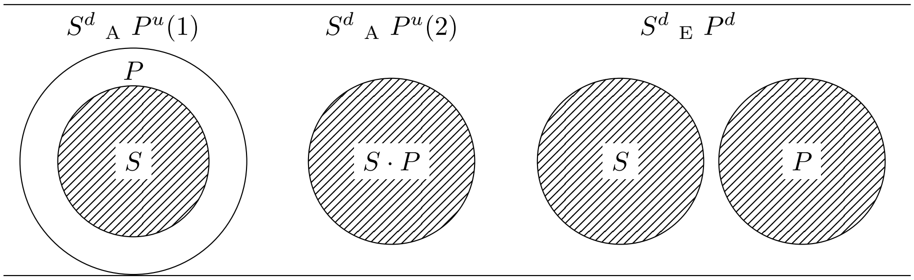
動物-猫の周延状況



この例では左側の同心円が周延の様子を, 右側にその周延の根拠をベン図を使って示しています. 右図の矢印は命題「この猫は動物である」と「猫はこの動物である」であり,「この猫は動物である」は対応関係が成立しても,「猫はこの動物である」は「猫」以外の動物を指示したときに対応関係は不成立 (Œ) です. 以上から「猫」は周延され,「動物」は不周延です. このことを記号で表示すると 猫^d, 動物^u で, 判断が全称肯定判断 (A) であることから「猫^d $_\mathrm{A}$ 動物^u」とまとめて表記できます.

現在, 周延関係はスコラ哲学風に代表理論で説明するよりも概念の外延を用いて説明されています. この外延を用いた説明では, 概念 S の外延が概念 P の外延に包含される状況を「概念 S は概念 P に周延されている」と呼び, 概念 S が概念 P に包含されていない状況を不周延と呼びます. この状況はオイラー図で可視化できます. ここでは A と E の周延状況をオイラー図で示しておきましょう:

AE の周延状況



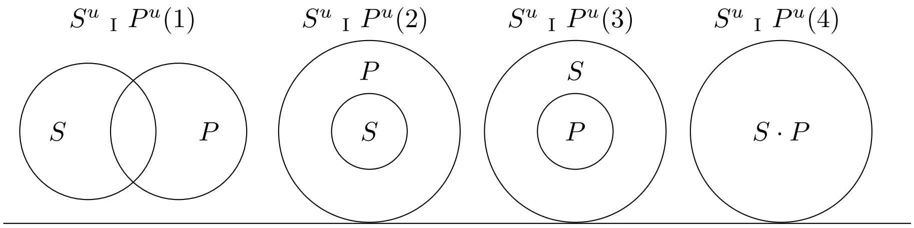
最初の二つのオイラー図が全称判断のものです. このオイラー図は命題間の関係を表現する図で, 二つの命題に関係があれば交差するように描きます. また, 周延している様子を斜線で表現します. 最初のオイラー図は S の外延が P の外延に含まれており, このような判断を「**包摂判断**」と呼びます. この場合は S の外延は全て周延されるために斜線を全体に入れます. 次の全称判断は「明けの明星は金星である」のように主辞と賓辞が同一概念, 等価概念となる場合で, このような判断を「**等価判断**」と呼びます. この場合は S, P の外延を重ねて描きます. ちなみにアリストテレスは周延について明瞭に述べていませんが,

日常語で「すべての S は P である (εἶναι)」と記述するところを「P は S のすべてにある (ὑπάρξειν)」と S が周延されている様子が判り易い表現, いわば, 一種の人工語になっています.

次のオイラー図が否定判断の周延状況を示したもので, 否定判断の場合は主辞と賓辞の外延双方に関係がないために, このような分離した表記になります.

次に「ある S は P である 」の I 判断に対応するオイラー図を示します:

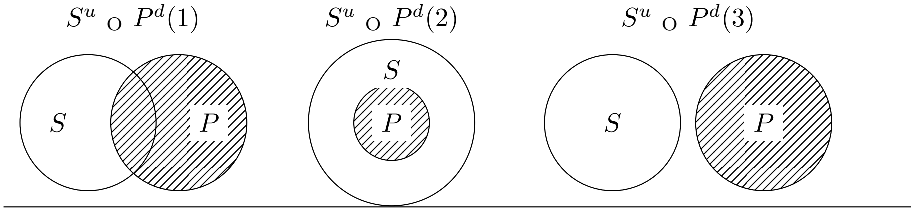
I 判断の周延状況



I 判断の場合は 4 つの場合が考えられますが, 何れも全体が周延されることはありません. 最初の (1) は「ある学生は男性である」のように外延が交差している状況であり, このような判断を「**交叉判断**」と呼びます. 次に (2) に対応するのが「ある三角形は多角形である」のように主辞の外延が賓辞の外延に含まれる場合, それから (3) に相当するのが「ある人は学生である」のように A 判断の包括判断の部分的な判断, 最後に (4) に対応するのが「ある正方形は長方形である」のように A 判断の同一判断の部分的な判断に留まっている状況です.

最後に O 判断の周延状況のオイラー図を示しておきます:

O 判断の周延状況



O 判断の場合は 3 つの場合があります. 最初のものが「ある学生は男性でない」のように交叉判断のように外延が交叉しており, P の外延が周延されており, (2) は「ある人は学生でない」と同様に P が周延され, (3) は「ある人は石ではない」のように S と P が無関係で P が周延された状況になっています.

このように A 判断では S が必ず周延され, O 判断では P が必ず周延されます.

なお, アリストテレスは周延関係の説明で棒を使って図示していたようです [5]. そして, この周延関係は基本トポスの定義で必要な「**部分対象分類子 (subobject classifier)**」と

しても現れます．次に判断の意味を変えずに判断を変形する「**換質法**」と「**換位法**」について解説しておきましょう．

2.3.2 換質法

「**換質法**」はもとの判断と同じ意味になるように判断の質を変更する方法で, 主辞はそのままで賓辞の「**矛盾概念**」で賓辞を置き換え, 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ます．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
ある生徒の生活は規則正しい	→	ある生徒の生活は不規則ではない
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

最初の「天空を移動」の矛盾概念は「天空で停止」です．同様に「死すべき存在」の矛盾概念が「不老不死」です．つぎの生徒の例では「規則正しい生活」と「不規則な生活」の二種類の生徒にバッサリと分類しています．そして, 東海道新幹線に「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」です．この変形で判断の真偽といった意味が変化するものではありません．この換質法による AEIO の変形をまとめておきます．ここで賓辞 P に対する $\overline{P}$ は賓辞 P の概念の矛盾概念である「非 P」を指示します:

換質法による AEIO

A	:	$S^d \ _{\mathrm{A}}\ P^u$	→	$S^d \ _{\mathrm{E}}\ \overline{P}^u$
E	:	$S^d \ _{\mathrm{E}}\ P^d$	→	$S^d \ _{\mathrm{A}}\ \overline{P}^u$
I	:	$S^u \ _{\mathrm{I}}\ P^u$	→	$S^u \ _{\mathrm{O}}\ \overline{P}^d$
O	:	$S^u \ _{\mathrm{O}}\ P^d$	→	$S^u \ _{\mathrm{I}}\ \overline{P}^u$

ここで注意すべきことは矛盾概念が日常語の「**反対**」でないことです．たとえば,「不味いラーメン」の矛盾概念は「旨いラーメン」ではありません．実際,「不味いラーメン」の矛盾概念の「非 (不味いラーメン)」には「旨いラーメン」だけではなく「普通のラーメン」や「不味くはないが美味しくもないラーメン」といった微妙なものもあり, 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, 同じ意味の判断を構築する方法です[*32]. ただし, もとの判断の主辞と賓辞の周延関係で, 周延されていたものが新しい判断で不周延になっても構いませんが, 不周延のものを周延にしてはいけません. このことに注意して AEIO で換位法を適用してみましょう.

■全称肯定判断の場合:　周延が $S^d\ {}_{\mathrm{A}}\ P^u$ であるために主辞と賓辞の入れ替えが可能で,「**すべての S は P である**」から「**ある P は S である**」と全称肯定から特称肯定になり, この置換を「**限定置換**」と呼びます. ここで「**すべての理性的動物は人間である**」と「**すべての人間は理性的動物である**」の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, 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 $S^d\ {}_{\mathrm{E}}\ P^d$ と主辞も賓辞も周延しているために単位置換ができます. 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**」にできます.

■特称肯定判断の場合:　周延が $S^u\ {}_{\mathrm{I}}\ P^u$ と主辞も賓辞の双方が周延していないために「**ある S は P である**」から「**ある P は S である**」に単位置換が行えます.

■特称否定判断の場合:　周延が $S^u\ {}_{\mathrm{O}}\ P^d$ のため, S^u を賓辞の位置に持って行くと否定判断では S^d にならなければならないため, この判断の換位ができませんが, 特称否定判断であっても換質法で特称肯定判断に変形できます.

これらの結果を以下の表にまとめておきましょう:

換位法による AEIO

A	:	$S^d\ {}_{\mathrm{A}}\ P^u$	$\to$	$P^u\ {}_{\mathrm{O}}\ S^u$
E	:	$S^d\ {}_{\mathrm{E}}\ P^d$	$\to$	$P^d\ {}_{\mathrm{E}}\ S^d$
I	:	$S^u\ {}_{\mathrm{I}}\ P^u$	$\to$	$P^u\ {}_{\mathrm{I}}\ S^u$
O	:	$S^u\ {}_{\mathrm{O}}\ P^d$	$\to$	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可能であり, この変形は判断が特殊否定判断 (I) になった時点で停止します.

[*32] ここでの「換位」に対応するギリシャ語「ἀντιστρεφειν」の意味はものの順位や方向を逆転・反対にすることでです [5].

2.3.4　三段論法 (Syllogism)

換質法と換位法は, 与えられた命題を同じ意味のより判断し易い命題で置き換える操作で, この操作による判断を「**直接推論**」と呼びます. 伝統的論理学には「**三段論法 (Syllogism)**」と呼ばれる推論形式もあります. この三段論法は二つの前提となる判断と一つの結論の合計三つの命題から構成され, 判断の種類から「**定言三段論法**」,「**仮言三段論法**」と「**選言三段論法**」の三種類に分類されます. ここでは定言三段論法について解説しておきましょう. この定言三段論法は 3 つの概念 S, M, P に対し, S と M, P と M の関係から S と P の関係を求める推論で, 概念 S と P の仲立ちをする概念 M を「**媒概念**」と呼び, 媒概念の判断の位置, つまり, 主辞か賓辞であるかで, 次の 4 つの「**格 (figure)**」に分類されます:

定言三段論法の格

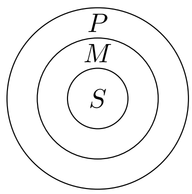
第一格	第二格	第三格	第四格
M - P	P - M	M - P	P - M
S - M	S - M	M - S	M - S
S - P	S - P	S - P	S - P

ここでは格として 4 つ挙げていますが, アリストテレスはその内の第一, 第二, 第三格を挙げており, 第四格はガレノスが形式整備のために補完した格です. この表の判断は A, E, I, O の何れかになりますが, 格によっては使えない判断の組合せがあります. ここで典型的な定言三段論法の例を示しておきます:

定言三段論法の例

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

この例では概念 S が「ソクラテス」, 概念 P が「死すべき存在」, 概念 M が「人間」で,「ソクラテス」は「人間」に「人間」は「死すべき存在」に包含されて概念 M を介して外延の包含関係 $S \subset M \subset P$ から $S \subset M$ が得られて概念 S と P の判断ができます. この包含関係から概念 S を「**小概念**」, 概念 P を「**大概念**」, 媒概念 M を「**中概念**」と呼び, 大概念に関わる前提を「**大前提**」, 小概念に関わる前提を「**小前提**」と呼びます. また, 定言三段論法を構成する概念 S, P, M に次の公理があります:



三段論法の公理

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

ここから 6 個の規則とそれから派生する 3 個の規則が得られ, 三段論法の格における判断の組合せから次の規則が定まります:

三段論法の規則

1. 三段論法は 3 個の概念から構成され, これら 3 個の概念に限定される.
2. 三段論法は 3 個の判断のみで構成され, 3 個の判断に限られる.
3. 媒概念は二つの前提判断のどちらかで必ず周延されていなければならない.
4. 前提で一度も周延されていない概念を結論で周延させてはならない.
5. 前提が二つとも否定であれば結論が得られない.
6. 前提が二つとも肯定であれば結論も肯定, 前提の一方が否定であれば結論は否定である.

■**規則** 1:　名辞では一致していても文脈上で異なる概念があることに注意が必要です. たとえば,「病気」も病院で治療すべき疾病なのか, 単なる悪癖かもしれません. このように概念に曖昧さがあれば三段論法に現われる概念が実質的に 4 個になって規則 1 に反します. この誤謬を「**媒概念曖昧の誤謬**」と呼びます:

媒概念曖昧の誤謬の例

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

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

■**規則** 3:　三段論法は概念の包含関係に基づく推論であるために全ての概念が不周延のときは相互の包含関係が不明慮になるために結論が得られません. つまり, 三段論法では概念 S と P の間を媒概念 M が仲立して S と P の包含関係が明瞭になることを利用していますが, 媒概念 M が不周延で, 概念 S と概念 P の仲立ができていない状況であれば結論が出せません. この状況を「**媒概念不周延の誤謬**」と呼びます:

媒概念不周延の誤謬
すべての男の子は人間である
すべての女の子は人間である
ゆえに, すべての男の子は女の子である

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

■規則 4: 前提で不周延な概念は媒概念との関係上, 部分的な合致に留まり, 全体の合致や周延関係があることを主張できません. 小前提にのみに現れる概念が不周延で, それを結論で周延させたときに「**小前提不当周延の誤謬**」, 大前提のみに現われる概念が不周延で, それを結論で周延させたときに「**大前提不当周延の誤謬**」と呼びます:

小前提不当周延の誤謬
すべての男の子はゲーム好きある
すべての男の子はサッカーファンである
ゆえに, すべてのサッカーファンはゲーム好きである

これは「小前提不当周延の誤謬」の例で, 小概念が「サッカーファン」, 大概念が「ゲーム好き」, 媒概念が「男の子」で, まず, 小前提で「サッカーファン」は周延されていませんが結論では周延されています. ところでサッカーファンは不周延, つまり, 男の子以外のファンもいるためで, 前提からゲーム好きと結論付けられません. もう一つの例を挙げておきましょう:

大前提不当周延の誤謬
すべての OL はスイーツが好きである
このレストランの客はみんな OL ではない
ゆえに, このレストランの客はスイーツが好きではない

これも「大前提不当周延の誤謬」の例で, 小概念が「このレストランの客」, 大概念は「スイーツ好き」で媒概念が「OL」です. そして, 大前提で「スイーツ好き」が不周延ですが, 結論が全称否定判断であるために大概念が周延されて間違った結論になります.

■規則 5: 三段論法の公理 3 の小概念と大概念が媒概念に一致しないときは小概念と大概念の周延関係が定まりません. この規則に反したときに「**否定二前提の誤謬**」と呼びます.

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

規則 6 から派生する系

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

格は 4 種類, 判断は AEIO の 4 種類あるために単純計算で $4^3 = 64$通り の格と判断の組合せが考えられますが, 上述の規則と系によって制約が入ります.

■第一格の場合:　大前提が $M - P$, 小前提が $S - M$ で, 媒概念が前提で周延しなければならないために大前提が全称判断, そうでなければ小前提が否定判断でなければなりません. ところで小前提が否定判断のときに結論も否定判断で, P も周延されていなければなりません. したがって, 大前提は否定判断でなければなりませんが, ここで前提が二つとも否定のために「否定二前提の誤謬」になります. そのために小前提は肯定判断でなければなりません. 以上から AAA, AII, EAE, EIO のいずれかに限定されます.

■第二格の場合:　大前提が $P - M$, 小前提が $S - M$ で, 媒概念 M が周延されるためには大前提か小前提のどちらか一方が否定判断でなければなりません. ここで前提の一つが否定判断であれば結論が否定判断になるために P が周延されるために大前提が全称判断でなければなりません. 以上から AEE, AOO, EAE, EIO に限定されます.

■第三格の場合:　大前提が $M - P$, 小前提が $M - S$ で, 媒概念 M が周延されるためには前提が全称肯定か全称否定のいずれかでなければなりません. まず, 大前提が全称肯定判断 (A) であれば P は不周延, そして, 結論は否定判断にならないために小前提は肯定判断でなければなりませんが, このときに S は不周延になるために結論は特称肯定判断 (I) になります. また, 大前提が全称否定判断 (E) であれば, 小前提は否定二前提の誤謬を避けるために肯定判断に限定されるために S は不周延, P が周延されるため結論は特称否定判断 (O) になります. 最後に小前提が全称肯定判断 (A) のとき, 大前提が全称判断であれば先程の考察に該当するために大前提が特称判断の場合だけを考察すればよく, ここで大前提が特称肯定判断 (I) であれば S, P ともに不周延のために結論は特称肯定判断 (I), 大前提が特称否定判断 (O) であれば S は不周延, P が周延であるために結論は特称否定判断 (O) になります. 以上から AAI, AII, EAO, EIO, IAI, OAO に限定されます.

■第四格の場合:　大前提が $P - M$, 小前提が $M - S$ で, 媒概念 M が周延されるためには大前提が否定判断であるか, 小前提が全称判断でなければなりません. まず, 大前提が全称否定判断 (E) のときに小前提は肯定判断でなければならないために S は不周延, P が周延され, 結論が特称否定判断 (O) になります. また, 大前提が特称否定判断 (O) であれば P が不周延になりますが, 二つの前提の一方が否定判断であれば結論が否定判断になることと, そのときに P が周延されていなければならないために大前提が特称否定判断

(O) であることはあり得ません．小前提が全称肯定判断 (A) のときに S は不周延であるために結論は特称判断になります．それから, 大前提が全称肯定判断 (A) のときと特称肯定判断 (I) のときは結論は特称肯定判断 (I) になります．そして, 小前提が全称否定判断 (E) のときは S が周延され, 結論も否定判断になるために P も周延されていなければなりません．以上から大前提と結論の双方が全称否定判断 (E) になります．以上から, EAO, EIO, AAI, IAI, AEE に限定されます．

これらの考察を以下の表にまとめておきましょう[*33]:

認められる判断の組合せ

第一格:	AAA(Barbara)	EAE(Celarent)	AII(Darii))	EIO(Ferioque)
第二格:	EAE(Cesare)	AEE(Camestres)	EIO(Festino)	AOO(Baroco)
第三格:	AAI(Darati)	IAI(Disamis)	AII(Datisi)	EAO(Felapton)
	OAO(Bocardo)	EIO(Ferison)		
第四格:	AAI(Bramantip)	AEE(Camenes)	IAI(Dimaris)	EAO(Fesapo)
	EIO(Fresison)			

このように定言三段論法を適用するにも周延関係に基くために大前提, 小前提や結論に煩雑な格の組合せがあります．さらに概念の外延の存在が前提にあるために大前提に使える命題は,「**明らかに真であると判断できるもの**」か「**帰納的に求められるもの**」でなければなりません．ここで仮説に三段論法を適用するためには, その仮説が前述のように経験的に正しいか, 帰納的に正しいと言えなければなりませんが, さらに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．とは言え, 経験的にも正しいかどうか吟味が不十分な仮説では, どのような結論が出ても不思議がありません．実際, イスラム神学の一派で, アッバース朝の公認神学の「**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異端的な結論も導出していたといいます [7]．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比喩として捉え, 神を**知識**や**理性**と見なしました[*34]．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**」という考えを持っていましたが, やがて,「**正統派**」によってそ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, そして改革を受けれられない大衆によって再三, 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

[*33] この表で括弧内は中世の学生が暗記するために使った単語で, 子音を外すと格が現われるという仕組です．

[*34] 神を νους や λόγος とみなすために同時代の神学者からは「**「神よ!」と呼びかけるのではなく,「知恵よ!」と呼びかければ良いではないか**」と皮肉られている程です．

類似していなくもありません[*35].

2.4　集合論について

2.4.1　集合論言語について

「それが何であるか?」という問に対する説明規定が概念であり, 概念で説明され得るものの集まりが外延ですが,「**それ自身でないもののあつまり**」という命題には外延が存在しません. どのような命題にも外延が存在しているという前提でフレーゲが開始した論理主義は厳密な数学の基礎を与えるかのように見えましたが, この命題から体系に矛盾が生じて呆気なく破綻しています. 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. ラッセルは述語そのものの考察を行い, それが 1908 年に「型の理論」[51] になりますが, 同時期にツェルメロ (Zermelo) の論文[*36]で集合そのものに制約を入れるという公理的集合論を提唱します. 現在の公理的集合論の公理系は, ツェルメロの公理系 (Z) を基にフレンケル (Frankel) の公理系 (F) と, それらの公理と独立した「**選択公理**」と呼ばれる重要な公理 (C) があり, これらの公理の組み合わせで Z, ZF や ZFC 等と略記されます. そして, この集合論には「**集合論言語**」と呼ばれる, その体系で扱う対象を語るための「**言語**」があります. この言語の記号系を以下に示します:

集合論で用いる記号系

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

元が集合に属するという意味で用いる記号 “$\in$” と対象の同一性を示す記号 “$=$” の他は論理式の論理和 “$\vee$”, 論理積 “$\wedge$”, 否定 “$\neg$” と含意 “$\supset$”, 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 “$\overset{\mathrm{Def.}}{=}$” を導入することで ‘$a \overset{\mathrm{Def.}}{=} b$’ 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以下で定義します:

[*35] 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.

[*36] 「Investigations in the foundation of set theory」[51], pp199-215

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

これらの記号を用いて集合論の論理式を次の形成規則で定義します:

論理式の形成規則

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

この論理式の形成規則を持つ系を「**集合論言語**」と呼び, $\mathscr{L}$ と表記します. この形成規則は帰納的で, この表記を §3.3 で述べた BNF 記法を使っても表現できますが, こ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, その意味や意義は集合論の公理系が規定します. さらに言語的には $A = B$ が充される状態と $\neg(A = B)$ が充される状態がそれぞれ真と偽に対応し, 真や偽といったオブジェクトが存在するというものではありません. そして, 対象 A, B の同一性と非同一性を示す状況のことを「**意味**」と呼びます. したがって, 記号 $\equiv$ は論理式の意味が同一であることを指示する記号であり, 記号 $\neg$ は論理式の意味を反転することを意味する記号です.

2.4.2　集合論の公理系

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

集合論の公理系

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

■外延性公理 (Axiom of extensionary):　外延から集合が一意に定まることを保証します. 集合の外延の記述は $\{a, b, c, d\}$ のように括弧 $\{\ \}$ の中に区切記号 “,” で元の列を記載し, 外延性公理から列の順番に依存せずに集合が一意に定まります. ここで二つの記号 $\subseteq$ と $\subset$ を定めておきます:

$$\begin{aligned} a \subseteq b &\overset{\text{Def.}}{=} \forall x(x \in a \supset x \in b) \\ a \subset b &\overset{\text{Def.}}{=} a \subseteq b \wedge \neg(a = b) \end{aligned}$$

集合 x, y に対して $x \subseteq y$ のときに集合 x を集合 y の「**部分集合**」, $x \subset y$ のときに集合 y の「**真部分集合**」と呼びます. これらの集合は外延性公理から一意に定まります. また, 部分集合を使うと外延性公理は $\forall x \forall y(x \subset y \wedge y \subset x \supset x = y)$ と書換えられます.

■対公理 (Axiom of pairing):　集合 x, y を成分とする「**対集合**」の存在を保証します. 集合 x, y の対集合を $\{x, y\}$ と表記しますが, 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. また, $n > 2$ 個の集合からも対集合が帰納的に構成可能です. 実際, 対集合の公理から $\{\{x_1, x_2\}, x_3\} = \{x_1, \{x_2, x_3\}\}$, $\{\{x_1, x_2, \ldots, x_{n-1}\}, x_n\} = \{x_1, \{x_2, \ldots, x_{n-1}, x_n\}\}$ であることから $\{x_1, \ldots, x_n\}$ と表記します. また,「集合の集合」を「**集合族**」と呼びます. 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 との関係は何も述べていません. そこで, $\langle x, y\rangle \overset{\text{Def.}}{=} \{\{x\}, \{x, y\}\}$ 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, 成分が 3 以上の順序対 $\langle x_1, x_2, \ldots, x_n\rangle$ も $\langle x_1, \langle x_2, \ldots, x_n\rangle\rangle$ で帰納的に定義できます．

■和集合公理 (Axiom of union set):　集合 x の成分を全て含む集合の存在を保証します．特に x が**集合族**」, すなわち, 集合の集合のときに集合族の全ての集合の元を含む集合の存在を保証し, この集合を「**和集合**」と呼び, $\cup x$ と表記します．特に, 対集合 $\{x, y\}$ の和集合を $x \cup y \overset{\text{Def.}}{=} \cup\{x, y\}$ で定め,「**集合 x, y の和集合**」と呼びます．

■冪集合公理 (Axiom of power set):　集合 x の任意の成分をその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する集合を「**冪集合**」と呼び, 集合 x の冪集合を $\mathfrak{P}(x)$ と表記します．なお, 冪集合公理は $\forall x \exists y \forall z (z \in y \equiv z \subseteq x)$ と表記可能で, さらに $z \in \mathfrak{P}(x) \equiv z \subseteq x$ も証明できます．

■空集合公理 (Axiom of empty set):　元を持たない集合の存在を保証します．この公理と外延公理から唯一存在する集合を「**空集合**」と呼び, “$\emptyset$” と表記します．この空集合　は Lisp の null や Python の None のように便利に用いられます．

■無限集合公理 (Axiom of infinity set):　無限集合のひとつの創り方を定める公理で, この公理の定める集合の創り方は, v が集合のときに $\emptyset \cup \{v\}$ も集合であることを保証します．ここで集合 v を空集合 $\emptyset$ とすると, この公理から

$$\emptyset,\ \emptyset \cup \{\emptyset\},\ \emptyset \cup \{\emptyset \cup \{\emptyset\}\},\ \emptyset \cup \{\emptyset \cup \{\emptyset \cup \{\emptyset\}\}\},\ \ldots$$

という集合の無限列が構成できますが, この無限列が自然数になります:

自然数の定義

$$\begin{array}{llll}
0 & \overset{\text{Def.}}{=} & \emptyset & \\
1 & \overset{\text{Def.}}{=} & \emptyset \cup \{\emptyset\} & = \{0\} \\
2 & \overset{\text{Def.}}{=} & \emptyset \cup \{\emptyset \cup \{\emptyset\}\} & = \{0, 1\} \\
3 & \overset{\text{Def.}}{=} & \emptyset \cup \{\emptyset \cup \{\emptyset \cup \{\emptyset\}\}\} & = \{0, 1, 2\} \\
\cdots & \cdots & \cdots & \\
n+1 & \overset{\text{Def.}}{=} & \emptyset \cup \{\emptyset \cup \{\emptyset \cup \{\emptyset \cup \ldots\}\}\} & = \{0, 1, 2, \ldots n\} \\
\cdots & \cdots & \cdots &
\end{array}$$

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は自然数をその体系内に包含します．そして, 自然数 a に対して

$a+1 \stackrel{\text{Def.}}{=} \emptyset \cup \{a\}$ と定義し, この $a+1$ を a の「**後続**」, あるいは「**後者**」と呼びます*37. この列の行き着く先が「**超限順序数**」で, すべての自然数の集合として定義されます:

超限順序数

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

超限順序数 ω は末端ではありません. この超限順序数に対しても後者関係から $\omega+1, \omega+2, \ldots, 2\omega = \omega+\omega, \ldots$ と続々と数が構成できます.

さらに自然数 a, b に対して $a < b \stackrel{\text{Def.}}{=} a \in b$ と $a \leq b \stackrel{\text{Def.}}{=} a \in b \vee a = b$ で記号 “$<$” と記号 “$\leq$” を定義することで「**大小関係**」が自然に導入できます.

■置換公理図式 (Axiom schema of replacement):　述語 ϕ があ式中に含まれるために「図式 (schema)」です. この公理図式はフレンケルが導入したもので, ツェルメロの論文では「**分出公理 (Axiom of seperation)**」と呼ばれる次の公理です:

分出公理 (Axiom of seperation)

$$\text{A7}' \quad \forall x \exists y \forall z (z \in y \equiv (z \in x \wedge \phi(z)))$$

この分出公理は, 集合 x 上で命題 ϕ を充す元はある集合 y の元になるという意味ですが, 既存の集合から命題 ϕ を使って切り出したものも集合であると言い換えられます. ところで, 分出公理の函数 ϕ は 1 変数ですが, 置換公理図式の ϕ は 2 変数の函数で, 第 1 変数に対しては単射写像 $\phi_x() = \phi(x,)$ を定めることが条件です. そのために図式の函数 $\phi(x, y)$ を $\psi(x) \wedge x = y$ で置き換えることで分出公理が得られます. また, 分出公理の述語 ϕ を充す集合 x の元から構成される集合を $\{u \in x \mid \phi(u)\}$ と表記します. ただし, 一般的な命題 $\phi(x)$ の外延 $\{x \mid \phi(x)\}$ は, その元が集合の元である保証がないために集合であると断言できません. そこで, この一般的な命題の外延を「**類**」, あるいは「**クラス (class)**」と呼んで「**集合**」と区別します. オブジェクト指向の「**クラス**」が「**クラス**」と呼ばれることも, クラスが複数の「**述語**」に対応する「**属性値**」や「**メソッド**」から構成され, それらが定める外延が既存の「**集合**」から切り出したものであるとは限らないためです. ここで素朴集合論で問題になった「**ラッセルの逆理**」をもう一度考えてみましょう. 命題 $x \notin x$ を充す元を集合 x から切り出さなければなりませんが, そのようなことができないために結果として公理的集合論の体系から排除できます.

また, 分出公理から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 $\{u \in x \mid u \in y\}$ 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 $\{\langle u, v \rangle \mid u \in x \wedge v \in y\}$ 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, 共通集合は集合 x から $\tau \in y$ を充す元を取り出したも

*37 自然数の後者関係についてはフレーゲの「概念記法」[31] ではじめて厳密に述べられています.

ので, 直積集合は対集合の元 $\langle u, v\rangle$ で $u \in x$, $v \in y$ を充すものを切り出すことで得られています. ところで, フレンケルが置換公理図式で分出公理を置き換えた理由に「**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」[25] 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

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

で定めます. このとき函数 f の値域 $\mathrm{rng}(f)$ は

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

によって与えられますが, この値域 $\mathrm{rng}(f)$ が集合になることは分出公理からは導出できません. ところで, 公理的集合論では後述するように集合論の公理から体系内に自然数が包含されます. したがって, 函数 f は体系内に自然に定義される函数で, このときに $\phi(n, w) \stackrel{\mathrm{Def.}}{=} \mathfrak{p}^{n-1} = w$ とすると, 置換公理図式から, このような函数による集合の像も集合になることが保証され, $\mathrm{rng}(f)$ も集合になります. 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 (Axion of regularity): 基礎公理 (Axiom of foudation) とも呼ばれる公理で, この公理によって論理式 $a \in a$ の外延が集合から排除され, さらに集合と集合の元が区別されます. また, この公理は $\neg(x = \emptyset) \supset \exists y(y \in x \wedge y \cap x = \emptyset)$ とも記載され, 空集合 $\emptyset$ のみが集合であると同時に, 任意の集合に包含される元でもあります. この正則公理によって $\ldots, x_3 \in x_2, x_2 \in x_1, x_1 \in x_0$ を充す**集合の底なしの無限列**:「$\ldots x_3, x_2, x_1, x_0$」も排除されます. この底なしの無限列があると困ることを説明しておきましょう. 空集合公理と無限公理から自然数と大小関係が導入できますが, 正則性公理があれば $\ldots \in x_2 \in x_1 \in x_0$ 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 $x_n \in \ldots \in x_2 \in x_1 \in x_0$ 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*38. 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′:

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

無限降下列の非存在性

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

になります．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 ⊃ A8′ が成立しますが, その逆の A8′ ⊃ A8 が成立するためには次の「**選択公理**」が必要になります [25].

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

バナッハ-タルスキの逆理

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

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

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

この逆理は, ゴルフボールの表面を適当に分割し, それらを貼り合せるだけで地球が覆えると主張しています．牛の皮程もない蜜柑の皮で, 砦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[*39]．さすがにこの定理は日常的な常識から大きく外れていますが, 選択公理自体はそれを認めたときの御利益が圧倒的に大きな公理です．なお, 選択公理を認めないときは, 自然数に対しては任意の自然数の部分集合が最小元を持つことを利用します.

ここで A1 から A9 までの公理系の組み合せ表を示しておきましょう:

集合論の公理系

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

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

[*39] 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.

2.4.3 順序数

ZFC 公理系にて順序数を次で定義します.

順序数の定義

$$\begin{array}{lcl} \mathrm{Trans}(u) & \overset{\mathrm{Def.}}{=} & \forall x, y(x \in u \wedge y \in x \supset y \in u) \\ \mathrm{Ord}(\alpha) & \overset{\mathrm{Def.}}{=} & \mathrm{Trans}(\alpha) \wedge \forall x, y \in \alpha(x \in y \vee x = y \vee y \in z) \end{array}$$

最初の述語 $\mathrm{Trans}(u)$ は集合 u が推移的であることの定義です. ここで述語 $\mathrm{Trans}(u)$ 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 $\mathrm{Ord}(\alpha)$ 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 $\mathrm{Ord}(\alpha)$ 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 $\omega = \{0, 1, 2, 3, \ldots\}$ の元 u は $\mathrm{Trans}(u)$ を充すために推移的で, さらには $\mathrm{Ord}(u)$ を充すために順序数になります. 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 $\{\omega, \mathfrak{P}(\omega), \mathfrak{P}^2(\omega), \ldots\}$ 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, $\alpha \in \mathrm{Lim}$ 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

$$\mathrm{OR} \overset{\mathrm{Def.}}{=} \{\alpha \mid \mathrm{Ord}(\alpha)\}$$

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 $\mathrm{OR} \in \mathrm{OR}$ になって OR の後者 $\mathrm{OR}+1$ を考えられますが推移律から $\mathrm{OR} \in \mathrm{OR}+1$, 一方で OR は順序数の全体なので $\mathrm{OR} + 1 \in \mathrm{OR}$ と矛盾が生じます. これが素朴集合論

で「**ブラリ＝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 では正則性公理から「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 上で集合論言語 $\mathscr{L}$ が定められているとします．このことを $\langle M, \in \rangle$ と表記し，集合論言語 $\mathscr{L}$ の「**$\in$-構造**」，「**$\in$-モデル**」，あるいは単に「**モデル**」と呼び，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 $\mathscr{L}$ 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 $\mathscr{L}$ 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」が誰を指示するのか不明なために真偽の判断ができません．これはモデルでも同様で，モデル M で文 φ の意味が真であったとしても別のモデル N では偽となることがあります．ところが，文 $A \supset A$，日常語なら「**A は A である**」のように文脈と無関係に常に真になる文もあります．このように文脈とは無関係に常に真となる文のことを「**恒真式**」あるいは「**トートロジー (tautlogy)**」と呼びます．

2.5　函数と関係

2.5.1　函数について

論理学に函数概念を最初に導入した人はフレーゲです．フレーゲによると函数の本質は「**不飽和**」であると述べています．この不飽和の意味は，たとえば，x^2 といった函数は 2^2 と異なり変数 x に値が設定されない限り，その値を持ちません．つまり，函数は意味を持たせるために「**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．この表記ではチャーチの λ-表記法に似た表記で，函数の表示でメタ変数を用います．たとえば「与えられた数の二乗を返す函数」は x^2 という表記ではなく，メタ変数 ξ を導入して ξ^2 と表記し，二変数の函数であれば $f(x, y)$ と $f(y, x)$ が異なることから変数のタプルの順番，つまり，「**項の位置**」を導入し，第 1 変数に ξ，第 2 変数に ζ というメタ変数を割り当て，$f(\xi, \zeta)$ と記述します．ちなみに λ-表記では $\lambda(x, y).f(x, y)$ になります．また，三変数以上の多変数函数については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 $\mathfrak{f}$ 等のドイツ文字で表記しています．ところで，函数は，その構造に関心がな

い限り, 一般的にはメタ記号を用いた表記は用いずに ‘$f(x)$’ や ‘$g(x,y,z)$’ のように f,g 等の函数名と $(x),(x,y,z)$ のような変数を指示するタプル, つまり, 変数の列を括弧 “( )” で括ったものの結合として表記します．この本では, 函数それ自体を明記する目的で λ-式表記を用い, たとえば, ‘$\lambda(x,y).f(x,y)$’ と表記しますが, 簡略化した表記として, 変数を $x,y,z,\ldots$ とアルファベットのうしろ側を用いて ‘$f(x.y)$’ と表記します.

2.5.2　関係について

集合 S での関係は任意の $x,y \in S$ に対して ‘$x = x$’ と同じ意味を持つものが真, 任意の $x,y \in S$ に対して ‘$x = y$’ と同じ意味を持つものが偽です．真と偽は True, False といった真理値として表現され, 関係は真理値を返す二変数函数とも言い代えられます．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ます. 関係の表記は二項の間に函数名を配置する中値表記が用いられるため, ここでは a と b の間の関係を ‘$a \mathrm{R} b$ 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: $x \mathrm{R} x$
2. 対称律: $(x \mathrm{R} y) \supset (y \mathrm{R} x)$
3. 反対称律: $((a \mathrm{R} b) \wedge (b \mathrm{R} a)) \supset (a = b)$
4. 推移律: $((x \mathrm{R} y) \wedge (y \mathrm{R} z)) \supset (x \mathrm{R} z)$
5. 全順序律:$(a \mathrm{R} b) \vee (b \mathrm{R} a)$

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: $x \mathrm{R} x$
2. 対称律: $(x \mathrm{R} y) \supset (y \mathrm{R} x)$
3. 推移律: $((x \mathrm{R} y) \wedge (y \mathrm{R} z)) \supset (x \mathrm{R} z)$

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 $\dfrac{a}{b} \sim \dfrac{m}{n}$ という関係 “$\sim$” を次で定めましょう:

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

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 $\dfrac{a}{b}$ と $\dfrac{c}{d}$ が分数 $\dfrac{m}{n}$ 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 $a = h \cdot m, b = h \cdot n$, $c = k \cdot m, d = k \cdot n$ 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:

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

と, この結果から $\dfrac{a}{b} = \dfrac{c}{d}$ であれば $\dfrac{a}{b} \sim \dfrac{c}{d}$ になることが判りました. 実際は $a \cdot n - b \cdot m = 0$ の両辺を $b \cdot n$ で割ると $\dfrac{a}{b} = \dfrac{m}{n}$ になるために約分から同じ分数が得られることと関係 "$\sim$" を充たすことは同値です.

分数 $\dfrac{a}{b}$ 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 $\mathbf{N} \times (\mathbf{N} \backslash \{0\})$ として表現できます. しかし, $(1, 2) \neq (2, 4)$ であっても $\dfrac{1}{2} = \dfrac{2}{4}$ 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 "$\sim$" 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 $\sim$ 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 $\dfrac{1}{2}$ の同値類が $\left\{ x \in \mathbf{N} \times (\mathbf{N} \setminus \{0\}) \mid x \sim \dfrac{1}{2} \right\}$ で, 分数 $\dfrac{1}{2}$ 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 (preoder)**」,「**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**と呼びます:

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

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: $x \mathrm{\ R\ } x$
2. 推移律: $((x \mathrm{\ R\ } y) \wedge (y \mathrm{\ R\ } z)) \supset (x \mathrm{\ R\ } z)$
3. 反対称律: $((a \mathrm{\ R\ } b) \wedge (b \mathrm{\ R\ } a)) \supset (a = b)$

前順序と半順序では $a \mathrm{\ R\ } b$, $b \mathrm{\ R\ } a$ 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 $a \mathrm{\ R\ } b$, $b \mathrm{\ R\ } a$ のいずれかが必ず成立します:

全順序

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

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

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

2.6　代数的構造について

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

$$1 \div 2 = 0 \text{ あまり } 1$$

と商と剰余を併記したものを答とし, $1 \div 2 = \dfrac{1}{2}$ 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ましょう. そこで対象の自然数を集合 S, 足算や割算といった記号を記号 “$*$” と表記し, この記号を「**演算子**」, それから演算子 “*” の影響を受ける集合 S の元を「**被演算子**」と呼びます. それから集合と演算の対 $(S, *)$ で表記し, 足算, 掛算の何れのことを話題にしているか明瞭にします. すると, ここで話題にしている演算の大きな性質は集合 S の二つの元から新たな集合 S の元を生成する能力で, この演算 “$*$” が集合 S の対 $S \times S$ から S への写像になっていることです. この性質は「**閉じている**」と呼ばれる演算の性質で, この集合 S と閉じた演算 “$*$” の対 $(S, *)$ を「**マグマ (magma)**」と呼びます:

マグマの定義

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

- 演算 “$*$” が閉じていること: $a * b \in S$.

この定義から自然数の集合 $\mathbf{N}$ の和 “$+$”, 積 “$\times$”, 商 “$\div$” といった計算処理は全て閉じた演算で, $(\mathbf{N}+)$, $(\mathbf{N}, \times)$ と $(\mathbf{N}, \div)$ はマグマになります[*40]. ところで, これらの演算の詳細を眺めてみましょう. まず, 足算 “$+$”, 掛算 “$\times$” に共通する性質に $(1+2)+3 = 1+(2+3)$, $(3 \times 4) \times 5 = 3 \times (4 \times 5)$ という性質があります. しかし, 割算 “$\div$” は $(12 \div 6) \div 2 \neq 12 \div (6 \div 2)$ と, 式の括弧を動かすと結果が違います. この括弧は演算の順番に対応し, 最初に $a * b$ を計算して c との演算を計算する方法の $(a * b) * c$ と $b * c$ を計算して a との演算を計算する方法の $a * (b * c)$ が等しくなるかどうかという状況に対応します. この ‘$(a * b) * c = a * (b * c)$’ という関係は「**結合律**」と呼ばれる重要な関係式です. ところで日常の言葉で括弧 “()” がなければ微妙な解釈の違いが生じることがあります. たとえば「**太った猫と犬**」という文を「**(太った猫) と犬**」と解釈するか「**太った (猫と犬)**」[37] と解釈するかで意味が異なります[*41]. 集合が結合律を充すということは, このような曖昧さも排除した, 演算の順番に依存せずに一意に演算結果が定まるということを意味します. その意味で, 割算は本質的に足算や掛算と異なる演算であることを示しています. そこで, 閉じた演算を持ち, 結合律を充たすという二つの性質を持つ集合に新しい概念を導入しましょう. これが「**半群 (semigroup)**」と呼ばれる概念です:

半群の定義

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

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

同じ集合であっても演算によって入る構造が異なります. 実際, $(\mathbf{N}, +)$ は半群になりますが, $(\mathbf{N}, \div)$ はマグマであっても前述のように結合律を充さないために半群にはなりません.

つぎに自然数 $\mathbf{N}$ に入れた演算を吟味してみましょう. まず, 足算 “$+$” と掛算 “$\times$” には面白い性質があります. これは $1+2 = 2+1 = 3$ や $2 \times 3 = 3 \times 2 = 6$ と演算の順序を入れ替えられる性質です. つまり, 任意の $a, b \in S$ に対して $a * b = b * a$ を充すという性

[*40] $(\mathbf{N}, \div)$ は商のみを返す演算とします.
[*41] 「太った」の参照範囲 (スコープ) が猫だけなのか, 猫と犬の両方なのかどうかが分からない文の構造のためです.

質です．このように二項演算子の被演算子の順序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．しかし, 割算 “÷” は $4 \div 2 \neq 2 \div 4$ であるために可換でなく, 引算 “$-$” も $1-2 \neq 2-1$ であるために可換にはなりません．このように可換でない演算子の性質を「**非可換 (noncommutative)**」と呼びます．

今度は自然数 $\mathbf{N}$ の元 $0, 1$ に注目してみましょう．まず, $(\mathbf{N}, +)$ で 0 は, 任意の自然数 a に対して $a+0=0+a=a$ を充します．同様に $(\mathbf{N}, \times)$ で 1 も $1 \times a = a \times 1 = a$ を充たしています．このように集合 S の閉じた演算 “$*$” で任意の $a \in S$ に対して $a * u = u * a = a$ を充たす集合 S の元 u を「**単位元**」と呼びます．そして, 単位元を持つ半群を「**単系 (モノイド, monoid)**」と呼びます．この単系の例には自然数 $(\mathbf{N}, +)$, $(\mathbf{N}, \times)$ と有理数 $(\mathbf{Q}, \times)$ が挙げられます．ここで有理数 $(\mathbf{Q}, \times)$ が単系であることは, まず, $1 \in \mathbf{Q}$ が単位元になり, さらに $a, b, c \in \mathbf{Q}$ に対して $a \times b \in \mathbf{Q}$, $a \times 1 = 1 \times a = a$, $(a \times b) \times c = a \times (b \times c)$ が成立することから演算 “$\times$” は $\mathbf{Q}$ で閉じた演算で, 結合律も成立することから判ります．この有理数 $\mathbf{Q}$ の 0 以外の元には興味深い性質があります．まず, $a \in \mathbf{Q}_+$ には 0 と異なる $a = \dfrac{m}{n}$ を充す二つの自然数の対 (m, n) が存在します．この自然数の対 (m, n) を入れ替えた自然数の対 (n, m) から定まる分数 $b = \dfrac{n}{m}$ と掛算をしてみましょう．すると掛算 “$\times$” の可換性から $a \times b = \dfrac{m \times n}{n \times m} = \dfrac{m \times n}{m \times n} = 1$ となり, 分数 $\mathbf{Q}$ 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 $\mathbf{Q}$ の例のように単位元 u を持つ単系 $(S, *)$ で $a \in S$ に対して $a * b = b * a = u$ を充たす元 b 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 $(\mathbf{Q}, \times)$ の全ての元は正則で, このように全ての元が正則になる単系を「**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, $(\mathbf{N}, +)$ と $(\mathbf{N}, \times)$ は群でしょうか? $(\mathbf{N}, +)$ では $a+b=0$ になる元は双方が 0 のときだけで, $(\mathbf{N}, \times)$ でも $a \times b = 1$ になる元は双方が 1 のときだけです．したがって, これらの単系は群にはなりません．これらの単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, $(\mathbf{N}, +)$ に負の数を導入して整数 $\mathbf{Z}$ を構築すれば, この整数 $\mathbf{Z}, +)$ は群になります．また, $(\mathbf{N}, \times)$ に分数を導入して $(\mathbf{Q}_+, \times)$ にすると群になります．なお, 演算が可換な群のことを「**可換群 (commutative group)**」と呼び, さらにこの可換群の演算を記号 “$+$”, 単位元を 0, そして, a の逆元を $-a$ と表記して「**加法群 (additive group)**」とも呼びます．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 $\mathbf{N}$ や整数 $\mathbf{Z}$ 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 $\mathbf{Z}$ は足算と掛算という二つの演算を持ち, 演算 “$+$” なら可換群, 演算 ”$\times$”　は演算 “$+$” の単位元 0 を除く集合 $\mathbf{Z}\setminus\{0\}$ は単系で, 群にはなりません．そして, 整数は足算と掛算が混在する計算で次の性質を充たします:

$$\begin{array}{ll} 1. & a\times(b+c)=a\times b+a\times c \\ 2. & (a+b)\times c=a\times c+b\times c \end{array}$$

この性質は括弧の外にある掛算 “$\times$” を各被演算子を分配している格好になっているために「**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.
- $(S\setminus\{0\},*)$ は半群である.
- $(S,+,*)$ は分配律を充す:
  1　$a*(b+c)=a*b+a*c$
  2　$(a+b)*c=a*c+b*c$

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 $\mathbf{Z}$ で, 環としての構造を明確にするときは「**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 $\mathrm{M}(n)$ で, 体の代表的な例としては有理数 $\mathbf{Q}$, 実数 $\mathbf{R}$ や複素数 $\mathbf{C}$ が挙げられます．また, ここでの環の定義から零の重要な性質 $x*0=0*x=0$ とが導かれます[*42]

[*42] 0 の性質として $x\times 0=0\times x=0, x+0=0+x=x$ を明確に記載された最古の著作は 7 世紀のブラマーグプタの「ブラーマ・シッダーンティカ (Brāhma Siddhāntica」であり, これがイスラム圏に伝わって 8 世紀末のアッバース朝アッラシードの治世に「アル・シンドヒンド (al-Sindhind)」で知られていたとのことです [24]

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 $\mathbf{Z}$ の元を加法群である有理数 $\mathbf{Q}$ 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 $\mathbf{Q}$ の元に整数 $\mathbf{Z}$ の元を左側から掛けたものは有理数 $\mathbf{Q}$ 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 $\mathbf{Z}$ の有理数 $\mathbf{Q}$ 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 $(2 \times 3) \times \dfrac{2}{5} = 2 \times \left(3 \times \dfrac{2}{5}\right)$ から判るように整数側で積演算して有理数に作用させたものと, 有理数に近い側から作用させていったものの結果が一致する性質があります. この性質は有理数と整数の積が有理数の分子との積になり, このときに双方が全て整数であることから結合律を充すために成り立つ性質です. これも作用の性質としてまとめると, 左作用であれば $(r_2 * r_1) * a = r2 * (r_1 * a)$, 右作用であれば $a * (r_1 * r2) = (a * r_1) * r_2$ 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

$$\begin{array}{ll} \mathrm{L}_1. & r * (x + y) = r * x + r * y \\ \mathrm{L}_2. & (r + s) * x = r * x + s * x \end{array}$$

右作用に関する分配律

$$\begin{array}{ll} \mathrm{R}_1. & (x + y) * r = x * r + y * r \\ \mathrm{R}_2. & x * (r + s) = x * r + x * s \end{array}$$

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 は有理数に手を加えない恒等写として作用しています. 実際, $1 = 1 \times 1 = 1 \times \ldots \times 1$ 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 にには写像 $* : R \times A \to A$ が存在する.
- $r * (x + y) = r * x + r * y$
- $(r + s) * x = r * x + s * x$
- $(r * s) * x = r * (s * x)$
- $1 * x = x$

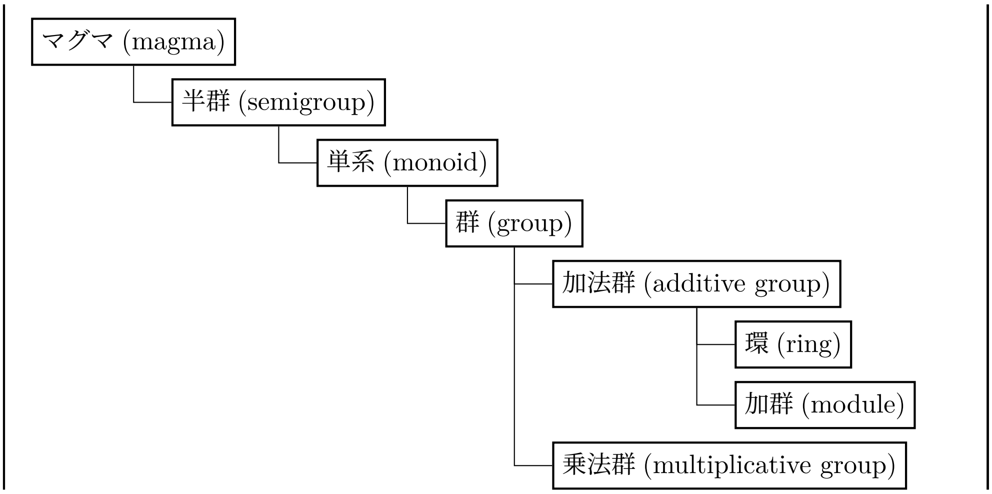
右 R -加群の定義

- 集合 $(A, +)$ は可換群である.
- 環 R と集合 A に写像 $* : A \times R \to A$ 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 $\mathbf{Z}$ の乗法 $\times$ による作用を持つ加法群 $\mathbf{Q}$, 環を実数 $\mathbf{R}$, 作用を通常の積 $\times$ による n 次の実ベクトル空間が挙げられます. また, n 次のベクトル空間も R-加群の一例です. 実際, n 次の (実) ベクトルの空間 $\mathrm{Vect}(n) = \{(v_1, \ldots, v_n) : v_1, \ldots, v_n \in \mathbf{R}\}$ とし, 係数環として実数 $\mathbf{R}$ の作用を各成分単位の積, つまり, $v \in \mathrm{Vect}(n)$, $a \in \mathbf{R}$ に対して $a \cdot v = (a \cdot v_1, \ldots, v_n)$ で定めると, この作用は左右の R-加群の条件を充たすためです.

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

代数的構造の階層



2.7　圏 (Category)

2.7.1　はじめに

ここからは数学の「**圏**」について解説します．この「**圏**」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 (κατηγοριά)**」，その日本語訳として「**範疇**」が対応します．このカテゴリーを最初に扱ったアリストテレスの著作「範疇 (カテゴリー) 論」は「**真実を探求するための道具**」としての「**道具 (オルガノン (ὄργανον))**」と呼ばれる著作群の筆頭に置かれ，哲学を学ぶ上で最初に読まれるべき書物とされていましたが [2]，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[54] は「**Category**」という言葉はアリストテレスとカントの「**カテゴリー**論」，「**函手 (functor)**」はカルナップ (Carnap) の著作に由来すると述べています．この圏論はそもそも代数的位相幾何学のホモロジーと呼ばれる数学的対象の計算から現れたものですが，現在では，数学全般，さらには哲学や人工知能 [60] にも関連しています．

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「**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**: $A, B, C, \ldots$
- **矢 (射)** ： $f, g, h, \ldots$
- **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 $\mathrm{dom}\ f$, 終域を $\mathrm{cod}\ f$ と表記する.
- **矢の表記**: 矢 f に対して $A = \mathrm{dom}\ f$, $B = \mathrm{cod}\ f$ とするとき $f : A \to B$, あるいは $A \xrightarrow{f} B$ と表記する.

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

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

ここでメタグラフ $\mathscr{C}$ の矢 $f : A \to B$ はメタグラフ $\mathscr{C}$ の二つの対象 A, B に順序を含めた関係を与え, この矢を ‘$\xrightarrow{f}$ と表記することで矢 $f : A \to B$ から $A \xrightarrow{f} B$ へと図式化が行えます. また, この図式化によって矢 f が対象 A と B を繋ぐ機能を持つものとしての性格が明瞭になります. その結果, メタグラフの矢は伝統的形式論理学での「**繋辞 (copula)**」と同様の機能を持つものと考えられます. 繋辞は命題「A は B である」の中の「**… は… である**」のように主語と述語の二項間を取り持ち, 命題を構成します [6]. 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, 関係の第一引数を ξ, 第二引数を ζ と表記しています. メタグラフの矢はこの関係をより抽象化し, その機能を明瞭にするために図式化をさらに推し進めたものになっています.

2.7.3　矢について

さて,「**メタグラフ**」に含まれる「**グラフ**」という言葉から「**函数のグラフ**」等の「**グラフ**」を連想される方も多いかと思います. 函数グラフは点 x における函数 f の値 $f(x)$ を

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

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

2.7.4 図式 (diagram) について

メタグラフ $\mathscr{C}$ の対象と対象間の結合律を充す矢の類を「**図式 (diagram)**」と呼びます. 図式は対象を頂点 (vertex), 矢の向きを持つ辺 (edge) の平面グラフとして図示できます. 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

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

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, $\mathrm{cod}\ f_i = \mathrm{dom}\ f_{i+1}$ を充たす矢の列 $f_1, \ldots, f_i, f_{i+1}, \ldots, f_n$ 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 “$\cdots$” を含めて図示します. 図式は対象, 矢と省略記号 “...” から構成され, 図式の可視化は対象を頂点 (vertex), 矢を辺 (edge) とし, これらに省略記号 “$\cdots$” で構成さ

れる 2 次元グラフになります. また, 辺を辿ることは矢の合成を行うことに対応し, 図式を構成する矢が結合律を充たすために, これらの合成は一意に定まります. このことからも図式で矢の合成に由来する曖昧さを排除するためには矢が結合律を充たさなければならなりません. また, 図式中のある対象を始域と終域とする複数の矢の道が存在し, それらの経路の何れを通っても道に対応する矢の合成が一致するときに「**可換図式**」と呼びます. ここで重要な可換図式を以下に示しておきます:

$$
\begin{tikzcd}
A \arrow[r, "f"] \arrow[dr, "h"'] & B \arrow[d, "g"] \\
& C
\end{tikzcd}
\qquad\qquad
\begin{tikzcd}
C \arrow[r, "h"] & A \arrow[r, shift left, "f"] \arrow[r, shift right, "g"'] & B
\end{tikzcd}
$$

左の図式は可換図式で最も重要な図式で対象 A から対象 C に向う道として合成矢 $g \circ f$ と矢 h の二つが存在し, これら二つの矢が一致することを意味します. この矢の合成 $g \circ f$ を矢 h の「**分解 (factorization)**」と呼びます. また, 右の図式では対象 C から B に向かう道として $C \xrightarrow{h} A \xrightarrow{f} B$ と $C \xrightarrow{h} A \xrightarrow{g} B$ の 2 つがあり, それぞれに対応する矢の合成 $f \circ h$ と $g \circ h$ が一致することを示しています. また, メタグラフ $\mathscr{C}$ 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 $f \in \mathrm{Hom}_{\mathscr{C}}(A, B)$, $g \in \mathrm{Hom}_{\mathscr{C}}(B, C)$ に対して

$$
\begin{tikzcd}
A \arrow[r, "f"] \arrow[dr, "f"'] & B \arrow[d, "\mathrm{id}_B"] \arrow[dr, "g"] & \\
& B \arrow[r, "g"'] & C
\end{tikzcd}
$$

を可換, すなわち, $\mathrm{id}_B \circ f = f \circ \mathrm{id}_B$, かつ, $\mathrm{id}_B \circ g = g \circ \mathrm{id}_B$ を充たす矢 $B \xrightarrow{\mathrm{id}_B} B$ 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 $\mathrm{id}_A = \iota_A$ 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できます.

2.7.5 単射, 全射, 同型

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

単射, 全射, 同型

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

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

これら単射, 全射, 同型といった矢の性質は対象が集合であれば通常の写像の単射, 全射, 同型に対応します．実際, 対象が集合であれば矢は通常の写像が対応するために単射で全射であれば同型になります．なお, 圏の対象が集合であると限らないために勝手が異なり, 矢 f が同型であれば矢 f は単射, かつ全射ですが, 矢 f が単射, かつ全射であっても同型になるとは限りません．また, 対象 $A, B \in \mathscr{C}$ 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せることを意味します．とは言え, $A \cong B$ 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, 対象の積や冪が定義されます．

分裂単射と分裂全射

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

矢 f が同型であることは, 矢 f が分裂単射かつ全射であることと矢 f が分裂全射かつ単射であることと同値です．

2.7.6 圏について

メタグラフ $\mathscr{C}$ が「**メタ圏**」であるとはメタグラフ $\mathscr{C}$ 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:

メタ圏 (metacategory)

メタグラフ $\mathscr{C}$ が次の性質を充すときにメタ圏と呼ぶ:

- **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が, メタ圏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ります. しかし, 対象や矢のあつまり (類) が集合になるとは限りません. そこでメタグラフやメタ圏に対象と矢が構成する類が集合となるという要請を入れましょう. この要請を入れることで, 対象や矢の類が集合として扱え, その結果, それらを扱う後述の函手や自然変換が通常の函数になることを意味します. この対象と矢の類がそれぞれ ZFC 公理系の集合になるメタグラフ $\mathscr{C}$ を「**グラフ**」, そのようなメタ圏を「**圏**」と呼び, 以後, これらを中心に考察します:

グラフ (graph) の定義

グラフ $\mathscr{C}$ を次の性質を充すものとして定義します:

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

なお, グラフ $\mathscr{C}$ の対象全体の集合 O を $\mathrm{Ob}\ \mathscr{C}$, 同様に矢全体の集合 A の集合を $\mathrm{Arr}\ \mathscr{C}$ と表記します[*43]. このグラフでは新たな矢を生成する矢の合成と矢の合成に関する結合律, それと可換図式で表現される同一矢の存在についての言及はありません. 矢の合成に関するこれらの性質を加味したグラフが「**圏**」になります:

[*43] グラフ $\mathscr{C}$ を有向グラフと看做すとき $(\mathrm{Ob}\mathscr{C}, \mathrm{Arr}\mathscr{C}, \mathrm{dom}, \mathrm{cod})$ を「**箙 (quiver)**」と呼びます.

圏 (category)

グラフ $\mathscr{C}$ が次の性質を充すときに圏と呼ぶ:

- **矢の合成**を持つ
- **同一矢の公理**を充す
- 矢の合成について**結合律**を充す

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

また, 圏 $\mathscr{C}$ 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 $\mathscr{A}$ が次の 3 つの条件を充たすときに圏 $\mathscr{C}$ の部分圏と呼ぶ:

- $\mathscr{A}$ の対象 A は圏 $\mathscr{C}$ の対象で, その恒等矢 id_A も $\mathscr{A}$ の矢である.
- $\mathscr{A}$ の矢 f は圏 $\mathscr{C}$ の矢で, その始域 $\mathrm{dom}f$ と終域 $\mathrm{cod}f$ が $\mathscr{A}$ の対象である.
- $\mathscr{A}$ の二つの矢 $A \xrightarrow{f} B$, $B \xrightarrow{g} C$ の合成 $A \xrightarrow{g\circ f} C$ も $\mathscr{A}$ 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做せます. また, 圏 $\mathscr{C}$ の部分圏 $\mathscr{A}$ が圏 $\mathscr{C}$ の全ての矢を包含しているとき, つまり, 任意の $A, B \in \mathrm{Ob}\mathscr{A}$ に対して $\mathrm{Hom}_{\mathscr{C}}(A, B) = \mathrm{Hom}_{\mathscr{A}}(A, B)$ を充たすときに部分圏 $\mathscr{A}$ を圏 $\mathscr{C}$ の「**充足部分圏 (full subcategory)**」と呼びます. たとえば群を対象, 群準同型を矢とする圏 **Grp** とアーベル群を対象, 群準同型を矢とする圏 **Ab** では, 圏 **Ab** が圏 **Grp** の充足部分圏になりますが, 対象を集合, 矢を連続写像とする圏 **Set** と対象を集合, 矢を全単射写像とする圏 $\mathscr{C}$ であれば圏 $\mathscr{C}$ は圏 **Set** の部分圏であっても充足部分圏になりません.

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

2.7.7　圏の例

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:

- $\mathbf{0} =$　(なにもない圏)
- $\mathbf{1} = \bullet$. (一つの対象のみ. 同一矢の他の矢を持たない圏)
- $\mathbf{2} = \bullet \longrightarrow \bullet$　(対象が二つで同一矢の他に矢が一つの圏)
- $\mathbf{3} =$ $\begin{tikzcd} \bullet \arrow[r] \arrow[dr] & \bullet \arrow[d] \\ & \bullet \end{tikzcd}$　(対象が三つで同一矢の他の矢が三つの圏)
- $\downdownarrows = \bullet \rightrightarrows \bullet$　(対象が二つで同一矢と異なる矢が二つ)

ここでの $\mathbf{0}$ と $\mathbf{1}$ は圏の名称であり, 後述の始対象 0 と終対象 1 と異なります.

つぎに矢が対象に対応する同一矢のみの圏は「**離散圏**」と呼ばれます. ここで対象 $A \in \mathscr{C}$ の同一矢 $A \xrightarrow{\mathrm{id}_A} A$ に繋辞の「**... は... である**」を対応させると $A \xrightarrow{\mathrm{id}_A} A$ 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 $A \in \mathscr{C}$ 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**」[*44]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
- $\mathbf{Set}_*$: 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
- $\mathbf{Vect}_{\mathbf{K}}$: 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
- $\mathbf{Top}_*$: 対象が基点付きの位相空間, 矢は基点を保つ連続写像
- **Cat**: 対象が圏, 矢が後述の函手
- $\mathscr{C}^{\mathrm{op}}$: 圏 $\mathscr{C}$ 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

*44 形而上学 [3] 5 巻 29 章, 1024b34

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 $\mathscr{C}$ が「**局所的に小さい**」とは任意の 対象 $A, B \in \mathscr{C}$ に対し, その矢の集合 $\mathrm{Hom}(A, B)$ 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 $\mathscr{C}_0$ と矢全体の集合 $\mathscr{C}_1$ 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述べました. 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 $\{\ \}$	$\{u, v\} \overset{\text{Def.}}{=} \{(x, y) \mid x \in u \wedge y \in v\}$
順序対 $\langle\ \rangle$	$\langle u, v\rangle \overset{\text{Def.}}{=} \{\{u\}, \{u, v\}\}$
直積 $\times$	$u \times v \overset{\text{Def.}}{=} \{\langle x, y\rangle \mid x \in u \wedge y \in v\}$
冪集合 $\mathfrak{P}$	$\mathfrak{P}(u) \overset{\text{Def.}}{=} \{v \mid v \subset u\}$
和集合 $\cup$	$\cup u \overset{\text{Def.}}{=} \{x \mid x \in u\}$

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)	$x \in u \wedge u \in U \supset x \in U$
(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)	$a \in U \wedge b \subset U \wedge a \to b$が上への写像 $\supset b \in U$

グロタンディーク宇宙 U 自体は ZFC 公理系の「**正則公理**」によって U の元として含まれることがありません. さらに, この宇宙 U が集合になるとも限りません. このように宇宙 U とその元には区分があり, この区分を対象が集合であれば宇宙 U の元を「**小集合**」,

対象が圏であれば U の元を「**小圏**」と宇宙 U の元を, その元の型の頭に「**小 (small)**」を付けます. 逆に宇宙 U は頭に「**大 (large)**」を付けます. たとえば対象が集合であれば「**大集合**」, 圏であれば「**大圏**」といったあんばいです.

2.7.9　始対象と終対象

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

始対象と終対象

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

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

ここで始対象と終対象の例を挙げておきます. 圏 **Set** の始対象 0 は空集合 $\emptyset$ です. 実際, $A \in \mathrm{Ob}\mathbf{Set}$ に対して $\emptyset$ から A への矢として $\emptyset \xrightarrow{\emptyset} A$ のみが存在するためです[*45]. 圏 **Rng** の始対象 0 は整数 $\mathbf{Z}$, しかし, 圏 **Field** には始対象が存在しません. 次に **Set**, **Grp**, **Ab** と **Rng** の各圏の終対象は成分が一つの集合, すなわち,「**シングルトン (singleton)**」です. しかし, 圏 **Field** には終対象も存在しません. また, 圏 **Set** では矢 $1 \longrightarrow A$ が集合 A の成分を定めます[*46].

2.7.10　函手

圏 $\mathscr{C}$ と圏 $\mathscr{D}$ に対して, 圏 $\mathscr{C}$ の対象を圏 $\mathscr{D}$ の対象に, 同時に圏 $\mathscr{C}$ の矢を圏 $\mathscr{D}$ の矢に対応付けられます. 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 $\mathscr{C}$ の同一矢 id_A を圏 $\mathscr{D}$ の

[*45] 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.

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

同一矢 id_B に写し, 矢の合成もそれらの像の合成になると都合が良くなります. すなわち, 圏 $\mathscr{C}$ 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 $F : \mathscr{C} \longrightarrow \mathscr{D}$ 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 $\mathscr{C}$ から圏 $\mathscr{D}$ の写像 F で以下の性質を充すものを「**共変函手**」と呼ぶ:

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

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

つぎに反変函手の定義を示しておきます:

反変函手 (contravariant functor)

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

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

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 $^{\mathrm{op}} : \mathscr{C} \to \mathscr{C}^{\mathrm{op}}$ が挙げられます. 実際, 函手 $^{\mathrm{op}}$ によって対象 $A \in \mathrm{Ob}\mathscr{C}$ と同一矢はそのままで, 矢 $A \xrightarrow{f} B$ は $B \xrightarrow{f^{\mathrm{op}}} A$ に, $B \xrightarrow{g} C$ との合成矢 $A \xrightarrow{g\circ f} C$ は $C \xrightarrow{f^{\mathrm{op}}\circ g^{\mathrm{op}}} A$ に写されるため

です．この双対 $^{\mathrm{op}}$ に対しては「**双対原理 (duality principle)**」と呼ばれる原理があります．これは S を圏 $\mathscr{C}$ 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 $\mathscr{C}^{\mathrm{op}}$ で成立し, 逆に, 双対圏 $\mathscr{C}^{\mathrm{op}}$ で成立する命題 S は圏 $\mathscr{C}$ で成立することです．双対函手 $^{\mathrm{op}}$ による圏 $\mathscr{C}$ と圏 $\mathscr{C}^{\mathrm{op}}$ 上の命題との代表的な対応関係を以下にまとめておきます:

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

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

圏 $\mathscr{C}$ から $\mathscr{D}$ への函手 $F : \mathscr{C} \to \mathscr{D}$ 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, 函手が単射, 全射, 同型になる場合が考えられます．まず, 函手 F に対して $GF = \mathrm{id}_{\mathscr{C}}$ と $FG = \mathrm{id}_{\mathscr{D}}$ を充す逆向きの函手 $G : \mathscr{D} \to \mathscr{C}$ が存在するときに函手 F を「**同型 (isomorphism)**」と呼び, 圏 $\mathscr{C}, \mathscr{D}$ に同型な函手が存在するときに $\mathscr{C} \cong \mathscr{D}$ 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 $\mathrm{Hom}_{\mathscr{C}}(A, B) \xrightarrow{F} \mathrm{Hom}_{\mathscr{D}}(FA, FB)$ が, 函手 F が反変なら写像 $\mathrm{Hom}_{\mathscr{C}}(A, B) \xrightarrow{F} \mathrm{Hom}_{\mathscr{D}}(FB, FA)$ 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「**充満 (full)**」と呼びます．また, 圏 $\mathscr{C}$ から圏 $\mathscr{D}$ の函手 F が「**稠密 (dense)**」であるとは, 任意の対象 $B \in \mathrm{Ob}\mathscr{C}$ に対して $FA \cong B$ を充す対象 $A \in \mathrm{Ob}\mathscr{D}$ が存在するときです．そして, 圏 $\mathscr{C}$ から $\mathscr{D}$ への函手 F が稠密, 忠実, 充満であるときに圏 $\mathscr{C}$ と $\mathscr{D}$ を「**同値 (equivalent**」であると呼び, $\mathscr{C} \simeq \mathscr{D}$ 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 $\mathscr{C}$ から小集合の圏 **Set** への函手を圏 $\mathscr{C}$ 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 $\mathscr{C}$ の対象 A から B の矢の類 $\mathrm{Hom}_{\mathscr{C}}(A, B)$ とその双対 $\mathrm{Hom}_{\mathscr{C}}(B, A)$ は圏の性質から集合になります．このときに $\mathrm{H}_A : B \to \mathrm{Hom}_{\mathscr{C}}(A, B)$ と $\mathrm{H}^A : B \to \mathrm{Hom}_{\mathscr{C}}(B, A)$ は圏 $\mathscr{C}$ 上の集合値函手で, H_A が圏 $\mathscr{C}$ から圏 **Set** への共変函手, H^A が圏 $\mathscr{C}^{\mathrm{op}}$ から圏 **Set** への反変函手です．なお, 集合値函手 F が $F = \mathrm{H}_A$ あるいは $F = \mathrm{H}^A$ となる対象 $A \in \mathrm{Ob}\mathscr{C}$ が存在するときに函手 F を「**表現可能 (representable)**」と呼びます．

2.7.11　自然変換

圏 $\mathscr{C}$ から圏 $\mathscr{D}$ への二つの函手の間に「**自然変換**」と呼ばれる対応関係が入ることがあります．まず, 二つの函手 $F, G : \mathscr{C} \to \mathscr{D}$ が与えられたとき, 圏 $\mathscr{C}$ の矢 $A \xrightarrow{f} B$ の始域と終域になる対象 $A, B \in \mathrm{Ob}\mathscr{C}$ は函手 F によってそれぞれ $F\ A, F\ B \in \mathrm{Ob}\mathscr{D}$ に写され, また, 矢 f 自体も圏 $\mathscr{D}$ の矢 $F\ A \xrightarrow{Ff} F\ B$ に写されます．これは函手 G も同様で, 対象は $G\ A, G\ B \in \mathrm{Ob}\mathscr{D}$ に矢は $G\ A \xrightarrow{G\ f} G\ B$ 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 $\mathscr{C}$ 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 $G\ f \circ \alpha_B$ と $\alpha_A \circ Ff$ が等しいことは自然な要請になります．この写像 α を自然変換と呼びます:

自然変換 (natural transformation)

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

$$\begin{tikzcd}
A \arrow[d, "f"] & FA \arrow[r, "\alpha_A"] \arrow[d, "Ff"'] & GA \arrow[d, "Gf"] \\
B & FB \arrow[r, "\alpha_B"'] & GB
\end{tikzcd}$$

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 $\mathscr{C}$ から圏 $\mathscr{D}$ への函手 F から G への自然変換のあつまりを $\mathrm{Nat}(F, G)$ と表記します．ここで自然変換と函手との間には「**米田の補題**」と呼ばれる非常に有名な補題があります:

米田の補題

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

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

$$\begin{tikzcd}
A \arrow[d, "f"] & \mathrm{H}_A(A) \arrow[r, "\eta_A"] \arrow[d, "H_A(f)"'] & F(A) \arrow[d, "F(f)"] \\
B & \mathrm{H}_A(B) \arrow[r, "\eta_B"'] & F(B)
\end{tikzcd}$$

が可換になります．ところで $f \in \mathrm{H}_A(B)$ で, f の自然変換 η_B による像 $\eta_B(f)$ は

$f = \mathrm{H}_A(f)(\mathrm{id}_A)$ であることと, この可換図式から $\eta(\mathrm{H}_A(f)(\mathrm{id}_A)) = F(f)(\eta_A(\mathrm{id}_A))$. このことから写像 θ による η の像は $\eta_A(\mathrm{id}_A)$ 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 $a^*_B : \mathrm{H}_A(B) \to F(B)$ を $a^*_B(f) \overset{\mathrm{def}}{=} F(f)a$ 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 $g \in \mathrm{H}_B(C)$ に対して図式

$$\begin{tikzcd} B \arrow[d, "g"] \\ C \end{tikzcd} \qquad \begin{tikzcd} \mathrm{H}_A(B) \arrow[r, "a*_A"] \arrow[d, "\mathrm{H}_A(g)"'] & F(B) \arrow[d, "F(g)"] \\ \mathrm{H}_A(C) \arrow[r, "a*_C"'] & F(C) \end{tikzcd}$$

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 $F(g)(a^*_B(f)) = a^*_C(\mathrm{H}_A(g)(f))$ 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

$$a^*_C(\mathrm{H}_A(g)(f)) = a^*_C(g \circ f) = F(g \circ f)(a)$$

になります. したがって a^* は自然変換であり, また, $\theta(a^*) = a$ であるために写像 θ が全射であることが判り, 以上から米田の補題が証明できます. なお, $\mathrm{Nat}(\mathrm{H}^A, F) \cong F(A)$ についても $\mathrm{H}^A(B) = \mathrm{Hom}_{\mathscr{C}}(B, A) = (\mathrm{Hom}_{\mathscr{C}}(A, B))^{\mathrm{op}}$ 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できます. すなわち, 対象を圏 $\mathscr{C}$ から圏 $\mathscr{D}$ への函手, 矢をそれらの間の自然変換とすることで圏が構成できます. この方法で定まる圏を $\mathscr{D}^{\mathscr{C}}$ と表記します. この構成による圏で重要なものに圏 $\mathbf{Set}^{\mathscr{C}^{op}}$ を挙げておきましょう. この $\mathbf{Set}^{\mathscr{C}^{\mathrm{op}}}$ 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 $Y : \mathscr{C} \to \mathbf{Set}^{\mathscr{C}^{\mathrm{op}}}$ を $A \in \mathrm{Ob}\mathscr{C}$ に対しては $Y(A) = \mathrm{H}^A$, $A \xrightarrow{f} B$ と $C \xrightarrow{g} A$ に対しては $Y(f)_{\mathscr{C}}(g) = f \circ g$ で定めると, 函手 Y は $\mathscr{C}$ から $\mathbf{Set}^{\mathscr{C}^{\mathrm{op}}}$ への充満な埋め込みになります. この函手 Y を「**米田の埋め込み (Yoneda embedding)**」と呼びます.

ここで自然変換を使って「**図式 (diagram)**」を定義しなおします:

J 型の図式 (diagram) の定義

圏 $\mathscr{C}$, J で定まる圏 $\mathscr{C}^J$ の対象 $D : J \to \mathscr{C}$ を圏 $\mathscr{C}$ 上の J 型 (J-type) の「**図式 (diagram)**」, さらに $i \in \mathrm{Ob}J$ に対して $Di \in \mathrm{Ob}\mathscr{C}$ 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:

$$
\begin{tikzcd}
J \arrow[d, "D"] \\
\mathscr{C}
\end{tikzcd}
\qquad
\begin{tikzcd}
i \arrow[r, "u"] \arrow[d, "D"'] & j \arrow[d, "D"] \\
D_i \arrow[r, "Du"'] & D_j
\end{tikzcd}
$$

この定義から図式は圏 $\mathscr{C}$ 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られます．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 $\Delta_J : \mathscr{C} \to \mathscr{C}^J$ は対角函手とも呼ばれ, $\Delta_J(C)$ は添字圏の対象全てを圏 $\mathscr{C}$ の対象 C に, 矢は全て C の同一矢 id_C に写す函手です．この定図式は錐と余錐の定義で用いられます．

2.7.12 コンマ圏

既存の圏と函手を使って新しい圏を構成することが可能です．まず, b を圏 $\mathscr{C}$ 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 $\mathscr{C}$ の矢 $c \xrightarrow{h} d$ で $g = h \circ f$ を充たす

$$
\begin{tikzcd}
& b \arrow[dl, "f"'] \arrow[dr, "g"] & \\
c \arrow[rr, "h"'] & & d
\end{tikzcd}
$$

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 $(b \downarrow \mathscr{C})$ 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られます．このときに対象は $c \xrightarrow{f} b$ で与えられ, この対象を「**対象 b の上にある対象**」と呼んで対 $\langle c, f \rangle$ と表記します．また, 矢は先程と同様に次の可換図式が成立する矢 h で与えられます:

$$
\begin{tikzcd}
c \arrow[rr, "h"] \arrow[dr, "f"'] & & c' \arrow[dl, "f'"] \\
& b &
\end{tikzcd}
$$

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 $(\mathscr{C} \downarrow b)$ 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 $\mathscr{C}/b$ と表記されます．さらに二つの圏 $\mathscr{C}, \mathscr{D}$ に対して函手 $\mathrm{S} : \mathscr{D} \longrightarrow \mathscr{C}$ が与えられたときに対象 $b \in \mathscr{C}$ を固定します．それ

から $d \in \mathscr{D}$ に対して矢 $f : b \to \mathrm{S}d$ を $\langle f, d\rangle$ と表記します．つぎに $\langle f, d\rangle$ と $\langle f', d'\rangle$ を考えます．ここで矢 $d \xrightarrow{h} d'$ で以下の可換図式が成立するものを $\langle f, d\rangle$ と $\langle f, d'\rangle$ の矢 h とします:

$$\begin{tikzcd} & b \arrow[dl, "f"'] \arrow[dr, "f'"] & \\ \mathrm{S}d \arrow[rr, "\mathrm{S}h"'] & & \mathrm{S}d' \end{tikzcd}$$

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 $(b \downarrow \mathrm{S})$ と表記し, b 上の S の圏と呼びます．また, この圏 $(s \downarrow \mathrm{S})$ の双対を $(\mathrm{S} \downarrow s)$ 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:

$$\mathscr{E} \xrightarrow{T} \mathscr{C} \xleftarrow{S} \mathscr{D}$$

を考え, $d \in \mathscr{D}, e \in \mathscr{E}$ に対し矢 $\mathrm{T}e \xrightarrow{f} \mathrm{S}d$ 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:

$$\begin{tikzcd} Te \arrow[r, "Th"] \arrow[d, "f"] & Te' \arrow[d, "f'"] \\ Sd \arrow[r, "Sh"'] & Sd' \end{tikzcd}$$

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 $\langle e', d', f'$ への矢 $\langle h, k\rangle$ を定め, これらの対象と矢で構成される圏を「**コンマ圏**」と呼び, $(T \downarrow S)$, あるいは (T, S) と表記します．

このコンマ圏が上述の圏を一般化したものであることを確認しておきましょう．まず圏 $\mathscr{C}$ 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 $\mathscr{C}$ 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られます．そして対象 $\langle e, d, f\rangle$ も $\langle 1, d, f\rangle$ となりますが実質的には $\langle d, f\rangle$ であり, 矢 $\langle h, k\rangle$ も $\langle \mathrm{id}, k\rangle$ で, この矢が充たすべきの可換図式も

$$\begin{tikzcd} T1 = b \arrow[r, "\mathrm{id}"] \arrow[d, "f"] & T1 = b \arrow[d, "f'"] \\ Sd \arrow[r, "Sh"'] & Sd' \end{tikzcd}$$

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 $\mathscr{C}$ の同一矢とすると圏 $(b \downarrow \mathscr{C})$ が得られます.

2.7.13 圏の演算

圏 $\mathscr{C}$ の対象について積や冪が定義できます. 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 $\mathscr{C}$ の対象 X を $A \times B$ と表記し, 対象 A, B の積と呼ぶ:

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

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

圏の積を可換図式として表現できます:

$$
\begin{tikzcd}
 & C \arrow[dl, "f"'] \arrow[d, "\langle f,g \rangle"] \arrow[dr, "g"] & \\
A & A \times B \arrow[l, "\pi_1"] \arrow[r, "\pi_2"'] & B
\end{tikzcd}
$$

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, $A \to B$ 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

[*47] この普遍性は後述の極限に関連します.

$$
\begin{tikzcd}
 & & P \arrow[dll, "\pi_1'"'] \arrow[d, "\langle \pi_1', \pi_2' \rangle"] \arrow[drr, "\pi_2'"] & & \\
A & & A \times B \arrow[ll, "\pi_1"] \arrow[rr, "\pi_2"] \arrow[d, "\langle \pi_1, \pi_2 \rangle"] & & B \\
 & & P \arrow[ull, "\pi_1'"] \arrow[urr, "\pi_2'"'] & &
\end{tikzcd}
\qquad
\begin{tikzcd}
 & & A \times B \arrow[dll, "\pi_1"'] \arrow[d, "\langle \pi_1, \pi_2 \rangle"] \arrow[drr, "\pi_2"] & & \\
A & & P \arrow[ll, "\pi_1'"] \arrow[rr, "\pi_2'"] \arrow[d, "\langle \pi_1', \pi_2' \rangle"] & & B \\
 & & A \times B \arrow[ull, "\pi_1"] \arrow[urr, "\pi_2"'] & &
\end{tikzcd}
$$

から $\langle \pi_1, \pi_2 \rangle \circ \langle \pi_1', \pi_2' \rangle = \mathrm{id}_P$, $\langle \pi_1', \pi_2' \rangle \circ \langle \pi_1, \pi_2 \rangle = \mathrm{id}_{A \times B}$ が得られるために $A \times B \cong P$ が判り, 積自体は iso な対象で一意に定まることを意味します. なお, 圏を小集合の圏 **Set** とするときに対象の積は $A \times B = \{\langle x, y \rangle \mid x \in A \wedge y \in B\}$ と小集合のデカルト積になりますが, 圏 $\mathscr{C}$ 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 $\mathscr{C}$ の対象 A_1, A_2 に対して部分圏 $\mathscr{C} \mid A_1, A_2$ 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 $\mathscr{C}|A_1, A_2$ の終対象として考えられます.

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 $\mathscr{C}$ の対象の集合 $\mathrm{Ob}\mathscr{C}$ には iso による関係が入るもの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 $A_1 \times A_2 \times \ldots \times A_n$ 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 $A \times \underset{n}{\ldots} \times A$ を A^n と表記します. さらに圏 $\mathscr{C}$ 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

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

これらのことは積の可換性と以下の可換図式から判ります:

$$
\begin{tikzcd}
& A \arrow[dl, "!_A"'] \arrow[d, "\langle !_A, \mathrm{id}_A \rangle"] \arrow[dr, "\mathrm{id}_A"] & \\
1 & 1 \times A \arrow[l, "!_{1\times A}"] \arrow[r, "\pi_2"'] \arrow[d, "\pi_2"] & A \arrow[dl, "\mathrm{id}_A"] \\
& A &
\end{tikzcd}
\qquad
\begin{tikzcd}
& 1 \times A \arrow[dl, "!_{1\times A}"'] \arrow[d, "\pi_2"] \arrow[dr, "\pi_2"] & \\
1 & A \arrow[l, "!_A"] \arrow[r, "\mathrm{id}_A"'] \arrow[d, "\langle !_A, \mathrm{id}_A \rangle"] & A \\
& 1 \times A \arrow[ul, "!_{1\times A}"] \arrow[ur, "\pi_2"'] &
\end{tikzcd}
$$

まず, 左の可換図式から $\pi_2 \circ \langle !_A, \mathrm{id}_A \rangle = \mathrm{id}_A$ が判ります. また, 右の図式で 1 が終対象のために矢 $A \to 1$ 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 $\langle !_A, \mathrm{id}_A \rangle \circ \pi_2 = \mathrm{id}_{1\times A}$ 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 $\langle \mathrm{id}_A, \mathrm{id}_A \rangle$ を特に対角矢と呼び記号 Δ_A で表記される.

$$
\begin{tikzcd}
& A \arrow[dl, "\mathrm{id}_A"'] \arrow[d, "\Delta_A"] \arrow[dr, "\mathrm{id}_A"] & \\
A & A \times A \arrow[l, "\pi_1"] \arrow[r, "\pi_2"'] & A
\end{tikzcd}
$$

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 $\mathscr{C}$ 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:

- $\mathscr{C}$ の二つの矢 $A \xrightarrow{i_1} X$ と $B \xrightarrow{i_2} X$ が存在
- $\mathscr{C}$ 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:

$$
\begin{tikzcd}
 & C & \\
A \arrow[ur, "f"] \arrow[r, "i_1"'] & A+B \arrow[u, "\langle f,g \rangle"'] & B \arrow[ul, "g"'] \arrow[l, "i_2"]
\end{tikzcd}
$$

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 $A_1+A_2+\ldots+A_n$ と括弧を外した形や $\amalg_1^n A_i$ と表記します. さらに圏 $\mathscr{C}$ 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 $\mathscr{C}$ の対象の集合 $\mathrm{Ob}\mathscr{C}$ 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**」も定義できます:

矢の積

$\mathscr{C}$ の二つの矢 $A\ \xrightarrow{f}\ B$ と $C\ \xrightarrow{g}\ D$ に対し $\langle f\circ\pi_1, g\circ\pi_2\rangle : A\ \times\ C\ \to\ B\ \times\ D$ 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せます:

$$
\begin{tikzcd}
A \arrow[r, "f"] & B \\
A\times C \arrow[u, "\pi_1"] \arrow[r, "f\times g"] \arrow[d, "\pi_2"'] & B\times D \arrow[u, "\pi_1"'] \arrow[d, "\pi_2"] \\
C \arrow[r, "d"] & C
\end{tikzcd}
$$

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

対象の冪

積を有する圏 $\mathscr{C}$ をの対象 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 $\mathscr{C}$ の対象 B^A と矢 $B^A \times A \xrightarrow{\mathrm{ev}} B$ 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, $\mathscr{C}$ の対象 B^A のことを冪と呼ぶ:

$$\begin{tikzcd}
B^A \times A \arrow[dr, "\mathrm{ev}"] & \\
C \times A \arrow[u, "\hat{g} \times 1_A"] \arrow[r, "g"'] & B
\end{tikzcd}$$

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. さらに対象の冪では $\mathrm{Hom}(C \times A, B) \cong \mathrm{Hom}(C, B^A)$ が成立します. 実際, $g \in \mathrm{Hom}(C \times A, B)$ に対して冪 B^A の定義から一意に $\hat{g} \in \mathrm{Hom}(C, B^A)$ が存在するために $\mathrm{Hom}(C \times A, B)$ から $\mathrm{Hom}(C, B^A)$ への写像が存在します. そして, $h \in \mathrm{Hom}(C, B^A)$ に対して $g = \mathrm{ev}(h \times 1_A)$ と定めることで $g \in \mathrm{Hom}(C \times A, B)$ が得られますが, 冪の定義から g の転置 $\hat{g}$ が一意に定まるために $\hat{g} = h$, したがって, $\mathrm{Hom}(C \times A, B)$ から $\mathrm{Hom}(C, B^A)$ 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 $A \overset{f}{\underset{g}{\rightrightarrows}} B$ 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:

$$\begin{tikzcd}
D \arrow[d, "h"'] \arrow[dr, "k"] & & \\
C \arrow[r, "e"'] & A \arrow[r, shift left, "f"] \arrow[r, shift right, "g"'] & B
\end{tikzcd}$$

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, それぞれが $e \circ h = e \circ k$ を充すときに二つの図式:

$$
\begin{tikzcd}
D \arrow[d, "h"'] \arrow[dr, "e\circ h"] & & \\
C \arrow[r, "e"'] & A \arrow[r, shift left, "f"] \arrow[r, shift right, "g"'] & B
\end{tikzcd}
\qquad
\begin{tikzcd}
D \arrow[d, "k"'] \arrow[dr, "e\circ k"] & & \\
C \arrow[r, "e"'] & A \arrow[r, shift left, "f"] \arrow[r, shift right, "g"'] & B
\end{tikzcd}
$$

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 $A \underset{g}{\overset{f}{\rightrightarrows}} B$ 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:

$$
\begin{tikzcd}
 & & D \arrow[d, "h"] \\
A \arrow[r, shift left, "f"] \arrow[r, shift right, "g"'] & B \arrow[ur, "k"] \arrow[r, "e"'] & C
\end{tikzcd}
$$

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

2.7.15　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 $E \in \mathscr{C}$ 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.

$$\begin{tikzcd} P \arrow[r, "f'"] \arrow[d, "g'"'] & B \arrow[d, "g"] \\ A \arrow[r, "f"'] & C \end{tikzcd} \qquad \begin{tikzcd} E \arrow[dr, "l"] \arrow[drr, bend left, "k"] \arrow[ddr, bend right, "h"'] & & \\ & P \arrow[r, "f'"] \arrow[d, "g'"] & B \arrow[d, "g"] \\ & A \arrow[r, "f"'] & C \end{tikzcd}$$

ここで圏 $\mathscr{C}$ 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 $\mathscr{C}$ の矢が通常の写像の圏 **Set** のときに引き戻しの対象 P は $A \times_C B = \{\langle x, y\rangle \mid (x \in A) \wedge (y \in B) \wedge (f\ x = g\ y)\}$ と外延として記述することが可能です. このように引き戻しは特殊な積としても考えられます.

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:

$$\begin{tikzcd} P \arrow[r, tail, "f'"] \arrow[d, "g'"'] & B \arrow[d, "g"] \\ A \arrow[r, tail, "f"'] & C \end{tikzcd}$$

- 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外側の四角の図式も引き戻しになります:

$$\begin{tikzcd} P \arrow[r, "g'"] \arrow[d, "f'"'] & Q \arrow[r, "h'"] \arrow[d, "f"'] & D \arrow[d, "f''"'] \\ A \arrow[r, "g"'] & B \arrow[r, "h"'] & C \end{tikzcd}$$

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

$$\begin{tikzcd}
A \times_C B \arrow[r, "\pi_A"] \arrow[d, "\pi_B"'] & B \arrow[d, "g"] \\
A \arrow[r, "f"'] & C
\end{tikzcd}$$

は引き戻しになりますが, ここで

$$\begin{tikzcd}
D \arrow[dr, "l"] \arrow[drr, bend left, "k"] \arrow[ddr, bend right, "h"'] & & \\
& A \times_C B \arrow[r, "\pi_A"] \arrow[d, "\pi_B"] & B \arrow[d, "g"] \\
& A \arrow[r, "f"'] & C
\end{tikzcd}$$

をよくよく考えると次の集合の積と等化が現われます:

$$\begin{tikzcd}
& D \arrow[dl, "h"'] \arrow[d, "l"] \arrow[dr, "k"] & \\
A & A \times_C B \arrow[l, "\pi_B"] \arrow[r, "\pi_A"'] & B
\end{tikzcd}
\qquad
\begin{tikzcd}
D \arrow[d, "l"'] \arrow[dr, "h \times k"] & & \\
A \times_C B \arrow[r, "e"'] & A \times B \arrow[r, shift left, "f \circ \pi_A"] \arrow[r, shift right, "g \circ \pi_B"'] & C
\end{tikzcd}$$

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 $\mathscr{C}$ 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:

$$
\begin{tikzcd}
D \arrow[r, "e"] & A \times B \arrow[r, "\pi_A"] \arrow[d, "\pi_B"] & B \arrow[d, "g"] \\
& A \arrow[r, "f"'] & C
\end{tikzcd}
$$

このときに

$$
\begin{tikzcd}
D \arrow[r, "\pi_A \circ e"'] \arrow[d, "\pi_B \circ e"] & B \arrow[d, "g"] \\
A \arrow[r, "f"'] & C
\end{tikzcd}
$$

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:

$$
\begin{tikzcd}
E \arrow[rrr, bend left, "k"] \arrow[rr, "\langle h\, k \rangle"] \arrow[ddrr, bend right, "h"'] & & & \\
& D \arrow[r, "e"] & A \times B \arrow[r, "\pi_A"] \arrow[d, "\pi_B"] & B \arrow[d, "g"] \\
& & A \arrow[r, "f"'] & C
\end{tikzcd}
$$

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:

$$
\begin{tikzcd}
E \arrow[rrr, bend left, "k"] \arrow[rr, "\langle h\, k \rangle"] \arrow[dr, "l"] \arrow[ddrr, bend right, "h"'] & & & \\
& D \arrow[r, "e"] & A \times B \arrow[r, "\pi_A"] \arrow[d, "\pi_B"] & B \arrow[d, "g"] \\
& & A \arrow[r, "f"'] & C
\end{tikzcd}
$$

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 $E \in \mathscr{C}$ 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:

$$
\begin{tikzcd}
C \arrow[r, "f"] \arrow[d, "g"] & B \arrow[d, "g'"] \\
A \arrow[r, "f'"'] & P
\end{tikzcd}
\qquad
\begin{tikzcd}
C \arrow[r, "f"] \arrow[d, "g"] & B \arrow[d, "g'"] \arrow[ddr, bend left, "k"] & \\
A \arrow[r, "f'"'] \arrow[drr, bend right, "h"] & P \arrow[dr, "l"] & \\
 & & E
\end{tikzcd}
$$

この押し出しの定義は引き戻しの定義の矢の向きが逆になったもの, すなわち, 引き戻しの双対であることに注目して下さい. そのために対象 C が始対象 0 であれば押し出しの対象 P は対象 A, B の積 $A \times B$ の双対の $A \amalg B$ になり, 圏 $\mathscr{C}$ 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 $\mathscr{C}$ に対して図式 $D : J \to \mathscr{C}$ に対して「**錐 (cone)**」を次で定めます:

錐 (cone) の定義

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

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

$$\begin{tikzcd} & C \arrow[dl, "\xi_i"'] \arrow[dr, "\xi_j"] & \\ D_i \arrow[rr, "\delta_{ji}"'] & & D_j \end{tikzcd}$$

が可換であることを意味します．また, 錐の定義として定図式 $\Delta_J(C)$ を用いれば $\Delta_J(C)$ から D への自然変換 ξ として与えられます:

$$\begin{tikzcd} i \arrow[d, "u"] & C \arrow[r, "\xi_i"] \arrow[d, equal, "\Delta_J(C)(u)=\mathrm{id}_C"] & D_i \arrow[d, "D(u)=\delta_{ji}"] \\ j & C \arrow[r, "\xi_j"'] & D_j \end{tikzcd}$$

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 $\mathscr{C}$ 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

$$\begin{tikzcd} D_i \arrow[rr, "\delta_{ji}"] & & D_j \\ & C \arrow[ul, "\zeta_i"] \arrow[ur, "\zeta_j"'] & \end{tikzcd}$$

が可換であることを意味します．また, 余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:

$$\begin{tikzcd} i \arrow[d, "u"] & D_i \arrow[r, "\zeta_i"] \arrow[d, "D(u)=\delta_{ji}"'] & C \arrow[d, equal, "\Delta_J(C)(u)=\mathrm{id}_C"] \\ j & D_j \arrow[r, "\zeta_j"'] & C \end{tikzcd}$$

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

$$
\begin{tikzcd}
i \arrow[dd, "u"] & & D_i \arrow[dd, "Du"] & D_i \arrow[dd, "Du"'] \arrow[dr, "\rho_i"] & \\
& C \arrow[ur, "\pi_i"] \arrow[dr, "\pi_j"'] & & & C \\
j & & D_j & D_j \arrow[ur, "\rho_j"'] &
\end{tikzcd}
$$

と対象 C を纏めた図式で錐と余錐を表記します. これら錐と余錐を使って極限と余極限を定義しましょう. 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 $\mathscr{C}$ の対象 L が図式 $D \in \mathscr{C}^J$ 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 $\mathscr{C}$ の矢 $A \xrightarrow{h} L$ が一意に存在するときで, 図式 D の極限を $\varprojlim D$, あるいは $\lim D$ と表記します.

$$
\begin{tikzcd}
i \arrow[dd, "u"] & D_i \arrow[dd, "D(u)=\delta_{ji}"'] & & \\
& & L \arrow[ul, "\pi_i"'] \arrow[dl, "\pi_j"] & A \arrow[l, dashed, "h"'] \arrow[ull, "\xi_i"'] \arrow[dll, "\xi_j"] \\
j & D_j & &
\end{tikzcd}
$$

極限の双対が「**余極限 (colimit)**」で, その定義で余錐が用いられます:

余極限 (colimit) の定義

圏 $\mathscr{C}$ の対象 L が 図式 $D \in \mathscr{C}^J$ 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 $\mathscr{C}$ 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 $\varinjlim D$, あるいは $\mathrm{colim} D$ と表記します.

$$
\begin{tikzcd}
i \arrow[dd, "u"'] & D_i \arrow[dd, "D(u)=\delta_{ji}"'] \arrow[dr, "\rho_i"] \arrow[drr, "\zeta_i"] & & \\
 & & L \arrow[r, dashed, "k"] & A \\
j & D_j \arrow[ur, "\rho_j"] \arrow[urr, "\zeta_j"'] & &
\end{tikzcd}
$$

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 $\{\bullet \longrightarrow \bullet \longleftarrow \bullet\}$ 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 $D \in \mathscr{C}^J$ に $\varprojlim D$ が存在し, 図式 D の錐 ξ に対して

$$
\begin{tikzcd}
 & A \arrow[dl, "\xi_1"'] \arrow[d, "h"] \arrow[dr, "\xi_2"] & \\
D_1 & \varprojlim D \arrow[l, "\pi_1"] \arrow[r, "\pi_2"'] & D_2
\end{tikzcd}
$$

を充たす矢 h が一意に定まりますが, この可換図式は $\varprojlim D \cong D_1 \times D_2$ 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 $\downdownarrows$ を添字圏とし, 図式 $D \in \mathscr{C}^{\downdownarrows}$ に対してその極限 $\varprojlim D$ が存在するときに $\varprojlim D$ を頂点に持つ錐 π が D から $\varprojlim D$ への自然変換であることから

$$
\begin{tikzcd}
D_1 \arrow[rr, shift left, "F=Df"] \arrow[rr, shift right, "G=Dg"'] & & D_2 \\
& \varprojlim D \arrow[ul, "\pi_1"] \arrow[ur, "\pi_2"'] &
\end{tikzcd}
$$

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

$$
\begin{tikzcd}
D_1 \arrow[rr, shift left, "F=Df"] \arrow[rr, shift right, "G=Dg"'] & & D_2 \\
& \varprojlim D \arrow[ul, "\pi_1"] \arrow[ur, "\pi_2"'] & \\
& A \arrow[uul, "\xi_1"] \arrow[uur, "\xi_2"'] \arrow[u, "h"] &
\end{tikzcd}
$$

この可換図式を次の可換図式で置き換えられます:

$$
\begin{tikzcd}
\varprojlim D \arrow[r, "\pi_1"] & D_1 \arrow[r, shift left, "F=Df"] \arrow[r, shift right, "G=Dg"'] & D_2 \\
A \arrow[u, "h"] \arrow[ur, "\xi_1"] & &
\end{tikzcd}
$$

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

　この余極限については興味深い事例が挙げられます.「アベラールとエローイーズ」で有名な 12 世紀の哲学者アベラールは普遍についてこのように興味深いことを述べています ([39] pp. 43-44).

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

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

$$\begin{tikzcd}
D_1 \arrow[r, "\supset"] \arrow[d, "\cup"] \arrow[drr] & D_2 \arrow[r, "\supset"] \arrow[dl, "\in"] \arrow[dr, "\supset"] & D_3 \arrow[d, "\cup"] \arrow[dll] \\
\text{アリストテレス} & & H \arrow[ll, dashed, "\subset", "\supset"']
\end{tikzcd}$$

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

圏 $\mathscr{C}$ の任意の (有限) 図式が極限を持つときに圏 $\mathscr{C}$ のことを「**(有限) 完備 ((finite-)complete)**」と呼びます. 同様に圏 $\mathscr{C}$ の任意の (有限) 図式が余極限を持つときに圏 $\mathscr{C}$ 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 $\mathscr{C}$ が (余) 完備ならば圏 $\mathscr{C}$ に (余) 積と (余) 等価が存在し, そのときに限る.

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, 後半の積と等化から極限が構成できることを示さなければなりません. そこで圏 $\mathscr{C}$ を積と等化が存在する圏と仮定し, 与えられた図式 $D \in \mathscr{C}^J$ 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 $\bar{D} = \prod_{i \in \mathrm{Ob}J} D_i$ 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 $\underline{D} = \prod_{f \in \mathrm{Arr}J} D_{\mathrm{cod}f}$ 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 $i \xrightarrow{u} j \in \mathrm{Arr}J$ に対して次の図式を可換にする圏 $\mathscr{C}$ の矢 f, g が一意に存在します:

$$
\begin{tikzcd}
\bar{D} \arrow[r, "f"] \arrow[dr, "\pi_j"'] & \underline{D} \arrow[d, "\rho_j"] \\
& D_j
\end{tikzcd}
\qquad
\begin{tikzcd}
\bar{D} \arrow[r, "g"] \arrow[dr, "Du\circ\pi_i"'] & \underline{D} \arrow[d, "\rho_j"] \\
& D_j
\end{tikzcd}
$$

ここで圏 $\mathscr{C}$ で等化が存在することから図式 $\bar{D} \underset{g}{\overset{f}{\rightrightarrows}} \underline{D}$ の等化を $A \xrightarrow{h} \bar{D}$ とします. 次に $\pi \circ h$ が A を頂点とする図式 D の錐であることを示しましょう. そのためには以下の図式が可換図式であることを示さなければなりません;

$$
\begin{tikzcd}
& A \arrow[dl, "\pi_i\circ h"'] \arrow[dr, "\pi_j\circ h"] & \\
D_i \arrow[rr, "Du"'] & & D_j
\end{tikzcd}
$$

ここで矢 f に関わる可換図式から $\pi_j \circ h = \rho_j \circ f \circ h$, $A \to \bar{D}$ が等化であることから $f \circ h = g \circ h$ より $\pi_j \circ h = \rho_j \circ f \circ h = \rho_j \circ g \circ h$. それと, 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 $B \xrightarrow{\xi} \bar{D}$ に対して次の図式が可換になる矢 $B \xrightarrow{k} A$ が一意に存在します:

$$\begin{tikzcd}
A \arrow[r, "h"] & \bar{D} \arrow[r, shift left, "f"] \arrow[r, shift right, "g"'] & \underline{D} \\
B \arrow[u, "k"] \arrow[ur, "\xi"'] & &
\end{tikzcd}$$

では $\pi \circ \xi$ も図式 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

$$\begin{tikzcd}
& B \arrow[dl, "\pi_i\circ\xi"'] \arrow[dr, "\pi_j\circ\xi"] & \\
D_i \arrow[rr, "Du"'] & & D_j
\end{tikzcd}$$

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 $Du \circ \pi_i \circ \xi = Du \circ \pi_i \circ h \circ k$ であり, $\pi \circ h$ が図式 D の錐であることから $Du \circ \pi_i \circ h \circ k = \pi_j \circ h \circ k = \pi_j \circ \xi$ を充します．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

$$\begin{tikzcd}
& B \arrow[d, "k"] \arrow[ddl, "\pi_i\circ\xi"'] \arrow[ddr, "\pi_j\circ\xi"] & \\
& A \arrow[dl, "\pi_i\circ h"] \arrow[dr, "\pi_j\circ h"'] & \\
D_i \arrow[rr, "Du"'] & & D_j
\end{tikzcd}$$

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

極限が存在することが示せました．同様に双対を考えることで，余積と余等化が存在するときに余極限が存在することと，その逆が示せます．

このように圏 $\mathscr{C}$ が完備であることと，圏 $\mathscr{C}$ 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 $\mathscr{C}$ が完備であることと，引き戻しと終対象を持つことも同値です．実際，圏 $\mathscr{C}$ が完備のときに図式 $\{\ \}$ の極限が終対象 1，図式 $A \leftarrow 1 \rightarrow B$ の極限が積 $A \times B$ になります．また，その逆は $A \xrightarrow{\langle f,g \rangle} B \times B \xleftarrow{\Delta_B} B$ の引き戻し:

$$\begin{tikzcd}
E \arrow[r, "f \circ e = g \circ e"] \arrow[d, "e"'] & B \arrow[d, "\Delta_B"] \\
A \arrow[r, "\langle f,g \rangle"'] & B \times B
\end{tikzcd}$$

から $A \underset{g}{\overset{f}{\rightrightarrows}} B$ の等化 $E \xrightarrow{e} A$ が得られるために引き戻しと終対象を持つ圏 $\mathscr{C}$ は積と等化を持つことになり，以上から圏 $\mathscr{C}$ 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　随伴関係

2.8.1　普遍矢について

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[*48]．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ため[*49],「三毛猫」は「猫」よりも下位の普遍です．この「三毛猫」という概念は「猫」という概念よりも下部の概念で，この「猫」という類概念を「毛並み」という差異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，この特徴を差異とすると類概念を差異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

[*48] 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．

[*49] 「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．

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 $S_1, \ldots, S_n$ 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 $[a] \neq [b]$ 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 $\mathscr{C}$ と, その部分圏 $\mathscr{D}$ を考えます. この圏 $\mathscr{D}$ は $A, B \in \mathrm{Ob}\mathscr{D}$ 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 $\mathscr{D}$ 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 $\{$三毛猫, 白猫, 黒猫, 虎猫, $\ldots\}$ が対応します. そして, 圏 $\mathscr{D}$ が骨格的で, 圏 $\mathscr{D}$ から圏 $\mathscr{C}$ への函手 F が忠実, かつ充足で, さらに任意の $B \in \mathrm{Ob}\mathscr{C}$ に対して $FA \cong B$ を充たす $A \in \mathrm{Ob}\mathscr{D}$ が存在するときに圏 $\mathscr{D}$ を圏 $\mathscr{C}$ 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 $F : \mathscr{C} \longrightarrow \mathscr{D}$ で, 任意の $B \in \mathrm{Ob}\mathscr{D}$ に対して $FA \cong B$ を充す $A \in \mathrm{Ob}\mathscr{C}$ 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 $\mathscr{C}$ と圏 $\mathscr{D}$ が「**同値 (equivalent)**」と呼び $\mathscr{C} \simeq \mathscr{D}$ 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 $\mathscr{C}$ とし, それから「**猫の分類の集合**」をここでは $\mathscr{D} = \{$三毛猫, 白猫, 黒猫, 虎猫, 斑猫, その他$\}$ とします. このときに $\mathscr{C}$ から $\mathscr{D}$ への写像として包含写像から誘導される写像 F, $\mathscr{D}$ から $\mathscr{C}$ 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 $(A, B \in \mathscr{C})$ のときに恒等矢 1 に対応する関係を「A は A である」, $A, B \in \mathscr{C}$ 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 $F : \mathscr{C} \to \mathscr{D}$ 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 $\mathscr{C}$ 内の毛並みに関する関係は猫の毛並みによる分類 $\mathscr{D}$ でも同様の関係があり, 逆に $\mathscr{D}$ での関係も $\mathscr{C}$ に対応する関係があります. したがって, $\mathscr{C}, \mathscr{D}$ を圏とみなすときに写像 F, G は函手になり, 特に $\mathscr{D}$ は $\mathscr{C}$ と同型 $\mathscr{C} \simeq \mathscr{D}$ になります.

ここで毛並みの分類については次の可換図式が得られます:

$$
\begin{tikzcd}
\text{みけ} \arrow[r, "\text{同じ毛並み}"] \arrow[dr, "\text{違う毛並み}"'] & \text{たま} = G\text{三毛猫} \arrow[d, "\text{違う毛並み}"] & \text{三毛猫} = [\text{たま}] \arrow[d, "\text{同類ではない}"] \\
& \text{しろ} = G\text{白猫} & \text{白猫} = [\text{しろ}]
\end{tikzcd}
$$

この例は左側の可換図式が猫全体: 圏 $\mathscr{C}$ での話で, 右側が猫の分類: 圏 $\mathscr{D}$ での話です．この $\mathscr{C}$ と $\mathscr{D}$ の双方を繋いでいるものが函手 G です．まず, 函手 G は, 対象に関しては猫の種からその代表を返す函数で, 圏 $\mathscr{D}$ の矢「違う毛並み」は圏 $\mathscr{C}$ でも「違う毛並み」に写されます．これらの図式の面白いところは, 左側の「みけ」と「しろ」に関する矢が現われると, 可換になるようにその種の代表である「たま」から「しろ」への矢が一意に存在し, その矢はさらには種の間の矢と対応関係があることです．さらに「みけ」から「たま」への矢では注目すべき性質があります．これは函手 F によって圏 の矢:「同じ毛並み」は圏 $\mathscr{D}$ 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, $\langle$三毛猫, 同じ毛並み$\rangle$ 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, $\langle$白猫, 違う毛並み$\rangle$ という対は, 猫全体の集合を「白猫」と呼ばれる概念に対して「違う毛並み」という矢で帰属しないことを意味する対になっています．そして, これらの対を結ぶ種の間の矢として「同類ではない」が一つ存在しています．この対は次の可換図式でも現われます:

$$
\begin{tikzcd}
\text{みけ} \arrow[r, "\text{同じ毛並み}"] \arrow[dr, "\text{同じ毛並み}"'] & \text{たま} = G\text{三毛猫} \arrow[d, "1_{\text{猫}}"] & \text{三毛猫} = [\text{たま}] \arrow[d, "1_{\text{三毛猫}}"] \\
& \text{たま} = G\text{三毛猫} & \text{三毛猫} = [\text{たま}]
\end{tikzcd}
$$

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．これらのことを整理すると, ある圏 $\mathscr{C}$ の対象 A と圏 $\mathscr{D}$ 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 $\mathscr{D}$ の矢 $R \xrightarrow{f'} D$ が一意に存在しています:

$$
\begin{tikzcd}
A \arrow[r, "u"] \arrow[dr, "f"'] & GR \arrow[d, "Gf'"] & R \arrow[d, "f'"] \\
& GD & D
\end{tikzcd}
$$

ここで注目すべきことは左辺の圏 $\mathscr{C}$ に対して $\mathscr{D}$ 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 $\mathscr{C}$ の対象と，圏 $\mathscr{D}$ 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 $\mathscr{C}$ での関係には「イデア界」に対応する $\mathscr{D}$ 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 $G : \mathscr{D} \to \mathscr{C}$，対象 $C \in \mathscr{C}$ 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:

- $R \in \mathrm{Ob}\mathscr{D}$, $C \xrightarrow{u} GR$ とする
- 任意の $D \in \mathrm{Ob}\mathscr{D}, C \xrightarrow{f} CD$ 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できます:

$$
\begin{tikzcd}
C \arrow[r, "u"] \arrow[d, "1_C"] & GR \arrow[d, "Gf"] & R \arrow[d, "f'"] \\
C \arrow[r, "f"'] & GD & D
\end{tikzcd}
$$

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 $p : C \to X$ 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 $\tilde{U}_i, i = 1, 2, ..$ 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られます．まず，$q : D \to B$ を B の被覆空間とし，それから $p : C \to B$ を B の被覆空間とすると，被覆写像 $f : D \to C$ が存在し，$p \circ f = q$ となるときに q を普遍被覆空間と呼びます．この関係は次の可換図式で表現できます:

$$\begin{tikzcd} D \arrow[r, "f"] \arrow[dr, "q"'] & C \arrow[d, "p"] \\ & B \end{tikzcd}$$

次に Mac Lane の本 [54] に出ている例を挙げておきましょう, まず集合の圏 $\mathbf{Set}$ と体 K を係数とするベクトル空間の圏 $\mathbf{Vect}_K$ 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 $\mathbf{Set}$ 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 $U(W)$ に対応させせることで矢 $X \xrightarrow{f} U(W)$ を構成できます. それからこの f の対応関係を基に体 K について線形になるように拡張することで新たに $\mathbf{Vect}_K$ の矢 $V_X \xrightarrow{f} W$ を構成できます. 以上から, 次の可換図式がえられます:

$$\begin{tikzcd} X \arrow[r, "j"] \arrow[dr, "f"'] & U(V_X) \arrow[d, "Uf'"] & & V_X \arrow[d, "f'"] \\ & U(W) & & W \end{tikzcd}$$

2.8.2　随伴関係

重要な関係で随伴関係があります:

随伴関係

二つの函手 $F : \mathscr{C} \longrightarrow \mathscr{D}$, $G : \mathscr{D} \longrightarrow \mathscr{C}$, について函手 F が函手 G と随伴関係にあるとは次の関係を充すときです:

$$C(FA, B) \cong C(A, GB)$$

このときに函手 F は 函手 G の左随伴と呼んで $F \dashv G$ と表記し, 函手 G は函手 F の右随伴と呼んで $G \vdash F$ と表記します.

この随伴関係は論理学の証明記号を使って $\dfrac{FA \longrightarrow B}{A \longrightarrow GB}$ とも記述されます.

この随伴関係の例として対象の冪が挙げられます．実際, $f \in C(C \times A, B)$ に対して $\hat{f} \in C(C, B^A)$ が唯一存在し, 可換図式

$$
\begin{tikzcd}
C \times A \arrow[r, "f"] \arrow[d, "\hat{f}\times 1_A"'] & B \\
B^A \times A \arrow[ur, "\mathrm{ev}"'] &
\end{tikzcd}
$$

と $FC = C \times A, GB = B^A$ と置換えることで $C(FC, B) \cong C(C, GB)$ であることが判り, このことから, この函手 F は函手 G の左随伴になります.

ここで重要な矢にユニットとコユニットがあります．まず, 随伴関係 $\dfrac{FA \longrightarrow B}{A \longrightarrow GB}$ について $B = FA$ とするときに $\dfrac{FA \longrightarrow FA}{A \longrightarrow GFA}$ を得ます．このときに $FA \xrightarrow{\mathrm{id}_{FA}} FA$ に対応する矢 $A \xrightarrow{\eta_A} GFA$ が存在します．この矢を「**ユニット (unit)**」と呼びます.

2.9 トポス (Topos)

2.9.1 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)**」[*50]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 ([4] の註を参照)．ここで述べる圏論のトポスはそれに似た働き, つまり, 判断の枠組を与える場所としての働きをします．以下, トポスがどのように判断の枠組を与えるかを見ましょう.

2.9.2 部分対象分類子 (subobject classifier)

終対象 1 を有する圏 $\mathscr{C}$ 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 $\mathscr{C}$ 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:

---

*50 τόποι は τόπος の複数形です.

部分対象分類子の定義

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

- 圏 $\mathscr{C}$ には終対象 1 が存在する.
- 圏 $\mathscr{C}$ の単射である矢 $A \overset{f}{\rightarrowtail} B$ 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.

$$\begin{tikzcd} A \arrow[r, "!_A"] \arrow[d, tail, "f"'] & 1 \arrow[d, tail, "\top"] \\ B \arrow[r, "\chi_f"'] & \Omega \end{tikzcd}$$

この図式の意味を小集合の圏 **Set** で説明しておきましょう. このとき, 対象 A, B の関係は $A \subset B$ として考えられます. それから Ω を $\{\text{True, False}\}$ 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[*51]. 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:

対角矢 Δ_A の特性矢 δ_A

圏 $\mathscr{C}$ 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:

$$\begin{tikzcd} A \arrow[r, "!_A"] \arrow[d, tail, "\Delta_A"'] & 1 \arrow[d, tail, "\top"] \\ A \times A \arrow[r, "\delta_A"'] & \Omega \end{tikzcd}$$

*51 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.

この図式の例として圏 **Set** にて部分対象分類子 Ω を $\{\mathrm{True}, \mathrm{False}\}$, $\top 1 = \mathrm{True}$ 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 $a =_A b \stackrel{\mathrm{Def}}{=} \delta_A(a, b)$ 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 $\vdash\!\smile^{\mathfrak{a}}\, \mathfrak{a} = \mathfrak{a}$ すなわち，$\forall x(x = x)$ を真 (True)，概念記法で $\vdash\!\smile^{\mathfrak{a}}\, \mathfrak{a} \neq \mathfrak{a}$，すなわち，$\forall x(x \neq x)$ を偽 (False) として定義していますが [32]，ここでの同値性 “$=_A$” の定義で真理値集合に該当する部分対象分類子 Ω が与えられていなければならないために，フレーゲの真理値の定義が妥当でないことが分かります．実際，真理値 True と $\forall x(x = x)$ の値が一致することが主張できても演算子 “$=$” 自体が真理値に依存するために $\forall x(x = x)$ を真理値 True の定義にできないためです．ところで，アリストテレスによると真理値 は「**真や偽は命題の状態を示すもの**」で，「**存在するものを存在すると言い，あるいは存在しないものを存在しないと言うこと**」が真で，「**存在するものを存在しないと言い，あるいは存在しないものを存在すると言うこと**」が偽である ([3]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 $\mathscr{C}$ 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

$$\begin{tikzcd} 1 \arrow[r, "!"] \arrow[d, tail, "\langle\top,\top\rangle"'] & 1 \arrow[d, tail, "\top"] \\ \Omega \times \Omega \arrow[r, "\wedge"'] & \Omega \end{tikzcd}$$

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

矢 $\neg$ の定義

$$\begin{tikzcd} 0 \arrow[r, "!"] \arrow[d, tail, "0_1"'] & 1 \arrow[d, tail, "\top"] \\ 1 \arrow[r, "\bot"'] & \Omega \end{tikzcd} \qquad \begin{tikzcd} 1 \arrow[r, "!"] \arrow[d, tail, "\bot"'] & 1 \arrow[d, tail, "\top"] \\ \Omega \arrow[r, "\neg"'] & \Omega \end{tikzcd}$$

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

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

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

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

トポスの定義

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

トポスになる圏として代表的なものとして小集合から構成される圏 **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.

2.9.4　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.

$$\begin{tikzcd}
 & N \arrow[r, "\sigma"] \arrow[d, "f"] & N \arrow[d, "f"] \\
1 \arrow[ur, "\xi"] \arrow[dr, "g"'] & & \\
 & A \arrow[r, "h"'] & A
\end{tikzcd}$$

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:

$$\begin{tikzcd}
 & A \times N \arrow[r, "\mathrm{id}_A \times \sigma"] \arrow[d, "f"] & A \times N \arrow[d, "f"] \\
A \arrow[ur, "\zeta"] \arrow[dr, "g"'] & & \\
 & B \arrow[r, "h"'] & B
\end{tikzcd}$$

なお, $\zeta = \langle \mathrm{id}_A, \xi_N \rangle$, 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 $h' : B^A \to B^A$ が存在します:

$$
\begin{tikzcd}
B^A \times A \arrow[r, "\mathrm{ev}"] \arrow[d, "h' \times \mathrm{id}_A"'] \arrow[dr, "h \circ \mathrm{ev}"] & B \arrow[d, "h"] \\
B^A \times A \arrow[r, "\mathrm{ev}"'] & B
\end{tikzcd}
$$

この矢 h' は具体的には $h \circ \mathrm{ev}$ の転置: $(\widehat{h \circ \mathrm{ev}})$ 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:

$$
\begin{tikzcd}
B^A \times A \arrow[dr, "h \circ \mathrm{ev}"] & \\
1 \times A \arrow[u, "g' \times \mathrm{id}_A"] \arrow[r, "g \circ \pi_2"] \arrow[d, "\pi_2"'] & B \\
A \arrow[ur, "g"'] &
\end{tikzcd}
$$

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

$$
\begin{tikzcd}
 & N \arrow[r, "\sigma"] \arrow[d, "k"] & N \arrow[d, "k"] \\
1 \arrow[ur, "\xi"] \arrow[dr, "g'"'] & & \\
 & B^A \arrow[r, "h'"'] & B^A
\end{tikzcd}
$$

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:

$$
\begin{tikzcd}
A \times N \arrow[r, "f"] \arrow[d, "\mathrm{id}_A \times k"'] & B \\
A \times B^A \arrow[ur, "\mathrm{ev}"'] &
\end{tikzcd}
$$

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:

$$
\begin{tikzcd}
 & N \times N \arrow[r, "\mathrm{id}_N \times \sigma"] \arrow[d, "+_{\mathbf{E}}"] & N \times N \arrow[d, "+_{\mathbf{E}}"] \\
N \arrow[ur, "\zeta"] \arrow[r, "\mathrm{id}_N"'] & N \arrow[r, "\sigma"'] & N
\end{tikzcd}
$$

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

2.10　トポスの基本定理

トポスの基本定理

- 圏 $\mathscr{E}$ がトポスであり, B が $\mathbf{E}$ の任意の対象であるときにコンマ圏 $(\mathbf{E}\downarrow B)$ 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 $f^* : (\mathbf{E}\downarrow B) \to (\mathbf{E}\downarrow A)$, $\Sigma_f : (\mathbf{E}\downarrow A) \to (\mathbf{E}\downarrow B)$ と $\Pi_f : (\mathbf{E}\downarrow A) \to (\mathbf{E}\downarrow B)$ が存在して $\Sigma_f \dashv f^* \dashv \Pi_f$ を充たす.

この基本定理の証明を順番に行いましょう.

まず, 圏 $\mathbf{E}$ が基本トポスであればコンマ圏 $(\mathbf{E}\downarrow B)$ も基本トポスであることを示しましょう. そのためにはコンマ圏 $(\mathbf{E}\downarrow B)$ で終対象の存在, 積の存在, 冪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 $B \xrightarrow{\mathrm{id}_B} B$ は任意の $A \xrightarrow{f} A$ に対して次の図式が可換になります:

$$
\begin{tikzcd}
A \arrow[rr, "f"] \arrow[dr, "f"'] & & B \arrow[dl, "\mathrm{id}_B"] \\
& B &
\end{tikzcd}
$$

このことから $B \xrightarrow{\mathrm{id}_B} B$ 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:　圏 $\mathbf{E}$ 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:

$$
\begin{tikzcd}
E \arrow[drr, "k"] \arrow[drrr, "l"] \arrow[ddrr, "h"'] & & & \\
& & D \arrow[r, "f'"] \arrow[d, "g'"] & C \arrow[d, "g"] \\
& & A \arrow[r, "f"] & B
\end{tikzcd}
$$

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

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

$$
\begin{tikzcd}
A \arrow[r, "!_A"] \arrow[d, tail, "f"'] & 1 \arrow[d, tail, "\top"] \\
B \arrow[r, "\chi_f"'] & \Omega
\end{tikzcd}
$$

$$
\begin{tikzcd}
A \arrow[rr, "f"] \arrow[dr, "f"] \arrow[dd, tail, "h"'] & & B \arrow[dl, "\mathrm{id}_B"'] \arrow[dd, tail, "\top"] \\
& B & \\
C \arrow[ur, "g"] \arrow[rr, "\chi_h"'] & & \Omega \times B \arrow[ul, "\pi_2"']
\end{tikzcd}
$$

■**関手 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:

$$
\begin{tikzcd}
 & C \arrow[dd, "g"] \arrow[dr, "k"] & \\
 & & Q \arrow[dl, "h"] \\
A \arrow[r, "f"] & B &
\end{tikzcd}
$$

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:

$$
\begin{tikzcd}
D \arrow[rr, "f'"] \arrow[dd, "g'"'] \arrow[dr, "k'"] & & C \arrow[dd, "g"] \arrow[dr, "k"] & \\
 & P \arrow[rr, "f''"] \arrow[dl, "h'"] & & Q \arrow[dl, "h"] \\
A \arrow[rr, "f"'] & & B &
\end{tikzcd}
$$

ここで $h, g \in \mathrm{Ob}(\mathbf{E} \downarrow B)$ に対して $h', g' \in \mathrm{Ob}(\mathbf{E} \downarrow A)$ 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 $f^* : \mathbf{E} \downarrow B \longrightarrow \mathbf{E} \downarrow A$ が構築できます.

2.11　高階論理 λ-h.o.l. とトポス

基本トポスでは自然に言語を導入できます．この言語は「圏論による論理学」[18] では「函数型高階論理 λ-h.o.l.」,「Toposes and Local Set Theories」[46] では「locat set language」, 一般的には「Mitchell-Benabou Language」と呼ばれるトポスに内在的な言語です．ここでは基本トポスを小集合の圏 **Set** とし, 部分対象分類子 Ω を真理値 $\{\mathrm{True}, \mathrm{False}\}$ とする, いわゆるブーリアン・トポス $\mathbf{E}$ の事例を紹介します．なお, 圏を小集合の圏 **Set** とするために対象は集合になりますが, 函数型言語としては, その対象である集合を「**型 (type)**」と呼ぶために λ-h.o.l. の言語としての性格は型付きの言語になります．また, 型が小集合であるために各小集合には集合論から定義される言語を既に保持していますが, λ-h.o.l. は対象としての小集合に限定さることなく大域的に定義される言語になります.

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

2.11.1　高階論理 λ-h.o.l. について

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

函数の型の表記

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

まず, 1. については単なる略記として認めることは問題がないでしょう．2. の記号の表記については, 記号 $\langle\ \rangle$ を二項演算子として見なしたときに右側の記号との結合が強いとする立場を採ることを意味しており, このような表記を「**右結合**」と呼びます．ここで写像が通常の集合の写像であれば $f\circ(g\circ h) = (f\circ g)\circ h$ であるために, この結合の順序はそ

れほど問題にはなりません. ちなみに論理学に函数概念を最初に導入したのはフレーゲですが, 彼は函数を二項間の関係とみなしており, その場合, 項の場所に注目し, 三変数以上の函数を二項関係の延長として捉えています.

それから λ-h.o.l. の型については次のように定めています:

λ-h.o.l. の型 (type)

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

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

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

λ-h.o.l. の基本記号

1.	論理常項:	$=_{\alpha\langle\alpha t\rangle}$
2.	変項:	$x_\alpha, y_\beta, z_\gamma, \cdots$
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 $\alpha\alpha t$ 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 $((=_{\alpha\langle\alpha t\rangle}\ a)b)$ 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

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

高階論理 λ-h.o.l. はこれらの基本記号から次の項の定義に沿って生成される項から古典

的論理学の $\wedge, \neg, \supset$ といった論理記号が生成されます.

λ-h.o.l. の型付き項の定義

1. $x_\alpha, y_\alpha, \cdots$ 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. $(\lambda x^i_\alpha.A_\beta)$ 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	$\stackrel{\text{Def.}}{=}$	$\lambda x_t.\mathrm{T} = \lambda x_t.x_t$
3.	$\neg_{tt}$	$\stackrel{\text{Def.}}{=}$	$(\lambda x_t.(\mathrm{F} = x_t))$
4.	$\wedge_{t\langle tt\rangle}$	$\stackrel{\text{Def.}}{=}$	$\lambda x_t.\lambda y_t.(\lambda f_{t\langle tt\rangle}.(f_{t\langle tt\rangle}\mathrm{TT}) = \lambda f_{t\langle tt\rangle}.(f_{t\langle tt\rangle}x_t y_t)))$
5.	$\supset_{t\langle tt\rangle}$	$\stackrel{\text{Def.}}{=}$	$\lambda x_t.(\lambda y_t.(x_t = (x_t \wedge_{t\langle tt\rangle} y_t)))$

ここで T, F といった真理値が右辺の式で定められるというよりは, むしろ, 言語に包含さ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 '¬', 論理式の連言 '∧' と含意 '⊃' 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.	$A_t \supset B_t$	$\stackrel{\text{Def.}}{=}$	$((\supset_{t\langle tt\rangle} A_t)B_t)$

2.11.2 高階論理とトポスとの関係

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

型と対象の対応関係

型 e	$\to$	対象 E
型 t	$\to$	対象 Ω
型 $\langle \alpha, \beta \rangle$	$\to$	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えます．トポス $\mathbf{E}$ の対象 a に対してその終対象 1 からの矢 $1 \to a$ 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, 冪や矢の積が存在するといった性質もあります．これらのことを利用して λ-h.o.l. の項 a とトポス $\mathbf{E}$ の矢 $|a|$ との対応付けを行ってみましょう.

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

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

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

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

$$
\begin{tikzcd}
 & B^A & \\
1 \arrow[ur, "|C_{\alpha\beta}|"] \arrow[r, "\langle |C_{\alpha\beta}|{,}|D_\alpha| \rangle"] \arrow[dr, "|D_\alpha|"'] \arrow[rr, bend right=80, "|C_{\alpha\beta}D_\alpha|"'] & B^A \times A \arrow[u, "\pi_1"'] \arrow[d, "\pi_2"] \arrow[r, "ev"] & B \\
 & A &
\end{tikzcd}
$$

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

$$
\begin{tikzcd}
 & 1 \arrow[r, "|C_{\alpha\beta}|"] & B^A & \\
1\times A \arrow[ur, "\pi_1"] \arrow[rr, "|C_{\alpha\beta}|\times|x_\alpha|"] \arrow[dr, "\pi_2"'] \arrow[rrr, bend right=70, "|C_{\alpha\beta}x_\alpha|"'] & & B^A\times A \arrow[u, "\pi_1"'] \arrow[r, "\mathrm{ev}"] \arrow[d, "\pi_2"] & B \\
 & A \arrow[r, "|x_\alpha|"'] & A &
\end{tikzcd}
$$

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

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

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

$$
\begin{tikzcd}
B^A & B^A\times A \arrow[l, "\pi_1"'] \arrow[r, "\pi_2"] \arrow[drr, bend left=60, "\mathrm{ev}"] & A & \\
1 \arrow[u, "|\lambda x_\alpha.D_\beta|"] & 1\times A \arrow[l, "\pi_1"] \arrow[r, "\pi_2"] \arrow[u, "|\lambda x_\alpha.D_\beta|\times|x_\alpha|"'] \arrow[rr, bend right, "\mathrm{ev}\circ(|\lambda x_\alpha.D_\beta|\times|x_\alpha|)"'] \arrow[d, "\pi_1"] & A \arrow[u, "|x_\alpha|"'] & B \\
 & 1 \arrow[urr, bend right, "|D_\beta|"'] & &
\end{tikzcd}
$$

ここでも矢の積 $|\lambda x_\alpha.D_\beta|\times|x_\alpha|$ が判り易くなるように可換図を構築していますが, そのために $|\lambda x_\alpha.D_\beta|$ が $|D_\beta|$ の転置であることが判り難くなっています. そこで矢の積の箇所を説明する部位を除いた可換図式を以下に示しておきましょう:

$$
\begin{tikzcd}
B^A \times A \arrow[drr, "\mathrm{ev}"] & & \\
1 \times A \arrow[u, "|\lambda x_\alpha.D_\beta|\times|x_\alpha|"] \arrow[rr, "\mathrm{ev}\circ(|\lambda x_\alpha.D_\beta|\times|x_\alpha|)"] \arrow[d, "\pi_1"'] & & B \\
1 \arrow[urr, "|D_\beta|"'] & &
\end{tikzcd}
$$

この可換図式から $\mathrm{ev}\circ(|\lambda x_\alpha.D_\beta|\times|x_\alpha|) = |D_\beta| circ \pi_1$ 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 $\lambda x_\alpha.C_{\alpha\beta}x_\alpha$:　$1\times A \stackrel{|C_{\alpha\beta}x_\alpha|}{\longrightarrow} B$. なお, 以下の可換式で $|C_{\alpha\beta}x_\alpha| = |ev\circ(|\lambda x_\alpha.D_\beta|\times|x_\alpha|)|$ になる結果を用いています:

$$
\begin{tikzcd}
B^A & B^A \times A \arrow[l, "\pi_1"'] \arrow[r, "\pi_2"] \arrow[drr, bend left=60, "\mathrm{ev}"] & A & \\
1 \arrow[u, "|\lambda x_\alpha.C_{\alpha\beta}x_\alpha|=|C_{\alpha\beta}|"] & 1\times A \arrow[l, "\pi_1"] \arrow[r, "\pi_2"'] \arrow[u, "|\lambda x_\alpha.C_{\alpha\beta}|\times|x_\alpha|"] \arrow[rr, bend right=50, "|C_{\alpha\beta}x_\alpha|"'] & A \arrow[u, "|x_\alpha|"'] & B
\end{tikzcd}
$$

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:

$$
\begin{tikzcd}
B^A \times A \arrow[drr, "\mathrm{ev}"] & & \\
1 \times A \arrow[u, "|\lambda x_\alpha.C_{\alpha\beta}|\times|x_\alpha|"] \arrow[rr, "\mathrm{ev}\circ(|\lambda x_\alpha.D_\beta|\times|x_\alpha|)"'] & & B
\end{tikzcd}
$$

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

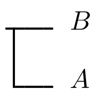
■**項** $C_\alpha \ = \ D_\alpha$:　$1 \overset{\delta_A \circ \langle |C_\alpha|, |D_\alpha| \rangle}{\longrightarrow} \Omega$

$$\begin{tikzcd}
 & A & \\
1 \arrow[ur, "|C_\alpha|"] \arrow[r, "\langle |C_\alpha|, |D_\alpha| \rangle"] \arrow[dr, "|D_\alpha|"'] & A \times A \arrow[u, "\pi_1"] \arrow[d, "\pi_2"] \arrow[r, "\delta_A"] & \Omega \\
 & A &
\end{tikzcd}$$

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．

2.11.3　λ-h.o.l. の解釈について

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

ここで現代の論理学の創始者と言えるフレーゲは含意 ‘$A \supset B$’ を  , 否定 ‘$\neg A$’ を ‘ a’ とし, それと「**Modus Ponens(MP)**」と呼ばれる推論規則*52, 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:

*52 ‘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 ⊃ の定義

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

このように λ-h.o.l. では論理常項 “$=_{t\langle t,t\rangle}$” と連言 “∧” から含意が定義されるので, トポス **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 **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 **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 “∀” 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

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

で定義されます. この定義もトポス **E** の下で

トポス E における量化詞

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

になることがわかります.

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

$$\begin{array}{ll} A.1 & (x_\alpha = y_\alpha) \supset (A_{\alpha t}x_\alpha = A_{\alpha t}y_\alpha) \\ A.2 & (A_{\alpha\beta} = \forall x_\alpha(A_{\alpha\beta} = B_{\alpha\beta}) \\ A.3 & (\lambda x^i_\alpha.A_\beta)B_\alpha = A_\beta\left[x^i_\alpha := B_\alpha\right] \\ A.4 & (A_{tt}T_t \wedge A_{tt}F_t) = \forall x_t A_{tt}x_t \end{array}$$

第 3 章

Python について

3.1　Python の概要

3.1.1　言語仕様としての Python

Python は一つの言語仕様であり, 通常, Python と呼ばれているソフトウェアは C で記述された CPython を指します. この CPython の他には Java による Jython, .Net Framework/Mono 上の IronPython, Python 自体[*1]による PyPy, C ライブラリを効率よく利用するために開発された Cython と目的に応じた実装があります.

SageMath は Python を基盤とし, 処理速度が重要視される処理等で Cython を用いています. 実際, CPython の処理自体は高速ではなく, そのために高速処理が可能なモジュールや §4.3 で紹介する NUMBA を利用する方法がありますが, それでも解決しなければ SageMath のように Cython を利用するか, 他の Julia のような高速処理が可能な言語を用います. なお, Python は 2 系と 3 系が互換性の問題のために並立しており, SageMath もライブラリの互換性の問題などから Python 2.7 をその基盤としていましたが, SageMath 9.0 で Python 3 に完全に移行したために, ここでの解説では Python 3 系を中心に解説します.

3.1.2　簡素化された構文

Python は構文が簡素で, その機能拡張はモジュールで行います. Python では変数宣言は不要で, 条件分岐は if 文, 例外処理は try 文, 反復処理は for 文と while 文と必要最低限に抑えられ, その結果, Perl[*2] や *Mathematica* に見られる複雑な処理を一行に閉じ込める「**超絶的技巧**」を駆使したプログラミングではなく, 常識的なプログラミングで妥当な結果が得られる均質的なプログラムに収斂されます. このことに加え, PEP-8 [*3]に代表されるプログラム記述のためのガイドラインがあり, これらのガイドラインにしたがってプログラムを構築すれば文書化も含めて, ある程度の品質のプログラムが得られるという実用本位の言語です.

3.1.3　構文要素としての字下げ

多くの計算機言語で「**インデント (indent, 字下げ)**」はプログラムの構造を視覚的に把握するために用いられていますが, Python の字下げは構文上の必要不可欠な要素です.

[*1] 正確には Python に幾つかの制約を加えた RPython です.
[*2] 駱駝形のプログラム例 (camel code): https://gist.github.com/cgoldberg/4332167 ,4 頭のラクダを ASCII アートで描くプログラムで, それ自体がラクダの ASCII アートになっています.
[*3] PEP(=Python Enhancement Proposal): Python 改善提案書

実際, C の if 文は空白文字の Space や TAB を使った字下げはプログラマの意図は別にあってもなくても良いものですが, Python の字下げ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, それらの行の並びを揃えて字下げを行わなければなりません[*4].

つまり,

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

のように if 文内部の文節 (ここでは 'y = 1' と 'y = 0') と if 文を構成する文節 if と else を同水準で揃え, その間に挟まれた文を一つのブロックとして字下げの水準を揃えます. したがって,

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

は許容されても, 記号 ":" を一つの行に二つ以上含む

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

や字下げ水準がチグハクでプログラムの構造が十分に 2 次元的に表現できていない

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

の双方は構文エラーになります.

3.1.4 文書文字列 (docstring)

Python にはプログラム内部に解説や例題等の文書を組み込んで文書としての価値を高める工夫があります. この埋め込まれた文字列を「**文書文字列 (docstring)**」と呼びます. この文書文字列を函数定義文に所定の位置に書き込めば Python インタプリタ等の UI からのヘルプで表示される文書として使えます. ここで実際に Python インタプリタを仮想端末から起動して操作してみましょう:

[*4] 仕様として字下げを持つ昔からの言語に FORTRAN77 もありますが, FORTRAN77 はカード読取機の制約に由来し, プログラムの構造の可視化の意図はありません.

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

Help on function add2 in module __main__:

add2(x)
    2を足すよー
>>> add2.__doc__
'\n    2をたすよー\n    '
>>>
```

ここで, “> > >” がインタプリタの通常のプロンプトで, このプロンプトに続けて入力します. また, Python には行末端を示す文字はなく, インタプリタ上で行を入力し, Enter/Return キーを押すと一行で評価できる式であれば結果を返し, ここでの函数の定義のような文の記述の途中であればプロンプトが “... ” に切り替わります. ここからはプログラムの構造に従ってインデントも含めて入力します. インデントとしては一段が 4 文字の Space で行うことが推奨されています. さて, 文書文字列は三連続の二重引用符 ("), あるいは三連続の単引用符 (') で括られた文字の列であり, ここでは文書文字列のエンコーディングが UTF-8 であることを明示的に指示するために接頭辞 “u” を配置しています. なお, Python 3 から文字コードが標準で UNICODE になるため, 他の文字コードに設定していなければ不要です. この函数の定義中はプロンプトが “... ” に切り替わったままですが, 通常のプロンプトに復帰したければ単に Enter キーか Return キーだけを押します. これで函数 add2() が定義できました. それからオンラインマニュアルの表示を函数 help() で行います. ここで, ‘help(add2)’ と入力すれば函数中に記載した文書文字列が表示されます. この定義した函数 add2() の文書文字列は函数 add2() の属性__doc__に割り当てられています.

文書文字列は三連続の二重引用符 (""") か三連続の単引用符 (''') で括られた文字列のために三重の引用符でなければ書文字列内部に引用符や改行が記入できます. この文書文字列に関する PEP には PEP-257 があり, PEP-287 では「**組版指示 (markup) 言語**」の「**reST(reStructuredText)**」での記述が提唱されています. ここで reST は Markdown と同様の組版指示言語で, プレーンテキスト上で見出しや箇条書をアスキーアート風に記述するために記述性と可読性く, reST の文書は文書生成ツールの Sphinx[*5]を使えば

[*5] http://www.sphinx-doc.org/en/stable/index.html と http://docs.sphinx-users.jp/ を参照.

LaTeX, HTML や PDF 等の書式の文書に変換できます．このように Python はプログラムの文書化も重要視した仕様です．

3.1.5 クラスに基づくオブジェクト指向プログラミング言語

§2 で述べたようにオブジェクト指向プログラミング言語では扱う対象全てを「**オブジェクト**」として捉え, さらにクラスに基づくオブジェクト指向プログラミング言語であれば, 扱うべき対象を抽象化して「**概念**」として捉え,「**クラス**」をオブジェクトを説明規定するもの, すなわち, **概念の内包**や**概念の外延**として表現します．ここで実際に扱うデータは現実のもの, あるいはそれに最も近接するもののために「**概念の外延を構成する個体**」に相当し, クラスが実体化することを「**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)**」と呼びます．そして, クラスがあるクラスのインスタンスであれば, この「**インスタンス化で得られたクラス**」を「**クラス・オブジェクト (class object)**」,「**クラスのクラス**」として, クラスの雛型に相当するクラスを「**メタクラス (metaclass)**」, 実体化したオブジェクトを「**インスタンス・オブジェクト (instance object)**」と呼びます．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はクラスタイプです

```
class TEST:
    pass
```

pass 文は「**何もしない**」ことを意味する文です． クラス TEST のインスタンス化は ‘TEST()’ で行います．ところで, クラス TEST では好き勝手ができます:

```
>>> class TEST:
....     pass
....
>>> a = TEST()
>>> a.name = 'mike'
>>> a.weight = '10kg'
>>> a.age = '10years'
>>> a.name
'mike'
>>> a.weight
'10kg'
>>>
```

この例では TEST クラスの定義と実体 (インスタンス) 化したオブジェクト (インスタンス) を名前 a に束縛し, a.pet, a.weigth と a.age という属性に値を設定して C の構造体に類似した使い方をしています．ところで, 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．この属

性の表記は先頭がインスタンス名, 区切記号が記号 "." で, そのうしろが属性です. この例のように演算子 "=" を用いて, 右側の被演算子が列のときに同じ成分数の名前のタプルを左側に配置すると, タプルを構成する名前に対して各オブジェクトの割当ができます:

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

ここでは 4 成分の変数のタプル 'x, y, z, w' にリスト '[1, 2, 3, 4]' の各成分を束縛させていますが, このような変数への束縛は演算子 "=" の双方の被演算子が列として長さが等しければできます. つまり, 括弧なしのタプルとして変数列の長さが右側の列と同じ長さであることが必要です.

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 の構造体に類似しています:

```
class TEST:
    x = 1
    y = 1
```

このクラスには二つの属性 x と y があり, 値として 1 を設定しています. 実際に使ってみましょう:

```
>>> class TEST:
....        x = 1
....        y = 1
....
>>> a1 = TEST()
>>> a1.x
1
>>> a1.y
1
xx z
>>> a1.x = 128
>>> a1.y = 0
>>> a1.x
128
>>> a1.y
0
>>>
```

この例ではクラス TEST を定義し, ‘a1 = TEST()’ でインスタンス化と同時に名前 a1 にオブジェクトを束縛させ, その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．

さて, クラスには値としての属性だけではなくメソッドという, そのクラスに所属するオブジェクト固有の能力/機能を表現する属性があります．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**」という習性があり, 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ですが, 犬のポチがその仕草をしたからといって洗濯物を取り込む理由になりません．つまり, ミケやポチといったインスタンスが属するクラスは「**猫**」と「**犬**」で, ミケが包含される猫にそのような属性 (習性) があっても, ポチが包含される犬にはありません．このように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**」という問に対する一つの回答ですが, 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に対応します．メソッドはクラスに結び付けられた函数[*6]として表現され, そのクラスのインスタンスか, そのクラスの派生クラスのインスタンスでなければ使えません．

また属性の役割を考えると, そのクラスを特徴付ける種差や特有性のようにクラス単位で共通になる値が設定される属性, 個々のインスタンスごとに異なる値が設定される属性の二種類が考えられます．ここで前者の属性のようにクラス全体で共通になる値が設定される属性を「**クラス変数**」, 後者のようにインスタンスごとに異なる値が設定される属性を「**インスタンス変数**」と呼びます．なお, Python には C++ のようなクラス変数やインスタンス変数に対する型宣言がないために変数の使い方で両者を区別します．たとえば,「**クモ**は足の数が 8 本」の「**足の数が 8 本**」はクモというクラスを特徴付ける属性の一つで,「**足の数**」がクラス変数です．また,「**犬**のニコの年齢は二才」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) ごとに異なるために, 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．なお, Python の属性には他の言語の「**公開 (public)**」や「**非公開 (private)**」といった弱く, Python では非公開にしたい属性名に ‘__’ を先頭に付けて属性の隠蔽を行います．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クラス object を継承して「**記述子 (ディスクリプタ)**」を利用します．

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「**インスタンスメソッド (instancemethod)**」と函数的なメソッドである「**静的メソッド (staticmethod)**」があり, クラスメソッドなら ‘@classmethod’,

[*6] ここでの函数は数学での函数ではなく, 計算機のある「機能 (functionality)」を実装したものです．

静的メソッドなら '@staticmethod' といった「**デコレータ**」で定義できます．これらのメソッドには定義の際の引数に違いがあり，クラスメソッドには第 1 引数に 'cls' というクラスそれ自体に対応する固定の名前があり，そのクラスの属性の参照ができます．しかし，静的メソッドはクラスそれ自体を示す引数を持たず，参照するクラス名を直接指定しなければなりません．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「**動的 (dynamic) な参照**」[*7]が行われますが，静的メソッドは名前で直接指定したクラスの参照に留まり，「**属性の参照を遡らずに指定した範囲内で行われる**」という意味で「**静的 (static) な参照**」になります．

次に 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メソッドを追加したクラス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
```

メソッドの定義自体は Python の函数定義と同様ですが，インスタンスメソッドで操作するインスタンスを 'self' いう名前で指示し，第 1 引数に必ず配置し，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指示します．ここで定義するメソッド wa() は 'return self.x+self.y' でそのインスタンスのクラス属性 x と y の和を返却します．実際に動かしてみましょう:

```
>>> class TEST:
...     x = 1
...     y = 1
...     def wa(self):
...         return self.x + self.y
...
>>> a1 = TEST()
>>> a1.x = 10
>>> a1.y = 2
>>> a1.wa()
12
>>>
```

[*7] MRO，あるいは C3 linearization がそのアルゴリズムとして用いられます．詳細は §3.10 を参照．

つぎに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上で函数 dir() を使った結果を示します:

```
>>> dir()
['__builtins__', '__doc__', '__name__', '__package__']
>>> globals()
{'__builtins__': <module '__builtin__' (built-in)>, '__name__': '
    __main__',
 '__doc__': None, '__package__': None}
>>> __name__
'__main__'
>>>
```

この組込函数 dir() は「**参照範囲 (スコープ) 内**」にある名前のリストを返却する函数で, 引数がなければ大域変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. なお, 大域変数の情報を返す函数に函数 globals() もあり, こちらは指定したスコープ内の大域変数の情報を辞書と呼ばれる型で返却します.

ところで大域変数 __name__ の先頭の文字 "_" には「**非公開 (private)**」という意味があり, 特に名前の先頭に文字列 '__' を持つメソッドや属性は隠蔽されるべきオブジェクトを意味します. そこで, この文字 "_" の働きを確認しておきましょう:

```
>>> dir()
['__builtins__', '__doc__', '__name__', '__package__']
>>> class test:
...     x = 1
...     _y = 1
...     __z = 1
...
>>> dir()
['__builtins__', '__doc__', '__name__', '__package__', 'test']
>>> a1 = test()
>>> dir()
['__builtins__', '__doc__', '__name__', '__package__', 'a1', 'test
    ']
>>> a1.x
1
>>> a1._y
1
>>> a1.__z
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
AttributeError: test instance has no attribute '__z'
>>> dir(test)
['__doc__', '__module__', '_test__z', '_y', 'x']
>>> a1._test__z
1
```

この例ではクラス test の属性として x, _y と__z を定め, クラス test のインスタンスを a1 で属性値の確認を行います. クラス test を定義したことで函数 dir() が返すリストに test が現れ, それから test のインスタンスとして a1 を生成したことで函数 dir() の結果に a1 が追加されます. 次に a1 の属性を参照しますが, ここで名前の先頭に文字 "_" が二つある属性 a.__z だけ参照ができません. このように名前の先頭に文字 "_" が二つある属性を隠蔽していますが, 函数 dir() でクラス test の中にある名前を見ると属性__z だけに文字の列 '_test' が先頭に置かれています. つまり, 文字の列 '__' を名前の先頭に持つ属性は文字列 _⟨クラス名⟩ が先頭に置かれた名前に置換されて本来の属性名で参照できなくても a1._test__z で本来の属性__z の値の参照ができます. さらに他の属性と同様に, その属性値の書換もできます. このように Python の属性の隠蔽の方法は不徹底な方法です. なお, 函数 dir() と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) という概念がなく, 安易に見えなくするだけで属性値の保護や型検証もありません. そこで, 属性値の管理を行う必要があるときは「**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[*8]. つまり, これらのメソッドは属性の設定, 取得, 削除の「**束縛動作 (binding behavior)**」に関わるメソッドで, これらのメソッドの上書きで束縛動作での振舞いを利用者の目的に適ったものにします.

ところで, オブジェクト指向プログラミング言語の大きな有難味の一つは「**継承**」と呼ばれる機能, つまり, 既存のクラスを土台に新しいクラスを効率的に構成できる機能です. この機能を活用することで既存の資産を新たなシステムの開発に生かせます. そこで, 自然数を拡張して有理数を構築することで後利益を体験してみましょう.

3.2 有理数を構築してみよう

3.2.1 有理数の表現

有理数を定義するためには, 有理数が「**何であるか**」,「**どのようなものであるか**」を明確にしておく必要があります. まず, 有理数は二つの整数 n, m を用いて n/m の書式で表現される数のため, 整数のクラスを使って整数対のクラスを定義します. この整数対は有

[*8] Python 2 系で, これらのメソッドはクラス object に実装されています.

理数ごとに異なるために, これら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整数対が定まるべきです．また, この整数対のインスタンス名を入力すると ‘n/m’ と分かり易い書式で表示されると良いでしょう．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**」と呼ばれるメソッドがあり, これらを上書きすれば上記の目的が達成できます．これら特殊メソッドの詳細は §3.8 で述べますが, ここでは分母と分子の値の設定をインスタンス化の時点で行うためにインスタンス属性の初期化を行う特殊メソッド___init___(), インスタンスが割当てられた名前が入力されたときに表示内容を定める特殊メソッド___repr___() の二つを用いて整数対のクラス PairOfInts を以下で定義します:

```
class PairOfInts:
    def __init__(self, numer, denom):
        self.numer = numer
        self.denom = denom
    def __repr__(self):
        return '%s/%s' %(str(self.numer), str(self.denom))
```

最初の def 節で定義するクラス名を指示し, 継承するクラスがあればその引数として記載します[*9]．それから特殊メソッド___init___() の第一引数の ‘self’ はインスタンスそれ自体を指示し, メソッドの定義では必ず記載しなければなりませんが, 実際の利用ではそれ以降の引数のを用います．つまり, 引数 numer と denom が実際のインスタンスの生成で必要な引数, つまり, インスタンス変数に束縛すべき値です．ここで二つのインスタンス変数の numer が分子, denom が分母に対応して ‘a = PairOfInts(1,2)’ で対象を生成します．この PairOfInts() の働きから PairOfInts() のことを「**構築子/コンストラクタ**」と呼びます．構築子で生成され, 変数 a に束縛されたオブジェクトに関しては, a.numer で属性 numer の値, a.denom で属性 denom の値の参照や名付けができます．それから二番目に定義されているメソッド___repr___() がオブジェクトを指示する名前が入力されたときや函数 print[*10]でどのような表示を行うべきかを定めます．ここでは出力書式を文字列’%s/%s’ で指定して ‘a = PairOfInts(1,2)’ でインスタンスを生成して名前 a をインタプリタに入力すると文字列 ’1/2’ が表示されることを意味します．

このことを実際に試してみましょう．ここではカレントディレクトリ[*11]上に PairOfInts クラスの定義をファイル PairOfInts.py に記載して import 文でインタプリタに読み込み

[*9] Python 2 で継承するクラスを指定しなければ自動的に古典的クラス型になるためにクラスタイプ (新クラス) を利用したければ必ず ‘object’ を引数として記載します

[*10] Python 2 では文の書式です．

[*11] os モジュールの函数 getcwd() で確認できます．変更は os モジュールの函数 chdir() の引数に経路を与えます．

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で, 複数のモジュールを一つにまとめた対象を「**パッケージ**」と呼びます．それから ‘from PairOfInts import PairOfInts’ でクラス PairOfInts() を直接読込みます:

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

この例では PairOfInts クラスを import 文で読込み, そのインスタンスを生成して名前 a に束縛させて名前 a を入力すると表示 ‘1/2’ を得ます．では, メソッド__repr__() がなければどうなるでしょうか?

```
>>> class TEST:
....     def __init__(self,numer,denom):
....         self.denom=denom
....         self.numer=numer
....
>>>
>>> b = TEST(1,2)
>>> b
<__main__.TEST instance at 0x4f607a0>
>>> [b.numer,b.denom]
[1, 2]
>>> repr('%s/%s' %(b.numer,b.denom))
"'1/2'"
```

ここでは PairOfInts クラスからメソッド__repr__() を除いた TEST クラスをインタプリタ上で直接定義し, それから生成したオブジェクトを名前 b のインスタンスとして割当てて, 名前 b を直接入力すると ‘<__main__.TEST instance at 0x4f607a0>’ と表示されます．ちなみに ’0x4f607a0’ という値は組込函数 id() が返却するオブジェクトの識別値と一致するためにオブジェクトの先頭番地であることが判ります．このようにメソッド__repr__() 等で表示内容を指示しない限り, 名前を入力してもオブジェクトの番地が返されるだけです．また, メソッド__repr__() が組込函数の函数 repr() に対応し, メソッド__repr__() が定義されていなくてもインスタンスの属性とそれらの書式を函数 repr() に引き渡せば同様の表示を得られます．このように特殊メソッド__repr__() を記述することでインスタンスの表示が定められ, 逆に言えば, 二つのオブジェクトの表示の一致からオブジェクトの同一性が保証できません．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)**」, メソッドの引数の型が異なるときは「**多重定義 (overload)**」と呼びます．

3.2.2　有理数のクラスの構築

「**有理数とは何なのか**」という問掛けに対して有理数を分母と分子の整数対で表現しましたが, では, 二つの整数対が有理数として「**同じ**」であると判断するためにはどのような基準が必要でしょうか? 最も安易な判断基準は分母と分子がそれぞれ等しいという条件でが, 1/2 と 2/4 は有理数として同じであっても整数対では (1, 2) と (2, 4) で同じではありません. この「同じ」という関係を「**与えられた自然数の対 (a, b) と (c, d) が ‘ad - bc = 0’ を満すとき**」と定めます. そして, 有理数は最終的に一意の整数対で表現されるべきです. 実際, (a, -b) と (-a, b) が等しい自然数であるため, 分母に相当する整数が常に正となるように符号を付け直し, 0 と異なる整数 c に対して (a*c, b*c) であれば c を除去して (a, b) とすべきです. つまり, RationalNumber クラスは自然数の対のクラス PairOfItnts にこれらの正規化を行ったクラスであるべきです. では,「**同じでない**」ということはどうでしょうか? 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, 大小関係があり, この関係は有理数にもあります. ここで有理数に対して正規化を行っていれば常に分母は 0 以外の正整数にすることができるために ‘ad > bc’ と ‘(a, b) > (c, d)’ が同値です. これらのことを踏まえて RationalNumber クラスを構築しましょう:

```
from PairOfInts import PairOfInts
class RationalNumber(PairOfInts):
    def __gcd__(self):
        __a = self.numer
        __b = self.denom
        if  __a == 0:
            if  __b != 0:
                __b = 1
        elif __b == 0:
            __a = 1
        elif __a > __b:
            __d = __a / __b
            __r = __a -__d * __b
        else:
            __c = self.conv()
            return c.__gdc__()
```

```
    def __cmp__(self, other):
        u"""
        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):
        u"""
        逆元を返すメソッド
        """
        tmp = self.numer
        if tmp == 0:
            self.numer = 1
            self.denom = 0
        else:
            self.numer = self.denom
            self.denom = tmp

    def rexpr(self):
        u"""
        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
```

最初に import 文でクラス PairOfInts をファイル PairOfInts.py から読込みます．つぎの

'class RationalNumber(PairOfInts):' で用いられた手法を「**継承**」と呼び, 引数で指示されたクラスの属性やメソッドを引継げます. この例では, オブジェクトが整数対で, その初期化メソッドと表示のメソッドが引継がれるために, これらを再び定義することなしに, 追加されるべきメソッドや属性を記載するだけで新たなクラスの定義ができます. ここではクラス RatinalNumber 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, 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して大小関係の二項演算子 ">", "<" と等価 "==" がこのクラスでも使えるようになります[*12]. そして, クラス RationalNumber の基になった PairOfInts クラスを「**基底クラス (base class)**」, 逆に継承する側のクラスを「**派生クラス (derived class)**」と呼びます. また, 継承関係を親子関係にたとえれば基底クラスを「**親クラス**」, 派生クラスを「**子クラス**」, さらに継承関係を上下関係として捉えると基底クラスを「**スーパークラス (super class)**」, 派生クラスを「**サブクラス (subclass)**」と呼びます. クラスは処理の対象が「**何であるか**」ということと,「**どのようなものであるか**」を語るものであり, 対象への理解が深まることでより詳細に分類され, その結果, 下層のサブクラスは上位のクラスよりもより一層, 現実の事物に近いために具象性が増し, 逆に上位のクラス程, 下位のクラスの共通性を引き出すものであるためにより抽象的 (普遍的) になります.

3.2.3 特殊メソッドによる四則演算の導入

このクラス RationalNumber にまだ足りないものがあります. それは和, 差, 積と商の四則演算で, これらの実装は「**特殊メソッド**」の上書きで行います:

```
    def __add__(self, other):
        u"""
        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.rexpr()
        return
```

[*12] ここでの「**等価**」はオブジェクトの持つ値が「**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. ただし, 数オブジェクトのように等価であれば同一であるオブジェクトもあります.

```
    def __sub__(self, other):
        u"""
        有理数の差を定義するメソッド
        """
        numer = self.numer * other.denom - \
                self.denom * other.numer
        denom = self.denom * other.denom
        c = RationalNumber(numer, denom)
        c.rexpr()
        return

    def __mul__(self,other):
        u"""
        有理数の積を定義するメソッド
        """
        numer = self.numer * other.numer
        denom = self.denom * other.denom
        return RationalNumber(numer, denom)

    def __truediv__(self,other):
        u"""
        有理数の商を定義するメソッド
        """
        numer = self.numer * other.denom
        denom = self.denom * other.numer
        return RationalNumber(numer, denom)
```

最初の特殊メソッド__add__() が和演算子 “+”, メソッド __sub__() が差演算子 “-”, それからメソッド__mul__() が積演算子 “*”, 最後のメソッド__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, 被演算子が二つあるために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 に加えて同じクラスのもう一つのインスタンスを変数 other で指示します．ところで, ‘1/2 + 2/3’ の処理ができるようになりましたが ‘1/2 + 2/3 +3/4’ の処理はできるでしょうか? 実際は ‘(1/2 + 2/3) + 3/4’ か ‘1/2 + (2/3 + 3/4)’ とあくまでも二項演算として括らなければダメです．ここで ‘(1/2 + 2/3)

+ 3/4 = 1/2 + (2/3 + 3/4)' を保証して '1/2 + 2/3 + 3/4' と書ける性質が「**結合律を充す**」という「**代数的な性質**」です．Python を処理言語として用いている SageMath には, このような演算の代数的性質があらかじめ定義されており, 類似するクラスの継承で代数的性質を容易に表現できます．だからこそ, Python 言語の習得に意味があります．ここで参考文献の「Python 言語マニュアル」では構文の解説で EBNF と呼ばれる記法を用いており, 次の節ではこの EBNF について解説しましょう．

3.3 バッカス・ナウア記法 (BNF) について

「**バッカス・ナウア記法 (Backus-Naur form, BNF)**」はジョン・バッカス (John Backus) がプログラム言語 ALGOL [*13]の文法の説明で用い, ピーター・ナウア (Peter Naur) が改良した表記法で, 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$_n$

記号 "::=" を挟んで左辺の「**非終端記号**」が右辺の '定義$_{i\in\{1,\ldots,n\}}$' で指示される構文の何れか一つが採用されることを意味します．つまり, 生成規則がひとつであれば右辺は '定義$_1$' だけ, 右辺が '定義$_1$|定義$_2$' であれば '定義$_1$' でなければ '定義$_2$' の書式で左辺の「**非終端記号**」が構成されます．ところで, この「**非終端記号 (Nonterminal Symbol)**」は「**形式文法 (Formal Grammer)**」の言葉で, "::=" の左側に記載された定義に沿って変化する記号で, 変数と同様の動作をするもののために「**構文変数**」とも呼ばれます．また,「**終端記号**」は各種定義と無関係で変化しない, ちょうど定数に対応する記号で, BNF で "::=" の右辺のみに現われます．

BNF の例として数字の構成を挙げておきましょう．まず,「**数字**」は ASCII 文字の "0" から "9" です．これを BNF で表記するなら

BNF による数字の定義

数字 ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"

になります．このときに ASCII 文字の "0" から "9" が終端記号であることに異義はないでしょう．では, 自然数の BNF はどうなるでしょうか? この自然数は複数の非終端記号の定義行を組合せて表現できます:

[*13] ALOGOrithmic Language:1950 年代に開発された Pascal の先祖になる言語．

BNF による自然数の定義

自然数 ::= 数字 | 0 以外の数字 自然数
数字 ::= “0” | 0 以外の数字
0 以外の数字 ::= “1” | “2” | “3” | “4” | “5” | “6” | “7” | “8” | “9”

最初の「**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という演算子 “::=” の左右に「**自然数**」が現われる「**帰納的な定義**」, つまり, 既存の要素と自分自身を使って自分自身を定義する手法です. そして, 第 1 式の右辺に現われる「**数字**」は第 2 式で「**“0”**」または「**0 以外の数字**」から構成されると定義され, 最後に「**0 以外の数字**」は “1”, “2” から “9” までの数字と定義されています. このように「**自然数が何であるか?**」という本質的な問掛けに答えるものではありませんが, 自然数が数字をどのように使って構築されたものであるかを字句的に示す記述であり, これによって自然数であることを判断する上で機械的な処理が可能になったことを意味します.

この BNF は字句的な定義だけではなく構文の定義もできます. たとえば, 論理学の「**命題論理式の定義**」は次のものです:

命題論理式の定義

(1) 真理値の真 $\top$ は命題論理式である.
(2) 真理値の偽 $\bot$ 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. ここでも (5) に帰納的な定義が入っています. 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	::=	$\top$ | $\bot$ | 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, 命題論理式の定義の (5) の帰納的な論理式の定義も BNF の構造で表現されます. このように「**文の成り立ち**」の表記にも BNF は使えます. ところで,「Python 言語マニュアル」の BNF

は, 上述の BNF に対して正規表現を追加した「**拡張 BNF(EBNF)**」を採用しており, 以下に拡張の要旨を纏めておきます:

EBNF の特徴

- 項目のグループ化は丸括弧 “( )” で行います.
- 文字リテラル (後述) は二重引用符 (") で括ります.
- 角括弧 “[ ]” で括られた項目は 0 個か 1 個出現します.
- 順序を持つアルファベットや数字に対して “...” の直前の項目から開始して直後の項目のいずれか一つが現われます.
- 記号 “*” の直前の項目は 1 個以上出現します.
- 記号 “+” の直前の項目は 0 個以上出現します.

ここでの拡張は正規表現で見られる事項のグループ化や出現回数に関する指示とリテラル (=Python の文字列) の処理に関わります. 以下に EBNF の具体例を幾つか示しておきます:

整数の EBNF

```
整数 ::= ["-"] 自然数
```

これは角括弧 “[ ]” を使った例で, 角括弧 “[ ]” はあってもなくてもよい部位を角括弧 “[ ]” で括って表記しています. また, 自然数がその言語体系で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.

最後に「Python 言語リファレンス」の例を示しておきます. なお, ここで定義している name は Python で重要な意味を持つ「**名前 (name)**」ではなく, 単に ‘name’ というものの EBNF です:

name の EBNF

```
name ::= lc_letter(lc_letter | "_")*
lc_letter ::= "a" ... "z"
```

Python の EBNF では二重引用符 (") で "_", "a", "z" と文字を括ります. また, 第 1 式の ‘(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があります. 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? この定義が第 2 式で,

lc_letter が文字 "a" から "z" までの小文字で構成されることを意味します．ただし, 通常の正規表現であれば ‘a-z’ となるところですが, Python の EBNF では記号 “...” 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小文字, 以後はアルファベット小文字や記号 “_” で構成された文字列であり, ‘_a’ のように先頭がアルファベットでない文字列, ‘a1’ や ‘A_v0.1’ のように BNF に記述のないの文字が入った文字列は name に適合しません．このように EBNF を使って具体的にどのような記述がされ得るかを表記できます.

3.4 Python の字句解析について

3.4.1 トークン (token) について

Python のプログラムは Python 処理系の構文解析器によって Zephyr ASDL (Abstract Syntax Description Language) [*14]を使った表現 (ASTs, Abstruct Syntax Trees) で置換えられます．ASDL の書式は CPython のソースファイル Parser/Python.asdl [*15]に記載されており, BNF 以上に簡素化された引数の型宣言を含む函数表現です．そして, ASDL ではトークンの列に変換されます．ここで「**トークン (token)**」はプログラム内で「**意味**」[*16]を持つ最小単位で, 自然言語の「**語彙素**」に相当します．まず,「**NEWINE**」,「**INDENT**」と「**DEDENT**」が行と文の構造に関わるトークンで, NEWLINE が論理行の区切, INDENT と DEDENT が対で §3.6 で説明する「**複合文 (compound statement)**」の字下げに関係します． そして, 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[*17]．また,「**空白文字 (blank character)**」[*18]とよばれる文字の中で「**欧文間隔 (Space)**」,「 **水平タブ (TAB)**」と「**改ページ (FF, Form Feed)**」にはトークンを区切る作用があります．たとえば 'ab' と文字の間に Space を入れた 'a b' では前者が '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.

3.4.2 行構造について

トークン列は「**NEWLINE**」を区切として複数の「**論理行**」に, そして, 論理行は「**物理行**」へと分解されます:

■論理行 (logical line): 入力されたプログラムが分割されたもので, 先頭に「**字下げ/インデント (indentation)**」と呼ばれる ASCII 文字の **Space**, **TAB** による空白文字があり, 列の末端に「**NEWLINE**」があります.

■物理行 (physical line): 論理行を行末端文字で分割された文字の列のことです．通常, 行末端文字は計算機環境で異なり, UNIX 環境では ASCII 文字の **LF(行送り)**, Windows

---

[*14] 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 Ζεφυρος に由来します.

[*15] CPython のソース．https://github.com/python/cpython

[*16] ここでの「意味」は言語の処理系で固有の処理を指示する「機能」と言えるでしょう.

[*17] 演算子と区切文字は Parser/tokenizer.c にそれらの名前が定義されています.

[*18] 空白文字と呼ばれる ASCII 文字には水平タブ (TAB), 垂直タブ (VT), 改行 (LF/NL), 改ページ (FF), 行頭復帰 (CR), 欧文間隔 (Space) の 6 文字あり, Python では文字列クラスのメソッド isspace() で空白文字かどうかが判断できます．なお, 各言語の間隔文字は空白文字と区別します.

環境では **CR+LF**, MacOS では「**CR(復帰)** ですが, Python では C 同様に行末端文字として ASCII 文字 “**LF**” に対応するエスケープシーケン ‘\n’[*19]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[*20]．なお, 注釈は論理行を終らせる働きがあり, 後述する「行継続」で注意が必要です．
- **符号化宣言 (エンコーディング．encoding):**　符号化宣言はプログラムで用いる文字が属する文字コード[*21]を明示的に指示するための宣言です．既定値の文字コードは Python 2 で ‘ascii’, Python 3 では ‘utf_8’ です．PEP-263 にその規定があり, 注釈と同様に文字 “#” から開始して物理行の行末端文字で終える論理行とされ, プログラムの先頭の一行目か二行目に配置され, 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．これらの書式は GNU Emacs 風と vim の符号化設定書式に適合するもので, たとえばプログラムの文字コードが UTF-8 のときに GNU Emacs 風に記述するのであれば

  ```
  # coding = UTF-8
  ```

  となり, この設定を vim 風に行うのであれば

  ```
  # coding : UTF-8
  ```

  となります．この符号化宣言でプログラムで用いられる文字リテラルを構成する文字がどの言語のどの符号化であるかが明示的に指定されますが, これとは別に文字列の先頭に文字コードを示す接頭辞を使うことで文字列単位でも文字コードの指定ができます．
- **空行 (blank line):**　**Space**, **TAB**, **FF** といった空白文字, あるいは注釈だけで構成された論理行です．これらの空行には字句解析で **NEWLINE** が生成されず, プログラム内の内容的な区切以上の意味を持ちません．

---

*19 エスケープシーケンスの詳細は 3.4.4 を参照．なお, MS-Windows で用いられている文字コード: SHIFT_JIS(その実装の CP932) で記号 “\” が記号 “¥” で置換えられ, 多くの書籍でこれらの記号を同一視していますが, UTF-8 等の文字コードで別記号のため, この本では記号 “\” を記号 “¥” で置換えません．したがって, 日本語 MWindows 環境では適宜, 記号 “\” を記号 “¥” で読み直して下さい．

*20 記号 “#” は Python のリテラルに含まれません．

*21 日本語であれば JIS, SHIFT-JIS, EUC, UTF8 等です．

■字下げ/インデント (indentation):　論理行の先頭に配置された **Space**, **TAB** の個数で字下げの水準が計算されて入力文が纏められます．なお, PEP-8 で字下げは一段 4 個の Space のみが推奨されています．実際, 目視で TAB キーと Space の判別は困難で, それらが一貫した順序で並んでいなければなりません．さらに Jupyter のように TAB キーに入力補完機能を持たせたユーザ・インターフェイスもあるために Space のみの利用が実用的です．なお, この字下げの水準に不整合があるときは構文解析器から例外:TabError が送出されます.

■物理行の明示的/非明示的な分割:　注釈と符号化宣言を除く物理行を複数の物理行に置換できます．明示的に物理行に分割するときは行の継続を示す継続文字として記号 “\” を物理行の末尾 (行末端文字の直前) に置きます．なお, Python の構文解析器は継続文字直後の行末文字を削除して一つの物理行に変換しますが, 継続文字 “\” に続けて註釈が追記できません．なぜなら, 注釈で論理行が終わるためで, 注釈自体も継続文字を使って分割ができません．ここで改行の例外的な規則として丸括弧 “( )”, 角括弧 “[ ]”, 波括弧 “{ }” 内部で改行を行う際に継続行文字 “\” の併用を必要としません．同時にこれらの記号で括られたその内部で註釈を続けられます.

3.4.3　識別子とキーワードについて

「**識別子 (identifier)**」は「**名前 (name)**」に用いられ, この名前を介してオブジェクトが保持する値の参照が行われます:

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"

ここで Python 2 の識別子はアルファベット, 数字と記号 “_” のみで構成された ASCII 文字の列で, 日本語の漢字 “**三毛猫**” やいわゆる全角文字 “**ＡＢＣ**” といった ASCII 文字以外の文字は識別子ではありません．しかし, Python 3 では UNICODE 文字の漢字, キリル文字等の非 ASCII 文字も識別子として使えます．詳細は「PEP-3131 Supporting Non-ASCII Identifiers」を参照して下さい．なお, 以下に示す「**キーワード**」は式や文の構成要素であるために 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	

識別子は「**名前**」としてオブジェクトの参照で用いられます．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, 連想配列の「**辞書 (dictionary)**」として実装されています．なお, オブジェクトと対応関係にない名前がオブジェクトへの参照で用いられるときは例外:NameError が送出されます.

3.4.4　リテラルについて

「**リテラル (literal)**」は「**文字どおり**」,「**字義どおり**」を意味し, 記号論理学で命題記号 (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, プログラミングではコード内部で定数値となる文字列や数値といった値の記述を指します:

Python のリテラル

リテラル　::=　文字列リテラル | 数リテラル

この EBNF で示すように, Python のリテラルは「**文字列リテラル**」と「**数リテラル**」の二種類に分類されます．そして, これらは全て「**変更不能なデータ型**」, すなわち, オブジェクトの生成後に値の変更ができないオブジェクトです.

文字列リテラル

文字列リテラルは言語の文字列に相当するもので, 通常の文字や記号に加えてエスケープシーケンスと呼ばれるさまざまな機能を持った文字を加えて拡張したものです．この文字リテラルの EBNF を以下に示しておきます:

文字列リテラルの EBNF

```
文字列リテラル          ::=  [接頭辞](短文字列 | 長文字列)
接頭辞                  ::=  "r" | "u" | "ur" | "R" | "U" | "Ur" | "uR" | "b" |
                             "B" | "br" | "Br" | "bR" | "BR"
短文字列                ::=  " ’ " 短文字列本文* " ’ " |
                             " " " 短文字列本文* " " "
長文字列                ::=  " ’’’ " 長文字列本文* " ’’’ " |
                             " """ " 長文字列本文*" """ "
短文字列本文            ::=  短文字 | エスケープシーケンス
長文字列本文            ::=  長文字 | エスケープシーケンス
短文字                  ::=  記号 “\”, 改行や引用符を除く文字
長文字                  ::=  記号 “\” を除く文字
エスケープシーケンス    ::=  "\" 任意の ASCII 文字
```

文字リテラルには文字の文字コードを示す接頭辞が配置できます．たとえば, 文字列の先頭に “u” や “U’ といった接頭辞があれば, その文字列リテラルは UNICODE 文字列型として扱われます．このときに接頭辞と文字列リテラルの間に空白文字を入れてはいけません．もし, プログラムに符号化宣言が包含されず, リテラルに文字コードを示す接頭辞がなければ, リテラルを構成する文字は UNICODE 文字として解釈されます.

Python の文字列は引用符対で括られたオブジェクトですが, Python の引用符には単引用符 (’) と二重引用符 (") の二種類の引用符が存在するために "abc ’def’ hij" のように文字列リテラル内部に別の引用符が配置できます．さらに文字の列を引用符で一重で括るか, 三重に括るかで文字列の型が異なります．ちなみに Julia では単引用符 (’) で括った一つの文字は Char 型, 二重引用符 (") で括った一つの文字は String 型と区別されていますが, Python にはそのような型の区分がありません．そのために他の言語への移植で Python の文字列を何で括っているかを確認し, 移植先の言語でそのリテラルが何の型に対応するか注意していなければなりません．ここで,「**短文字列 (shortstring)**」は一重の引用符で括られた文字リテラルの型,「 **長文字列 (longstring)**」は三重の引用符で括られた文字リテラルの型です．具体的には「**’ 三毛猫’**」と「**"虎猫"**」が短文字列,「**’’’ 三毛猫’’’**」と「**"""虎猫"""**」が長文字列です．なお, 長文字列からは後述の「**文書文字列 (docstring)** 」が構成されますが, こちらはオンラインマニュアルとしての性格を持ち, 文書文字列の中で改行や単引用符 (’) や二重引用符 (") が入れられるために例題等を含む長い文書の記述ができます.

```
"""
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, 単引用符や二重引用符, それと行末端文字を含む文字リテラルを一つの長文字列に包含できます. また, 文字出力で改行を入れたいときは C 同様に出力文字列に対応する文字列リテラルに改行を入れたい箇所に文字列 "\n"(正確にはエスケープシーケンスの改行 (LF)) を挿入します. この長文字列は MATLAB の m-file で函数のヘルプや例題の記述で用いられる文書文字列 (docstring) を拡張・強化したもので, 長文字列を reSTructuredText や Markdown で記述すれば文書としての表現力が格段に向上します[*22].

次に,「**エスケープシーケンス (Escape sqeuence)**」は計算機の何らかの機能を表現するための文字の列であり, 多くの計算機環境でほぼ同じ文字列が用いられています:

[*22] Jupyter notebook では Markdown で記述した文書のレンダリングが可能です.

エスケープシーケンスの一覧

\\	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
\uxxxx	16 ビットの 16 進数値 xxxx を持つ Unicode 文字
\uxxxxxxxx	32 ビットの 16 進数値 xxxxxxxx を持つ Unicode 文字

この一覧の中で,「**UNICODE 文字名**」は「**Unicode Character Database**」[*23]に ASCII 文字で記載された Unicode 文字の名前を指示します. たとえば, 文字 “[” の UNICODE 符号点は 005B であり, UNICODE 文字名は ‘LEFT SQUARE BRACKET’ と名付けられています. ただし, 接頭辞をエスケープシーケンスに付けたときは単なる文字列として処理されます. たとえば, 接頭辞 ’r’, ’R’ に続けて “\” と記載しても, エスケープシーケンスとしてではなく, 通常の ASCII 文字として処理されます.

数リテラル

数リテラルの EBNF を以下に示します:

数リテラルの EBNF

数リテラル ::= 整数リテラル |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**」,「**浮動小数点数リテラル**」と「**虚数リテラル**」の 3 種類があります. なお, Python 2 の整数リテラルには整数リテラルに加えて「**長整数 (long integer) リテラル**」, があります. これは Python 2 までは整数に 64 ビット長の整数型と任意精度の整数の長整数型の二種類がありましたが, Python

*23 http://www.unicode.org/Public/UCD/latest/charts/CodeCharts.pdf を参照.

3 で整数型と長整数型が長整数型に統合され, そのリテラルは整数リテラルです. この統合に伴い, Python 2 の大域変数 sys.maxint に符号付き 32bit 整数の上限 214743647 が束縛されていましたが, Python 3 では廃止されています. なお, SageMath では独自の整数が用いられ, そのリテラルは整数リテラルに対応します. ちなみに長整数型は計算機の記憶容量内で表現可能な桁数の整数が扱え, 記憶容量の上限に到達して処理ができなくなると例外:MemoryError が送出されます. つぎに浮動小数点数リテラルは浮動小数点数は IEEE 754 で規定される実数の表現で, 近似値であることに注意が必要です. ところで, 数学で重要な数に複素数がありますが, 複素数は浮動小数点数リテラルと虚数リテラルの「**式**」として表現されるため, 何らかの型を持つ数ではありません.

つぎに整数リテラルの EBNF を示します. Python 3 では整数リテラルのみですが, SageMath では Python 2 を長く使っていたため, 長整数リテラルの BNF も併記しておきます:

整数リテラルの EBNF

整数リテラル	::=	10 進表示 | 8 進数示 | 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"
長整数リテラル	::=	整数リテラル ("l"|"L")

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, Python 3 で廃止された長整数のリテラルは末尾に “l” や “L’ といった文字を配置します. ただし, 小文字 “l” 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した.

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 +

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整数 '-1' との積演算の結果, すなわち式であるためです. 多くの言語では単精度と倍精度の二つの浮動小数点数の型を持ちますが, Python では倍精度浮動小数点数の float 型のみです[*24]. また, 特殊な数の表現に「**無限大 ('inf')**」と「**NaN('nan')**」があります[*25]. 無限大であれば float('inf'), NaN であれば float('NaN'), あるいは float('nan') でこれらの float 型のオブジェクトが生成できます.

最後に虚数リテラルの EBNF を示しておきます:

虚数リテラルの EBNF

虚数リテラル	::=	(浮動小数点数リテラル | 10 進表示) ("j" | "J")

虚数リテラルは「**浮動小数点数から構成されたリテラル**」で, このことが虚数リテラルの性格を決定付けます.

3.4.5 演算子と区切文字について

以下のトークンは Python 組込の二項演算子として用いられます:

Python 組込の演算子

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

ここで算術演算子は数リテラルに対して数リテラル, 論理演算子は Boolean に対して Boolean をその演算結果として返します. なお, 演算子 "/" は C と同様に割り切れれば整数, そうでなければ浮動小数点数を返却します. なお, 以前のバージョンの Python 2 で "/" は双方の引数が整数であれば整数を返すため, '3/2' の結果は整数の '1' のなるために注意が必要です. また, 演算子 "<>" と演算子 "!=" は同じ値でないことを判断するための演算子ですが, 演算子 "!=" の利用が推奨されています. そして, 演算子 "&" と演算子

[*24] 数の精度の拡張はパッケージ NumPy で行えます.
[*25] inf, NaN は IEEE 754 で規定されており, Python の NaN には qNaN が対応します.

“|” は Boolean に対しては論理積 “and”, 論理和 “or” と同じ意味ですが, 引数が整数型, あるいは長整数型のときに演算子 “&” と演算子 “|” は 2 進数で表現したときの各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の結果, そうでなければ演算子 “and” は右被演算子, 演算子 “or” は左被演算子を結果として返します. さらに演算子 “^” は排他的論理和 (XOR) の演算子です. さらに文字列に対して演算子 “+” は二つの文字列を結合して一つの文字列を生成する演算子です. このようにオブジェクトで演算子の意味が異なることもあり注意が必要です.

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)

<table>
<tr><td>括弧:</td><td>(</td><td>)</td><td>[</td><td>]</td><td>{</td><td>}</td><td>@</td></tr>
<tr><td>区切記号:</td><td>,</td><td>:</td><td>.</td><td>‘</td><td>=</td><td>;</td><td></td></tr>
<tr><td>累算算術演算記号 (1):</td><td>+=</td><td>-=</td><td>*=</td><td>/=</td><td>//=</td><td>%=</td><td></td></tr>
<tr><td>累算算術演算記号 (2):</td><td>&=</td><td>|=</td><td>^=</td><td>»=</td><td>«=</td><td>**=</td><td></td></tr>
</table>

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列の添字処理で使われます. そして, 下段の累算算術演算子は区切文字としても振舞います. また, 単引用符, 二重引用符, 記号 “#” と記号 “\” といった ASCII 文字は他のトークンの一部に用いられて特殊な意味を持ちます. 最後に “$” と “?” は Python では用いられず, 文字列リテラルと注釈以外ではエラーになりますが, IPython や Jupyter を Python のシェルとして使うときは記号 “?” に函数 help() を拡張した演算子としての働きを持たせているためにエラーになりません[*26].

[*26] “??” でソースファイルを表示させる機能も追加されています.

3.5 Python の式と文

3.5.1 式, 単純文と複合文について

「**式 (expression)**」は, 言語で何らかの機能や意味を持つ最小の単位です. たとえば, '1', '2', '3' のような数値, 'True', 'False' のような真理値, '1 + 2', 'math.sin(10)' のような数式, '(1, 2, 3)' や '['a', 'b', 'c']' のようなタプル型やリスト型, '{1, 2, 3} ' のような集合型, さらには文字列型等, 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)**」には手続が記載され, その構造から「**単純文 (simple statement)**」と「**複合文 (compound statement)**」に分類できます.

■**単純文**: 「**一つの論理行に収めることのできる文**」で, 例としては名前へのオブジェクトの参照関係を設定する名付け文や代入文, モジュールの読込を行う import 文が挙げられます.

■**複合文**: 「**複数の論理行を必要とする文**」で, 複数の文節とその文節に対応する文の区画 (ブロック) を持ちます. 条件分岐や反復処理, 函数やクラスの定義は複合文で, その先頭の文節の末尾に文字 ":" が置かれて字下げで可視化された階層構造を持つこともあります.

3.5.2 式の EBNF

最初に「**式**」と「**式のリスト**」の EBNF を示します:

式 (expression) の EBNF

式のリスト	::=	式 ("," 式)* [","]
式	::=	一次語 | 演算式 | if 式 | lambda 式

■**式のリスト**: 区切文字がカンマ "," の式の列が式のリストの書式です. 成分が 2 個以上の式のリストからはタプル型のオブジェクトが生成されます:

```
>>> True,True
(True, True)
>>> True,True,
(True, True)
>>> a=True,True,
>>> a
(True, True)
```

```
>>> type(a)
<type 'tuple'>
```

ここで示すように成分数が 1 以上のタプル型のオブジェクトの生成で丸括弧 “( )” は必須ではありません．この括弧 “( )” が必要になるのは空のタプルとタプルをひとまとめにして全体を処理するときです．また, 成分が一つのタプルはカンマを使って, たとえば ‘1,’ や ‘(1,)’ のように生成します[*27]．なお, 計算機言語の多くではリストは ‘(1, 2, 3)’ や ‘[1, 2, 3]’ のような括弧で囲われたリテラルの列ですが, ここでのリストはリテラルとしての列です．Python のリスト型のオブジェクトについては後述します．

■式 (expression): 処理系で評価した結果を返却するオブジェクトが式です．Python の式は EBNF に示すように「**一次語**」,「**演算式**」,「**if 式**」と「**lambda 式**」に大きくまとめられます．ここで一次語が最も原始的な式で, 演算式は算術演算式や論理演算式, および比較演算式があります．そして, if 式は if 節を使った処理の切替を伴い, lambda 式は無名函数の構築で用いられます．

3.5.3 一次語 (primary)

名前, 属性参照, 添字表記, 呼出等と基本になる表記です:

一次語の EBNF

一次語 ::= 原子要素 | 属性参照 | 添字表記 | スライス表記 | 呼出

一次語は Python の式の基本単位の原子要素, オブジェクトの名前, その名前から参照されるオブジェクトの属性への参照, またはオブジェクトが配列や辞書であればそれらの添字に対する値の参照で用いられる表記です．

原子要素 (atom)

Python の式を構成する基本単位で, 単体で意味や機能を持ち得ます:

原子要素 (atom)

原子要素	::=	シングルトン ｜ 識別子 ｜ リテラル ｜ 閉包
シングルトン	::=	True ｜ False ｜ None
閉包	::=	丸括弧形式 ｜ リスト表現 ｜ 集合表現 ｜ 辞書表現
		｜文字列変換 ｜ 生成式 ｜ 産出原子式

[*27] ‘(1)’ のように数値や文字列をカンマなしで括弧 “( )” で括ったときはグループ分けの演算子と判断されてタプル型のオブジェクトになりません．

「**原子要素**」は「**シングルトン (singleton)**」[*28],「**識別子 (identifier)**」,「**リテラル (literal)**」と「**閉包 (enclosure)**」の四種類に分類されます．まず, シングルトンは他と違って固有の意味 (値) と型を持ちます．識別子は「**名前 (Name)**」に用いられる文字に制限の入ったリテラルです．リテラルには「**数リテラル**」と「**文字列リテラル**」の二種類があり, 具体的なデータを構成します．これらシングルトン, 識別子とリテラルが最も基本的な原子要素です．つぎに「**閉包**」には「**丸括弧式**」,「**リスト表現**」,「**辞書表現**」,「**集合表現**」,「**文字列変換**」,「**生成式**」と「**産出原子式**」の七種類に分類されます．

■シングルトン (singleton): 識別子やリテラルと違って固有の型と意味を持ちます．Python の真理値集合 Ω の元には, ‘1 == 1’ と同じ意味であることを指示するオブジェクト True, ‘1 == 0’ と同じ意味であることを指示するオブジェクト False, さらに何もないことを指示するオブジェクト None の 3 つがあり, これらは型を持ちます．まず, 真理値集合 Ω の元の True と False の型は Boolean で, オブジェクト None の型は None 型です．また, オブジェクト True には整数の 1, オブジェクト False には整数の 0 が整数値として付与されていますが, None には何らの整数値も付与されていません．

■識別子 (identifier): 「**名前 (Name)**」に用いられ, その名前の参照で名前に束縛されたオブジェクトの値が返却され, 名前にオブジェクトが束縛されていないときは例外:NameError が送出されます．また, クラスの属性名としての識別子で, その先頭に二つ以上の記号 “_” があり, 末尾に二つ以上の記号 “_” がないものは隠蔽されるべき名前とみなされ, 名前空間内で別の名前に変換されます．「言語リファレンス」の例では, クラス名を Hom, 属性名を ‘___spam’ のときに名前は ‘_Hom___spam’ に変換されて ‘___spam’ で参照できないという方法で属性の隠蔽が行われます．

■リテラル (literal): 文字列リテラルと数リテラルから構成されます:

リテラルの EBNF

リテラル	::=	文字列リテラル | 整数リテラル | 浮動小数点数リテラル | 虚数リテラル

前述のように Python 3 で長整数リテラルは廃止されていますが, Python 2 の場合は, ここでの整数リテラルに長整数リテラルも含まれていると読んで下さい．

■閉包 (enclosure): 識別子やリテラルと違って一定の書式を持つ原子要素です．また, 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, 区切記号を “,”

[*28] 「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．

とするオブジェクトの列を囲んで得られます. なお, これらの書式が外延表現であるためにオブジェクトとして生成した時点で成分も定まり, 生成式と産出式から生成されるオブジェクトの成分はメソッド next() の呼出で都度, 生成されます. なお, 各閉包の生成で用いられる内包表現の詳細は §3.5.3 を参照してください.

■**丸括弧形式**: タプルの構成で用いられる書式です:

丸括弧形式の EBNF

```
丸括弧形式 ::= "(" [式のリスト] ")"
```

括弧 "( )" がタプル型のオブジェクトの構築で必須なときは空のタプル '( )' と一成分のタプル '(1)' 程度で, 多くの場合は不要です. たとえば '1,2,3' と '(1, 2, 3)' は同じタプルを生成します. また, タプルは後述のリスト型に類似していますが, リスト型と違ってその値が変更不可能なオブジェクトで, その生成もリスト型で使える内包表現が使えません. 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, ジェネレータ型のオブジェクトが生成されます.

■**リスト表現**: リスト型のオブジェクトを生成する表現で, タプルに似ていますがその生成で角括弧 "[ ]" が必須です:

リスト表現の EBNF

```
リスト表現 ::= "[" [式のリスト | 内包表現] "]"
```

リスト表現は「**式のリスト**」か「**内包表現**」を角括弧 "[ ]" で括った書式に分類され, 前者の式のリストが具体的な成分の表記, 内包表現では for 節が用いられます. この内包表現については §3.5.3 でその詳細を述べます. リスト表現は集合と異なり空のリスト '[ ]' を許容し, タプルと違ってオブジェクトの生成後に成分の挿入, 削除や入替が可能です. このリスト表現はパッケージ NumPy の配列の生成等, 幅広く用いられています.

■**集合表現**: 集合型のオブジェクトの生成で用いられ, 波括弧 "{ }" が必須です:

集合表現の EBNF

```
集合表現 ::= "{" (式のリスト | 内包表現) "}"
```

式の評価は式のリストの左側から行われます. なお, 空の式のリストや空集合 $\emptyset$ を返す内包表現は集合表現では扱えません. 実際, リテラル '{ }' は集合型ではなく辞書型のオブジェクトで, 空集合 $\emptyset$ は空のタプル '( )', あるいは空のリスト '[ ]' で表現します. なお, 集合型は for 文でリストやタプルと同様に利用できます.

■**辞書表現**: 辞書型のオブジェクトの生成で用いられ, 波括弧 "{ }" が必須です:

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

辞書表現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ていますが, 辞書表現の成分を指示する添字はここでの EBNF で鍵データ, つまり, 鍵 (キー) と呼ばれ, 一般的な式が鍵データになります．ここで辞書表現ではキーはカンマで区切られた式の左側の文字列になります．ここで簡単な例を示しておきましょう:

```
>>> a = {"Niko":1.95, "Mike":4.5}
>>> a["Niko"]
1.95
```

このように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が, リスト型では指示する成分の位置を, その列としての並び順に対応する自然数で指示するのに対して, 一般的には列の位置と無関係な式で成分を指示する点で異なります．なお, データ解析で広く用いられるパッケージ Pandas のデータフレーム (DataFrame) 型は辞書表現から定義されます．

■**文字列変換**: 式のリストから得られるタプル全体を評価し, 文字列型に変換します:

文字列変換の EBNF

文字列変換	::=	"‘" 式のリスト "‘"

ここで「**式のリスト**」として Space や TAB といった「**空白文字**」だけで構成されるものは許容されません:

```
>>> a =
  File "<stdin>", line 1
    a =
      ^
SyntaxError: invalid syntax
>>> 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する式で, タプルのように丸括弧 “( )” を用いますが, この括弧 “( )” はタプルと異なり必須です:

生成式の EBNF

```
生成式 ::= "(" 内包表現 ")"
```

リスト型の内包表現では内包表現をすべてを評価してオブジェクトが生成されますが, 生成式からはジェネレータ型のオブジェクトが生成され, 函数 next() やメソッド __next__() の呼出の都度, 評価され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))
>>> next(a)
1
>>> a.__next__()
4
>>> a.__next__()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>> a = (i for i in range(0,5) if (lambda x:math.sin(2*x)>0)(i))
>>> for i in a:
...     print(i)
...
1
4
```

なお, この生成型ではリスト型のように添字で成分を取り出せないために注意が必要ですが, 生成子 list() を使ってリスト型に変換できます:

```
>>> a = (i for i in range(0,5) if (lambda x:math.sin(2*x)>0)(i))
>>> list(a)
[1, 4]
```

■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え

ません. ジェネレータ型のオブジェクトは函数 next() やメソッド__next__() を呼出す度に yield 文の式が評価されて値が返却されます:

```
>>> def neko(x):
...     for i in range(4):
...         yield x**i
...
>>> a = neko(2)
>>> type(a)
<type 'generator'>
>>> a.__next__()
1
>>> a.__next__()
2
>>> a.__next__()
4
>>> a.__next__()
8
>>> a.__next__()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

■**map/filter オブジェクト**: Python 2 の函数 map() と filter() は函数をリスト/タプルに作用させてリスト/タプルを返却する函数でしたが, Python 3 の扱いは生成式に類似した map オブジェクトと filer オブジェクトを返却する函数になります. これらのオブジェクトも函数 next() やメソッド__next__() で内部の式を評価した結果を返します:

```
>>> b=map(lambda x:x**2,(1,2,3,4))
>>> type(b)
<class 'map'>
>>> b.__next__()
1
>>> b.__next__()
4
>>> b.__next__()
9
>>> b.__next__()
16
>>> b=map(lambda x:x**2,(1,2,3,4))
>>> c=list(b)
>>> c
[1, 4, 9, 16]
>>> d=filter(lambda x: x % 2, range(10))
```

```
>>> type(d)
<class 'filter'>
>>> next(d)
1
>>> next(d)
3
>>> next(d)
5
>>> next(d)
7
>>> next(d)
9
>>> list(filter(lambda x: x % 2, range(10)))
[1, 3, 5, 7, 9]
```

このように Python 3 では生成式に似たオブジェクトになり, リストやタプルで計算結果をそのまま持たせる Python 2 よりもメモリ消費の面で改善が行われています.

Python の内包表現

集合の内包表現は, その集合の成分を列記せずに各成分が充すべき命題を記載して集合を定める方法です. たとえば, 集合 S を定義するときに '$\{0,1,2,3,4,5\}$' で成分を列記する方法, '$\{x \mid x \in \mathbf{N} \wedge x \leq 5\}$' のように論理式 '$x \in \mathbf{N} \wedge x \leq 5$' を使って集合 S の元が充たすべき性質を記載する方法の二つの集合の成分の表現方法があり, 前者が外延による集合の表現, 後者が内包による集合の表現です. Python の内包表現では for 節で生成した対象を if 節で篩にかけて選別する表記もできます. この内包表現の 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" 反復式
if 節	::=	"if" 論理式 [else 反復式]
反復式	::=	生成式 |内包表現

「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格納されるオブジェクトを式で利用し, '1 for x in range(0,10)' のように標的リストを定数のリストとする表記も可能です. for 節で標的リストに代入可能なオブジェクトを提供する式が生成式で, 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__() やメソッド__getitem__() を持つクラスのインスタンスに限定されます. ここで '[i for j in k for i in j]' のような for 節の入れ子も可能です. この場合は左から順に選択する書式になっています:

```
>>> a = [[1, 2, 3], [4, 5, 6, 7]]
>>> [i for j in a for i in j]
[1, 2, 3, 4, 5, 6, 7]
```

この例では二成分のリストのリストから平坦なリストを生成します. 最初にリストの成分を左側の for 節で取り出し, その成分を構成する元を右側の for 節で取り出して対処しています. つまり, このような入れ子の構造を持つときの表現は, for 文で表現したときの for ブロックの外側から内側への関係に対応するように左側から順番に for 節を記載します. 要するに左側がより普遍であり, 右側に行くに従って成分に近くなる処理が記載されます.

また, 内包表現で if 節の配置には二通りあり, 一つは for 節のうしろに配置して for 節で生成した成分を篩にかける方法, もう一つが if 節を前方に配置する方法です. 前方に if 節を配置するときは else 節を有し, この else 節のうしろに for 節を置きます:

```
>>> [i for i in [1,2,3,4]]
[1, 2, 3, 4]
>>> [i for i in [1,2,3,4] if i%2==0]
[2, 4]
>>> [i if i%2==0 else -i for i in [1,2,3,4]]
[-1, 2, -3, 4]
```

これらの例は最初のものが if 節を持たない内包表現の例, 次が if 節を篩として持つ内包表現の例, 最後が if 節と else 節を持つ内包表現の例で, 篩として持つときは if 節を for 節のうしろに, if-else 節を持つときは逆に for 節をうしろに配置します. この内包表現を用いたオブジェクトの生成は, for 文を使って既存のオブジェクトから取り出す処理よりは多少は高速です. ただし, NumPy の ndarray 型の数値配列のように均質的なデータであればスライス表記のように添字を上手く利用してブロック単位で切り出す手法がより高速です. このことはデータが大きくなればなるほど顕著になります.

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, そのオブジェクトの属性を指示します. なお, 参照可能, すなわち, スコープ内にある属性は函数 dir() で調べられます.

添字表記

オブジェクトが文字列リテラル, タプルやリスト等の列や辞書のときに成分の参照で用いられます:

添字表記の EBNF

添字表記 ::= 一次語 "[" 式のリスト "]"

MATLAB 系言語と異なり丸括弧 '( )' ではなく鍵括弧 '[ ]' を使うことに注意が必要ですが, 逆に言えば, 引数の列を丸括弧で括る函数値と行列の添字による成分指定を混同せずに済みます. オブジェクトが列のときは「式のリスト」が整数値に限定され, 列が 1 次元であれば 0 が列の先頭 (左端) で, 列の長さが n であれば 'n-1' , あるいは '-1' が列の最後尾 (右端) の成分を指示します:

```
>>> a = [1, 2, 3, 4, 5]
>>> a[0], a[1]
(1, 2)
>>> a[4], a[-1], a[-2]
(5, 5, 4)
```

この例では長さが 5 のリスト型のオブジェクトの成分取出を行っています. まず, 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, 次に特殊な表記として添字が-1 でリストの末端の成分の指示になります. このように添字が負の数に対しては最後部からの成分の指示になり, 列の長さが判らなくても最後尾から数えたリストの成分を取り出せます. もう一つの特殊な表記として後述のスライス表記を使った列の成分取出しができます. なお, 成分の取出で, オブジェクトが辞書のときには「式のリスト」は辞書で用いられている鍵に限定され, 集合のときには成分の取出しが添字表記で行えません.

スライス表記

MATLAB 系言語で行列や配列の添字操作を, その表記方法も含めてタプル, リストや列で実現する表記です:

スライス表記の EBNF

スライス表記	::=	単純スライス表記 | 拡張スライス表記
単純スライス表記	::=	一次語 "[" 短スライス表記 "]"
拡張スライス表記	::=	一次語 "[" スライスリスト "]"
スライスリスト	::=	スライス項目 ("," スライス項目)* [","]
スライス項目	::=	式 | スライス本体 | 省略符号
スライス本体	::=	短スライス表記 | 長スライス表記
短スライス表記	::=	[下限] ":" [上限]
長スライス表記	::=	短スライス表記 ":" [刻幅]
上限	::=	式
下限	::=	式
刻幅	::=	式
省略符号	::=	"..."

スライス表記は MATLAB 系言語の配列処理と同様の表記で, 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 から開始します．そして,「上限」も MATLAB と異なり, その上限を越えないこと, つまり, Python で '[2:4]' は [2, 3]' で MATLAB の '[2:4]' は '[2, 3, 4]' であることに注意が必要です．また, Python の記号 ":" も最初と最後と刻幅が自明なときに添字の省略で, MATLAB 系言語で添字の省略記号 ":" と同じ操作になりますが, 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．それだけでなく, 数値配列の演算は Python のリスト型では行えません．したがって, Python で本格的に多次元配列や数値行列を扱うためには標準のリスト型では不十分で, パッケージ NumPy で定義される多次元配列 (ndarray) や Pandas のデータフレームを用いるべきです．NumPy に関しては §5.5 で詳細を述べますが, NumPy の多次元配列 (ndarray 型) の添字処理は MATLAB と同程度かそれ以上の処理が可能です．また, 省略記号 (Ellipsis)"..." 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)**」に近い機能を持ちます:

```
>>> 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ています. この乱数配列に対して省略記号 (Ellipsis) で省略された箇所の次元が保たれています.

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**」,「**キーワード引数**」と「**式**」があり,「**定位引数**」は通常の函数やメソッドの引数が対応し,「**キーワード引数**」はオプションの引数, つまり, 属性等の既定値の変更に用います.

3.5.4　演算式

Python の演算式は以下で分類されます:

演算式の EBNF

演算式	::=	単項演算式 | 二項演算式 | 論理演算式 | 比較演算式

ここで「**単項演算式**」は被演算子が一つだけの演算子から構成される演算式,「**二項演算式**」はその値が真理値以外の値の二項演算子で構成される式,「**論理演算式**」は被演算子と演算結果が真理値になる演算式, そして,「**比較演算式**」は演算子として比較演算子を用いた式です. なお, ここでの EBNF では構文の構成を記述するもので, 演算子が必要とする型の記載は含まれていません. そのため詳細は各演算子の項目を参照してください.

単項演算式

式 -1 や $+1$ の演算子 “+” や “−” のように演算子のうしろに一つだけ被演算子を取る前置演算子で構成される式です:

単項演算式の EBNF

```
単項演算式   ::=  負符合式 | 正符合式 | ビット反転式
負符合式     ::=  "-"  ( 一次語 | "(" 演算式 ")" )
正符合式     ::=  "+"  ( 一次語 | "(" 演算式 ")" )
ビット反転式 ::=  "~"  ( 一次語 | "(" 演算式 ")" )
```

単項演算式には算術演算子の加法 “+” や除法 “-” に加え, 2 進数のビット反転演算子 “~” 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. なお, Python では階乗 4! のような引数のうしろに演算子を配置する表記 (後置表現) がありません.

二項演算式

二つの被演算子を左右に配置する中値表現と呼ばれる書式の演算式で, その値が真理値以外のものです:

二項演算式の EBNF

```
二項演算式   ::=  乗法演算式 | 加法演算式 | 冪演算式 | 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 ")" )
冪演算式     ::=  ( 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になります．また, Python 2 で演算子 “/” は被演算子が整数のときは結果も整数で, 演算子 “//” と同じ結果ですが, Python 3 で演算子 “/” を使った式は C と同様に浮動小数点数になります．このように Python 2 で演算子 “/” を使うときは被演算子のどちらかが浮動小数点数でなければ浮動小数点数の結果が得られません．したがって, Python 2 では演算子 “/” を用いる前に構築子 float() による型変換が必要な場合があります.

■加法的演算式:　和演算子 “+” と減算演算子 “-” を持つ式です．このときの被演算子は数オブジェクトか Boolean ですが, 文字列やリストの結合処理として演算子 “+” が利用可能です.

■幂演算式:　Python で冪乗は演算子 “**” を用います．ただし, 多くの数式処理言語や MATLAB 系言語では演算子 “^” を冪乗の演算子として用い, これは SageMath の数学的オブジェクトに対しても同様です．ただし, SageMath で冪演算子が完全に演算子 “^” で置き換えられていないために SageMath の数学的オブジェクトと Python のオブジェクトで使い分ける必要があります．なお, Python で演算子 “^” は後述のビット演算子です.

■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, 2 進数で表示に対して処理を行う演算子です．演算子 "&", "|" と "^" は 2 進数表示した被演算子に対してビット単位で論理積 (AND), 論理和 (OR) と排他的論理和 (XOR) を行います．まず論理積は双方が 1 のときのみが 1 で他が 0, 論理和は双方が 0 のときだけが 0 で他が 1, 排他的論理和はどちらか一方が 1 のときだけが 1 で, それ以外は 0 を返す演算子です．また, 演算子 “<<” と “>>” は被演算子を 2 進数で表現したときにビットを左右に桁を移動させる演算で, 通常の算術演算子よりも優先順位が低くなっています．右被演算子が sys.maxsize[*29]を越えたときは例外 OverflowError を送出し, 負の数のときは例外 ValueError を送出します.

論理演算式

Boolean をその意味として持つ式を論理和, 論理積, 否定で処理する式です:

[*29] Py_ssize_t 型が取り得る最大値を示す整数で, 32bit 環境であれば $2^{32}-1$, 64bit 環境であれば $2^{64}-1$ です.

論理演算式の EBNF

```
論理演算式   ::= 一次語 | 論理和検証式 | 論理積検証式 | 否定式
                | 帰属検証式 | 比較演算式
論理和検証式 ::= ( 一次語 | 論理演算式 ) "or"
                ( 一次語 | 論理演算式 )
論理積検証式 ::= ( 一次語 | 論理演算式 ) "and"
                ( 一次語 | 論理演算式 )
否定式       ::= "not" 論理演算式
帰属検証式   ::= 式 "in" 式
比較演算式   ::= 式 ( "<" | ">" | ">=" | "<="
                | "is" | "==" | "<>" | "!=" ) 式
```

これらの演算式によって Boolean 値が返却されます. なお, Boolean の True は整数の 1, False は整数の 0 と等価で, このことを利用した算術演算で if 文の代用になります. この手法は MATLAB 系の言語でも同様で, 処理言語のオーバーヘッドを低減する手法としてきわめて有効です. また, 帰属検証式は Python の内包表現の for 節でも用いられています. この帰属検証式の右辺が外延を表現するジェネレータやイテレータ, あるいは内包表現そのものになります.

■帰属検証式: 左辺の被演算子としての式が右辺の被演算子としての式の成分であることを検証する式です. 基本的に右辺の式は閉包になります:

```
>>> 1 in (1, 2, 3)
True
>>> 1 in [2*i-1 for i in range(4)]
True
>>> 'neko' in 'mikeneko'
True
>>> import numpy as np
>>> 1 in np.arange(10)
True
```

帰属検証式式ではタプルやリストといったデータを包含するものであれば使えます. そのためタプルやリスト, 集合やハッシュ表以外の文字列や配列 (NumPy の ndarray) に対しても使えます.

■比較演算式: C の比較演算子と異なる優先順位を持ち, ‘a > b > c’ のような複合的な表記が可能です. 比較演算子による結果は True か False の Boolean であり, 比較演算子には通常の大小関係の演算子と同値性を示す演算子 “==”, 合同性を示す演算子としての演算子 “is” があります. 比較演算式では幾らでも繋げられます. C や Fortran の比較

の演算式は厳密に二項演算子で, 3 以上のアリティ持つ演算子ではありませんが, Python の比較演算子は 3 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[30]. 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. たとえば, 比較演算の式が 'a_1 op_1 a_2 op_2 ... a_{n-1} op_n a_n' であれば '(a_1 op_1 a_2) and (a_2 op_2 a_3) and ... and (a_{n-1} op_n a_n)' で置き換えた式と同じ結果になります.

if 式

if 節による条件分岐を含む式ですが, アリティが 3 の演算子としての性格も持ちます:

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
```

この式は一つのオブジェクトを返す内包表現の一つでもあります. つまり, 指示されるオブジェクトが if 節で語られるためです.

lambda-式

Python の lambda 式は無名函数, いわゆるチャーチの λ-式を構成します:

lambda-式の EBNF

lambda-式	::=	"lambda" [パラメータリスト]: 式
パラメータリスト	::=	"(" 識別子 ("," パラメータリスト) ")"

函数定義の def 文, クラス定義の class 文のように記号 ":" のうしろに改行を入れてはなりません.「式」を括弧 "( )" を使ってグループ化していれば改行を入れられますが, 通常は一行の式を記述します:

```
>>> a = lambda(x):( x**2+
... x*2 +
... 1)
>>> a(1)
4
```

[30] LISP の S 式として表記すると, 一つの S 式で表現できるという意味です.

await 式

Python 3.5 から導入された式で, Python 2 にはありません. asyncio モジュールを使った非同期 I/O 処理で利用されます.

3.5.5　単純文

「**単純文 (simple statement)**」は, その文全体を一つの論理行内に収められる文で, 複数の単純文をセミコロン “;” で区切って続けられます:

単純文の EBNF

```
単純文  ::=    式文 | 代入文 | 累積代入文
            | assert 文 | pass 文 | del 文 | print 文
            | return 文 | yield 文 | raise 文
            | break 文 | continue 文
            | import 文 | global 文 | exec 文 | nonlocal 文
```

なお, print 文と exec 文は Python 3 ではそれぞれ函数 print() と函数 exec() で置き換えられて廃止されています. また, nonlocal 文は Python 3 で導入された文です.

■式文:　式文は意味のある値を返さない函数で用いられます:

式文の EBNF

```
式文  ::=  式のリスト
```

■代入文:　名前とオブジェクトを関連付けるために用いられます:

代入文の EBNF

```
代入文      ::=  (標的リスト"=")+ (式のリスト | 生成式)
標的リスト  ::=  標的 ("," 標的)* [","]
標的        ::=  識別子
                 | "(" 標的リスト ")"
                 | "[" 標的リスト "]"
                 | 属性参照
                 | 添字表記
                 | スライス
```

代入文の「(標的リスト"=")+ (式のリスト | 生成式)」は Python の内包表現で述べた for 節の標的リストに対する処理と本質的に同じです．つまり, 標的リストは「式のリスト」や生成式が出力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 ‘$x, y, \ldots, z$’ , 右辺の式のリストを ‘$a, b, \ldots, k$’, 生成式が出力するオブジェクトを m とすると, これらの長さは一致し, $(x, y, \ldots, z)/(a.b, \ldots, k)$ と $(x, y, \ldots, z)/m$ が True で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を意味します．

■**assert 文**:　プログラム中にテストやデバッグのための仮定を仕掛けるための手法を提供します:

assert 文の EBNF

```
assert 文  ::=  "assert" 論理和検証式 ["," 論理和検証式式]
```

引数の式が一つの assert 文の ‘assert　式’ は以下の if 文と同等の機能を持ちます:

```
if __debug__:
    if not 式論理和検証式 raise AssertionError
```

また, 引数が二つの assert 文:

$$assert\ 論理和検証式_1\ ,\ 論理和検証式_2$$

は以下の if 文と等価です:

```
if __debug__:
    if not $論理和検証式_1$: raise AssertionError($論理和検証式_2$)
```

このように assert の直後の論理和検証式を充すときに処理が行われ, それ以外で例外:AssertionError が生じます．

■**pass 文**:　文字通り「**何もしない**」文です:

pass 文の EBNF

```
pass 文  ::=  "pass"
```

この文は構文的に何らかの文が必要なときに用います．実際, 形式的な函数が必要なときに, 函数本体にこの文のみを入れたものを用います．

■**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.

■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オブジェクトを記載します:

return 文の EBNF

return 文 ::= "return" [式のリスト]

return 文は yield 文と同一函数内部で混在できません.

■yield 文: 産出函数の定義で用いられます:

yield 文の EBNF

yield 文 ::= yield 式

yield 文は PEP-255 より導入された文で, return 文の代わりに yield 文を用いて定義された函数は産出函数と呼ばれます. また, 産出函数からジェネレータ型のオブジェクトが生成されます:

```
>>> def fibonacci():
...     a, b = 0, 1
...     while 1:
...         yield a
...         a, b = b, a + b
...
>>> a = fibonacci()
>>> next(a)
0
>>> next(a)
1
>>> next(a)
1
>>> next(a)
2
>>> next(a)
3
>>> next(d)
5
>>> type(fibonacci)
```

```
<type 'function'>
>>> type(a)
<type 'generator'>
```

この例はフィボナッチ数列を産出函数で定義したもので, 函数 next() で計算処理が行われます. ジェネレータ型のオブジェクトはリストや配列のように全てのデータをメモリ上に持たずに, 必要なときに都度, 生成できる点で優れています. なお, yield 文と return 文の双方を同一函数内に併記することはできません.

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

raise 文の EBNF

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

例外そのものについては §3.12.3 を参照してください. raise 文自体は try 文の try 節に記載し, raise 文等が送出した例外に対応した処理を except 節で受け取って処理します.

■**break 文**:　for 文, while 文による反復処理から抜けるために用いられる文で引数が不要です:

break 文の EBNF

break 文　::=　"break"

この文はある条件下で反復処理ブロックから抜けて次の処理を行うために用いられます. その反復処理を抜けずに, ある条件下であれば反復処理の内容を飛ばすときには次の continue 文が用いられます.

■**continue 文**:　引数が不要な文で, for 文や while 文による反復処理の継続で用います:

continue 文の EBNF

continue 文　::=　"continue"

この文は反復処理中に, ある条件下では反復処理の内容を全て飛して同じ反復処理の次の処理に移るとき等で用いられ, ある条件で反復処理自体を止めて次の処理に移行する break 文と, 逆の操作を行うと言えます.

■**import 文**:　モジュールやパッケージの読込で用いられる文です:

import 文の EBNF

import 文	::=	"import" モジュール ["as" 名前]
		("," モジュール ["as" 名前])*
		| "from" 関係モジュール "import" 識別子
		["as" 名前]("," 識別子 ["as" 名前])* [","] ")"
		| "from" モジュール "import" "*"
モジュール	::=	(識別子 ".")* 識別子
関係モジュール	::=	"." モジュール | "."+
名前	::=	識別子

import 文で読込まれたモジュールやパッケージは module 型のオブジェクトになり, その名前は as 節が未指定であれば import 文の引数であるモジュール, as 節があればその引数の識別子がオブジェクトの名前になります. ここで複数のモジュールを纏めて階層構造を入れたモジュールを「**パッケージ (package)**」と呼びます.

■**future 文**: 将来の Python のリリースで利用可能になるような構文や意味付けを行うための指示句です:

future 文の EBNF

future	::=	"from" "__future__" "import" 機能 ["as" 名前]
		("," 機能 ["as" 名前])*
		| "from" "__future__" "import" "(" 機能 ["as" 名前]
		("," 機能 ["as" 名前])* [","] ")"

future 文はモジュールの先頭に置かなければなりません. ここで future 文よりも先行して置けるものは文書文字列, 注釈, 空行と他の future 文に限定されます.

■**global 文**: コードブロック全体で維持される宣言文で, 後続の識別子を大域変数として扱うことを指示します:

global 文の EBNF

global 文	::=	"global" 識別子 ("," 識別子)*

ここで global 文で宣言する名前はプログラム中で global 文に先行して配置してはいけません. また, for 文での反復処理制御用の変数の名前, class 文によるクラスの定義や函数定義, import 文内で global 文で宣言した名前を仮変数として用いてはなりません.

■**exec 文**: Python コードの動的な実行に関係する文ですが, Python 3 では廃止され, 函数 exec() を代わりに用います:

exec 文の EBNF

exec 文	::=	"exec" 識別子 ("," 識別子)*

3.6　複合文

複合文は複数の文節と, それに関わる一群の文から構成されて複数行で記述されますが, 一行に纏められて記載されることもあります:

複合文の EBNF

複合文	::=	if 文 | whike 文 | for 文 | try 文
		| with 文 | funcdef 文 | classdef 文
		| asunc_with 文 | asunc_for 文 | asunc_funcdef 文
一揃いの文	::=	文の列 NEWLINE
		| NEWLINE INDENT 文 + DEDENT
文	::=	文の列 NEWLINE | 複合文
文の列	::=	単純文 (";" 単純文)*[";"]

3.6.1　条件分岐に関連する複合文

■**if 文**:　条件分岐を指示する文です:

if 文の EBNF

if 文	::=	"if" 式 ":" 一揃いの文
		("elif" 式 ":" 一揃いの文)*
		["else" ":" 一揃いの文]

Python で用意されている条件分岐は if 文のみです. ここでの式は基本的に真理値か数値を値に持つものです. なお, 真理値 True と同様に評価が行われるのは 0 以外の数値であり, 真理値 False と同様の評価が行われるのは None や数値 0 と浮動小数点数 0.0 のときです.

3.6.2　反復処理に関連する複合文

■**while 文**:　反復処理を行うために用いる文で, 与えられた式の値が False, None, 数値の 0 以外の値以外のときに while 文内部の処理を実行します:

while 文の EBNF

while 文 ::= "while" 式 ":" 一揃いの文

■**for 文**: 反復処理を行うために用意された文です. for 文は式のリストから標的リストに含まれる束縛変数に値を引渡し, その値を用いて for 文内部の式の評価を行います:

for 文の EBNF

for 文 ::= "for" 標的リスト "in" 式 ":" 一揃いの文

内包表現で述べたように, 標的リストに代入可能な形式のオブジェクトが式から出力されなければなりません. そのような式にはタプル, リスト, 集合, 辞書やジェネレータ型のオブジェクトがあります.

3.6.3 例外処理に関連する複合文

■**try 文**: 例外処理のために用意された文です:

try 文の EBNF

try 文	::=	try 文 1 | try 文 2
try 文 1	::=	"try" ":" 一揃いの文
		("except" [式 [("as" | ",") 標的] ":" 一揃いの文)+
		["finally" ":" 一揃いの文]
try 文 2	::=	"try" ":" 一揃いの文
		"finally" ":" 一揃いの文

基本的に try 節の文を実行し, そこで送出された例外を except 節で受け取って処理を行います. この例外の詳細は §3.12.3 で解説します.

3.6.4 隠蔽に関連する複合文

■**with 文**: try-except-finally 文の標準的な利用を Python から括り出すために PEP-343 で規定された構文で, そのためにコンテキストマネージャ型のオブジェクトが導入されています:

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]

with 文はコンテキストマネージャの起動に用いられる構文で, for 文のように入れ子にすることも可能です．このコンテキストマネージャ型のオブジェクトには try に対応するメソッド___enter___() と finally に対応する___exit___() があり, たとえば, ファイルのオープンを with 文で行うと, ファイルのオープンがメソッド___enter___() で実行され, 無事, 操作が終了したときや例外が生じたときにメソッド___finally___() でオブジェクトのクローズが実行されます．このコンテキストマネージャは contextlib.py で定義されており, クラス contextlib を継承することで try-except-finally で実行していた処理を with 文で簡潔に記載できるようになります.

3.6.5 定義に関連する複合文

■**函数定義**: 函数やメソッドの定義で用いられる文です:

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 ")"

函数の定義と, 函数の呼出, 実行は別もので, 函数が呼び出されたときのみに函数が実行されます．また, 函数定義によって局所的な名前空間で函数名に函数オブジェクトの束縛が行われます.

■**クラス定義**: class 文でクラス定義が行えます:

クラス定義の EBNF

クラス定義	::=	"class" クラス名 [継承] ":" 一揃いの文
継承	::=	"(" [式リスト] ")"
クラス名	::=	識別子

最初に継承リストがあればリストの評価を行います．ここで継承リストの各要素の評価結果はクラスオブジェクト, あるいはサブクラス可能なクラス型でなければなりません．それから実行フレーム内部にて局所名前空間と大域名前空間を用いてクラス内の変数への束縛が行われます．ここで実行フレームは無視されるものの局所名前空間は保持され, それから基底クラスの継承リストを用いてクラスオブジェクトが生成されて局所名前空間を属性値辞書として保存します．それから最後に局所名前空間でクラス名がクラスオブジェクトに束縛されます．次の節では, オブジェクト, 名前空間や例外の詳細について述べます．

3.7 オブジェクトについて

3.7.1 オブジェクトの概要

Python のオブジェクトは「**識別値 (identity)**」,「**型 (Type)**」と「**値 (Value)**」をその属性として持ちます．まず,「**識別値**」はオブジェクト生成時に決まって変更できないオブジェクト固有の値で, 具体的にはオブジェクトのメモリ上の番地に対応する整数値です．「**型**」もオブジェクトの生成時に決定されて以後, 変更できません．そして, 型 (type) 自体もまた一つの型です．そして, オブジェクトはその値が「**変更可能 (mutable)**」なものと, 逆に「**変更不可能 (immutable)**」なものに分類できますが, 変更不可能なオブジェクトであって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参照先の変更で実質的に内容の変更ができます．この変更不可能なオブジェクトでは, その生成時に一定のメモリを割り当てられ, さらに組込の変更不可能なオブジェクトは全て「**要約可能 (hashable)**」という性質を持ちます．オブジェクトが「**要約 (ハッシュ) 可能, hashable)**」であるとは, そのオブジェクトが生成されて, それが存在している間に一定の整数値, すなわち,「**要約値 (hash value)**」を持つことを意味します．この要約値は二つのオブジェクトが同じ値を持っているときに要約値が一致するという重要な性質があります．要約値はオブジェクトの持つ値が等しいことをオブジェクトの詳細を調べなくても判別できるようにするための値で, 要約値を計算するメソッドが__hash__() です．なお, 要約不可能なオブジェクトでは__hash__にオブジェクト None が割り当てられています．None は空集合 $\emptyset$ に対応するために空集合 $\emptyset$ を始域に持つ写像も空集合 $\emptyset$ になるという事実に合致します．

オブジェクトの参照で「**名前空間**」が使われます．この名前空間はオブジェクトと識別子の間に対応関係を与える仕組で, 名前空間で扱われる識別子を「**名前**」と呼び, オブジェクトの識別値, 型, 値の参照は名前を介して行われます．たとえば, オブジェクト固有の値である識別値の参照は名前を引数に函数 id() で, オブジェクトの型の参照はその名前を引数にして函数 type() で調べられます．また, 異なる二つの名前で指示されるオブジェクトの同一性の判断は演算子 “is” に二つの名前を引き渡して行えます．このように名前はオブ

ジェクトに取り付ける名札と例えられ, オブジェクトへの参照は名前を介して行われます. Python の名前空間の実装は連想配列の辞書型オブジェクトが用いられます. オブジェクトと識別子との対応関係は割当文で与えられます. この割当文は ‘x = 1’ のように演算子 “=” が用いられますが, Python 3.8 からは演算子 “:= ” を用いた割当式が導入される予定です[*31].

3.7.2　オブジェクトの生成と廃棄を行うメソッド

オブジェクトの生成を行う函数, メソッドを「**構築子/コンストラクタ (constructor)**」と呼びます. オブジェクトの生成ではオブジェクトに対する「**メモリの割当 (allocation)**」と属性値の設定等の「**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を「**消滅子/デストラクタ (destructor)**」[*32]と呼びます. Python では構築子に相当するメソッドに__new__() と__init__() の二つのメソッドがあり, メソッド__new__() でオブジェクト生成, メソッド__init__() で初期化を行います. メソッド__new__() が呼出されると自動的にメソッド__init__() も呼出され, これらのメソッド二つで構築子としての機能を持ち, 特にメソッド __init__() が C++ の構築子に類似した働きをします. なお, Python ではクラスの定義で構築子に対応するこれらの二つのメソッドを記載する必要がなく, これらのメソッドがクラスに記載されていなければより上位のクラスのメソッドがそのまま継承されます.

Python の消滅子としてはメソッド__del__() が該当しますが, このメソッドはオブジェクトを直接破棄するものではありません. Python でオブジェクトの破棄には次で説明する「**GC(Garbage collection)**」が深く関わります.

3.7.3　GC(garbage collection) について

Python ではオブジェクトの生成や破棄でメモリの割当や解放が自動的に行われますが, オブジェクトが破棄されるためにはオブジェクトと名前との関係が途切れた状態, すなわち「**到達不能の状態**」になることが必要で, 到着不能の状態でオブジェクトの破棄が許可されて「**GC(塵回収, garbage collection)**」でオブジェクトの回収と破棄が行われます. そのために利用者は不要になったオブジェクトを直接破棄できません.

CPython の GC では「**参照カウント (reference counting)**」が採用されています. この方式は全てのオブジェクトに参照カウントと呼ばれる整数値を付与し, オブジェクトの生成時に参照カウントに 0 が設定されます. 以降, オブジェクトがある名前に束縛され

[*31] PEP-572: Assignment Expressions.
[*32] Java ではファイナライザ (finalizer) と呼びます.

たり, 他のオブジェクトからの参照があれば参照カウントに 1 を加え, 参照関係がある名前に別のオブジェクトが束縛されて参照関係が解消されると参照カウントから 1 を減じます. そして, 参照カウントが 0 になった時点でオブジェクトの破棄が許可されます. ここで消去子__del__() の実行で名前との参照関係が外されるためにオブジェクトの参照カウントが 1 減じるだけで, 参照先のオブジェクトの破棄を行うメソッドではありません. オブジェクトの参照カウントはモジュール sys の函数 getrefcount() で確認できますが, 函数 getrefcount() で調べた値は本来の参照カウントよりも 1 多くなっています. これは函数 getrefcount() による参照が参照カウントに追加されるためです[*33]. ここでは簡単な例で確認しておきましょう:

```
>>> class A():
...     def __del__(self):
...         print("Nyao")
...
>>> for i in range(5):
...     print(i)
...     a = A()
...     a = 1
...
0
Nyao
1
Nyao
2
Nyao
3
Nyao
4
Nyao
>>>
```

循環参照を持たずにオブジェクトの間の参照関係のみであれば, 名前との参照関係が外れた時点でオブジェクトの参照カウンタが 0 になってメソッド__del__() が呼び出されていることが判ります. しかし, 参照カウントが 0 になってオブジェクトの破棄が許可されてもオブジェクトの総数が閾値に到達して GC が自動的に起動したとき, あるいは利用者がモジュール gc の函数 collector() を起動したときにオブジェクトが破棄されます. この閾値はモジュール gc の函数 get_threshold() で確認できます:

```
>>> import gc
>>> gc.set_debug(gc.DEBUG_STATS)
```

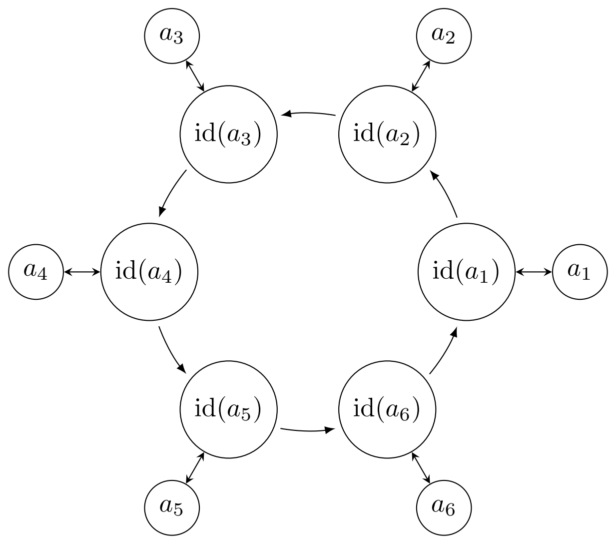
[*33] 一種の不確定性原理です. シュレディンガの猫とは違ってライフが充填されますが.

```
>>> gc.collect()
gc: collecting generation 2...
gc: objects in each generation: 346 484 5044
gc: objects in permanent generation: 0
gc: done, 0 unreachable, 0 uncollectable, 0.0053s elapsed
0
>>> gc.get_threshold()
(700, 10, 10)
```

この例では函数 set_debug() で統計量の出力を設定しているために函数 collect() でオブジェクトの実行で統計情報が表示されています．最後に実行した函数 gc_thershold() 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 世代の閾値です．この世代は到達可能なオブジェクトを乗り越えた GC の回数で分類したもので，最初に生成したオブジェクトを GC の 0 世代とし，第 1 世代が一度，GC を乗り越えた到達可能なオブジェクト，第 2 世代が二度以上 GC を乗り越えた到達可能なオブジェクトです．GC の自動実行は第 0 世代の総数が閾値を越えた時点で行われ，第 2, 第 1, 第 0 世代と古い順にオブジェクトの回収が行われます．

ここでオブジェクトを節点 (ノード, node)，参照関係を線分 (エッジ, edge) とすることで得られるグラフを考えましょう．たとえば，$i \in \{1,...,5\}$ 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 を参照するという参照関係をグラフ化すると次に示す円環がグラフに現れます:

循環参照の例



ここで名前 $a_{i,i\in\{1,...,6\}}$ に束縛されたオブジェクトを id(a_i) と表記します．なぜなら函数 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最終的に自己に戻る参照を「**循環参照 (circular reference)**」と呼びます．この例のオブジェクトは名前を介して相互に参照があるために全てのオブジェクトと名前との参照関係が途絶えてもオブジェクト間の参照関係 (内側の矢印) が残ります．このことは各オブジェクトの参照カウントが最終的に 0 にならないことを意味し, 参照カウントが 0 のオブジェクトを回収する方法で, これらの到達不能のオブジェクトが回収できません.

この弱点を克服する一つの手段がオブジェクトを世代別に分ける方法です．最初に生成直後のオブジェクトを第 0 世代とします．そして, 第 0 世代のオブジェクト総数が閾値を超えるか, gc モジュールの函数 collect() を起動することで GC が起動します．この GC ではオブジェクトが世代毎の到達可能性を調べ, 第 0 世代のオブジェクトで到達可能であれば第 1 世代, 第 1 世代と第 2 世代のオブジェクトで到達可能なものは第 2 世代と世代の繰り上げます．その結果, 到達不能なオブジェクトだけが残り, これらを回収すれば良いことになりますが, もし, オブジェクトにメソッド___del___() が定義されていると, どちらのメソッド___del___() を利用すべきかを判断できなくなるために, これらのオブジェクトが GC で回収できなくなります．Python 3 では PEP-442 でこの問題は改善されていますが, Python 2 では残っています．このことを例で確認しておきましょう．GC がどのように実行されているかを観察するために gc モジュールの函数 set_debug() を使って設定を行います．ここで統計情報を出力させるため gc.DEGUG_STATS の設定を行います．最初に循環参照があってもメソッド___del___() を持たないクラスの場合です．この例は Python 2, 3 で実質的な違いはありません:

```
>>> 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 —
167
gc: collecting generation 0...
gc: objects in each generation: 688 0 5817
gc: objects in permanent generation: 0
```

```
gc: done, 668 unreachable, 0 uncollectable, 0.0002s elapsed
168
── 略 ──
199
>>> gc.collect()
gc: collecting generation 2...
gc: objects in each generation: 131 12 5817
gc: objects in permanent generation: 0
gc: done, 134 unreachable, 0 uncollectable, 0.0020s elapsed
134
>>> gc.collect()
gc: collecting generation 2...
gc: objects in each generation: 3 0 5821
gc: objects in permanent generation: 0
gc: done, 0 unreachable, 0 uncollectable, 0.0019s elapsed
0
>>> gc.garbage
[]
>>>
```

閾値が 700 であるためにオブジェクトが 700 を越えた時点で最初の GC が行われ, この GC では第 1, 第 2 世代がまだないので第 0 世代のみです. 最初の GC のあとでもオブジェクトの生成が行われていために手動で 2 回, 函数 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おらず, 上手くオブジェクトの回収ができています. 次に $a_1, \ldots, a_6$ の相互参照で構成される循環参照の例を Python 2 で処理した様子を示しておきます:

```
>>> import gc
>>> gc.set_debug(gc.DEBUG_STATS)
>>> class A(object):
...     def __del__(self):
...         print "Nyao!"
...
>>> a1 = A()
>>> a2 = A()
>>> a3 = A()
>>> a4 = A()
>>> a5 = A()
```

```
>>> a6 = A()
>>> a2.x = a1
>>> a3.x = a2
>>> a4.x = a3
>>> a5.x = a4
>>> a6.x = a5
>>> a1.x = a6
>>> a1 = a2 = a3 = a4 = a5 = a6 = 1
>>> gc.collect()
gc: collecting generation 2...
gc: objects in each generation: 279 3282 0
gc: done, 12 unreachable, 12 uncollectable, 0.0018s elapsed.
12
>>> len(gc.garbage)
6
>>>
```

この例では函数 collect() で GC を実行しても円環を構築する 6 個のオブジェクトが残され, メソッド__del__() が実行されることもありません. また, この例では全ての名前に 1 を割り当てて参照関係を解消していますが, これを函数 del() で置き換えたとしても, 名前とオブジェクトとの参照関係が切断されて参照カウントが 1 減らされるだけで,「**どちらの消滅子を使えばよいのか?**」という問題のために肝心の処理が行われません. このように名前との参照関係を外すだけです. 同じことを Python 3.8 で行うとこの問題は上手く処理されていることが判ります:

```
>>> class A():
...     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: 28 0 10011
Nyao
Nyao
Nyao
Nyao
Nyao
Nyao
gc: done, 12 unreachable, 0 uncollectable, 0.0020s elapsed
12
>>> len(gc.garbage)
0
```

と, Python 2 の弱点であった循環参照を上手く処理ができていることが判ります. このように Python 3 は全体的にメモリの処理が Python 2 よりも改善されていることが判りますが, それでも__del__() の内容が必ず実行されるとは限らず, インタプリタを終了したときにメソッド__del__() が実行される保証がないことは以前と同様です. そのためにファイル等の外部リソースを参照するときはメソッド__del__() で解放するのではなく, 適宜, close() 等のリソースを解放するメソッドを使うべきです.

3.7.4 オブジェクトの値

Python のオブジェクトは値が「**変更可能 (mutable)**」なものと「**変更不能 (immutable)**」なものの二種類に分類され, この性質はオブジェクトの型で決定されます.

ここで簡単な例を示しておきましょう:

```
>>> a = b = []
>>> id(a)
3073670732L
>>> b.append(128)
>>> a is b
True
>>> id(a)
3073670732L
>>> id(b)
3073670732L
>>> c = [128]
>>> id(c)
3073670700L
```

この例では ‘a = b = [ ]’ でリスト型のオブジェクト ‘[ ]’ を生成して名前 a, b に同時に

束縛させています．そのために名前 a と b で参照されるオブジェクトが共通のためにその識別値が一致します．ところでオブジェクト '[ ]' は変更可能なオブジェクトのためにb.append(128) で名前 b で参照されているオブジェクトに '128' を追加できます．ところで, このオブジェクトを名前 a, b の双方で参照しているために名前 a で評価しても当然, '128' が追加されたものが表示されます．このことは 'a is b'[*34]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
>>> print id(c),id(d)
148353652 148353652
>>> c = 256
>>> print id(c),id(d)
148356068 148353652
>>> print c, d
256 128
```

この例では最初に 'c = d = 128' でオブジェクト 128 を名前 c, d に束縛させています．この時点で函数 id() の結果から名前 c と d で参照されるオブジェクトの識別値が一致するために名前 c, d ともに同じオブジェクトを参照していることが分かります．次に名前 c に 'c = 256' でオブジェクトの束縛を行なうと, 名前 c からの参照を解消し, 新たにオブジェクト 256 を生成して名前 c に束縛させます．ところが名前 d で参照されるオブジェクトにこの影響が及ばないために前回の変更可能なオブジェクトの例と異なり, 名前 d で参照されるオブジェクトは最初の '128' のままです．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, オブジェクト 128 への参照が名前 c と d 以外に存在しなければオブジェクト 128 は到着不能の状態で, いずれ, GC で回収されます．

Python のオブジェクトには他のオブジェクトへの参照を持つものがあり, 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

[*34] 二項演算子 "is" は「**オブジェクトが同一であるかどうか**」, 二項演算子 "==" は「**オブジェクトの値が等しいかどうか**」を判断する違いがあります．

変更に伴ってコンテナの実質的な値が変化します．要するにコンテナはアパートみたいなもので, 住人が入替っても部屋番号はそのままのために部屋番号を介して住人への問合せができますが, 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, 建物自体が以前のものと別物になるという訳です．コンテナの例としてタプルとリストを使って違いを確認しておきましょう:

```
>>> a=[1]; b=[2]; c=[3]
>>> l1 = [a, b, c]
>>> t1 = (a, b, c)
>>> l1
[[1], [2], [3]]
>>> t1
([1], [2], [3])
>>> id(l1)
26005416
>>> id(t1)
25557832
>>> c.append(128)
>>> id(l1)
26005416
>>> id(t1)
25557832
>>> t1
([1], [2], [3, 128])
>>> l1
[[1], [2], [3, 128]]
>>> t1[2]=[128]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> l1[2]=[128]
>>> l1
[[1], [2], [128]]
>>> id(l1)
26005416
>>> c = [129]
>>> print l1, t1
[[1], [2], [128]] ([1], [2], [3, 128])
```

この例では三成分のリストとタプルを生成して中身の入れ替えを行っています．タプルは変更不可能で, リストのように成分を直接変更できないものの, コンテナであるために参照しているオブジェクトが変更可能であれば, そのオブジェクトの値を変更することで結果として中身の変更ができます．ところで最後に名前 c にオブジェクト [129] を束縛させ

た場合はリストやタプルの側の旧来の参照が新しい参照に切り替わることがないために, そのまま名前 c に束縛されていたオブジェクト [3, 128] への参照が継続されます. つまり, オブジェクトの生成で名前を用いた場合, その名前に対応するオブジェクトが参照されることで新しいオブジェクトとの参照関係が発生しますが, 名前はそのオブジェクトの名札であって入れ物でないため, 名前に別のオブジェクトを束縛したからといって以前構築したオブジェクトが更新されることがありません.

3.7.5 オブジェクトの型

Python の組込のオブジェクトの型には次のものがあります:

オブジェクトの型

None	NotImplement	Ellipsis	数	列
集合	対応付け集合	呼出可能型	モジュール	クラス
クラスインスタンス	ファイル	内部型		

None, NotImplement 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, 真理値と違って数値を持たないために四則演算ができません. 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[*35]. 数は Scheme の「**数値塔 (Numerical Tower)**」にしたがって複素数に実数が, 実数に整数が包含されるという状況を, 複素 数を基盤に実数を, 実数を基盤に整数を載せた塔にたとえ, 塔を逆さに組み立てられないのと同様に上部と下部構造の入替ができません. なお, 数オブジェクト以外のオブジェクトはその生成に式や定義文を必要とするため, それらの EBNF は §3.5.2 で述べます.

■None: LISP の nil に似た値を持つオブジェクトの型で, そのオブジェクトが意味のある値を持たないことを指示します. None は組込の名前 None で参照され, その値は None そのもので if 文等の条件分岐で真理値 False として扱われますが False と同値ではありません. そして, この型を持つオブジェクトは None 以外に存在しません. このことを簡単な例で確認しておきましょう:

---

*35 PEP 3141:「A Type Hierarchy for Numbers」参照.

```
>>> 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 に割り当てています. 同時に名前 xx にも None を割当てていますが, 演算子 “is” で両者の同一性を調べると None 型を持つオブジェクトは一つのみ存在するために両者の識別子が一致し, そのために True が返却されます. このオブジェクト None の挙動は (小) 集合で構成される圏 **Set** で空集合 $\emptyset$ を始域とする矢 $\emptyset$ と同様の性質です. 実際, 圏 **Set** の矢は通常の写像が対応し, $A, B \in \mathrm{Obj}\ \mathbf{Set}$ 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 $\{\langle x, f(x)\rangle | x \in a\}$ 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 $\emptyset$ は圏 **Set** の対象であると同時に矢になります. このようにオブジェクト None は空集合 $\emptyset$ と同様の性質を持ちます.

■NotImplemented:　この型は単一の値しか持たず, この値を持つオブジェクトは NotImplemented のみです. 条件文では真理値 True として扱われますが, True と同一ではありません. この NotImplemented の値は NotImplemented そのものです.

■Ellipsis:　配列処理で用いられる拡張スライス構文にて配列の添字全体を示すリテラル ‘...’ で構成されるオブジェクトが持つ型です. Ellipsis は省略を意味する型で,None のように何もないことを意味する型と異なります. この型を持つオブジェクトは Ellipsis のみで, その値は Ellipsis そのものです. 条件文で真理値 True と同じ働きをしますが True と同一ではありません. この Ellipsis を用いたスライスの例を以下に示しておきます:

```
>>> from numpy import arange
>>> a = arange(16).reshape(2,8)
>>> a
array([[ 0,  1,  2,  3,  4,  5,  6,  7],
       [ 8,  9, 10, 11, 12, 13, 14, 15]])
>>> a[0, ...]
array([0, 1, 2, 3, 4, 5, 6, 7])
>>> a[1, ...]
array([ 8,  9, 10, 11, 12, 13, 14, 15])
```

ここで示すスライス操作は配列要素の取り出し操作で MATLAB 系言語でお馴染の添字操作です．例では最初に NumPy パッケージから函数 arange() の読込を行い，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

$$\begin{array}{lcl} a[0, ...] & \Rightarrow & \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{pmatrix} \\ a[1, ...] & \Rightarrow & \begin{pmatrix} 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \end{pmatrix} \end{array}$$

という処理です．この拡張スライス操作に現われるリテラル '...' は該当する添字の取り得る値の全てを意味する MATLAB の記号 ":" に相当する省略記号で，この記号が Ellipsis です．ただし，Python や MATLAB の記号 ":" は 1 次元の添字の省略記号ですが，この Ellipsis は 1 次元に限定されず，Yorick の「**ゴム添字 (rubber index)**」と同様に多次元である点で異なりますつまり，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**」されていること示すオブジェクトで，「**何もない**」ことを示す None とは異なります．実際，スライス操作で 1 次元的に記号 ":" を用いた操作で始点や終点を省略した表記が可能ですが，実際は None を記載したものと同値です:

```
>>> a = [0, 1, 2, 3, 4]
>>> a = [:4]
[0, 1, 2, 3]
>>> a = [None:4]
[0, 1, 2, 3]
```

このように None が Ellipse が意味する「**省略**」ではなく，「**何もない**」ことを意味していることが明瞭になるかと思います．また，Ellipsis は Python 2 では式や文を構成するオブジェクトで，独立したオブジェクトではありません．実際，Python 2 のインタプリタ上で '...' のみを入力するとエラーになりますが，Python 3 ではオブジェクトとしてリテラル '...' が許容され，Ellipsis 型のオブジェクトが生成されます．

■数 (numbers.Number): 数リテラルで生成され，算術演算や組込の算術函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．この型のオブジェクトの構築子は int() で, 函数 type() で調べると int が返却されます．なお, 演算結果や整数リテラルの入力が整数型の範囲を越えると自動的に長整数型に切替えられます．このことは構築子 int() を使って長整数型のオブジェクトを整数型に変換するときも同様で, 整数型の範囲を超えるときは長整数型のオブジェクトが自動的に生成されます[*36].
- 長整数型 (long integer): 長整数リテラルで生成され, 計算機のメモリに依存する任意桁数の整数が扱えます．そして, このクラスのオブジェクトの構築子は long() で, 函数 type() でこの型を調べると long が返却されます．なお, Python 3 の整数型は実質的に長整数型に統合されており, 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() [*37]で, 函数 type() でこの型を調べると bool が返却されます．なお, 構築子 bool() はリテラル 0, None と False をブール型の False に変換し, それ以外のほとんどをブール型の True に変換します．また, 四則演算を含む算術演算で True が整数型の 1, False が整数型の 0 として扱われ, これを利用してブール型のオブジェクトとの積をうまく使って if 文を使わない式の記述ができます．たとえば a, b が数オブジェクトのときに式 ‘a * True + b * False’ の結果は a になります．この処理は MATLAB 系の言語で if 文を減らすことによって言語のオーバヘッドを減らして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 は単精度の浮動小数点数の型を数リテラルに持ちません**．この型のオブジェクトの構築子は float() で, 函数 type() で調べると float が返却されます．なお, 構築子 float() で変換できるリテラルは整数, 長整数, 浮動小数点数とブール型で, ブール型の True は 0.0, False は 1.0 に変換されます．また, 特殊な数として無限大 (‘inf’) と非数 (‘NaN’) も float(’inf’) や float(’nan’) で生成できます．なお, 抽象基底クラスによる順序付では, このクラスはクラス Real の具象化ク

[*36] 数値処理言語によっては補数表現のために正負が入れ替わることもありますが, Python では自動的に長整数型に切り替わります.

[*37] Bool 型は 19 世紀の英国の数学者 George Boole(1815-1864) に由来しますが,「**Bool**」となぜか最後の “e” 省略で表記されます.

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 “+” で結合した式です. 構築子は complex() で, 函数 type() で調べると complex が返却されます. また, 複素数 z の実部は z.real, 虚部は z.imag で取り出せますが, 複素数型の性質上, これらは実数型, すなわち, 倍精度の浮動小数点数です. そのために近似の数であることに注意が必要で, そのこともあって虚部が 0.0 の複素数を構築子 int(), long() や float() で変換できません. なお, 抽象基底クラスによる順序付では, このクラスはクラス Complex の具象化クラスで, クラス Complex がクラス Number の子クラスです.

SageMath では高速処理のために, これらの数のクラスは全て SageMath の数のクラスで置換えられます. 詳細は §5 を参照して下さい.

■**列** (sequence): 列は自然数 $\mathbf{N}$ からオブジェクトへの対応があるオブジェクトです. すなわち, 列 S はある自然数 $N \in \mathbf{N}$ をその上限とし, 自然数 $i \leq N$ に対してオブジェクト $S(i)$ を対応させ, 0 が列の先頭, N が列の末尾に対応し, 列の濃度 $N+1$ を「**列の長さ**」と呼びます. 列には「長さ」があり, 函数 len() で調べられます. そして, 列を構成する成分の指定は ‘a[2]’ のように列の名前の直後の括弧 “[ ]” の中に対応すべき整数を与えます.

列に対する特殊な操作に「**スライス操作**」と呼ばれる部分列の生成操作があります. この操作は MATLAB 系言語でお馴染の操作で, 長さ n の列 a に対して $i < j$ を充す二つの正整数 $i, j \in \{0\ldots, n-1\}$ 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’ を取り出せます. つまり, [0:8:2] によって 0 から 8 までの間隔 2 の自然数の列 0, 2, 4, 6 が生成され, これらの自然数に相当する位置の文字が文字列 ’123456789’ から取り出された部分文字列 ’1357’ になります. また, 列を逆向きに取り出すときは刻幅を負の整数にします. たとえば ’123456’[5:2:-1] では 5 から開始して 2 を越えない間隔を-1 とする自然数の列 5, 4, 3, 2 に対応する文字列 ’6543’ になります. この拡張スライス操作で用いられる文字リテラル “...” は Ellipsis 型のオブジェクトへの参照になります. また, スライス操作で添字の省略もできます. これは添字が列の端部を指定するときに限って可能です. たとえば, リスト [1,2,3,4] で先頭から 3 成分を取り出したいときには a[0:3] と記述しますが, a[:3] と記述できます.

列は変更可能な型のオブジェクトと変更不能の型のオブジェクトの二種類に分類され, 変更可能なものがリスト型と ByteArrays 型, 変更不能なものが文字列型, UNICODE 文字列型とタプル型になります:

- リスト型 (list): 記号 “,” で区切られた任意の Python オブジェクトの列, すなわち, 式のリストを角括弧 “[ ]” で括って生成されるオブジェクトの型です．この型の構築子は list() で, この型のオブジェクトを函数 type() で調べると list が返却されます．
- ByteArray 型: 構築子 bytearray() で生成され, 0 から 255 までの整数の列をバイナリ形式の列として蓄えられます．
- 文字列型 (string): 接頭辞 ‘u’, ‘U‘ を含まない文字列リテラルで生成される型です．構築子は str() で, 函数 type() でオブジェクトを調べると str が返却されます．
- UNICODE 文字列型: 接頭辞 ’u’, ’U’ を含む文字列リテラルから生成される型です．構築子は unicode() で, この型のオブジェクトを函数 type() で調べると unicode が返されます．
- タプル型 (tuple): Python オブジェクトと記号 “,” を区切記号として並べた列, つまり, 式のリストの型がタプルです．ただし, 実用上, タプルの領域を明確にするためにリストの角括弧 “[ ]” ではなく, 丸括弧 “( )” で括ります．また, 成分が一つのタプルは特に「**シングルトン (singleton)**」と呼ばれます．この型の構築子は tuple() で, この型のオブジェクトを函数 type() で調べると tuple が返されます．

■集合 (set):　有限個のオブジェクトから構成されるオブジェクトです．波括弧 “{ }” で成員を囲った書式になります．成員の間には順序や対応付けを持たないために列や対応付け集合と異なり, 添字を用いた書式で成分の取出や参照ができません．なお集合の濃度は函数 len() で調べられます:

- 集合型 (set): 変更可能な型で構築子 set() で生成されます．
- FrozenSet 型: 変更不能な型で構築子 frozenset() で生成されます．集合型と違って要約可能 (hashable) なために別の集合の成分や辞書の鍵にできます．

■連想配列 (mapping):　LISP の「**連想リスト (association list, a-list)**」に相当するオブジェクトです．通常の列の成分取出は列 S の順位を表現する 0 から開始する自然数の部分集合を添字として指定することで行いますが, 連想配列では添字集合が自然数の部分集合だけではなく Python の有限個のオブジェクトの集合とすることが可能, より簡単に言うならば添字として自然数以外のオブジェクトが使える配列です．また, 連想配列 A の参照は添字の集合 I のときに $k \in I$ を使って $A[k]$ で行えます．また, 連想配列の濃度は列と同様に函数 len() で調べられます．現時点で Python に組込まれている連想配列は「**辞書 (dict) 型**」のみです:

- 辞書型 (dictionary): 変更可能な型で構築子は dict() です．また, 添字集合のことを「**鍵 (キー)**」, 添字集合の元を「**鍵値**」と呼びます．なお, 辞書は名前空間の実装

で用いられています.

■呼出可能 (callable):　呼出可能なオブジェクトの総称です. ここでの呼出とは, 形式的に名前等を函数の書式で扱うことで新たなオブジェクトの生成や既存のオブジェクトに対する処理操作のことです. 呼出可能なオブジェクトかどうかは組込函数 callable() で判断できます. 実装方法と挙動の違いから呼出可能のオブジェクトには以下のものがあります:

- ユーザ定義函数型 (user-definded function): 利用者による通常の函数定義から生成されるオブジェクトです. ここでの「通常」は yield 文を内部に包含しないという意味で, yield 文を包含する利用者定義の函数は「**生成函数型**」になります. ユーザ定義函数型のオブジェクトの呼出は函数定義で用いた引数の列と同じ長さ[*38]の列を引数にして行われ, 任意の属性の設定や取得ができます.

```
>>> def count(x):
...     return x + 1
...
>>> count(1)
2
>>> callable(count)
True
>>>
```

- ユーザ定義メソッド型 (user-defined method): クラス, クラス インスタンス, それと None を一次語とし, 記号 "." で任意の呼出可能なオブジェクトと結合させることで生成されるオブジェクトです. ここでの説明は neko という名前のクラスがあって, そのインスタンスメソッドに 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() でインスタンスメソッド CatchMouse() を呼び出せるという意味です.
  ユーザ定義のメソッドの簡単な例を示しておきます:

```
>>> class TEST(object):
...     def __init__(self, val):
...         self.val = val
...     def dbl(self):
...         return self.val * 2
...
>>> a = TEST(3)
>>> a.dbl()
6
>>> callable(TEST)
```

[*38] 引数の個数を函数のアリティ (arity) と呼びます.

```
True
>>>
```

メソッドの呼出はインスタンスの名前を一次語として記号 “.” に続けてメソッド名を記載することで行えます．このときにメソッドの定義で用いた引数 self は除きます．なお，メソッド__call__() を用いると，インスタンスを函数と同様の書式の記載で，このメソッドの呼出が可能になります:

```
>>> class TEST(object):
...     def __init__(self, start):
...         self.count = start
...     def __call__(self):
...         count = self.count
...         self.count = count + 1
...         return count
...
>>> a = TEST(0)
>>> a()
0
>>> a()
1
>>> a.__call__()
2
>>> callable(TEST)
True
>>>
```

この例ではメソッド__init__() でインスタンス変数 count の初期化を行い，メソッド__call__() を実行すると count の値が +1 されるというものです．このクラスのインスタンス a に対して ‘a()’ と函数の書式でメソッド__call__() の呼出が行われています.

- 生成函数型 (generator function): 内部に yield 文を持つ利用者定義の函数です.

```
>>> def genos(start):
...     count = start
...     while True:
...         yield count
...         count = count + 1
...
>>> a = genos(1)
>>> next(a)
1
>>> next(a)
```

```
2
>>> type(a)
<type 'generator'>
>>> callable(genos)
True
>>>
```

yield 文を有する函数で生成されるオブジェクトの型は generator で, 函数 next() で生成型のオブジェクトから次の値を取り出せます. この生成函数と同様の処理をメソッドで行うことも可能で, そのクラスのインスタンスが次に述べる反復子型になります.

- 反復子型 (iterator): メソッド__iter__() とメソッド__next__() の一対を持つ利用者定義のクラスです:

```
>>> class TEST(object):
...     def __init__(self, start):
...         self.count = start
...     def __iter__(self):
...         return self
...     def next(self):
...         count = self.count
...         self.count = self.count + 1
...         return count
...
>>> a = TEST(1)
>>> next(a)
1
>>> next(a)
2
>>> callable(TEST)
True
```

- 組込函数型 (built-in function): 組込函数オブジェクトは C の函数のラッパーです. このような函数の例に組込函数 dir() や 函数 math.sin()[*39]があります.
- 組込メソッド型 (built-in method): 組込函数を隠蔽したもので, C の函数に引き渡されるオブジェクトを何らかの非明示的な外部引数として持ちます.
- クラスタイプ型 (class type): クラスタイプ型のオブジェクトはインスタンスオブジェクトの生成で用いられ, このときに「**ファクトリクラス**」として振舞います. ここでメソッド__new__() の上書 (override) を行っても問題はありません. クラスを呼出したときの引数はメソッド__new__() に引渡され, このメソッドがクラ

[*39] 標準モジュール math に包含される函数 sin() を指示する名前になります.

スタイ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です. この呼出で用いられる引数はメソッド__init__() に引渡されるため, このクラスにメソッド__init__() がないときはクラスを引数なしで呼び出さなければなりません.
- クラスインスタンス型 (class instance): 古典的クラスのクラスにメソッド__call__() があるときに限って呼出可能型になります.

■モジュール (module):　Python の文を記載したファイルを import 文で読込むことで生成されるオブジェクトです. インタプリタに読込まれると, Python の文はコードオブジェクトに翻訳されて pyc ファイルに蓄えられます. このコードオブジェクトの実体はバイトコードと呼ばれるバイナリデータで, 入力式や文と比べて計算機が処理し易い書式になっており, このバイトコードを Python 仮想計算機 (PVM) が処理します. また, 複数のモジュールに階層構造を入れて管理できるようにしたものを「**パッケージ**」と呼びます. このパッケージの読込で生成されるオブジェクトの型も module 型です.

■クラス (class):　Python 3 は「**クラスタイプ型**」と呼ばれるクラスの型のみですが, Python 2 ではクラスタイプ型に加えて「**クラスオブジェクト型**」, 別名「**古典的クラス型**」があります. どちらも辞書で実装された名前空間を持っていますが, 古典的クラス型は Python 2 で基底クラスとして object を継承しないときに生成され, クラスタイプ型と違ってメタクラスが扱えず, 引数なしにメソッド super() で基底クラスへの属性等の参照ができません

クラス間の関係で, 継承する側のクラスを「**派生クラス**」, 継承される側のクラスを「**基底クラス**」と呼びます. 基底クラスは派生クラスよりも普遍的, すなわち上位のクラスです. この継承関係を親子関係にたとえるならば, 基底クラスが「**親**」, 継承する派生クラスが「**子**」にそれぞれ対応します. さらに, この継承関係を (素朴集合論の) 集合の包含関係として捉えることも可能で, そのときに派生クラスが「**部分集合 (subset)**」にする「**サブクラス**」, 基底クラスを「**スーパークラス**」と呼びます. さて, あるインスタンスがとあるクラスに包含されることと, あるクラスがとあるクラスを継承しているといった関係を矢で表現してみましょう. つまり, $C_1 \xrightarrow{\sigma} C_0$ でクラス C_1 がクラス C_0 を継承したもの, すなわち C_0 の派生クラスであることを示します. このときに次の図式が可換になります:

$$\begin{tikzcd} A \arrow[rr, "\sigma"] \arrow[dr, "\sigma"'] & & C \\ & B \arrow[ur, "\sigma"'] & \end{tikzcd}$$

この図式は 3 個のクラスの派生関係を示し, $A \overset{\sigma}{\to} B$ かつ $B \overset{\sigma}{\to} C$ であれば $A \overset{\sigma}{\to} C$ となること, つまり, 関係 $\overset{\sigma}{\to}$ は結合律を充します. ただし, 反射律 $A \overset{\sigma}{\to} A$ は, クラス定義で自分自身が未定義の状態で自分自身を用いる循環的定義が許容されません[*40]が, 既存のクラスを上書きする形で自分自身を定義することは可能です. だから, 反射律は循環的定義として不可でも, 既存の類定義の更新として可能です. それから, 下位のクラスの属性は新たにクラス変数に値の束縛を行うことで, そのクラスの属性値が更新されますが, 上位の基底クラスの属性に遡及して値が更新されることはありません. 逆に上位のクラスの属性を変更すると下位のクラスの属性の変更が生じます.

クラスには**特殊属性 (special attribute)** と呼ばれる属性があります. 属性 __name__ にクラス名, __module__ にクラスを定義したモジュール名, __dict__ にクラスの名前空間が入った辞書, __bases__ に基底クラス名を収納したタプル, そして, __doc__ にクラスを説明する文書文字列が束縛されています.

■**クラスインスタンス**: Python 3 で廃止された Python 2 の対象です. このクラスインスタンスは古典的クラスの呼出で生成される対象で, 辞書で実装された名前空間を持って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, それに伴なうクラス辞書の更新はありません.

■**ファイル** (file): 開かれたファイルを表現するオブジェクトです. このファイルは組込函数 open() ,os.popen() ,os.fdopen() , および, socket オブジェクトのメソッド makefile() 等で生成されます.

■**内部型** (internal type): インタプリタがその内部で用いる型で, これらの型の幾つかは利用者に公開されています. この内部型はインタプリタの仕様変更等で将来の変更が生じる可能性があります:

- **コードオブジェクト (code object):「バイトコード (bytecode)」**を表現するオ

[*40] 参照すべきオブジェクトがなければ例外が発生します.

ブジェクトで函数 type() で調べると code を返します．このバイトコードは式や函数のインタプリタ内部での表現で，バイナリ形式のオブジェクトです．Python 仮想計算機 (PVM) は入力された式や文を直接処理するのではなく，それらが翻訳されたバイトコードを処理しています．たとえば，函数のバイトコードはオブジェクトの属性___code___に束縛されています．組込函数 compile() で Python の式をバイトコードに変換し，組込函数 exec() を使って処理できます:

```
>>> a = compile('sum([1,2,3,4,5])','','single')
<code object <module> at 00000000036E2A30, file "", line 1>
>>> type(a)
<type 'code'>
>>> exe a
15
>>> def pet():
...     print "dog and ", cat
...
>>> pet.___code___
<code object pet at 0000000003615130, file "<stdin>", line 1>
>>> type(pet.___code___)
<type 'code'>
>>> exec(pet.___code___,{'cat' : 'dog'})
cat and dog
```

ここでは式 'sum([1, 2, 3, 4, 5])' を函数 compile() でバイトコードに変換したものと定義した函数 pet() の属性___code___に束縛されたバイトコードを函数 exec() で実行させています．このバイトコードの中身はモジュール dis の函数 dis() を使って確認できます:

```
>>> import dis
>>> dis.dis(a)
  1           0 LOAD_NAME                0 (sum)
              3 LOAD_CONST               0 (1)
              6 LOAD_CONST               1 (2)
              9 LOAD_CONST               2 (3)
             12 LOAD_CONST               3 (4)
             15 LOAD_CONST               4 (5)
             18 BUILD_LIST               5
             21 CALL_FUNCTION            1
             24 PRINT_EXPR
             25 LOAD_CONST               5 (None)
             28 RETURN_VALUE
>>> dis.dis(pet)
  2           0 LOAD_CONST               1 ('dog and ')
```

```
 3 PRINT_ITEM
 4 LOAD_GLOBAL              0 (cat)
 7 PRINT_ITEM
 8 PRINT_NEWLINE
 9 LOAD_CONST               0 (None)
12 RETURN_VALUE
```

ここで表示されていることは PVM に引き渡す指示で, PVM はスタックマシンと呼ばれる与えられた命令を積み重ね (stack) て, それらの命令を逐次解釈する仕組みです.

import 文等でインタプリタに読み込まれたプログラムはバイトコードに翻訳され, そのバイトコードは '.pyc' ファイルに蓄えられます[*41]. 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. そのために '.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. また, バイトコードは異なる Python 仮想マシンでの互換性を保証するものではありません.

- **フレームオブジェクト (frame object):「実行フレーム (execution frame)」**を表現するオブジェクトです. プログラム実行時に呼び出し可能 (callable) なオブジェクトが呼出され, そのオブジェクトの実行が終えた時点で呼出した側のプログラムで呼出前の実行状態を保存しているオブジェクトから取り出す仕組になっています. このときの実行状態が保存されているときのオブジェクトをフレームオブジェクトと呼びます.
- **トレースバックオブジェクト (traceback object):「例外スタックトレース (exception stacktrace)」**を表現するオブジェクトです. この traceback オブジェクトは例外が発生した時点で生成され, 例外ハンドラを検索してスタックを戻すときに, その戻ったレベル毎にトレースバックオブジェクトが, その時点の traceback の前に挿入されます. 例外ハンドラに入るとスタックトレースをプログラムで利用できるようになります.
- **スライスオブジェクト (slice object):「拡張スライス構文」**を表現するために用いられるオブジェクトです. この構文は MATLAB 系言語で用いられるベクトル成分の取出に類似し, 配列処理で類似の機能を与えます.
- **静的メソッドオブジェクト (staticmethod object):** 組込の構築子 staticmethod() やデコレータ@staticmethod を使って生成されるオブジェクトです. インスタンスメ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.

[*41] バイトコードの詳細は次を参照のこと: https://www.ics.uci.edu/ brgallar/week9_3.html

クラス変数の参照ではクラスやクラス変数を直接呼び出すことで行います．またこのことから派生クラスで静的メソッドが定義されたクラスのクラス変数の参照が行われます．このように「**静的**」には「**継承において動的なクラス変数の参照が行われない**」という意味があります．なお, 静的メソッドはインスタンス変数を参照できません.

- **クラスメソッドオブジェクト (classmethod object):** 組込の構築子 classmethod() やデコレータ@classmethod を使って生成されるオブジェクトで, staticmethod と同様にインスタンス化しなくても利用できます．メソッドの定義では第一引数に ‘cls’ を必要とし, そのため継承で静的ではなく動的にクラス変数の参照が行われますが, 静的メソッドと同様にインスタンス変数の参照はできません.

静的メソッドとクラスメソッドの違いについて

静的メソッドとクラスメソッドの相違点を明確にするために簡単な例を以下に示しておきます:

```
>>>class A(object):
...     neko = 'mike'
...     def sm(x):
...         print "%s: neko = %s %s"%(A, A.neko, x)
...     def cm(cls,x):
...         print "%s: neko = %s %s"%(cls, cls.neko, x)
...     static_method = staticmethod(sm)
...     class_method = classmethod(cm)
>>>class B(A):
...     neko = 'tama'
>>>A.static_method(1)
<class 'A'>: neko = mike 1
>>>A.class_method(1)
<class 'A'>: neko = mike 1
>>>B.static_method(1)
<class 'A'>: neko = mike 1
>>>B.class_method(1)
<class 'B'>: neko = tama 1
```

この例ではクラス A とクラス A の派生クラスのクラス B を生成しています．さらにクラス A に静的メソッド static_method() とクラスメソッド class_method() を定義していますが, ‘cls’ を静的メソッドは引数として取らず, クラスやクラス変数への参照ではメソッド内部でそれらを直接呼び出しています．それに対してクラスメソッドでは第一引数に ‘cls’ を取り, クラスやクラス変数への参照は ‘cls’ を介して行います．そして両者の結果の相違は定義したクラス A ではありませんが, 派生クラス B で両者に違いが出て

きます．まず, ‘B.static_method(1)’ でクラス変数 neko の値は静的メソッドが定義されたクラス A のものが用いられています．またクラスも A のままですが, クラスメソッド ‘B.class_method(1)’ ではクラスが B に切り替わっており, それに伴ってクラス変数の値もクラス B のものになります.

ここでの定義では組込函数 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できます:

```
>>>class A(object):
...     neko = 'mike'
...     @static_method
...     def static_method(x):
...         print "%s: %s %s"%(A, A.neko, x)
...     @class_method
...     def class_method(cls,x):
...         print "%s: %s %s"%(cls, cls.neko, x)
```

このように静的メソッドやクラスメソッドの定義の直前にデコレータを配置することで定義できます.

3.8 特殊メソッド

3.8.1 特殊メソッドの概要

特殊メソッドは特定の演算や機能をクラスに実装するためにあらかじめ用意された一群のメソッドです．これはメソッドの「**上書き (override)**」を利用し, 派生クラスに含まれない特殊メソッドは基底クラス側のものがそのまま用いられ, 派生クラスで特殊メソッドを新たに定義することで基底クラスから継承されたメソッドの上書きが行われます．その結果, 派生クラスで基底クラスのものと別の改変されたメソッドが利用できます．たとえば, 同じクラスのオブジェクトの比較で「**拡張比較**」と呼ばれる一群のメソッドは, これから定義するクラスの二つのインスタンスが等しいかどうか, 同一クラスのインスタンス間の大小関係を判断するメソッドで, これらのメソッドを上書きすることで整数や実数で用いられる等号 “==” とその否定 “!=”, あるいは大小関係 (“<”, “>”, “<=”, “>=”) に新たな意味を持たせられます．ただし, ここでの拡張比較のメソッドは相反するメソッドのどちらか一方を上書きすることでもう一方が自動的に再定義されるものではありません．たとえば, 等号 “==” の否定は “!=” ですが, 等号 “==” を再定義しても, その否定の “!=” が自動的に定義されるものではありません.

以下, 項目別に特殊メソッドを列記しますが, クラス名を ‘cls’, オブジェクト名を ‘obj’, それからメソッドの引数をまとめて ‘args...’ と表記します．また, メソッドの引数に ‘[,

args...]’ とある場合は鉤括弧 “[ ]” で括った引数がオプションであることを指示します．また, 二項演算子の引数は同一クラスに属するオブジェクトが二つ必要になりますが, 引数として一つは前述の ‘self’, もう一つは ‘other’ を用います．そして, ‘cls’, ‘self’ と ‘other’ は ‘obj’ と異なり固定です．

3.8.2　オブジェクトの生成と削除に関連するもの

ここで述べるメソッドは構築子と消却子に相当します．

■obj.___new___(cls [, args...])　クラスのインスタンス生成で呼び出される静的メソッドです．インスタンス生成が要求されているクラス名 cls を第一引数に取り, 残りのオプションの引数をメソッド___init___() に引渡してオブジェクトの初期化を行います．このメソッド___init___() と併せて C++ の構築子に似た動作をします．なお, メソッド___new___() がクラス cls のインスタンスを返却するときに限ってメソッド___init___() が呼出されます．

■obj.___init___(self [, args...])　オブジェクトの初期化に関わるメソッドで, 前述のメソッド___init___() と併せて C++ の構築子と同様の働きをします．ここで基底クラスがメソッド___init___() を持つときに, その派生クラスのメソッド___init___() がインスタンスの基底クラスで定義されている部分が初期化されるようにする必要があります．なお「**構築子は値を返却してはならない**」という制約があるためにメソッド___init___() がこの制約に反して値を返すようにしていると実行時に TypeError が発生します．

■obj.___del___(self)　C++ の「**消滅子 (destructor)**」に似た動作を行うメソッドで, オブジェクトを削除するときに呼出され, 引数は self 以外はありません．なお, Python のオブジェクトは到達不可能になった時点から回収され, C++ の消滅子のように不要になった時点ではありません．一般的に基底クラスがメソッド___del___() を持っているときは派生クラスのメソッド___del___() で明示的に基底クラスのメソッド___del___() を呼出してオブジェクトを消去するようにしなければなりません．

3.8.3　値の表示に関するもの

利用者に適切なオブジェクトの持つ属性や値といった情報が表示されるようにするためのメソッドです．これらのメソッドの引数は self のみです．

■obj.___repr___(self)　組込函数 repr() や文字列への変換時に呼出され, オブジェクトをフロントエンド側で表示する「**公式の表示**」の生成を行います．この章の有理数の定義にて整数対を有理数として表示する例のように, 利用者とって分かり易い表示や必要とされ

る情報の表示に変更できます.

■obj.__str__(self)　組込函数 str() と print 文[*42]から呼出され, オブジェクトをフロントエンド側で表現する**非公式の文字列**の生成を行います. このメソッドの返却値は文字列オブジェクトでなければなりません. なお, メソッド__repr__() が定義されていれば, このメソッドが定義されていなくてもメソッド__repr__() が利用されます.

3.8.4　値の比較に関するもの

クラスに「**大小関係**」を導入するためのメソッドで, これらの演算子は当然のことながら演算子を被演算子の間に配置する中値表現の二項演算子です:

比較の演算子

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

これらの演算子の書き換えは他の演算子に影響しません. すなわち, 大小関係を新たに定義したクラスに導入するために演算子 “≥” を定義しても, その否定として演算子 “<” が自動的に定義されません. 大小関係の書き換えが一部でも生じた場合は全体を通して再定義が必要ですが, 大小関係の演算子 “<”, “>”, “≤”, “≥” 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が可能です. また, これらの演算子を書き換えていないときに用いられる比較の特殊メソッドが次の obj.__cmp__() です.

■obj.__cmp__(self, other)　二つの同一クラスのインスタンスを比較するときにインスタンスの識別子 (整数型) を使って大小関係の判断ができます.

3.8.5　整数演算に関連するもの

整数オブジェクトの二項演算に関係する特殊メソッドを以下にまとめておきます:

[*42] 3.x では函数 print().

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.__trudiv__(self, other)
商	//	a // b	obj.__floordiv__(self, other)
剰余	mod	a mod b	obj.__mod__(self, other)
冪	**	a ** b	obj.__pow__(self, other)

ここで Python 2. で演算子 “/” の二つの被演算子が整数のときに結果は整数であり, Python 3 では演算子 “//” が相当し, Python 3 の演算子 “/” は割り切れないときに浮動小数点数になります.

これらの整数演算に加えて整数オブジェクトには, 被演算子を 2 進数として表現したときの二項演算も加わります:

2 進数整数演算子 (二項演算)

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

3.8.6 浮動小数点数演算に関連するもの

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	+	a + b	obj.__radd__(self, other)
差	-	a - b	obj.__rsub__(self, other)
積	*	a * b	obj.__rmul__(self, other)
商	/	a / b	obj.__rtrudiv__(self, other)
商	//	a // b	obj.__rfloordiv__(self, other)
剰余	mod	a mod b	obj.__rmod__(self, other)
冪	**	a ** b	obj.__rpow__(self, other)

これらの演算子は被演算子のどちらか一方が浮動小数点数のときに浮動小数点数を返します. 演算子 “/” は単純な割算の演算子ですが, 演算子 “//” は商の整数部分のみを浮動小数点数の型で返却する演算子で, 演算子 “/” を除く演算子は被演算子が整数型のときの演算子を自然に拡張した演算になっています.

3.8.7　累算算術代入演算に関連するもの

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	obj.__iadd__(self, other)
a = (a - b)	-=	a -= b	obj.__isub__(self, other)
a = (a * b)	*=	a *= b	obj.__imul__(self, other)
a = (a / b)	/=	a /= b	obj.__itrudiv__(self, other)
a = (a // b)	//=	a //= b	obj.__ifloordiv__(self, other)
a = (a mod b)	mod=	a mod= b	obj.__imod__(self, other)
a = (a ** b)	**=	a **= b	obj.__ipow__(self, other)

累算算術代入演算子では本来の演算子が演算子を構成する文字 “=” の左側にあり, 被演算子は整数, 浮動小数点数といった数オブジェクトの型の制約はありません.

2 進数累算算術代入演算子 (二項演算)

同値な式	記号	Python の式	特殊メソッド
a = (a< <= b)	< <=	a < <= b	obj.__ilshift__(self, other)
a = (a > >= b)	> >=	a > >= b	obj.__irshift__(self, other)
a = (a & b)	&=	a &= b	obj.__iand__(self, other)
a = (a ^ b)	^=	a ^= b	obj.__ixor__(self, other)
a = (a | b)	|=	a |= b	obj.__ior__(self, other)

2 進数累算算術代入演算子も同様で, 本来の演算子が演算子を構成する文字 “=” の左側に現れます.

3.8.8　単項演算子

単項演算子では整数, 浮動小数点数のメソッドの区別はありません:

単項演算子

演算の意味	記号	Python の式	特殊メソッド
$a \to -a$	-	-a	obj.___neg___(self)
$a \to +a$	+	+a	obj.___pos___(self)
$a \to \|a\|$	abs( )	abs(a)	obj.___abs___(self)
$a \to \tilde{a}$	~	~a	obj.___invert___(self)

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 $\tilde{a}$, 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.

3.8.9　属性値の取得と設定に関連するもの

属性値の取得や変更, 削除について特殊メソッドをクラスに定義することで新たな意味付けができます. ここで説明するメソッドに現れる変数名で name を属性名, value をその値とします:

■obj.___getattr___(self, name)　属性の検索において self のインスタンス属性やクラスツリーでも検出されなかったときに呼出されます. このメソッドは計算された属性値か例外:AttributeError を送出しなければなりません. なお, 新スタイルクラスで実際に完全な制御を行う方法はメソッド___getattribute___() を参照してください.

■obj.___setattr___(self, name, value)　属性への値の束縛で呼出される特殊メソッドです. ここで name が属性名, value がその属性値です. なお, インスタンス側の属性に値を束縛させるときに 'self.name = value' とした場合は自己参照が生じるために行ってはなりません. 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ます.

■obj.___delattr___(self, name)　属性に束縛した値の削除を行います. このメソッドの実装は 'del obj.name' に意味のあるときに限定すべきです.

■obj.___getattribute___(self, name)　クラス型がクラスタイプのみに対して利用可能なメソッドで, 指定した属性の値を返却するメソッドです. なお, メソッド___getattr___() が実装されていれば例外:AttributeError が送出されない限り呼び出されません. このメソ

ッドの実装では再帰的な呼出を防止するために必要な属性全てへの参照で ‘obj.___getattribute___(self, name)’ のように基底クラスのメソッドを同じ属性名で呼び出さなければなりません.

3.8.10 その他

■obj.___hash___(self) 要約 (ハッシュ) 値を返却するメソッドです. 要約値は整数値であり, 同じ値を持つオブジェクトであればそれらの要約値は一致しなければなりません. つまり, ‘a == b’ が True であるなら ‘a.___hash___() == a.___hash___()’ も True になります. なお, 要約値は整数値になりますが, -1 をエラーフラグとして予約済にしている関係上, 内部計算で-1 が得られると-2 を返却する仕様になっています[*43]. 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[*44]. 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 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()[*45]の実現のために呼出されます. このメソッドは真理値の ‘True’(=真) か ‘False’(=偽), あるいはそれらと等価の整数 ‘1’(=真) か ‘0’(=偽) の何れかを返さなければなりません. このメソッドが定義されていないときはメソッド___len___() が呼出され, その結果が ‘nonzero’ であれば True, nonzero のときは False です. それからもしもメソッド___len___() と___nonzero___() の双方が実装されていなければ, そのクラスのインスタンスの真理値は全て ‘True’ とみなされます.

[*43] http://effbot.org/zone/python-hash.htm 参照
[*44] 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.
[*45] 何故か boole でない!

■obj.__unicode__(self)　組込函数 unicode() を実現するために呼出され, unicode オブジェクトを返却しなければなりません. このメソッドが定義されていないときは文字列リテラルへの変換が試みられ, その結果, 既定値の文字エンコードを用いて UNICODE 文字列に変換されます.

3.9 記述子 (descriptor)

ある特定の性質を持つオブジェクトが実装すべきメソッドを「**規約 (プロトコル, procotol)**」と呼びますが,「**記述子**」はその規約の一つで, クラスタイプ (object や type の派生クラス) のみに対応し, 属性の束縛に関わるメソッドです. 記述子は「**記述子規約 (descriptor protocol)**」と呼ばれる 3 個のメソッド__get__(), __set__() と __delete__() の何れかが上書きされ, メソッド __get__() と__set__() の双方が定義されている「**データ記述子**」とメソッド__get__() のみが定義されている「**非データ記述子**」の二種類に大きく分類されます. さらにメソッド__set__() の呼出で例外:AttributeError が送出されるデータ記述子を「**読取専用データ記述子**」と呼び, この記述子を導入することで属性の管理が可能になります.

記述子の呼出は属性への参照がその基点になります. たとえばオブジェクト a に対して属性 x の参照は a.x で行いますが, このときに a.x が基点になります. ここで引数がどのように記述子に結合されるかはオブジェクト a がクラスのインスタンスであるか, あるいはクラスそのものであるかに依存します:

記述子の呼び出し

- 直接呼出: 最も単純な呼出操作で ‘x.__get__(a)’ に変換されます.
- インスタンス束縛: クラスタイプのインスタンスに対する束縛で ‘a.x’ が ‘type(a).__dict__[’x’].__get__(a,type(a))’ に変換されます.
- クラス束縛: クラスタイプのクラスに対する束縛で ‘a.x’ が ‘a__dict__[‘x‘].__get___(None, a)’ に変換されます.
- スーパークラス束縛: a が super のインスタンスのときに束縛 super(b, obj).m() を行うと最初に a, 次に b に対して obj.__class.__mro__を検索し, それから呼出: ‘a__dict__[’m’].__get__(obj,obj.__class__)’ で構築子を呼出します.

インスタンス束縛で構築子の呼出の優先順序は定義内容に依存します. そして構築子は上述の 3 つのメソッドの任意の組合せで定義されますが, ここでメソッド__get__() が定義されていないときに該当する属性の参照が行われると構築子オブジェクト自体が返却されます. 前述のようにメソッド__set__と__delete__のどちらか一方が定義され, デー

タ構築子, 双方が定義されていなければ非データ構築子になります．組込函数 property() はデータ構築子として Python に実装されたもので，このときにインスタンスでは属性の上書ができません．その一方で staticmethod() と classmethod() を含む Python のメソッドは非データ構築子として定義され, そのためにインスタンスでメソッドを再定義され, インスタンスでメソッドの再定義や上書きができます．このことを利用して同じクラスのインスタンスでも個々の挙動に違いを持たせられます．また, 属性検索でデータ記述子やインスタンスの属性辞書, 非データ記述子の順番で検索が行われます.

3.9.1 記述子 (descriptor) の実装

■obj.__get__(self, instance, owner)　クラスの属性, インスタンスの属性の参照で呼出されます．ここで 'owner' はオーナークラスで, instance は属性への参照を仲介するインスタンス属性が 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() と似た動作です.

3.10 クラス属性の参照について

クラス属性の参照はクラス名が C で属性が x のときに C.x で行えます．この参照の実体は C.___dic___["x"] で, 目的の属性がクラス名で指示したクラスに見当らなければ上位の基底クラスで参照が行われます．ここで属性検索は検索しているクラスになければ継承関係が一つ上のクラスへと遡るという「**継承の深さ**」が関わります．たとえば, $C_0 \overset{\sigma}{\to} C_1, C_1 \overset{\sigma}{\to} C_2, \cdots, C_{n-1} \overset{\sigma}{\to} C_n$ 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): $C_0 \to \ldots \to C_n$ 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 $(C_0, C_1, \cdots, C_n)$ の並び順がクラス C_0 の「**MRO(Method Resolution Order**」と呼ばれるメソッドの検索順序で, $\mathcal{L}(C)$ と記述します．また, この検索順序を求める処理のことを「**線形化 (linealization)**」と呼びます．この MRO を説明するために幾つかの言葉を定義しておきます．まず, 検索順序はクラスのリスト: $(C_1, \ldots, C_n)$ で表現されていますが, これを語: $C_1 \cdots C_n$ 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 $C_0 + C_1 \cdots C_n$ と表記します．次に語 $C_0 C_1 \cdots C_n$ の先頭 C_0 を取り出す操作を head, 先頭以外の残りの

$C_1 \cdots C_n$ を取り出す操作を tail と表記し, 語 L に対して $\overline{L} \overset{\mathrm{def}}{=} \mathrm{head}(L)$, $\underline{L} \overset{\mathrm{def}}{=} \mathrm{tail}(L)$ と略記します. それから語 $B_1 \cdots B_m$ と語 $C_1 \cdots C_n$ が与えられたとき, 語 $B_1 \cdots B_m$ に含まれる各 $B_i (1 \le i \le m)$ を語 $C_1 \cdots C_n$ から取り除いた語を $C_1 \cdots C_n \setminus B_1 \cdots B_m$ と表記します. たとえば, $abcd \setminus bd$ は ac です. また, 検索で一度出たクラスを再度検索する必要はありません. このことから語 '$\cdots W_(i-1) W_i W_(i+1) \cdots W_(j-1) W_i W_(j+1) \cdots$' は語 '$\cdots W_(i-1) W_i W_(i+1) \cdots W_(j-1) W_(j+1) \cdots$' と検索順序として一致します. このように後続の一致する語を除いたものとの関係を '$\cdots W_(i-1) W_i W_(i+1) \cdots W_(j-1) W_i W_(j+1) \cdots \searrow \cdots W_(i-1) W_i W_(i+1) \cdots W_(j-1) W_(j+1) \cdots$' 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. この作用素 $\mathcal{L}$ は次の性質を持ちます:

作用素 $\mathcal{L}$ の性質

1. $\mathcal{L}(C) = C$
2. $\mathcal{L}(C_0(C_1)) = C_0 + \mathcal{L}(C_1)$
3. $L_1 \searrow L_0$ のとき $\mathcal{L}(L_1) = \mathcal{L}(L_0)$
4. $\mathcal{L}(C(B_1, \ldots, B_n)) = C + \mathcal{M}(\mathcal{L}(B_1), \cdots, \mathcal{L}(B_n))$

1. は継承関係を持たないクラス C のときに MRO は C のみであることを示します. つぎの 2. は $C_0 \overset{\sigma}{\to} C_1$ 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 $*$ の値は不変であることを示し, 最後の 4. は多重継承のときの処理です. 次に作用素 $\mathcal{M}$ を導入しておきましょう. この作用素 $\mathcal{M}$ の働きは, 継承関係を遡る MRO を基本に多重継承のあるクラスで継承を示すタプルをそのまま用いて基底クラスの検索順序を入れるというものです. つまり, クラスの属性で基底クラスを示すタプルの左側から順番に先祖を辿る方法:「**深さ優先, 左から右の順番規則 (left-to-right depth-first rule)**」と呼ばれる規則になります. この操作を表現する作用素 $\mathcal{L}$ は次の性質を持ちます:

作用素 $\mathcal{M}$ の性質

a. $\mathcal{M}(W) = \underline{W}$
b. $\mathcal{M}(\cdots, L, \cdots) = \mathcal{M}(\cdots, \underline{L}, \cdots)$
c. $\mathcal{M}(L_1, L_2, \cdots, L_n) = \underline{L_1} + \mathcal{M}(L_2 \setminus L_1, \cdots, L_n \setminus L_1)$

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 $\mathcal{M}$ の値は不変であることを示します. 最後の c. は引数の最も左側にある経路を外に出し, その語に含まれるクラス名を他の引数から除

去する処理方法を示しています．もし, 引数の語に共通するクラス名がなければ $\mathcal{M}$ は語に対する和になるだけです．

古典的クラス型でメソッド検索で用いられる順序は MRO ですが, この手法は多重継承で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:

$$\begin{tikzcd} & A & & \text{def save(self) : ....} \\ B \arrow[ur, "\sigma"] & & C \arrow[ul, "\sigma"'] & \text{def save(self) : ...} \\ & D \arrow[ul, "\sigma"] \arrow[ur, "\sigma"'] & & \end{tikzcd}$$

この図式は, クラス D はクラス B と C の双方を継承する多重継承の関係にあり, 同時にクラス B と C はクラス A の派生クラスで, クラス A と C でメソッド save が定義されていることを表現しています．ここで各クラスが古典的クラス型のときにクラス B や クラス C でメソッド save() を利用しようとすればクラス A のものがそのまま用いられ, クラス C で上書きされたメソッド save() はそのままではクラス D で用いられません．ここで実際に MRO を計算してみましょう:

菱形状の継承関係の RMO

$$\begin{aligned} \mathcal{L}(B) &= BA \\ \mathcal{L}(C) &= CA \\ \mathcal{L}(D) &= D + \mathcal{M}(\mathcal{L}(B), \mathcal{L}(C)) \\ &= D + \mathcal{M}(BA, CA) \\ &= D + BA + \mathcal{M}(CA - BA) \\ &= DBA + \mathcal{M}(C) \\ &= DBAC \end{aligned}$$

このように古典的クラスの属性検索 (MRO) ではクラス D, B, A, C, A の順番で検索が行なわれますが, この属性検索では D, B の次のクラス A のメソッド save() が発見された時点で検索が終了し, その結果, クラス A で定義されたメソッド save() が用いられてクラス A の派生クラス C で再定義されたメソッド save() が用いられません．すなわち, より近い側のメソッドが用いられないという問題が生じます．そのためにクラスタイプでは「**C3 MRO(Method Resolution Order)**」[*46]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

[*46] https://www.python.org/download/releases/2.3/mro/ や PEP-253 を参照

素 $\mathcal{M}_{c3}$ で置換えて, 次のようにまとめられます:

C3 での作用素 $\mathcal{L}$ の性質

1. $\mathcal{L}(C) = C$
2. $\mathcal{L}(C_0(C_1)) = C_0 + \mathcal{L}(C_1)$
3. $L_1 \searrow L_0$ のとき $\mathcal{L}(L_1) = \mathcal{L}(L_0)$
4'. $\mathcal{L}(C(B_1, \ldots, B_n)) = C + \mathcal{M}_{c3}(\mathcal{L}(B_1), \cdots, \mathcal{L}(B_n))$

作用素 $\mathcal{M}_{c3}$ は次の性質を持ちますが, 最初の a., b. は作用素 $\mathcal{M}$ の場合と同様, また c. の計算手順は $\mathcal{M}$ よりも階層を意識した検出方法に代ります:

作用素 $\mathcal{M}_{c3}$ の性質

a. $\mathcal{M}_{c3}(W) = \underline{W}$
b. $L_0 \searrow L_1$ のとき $\mathcal{M}_{c3}(\cdots, L_0, \cdots) = \mathcal{M}_{c3}(\cdots, L_1, \cdots)$
c. $\mathcal{M}_{c3}$ の計算は後述の方法で計算される.

$\mathcal{M}_{c3}$ の計算手順を $\mathcal{M}(L_1, L_2, \cdots, L_n)$ が与えられたときにどのように行われるかを纏めておきます:

$\mathcal{M}_{c3}$ 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 $k \in (1, \ldots, n)$ 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, $h_i + \mathcal{M}_{c3}(L_1, \cdots, L_n)$ を作用素 $\mathcal{M}_{c3}$ の結果として返却する.
4. $h_i \in \underline{L_j}(i \neq j)$ のとき $i \neq n$ ならば $i = i + 1$ として 2. に戻る. もし $i = n$ であればエラーを出力して処理を終える.

作用素 $\mathcal{M}_{c3}$ は引数の語との間に共通するものが何もなければ最初の作用素 $\mathcal{L}$ を拡張したものと同様に語の和として作用します. しかし, 共通する語が現われたときの処理がクラスの階層を合致させる働きになります. このことを先程の菱形状の継承関係で C3 MRO を計算することで確認してみましょう.

菱形状の継承関係の C3 RMO

$$
\begin{aligned}
\mathcal{L}(B) &= BA \\
\mathcal{L}(C) &= CA \\
\mathcal{L}(D) &= D + \mathcal{M}_{c3}(\mathcal{L}(B), \mathcal{L}(C)) \\
&= D + \mathcal{M}_{c3}(BA, CA) \\
&= D + B + \mathcal{M}_{c3}(A, CA) \\
&= DB + C + \mathcal{M}_{c3}(A, A) \\
&= DBCA
\end{aligned}
$$

C3 MRO では $DBCA$ と MRO の $DBAC$ と異なりクラス C の方が大本のクラス A よりも先に検索が行われるためにより新しいクラス C のメソッド save() が用いられて MRO よりも妥当な結果が得られます．さらに C3 MRO の長所は間違った継承関係が検出できることです．たとえば次の継承関係を想定しましょう:

$$
\begin{tikzcd}
X & & Y \\
Y \arrow[u, "\sigma"] & & X \arrow[u, "\sigma"'] \\
& A \arrow[ul, "\sigma"'] \arrow[ur, "\sigma"] &
\end{tikzcd}
$$

この継承関係は $X \overset{\sigma}{\to} Y$ かつ $Y \overset{\sigma}{\to} X$ と, クラスの定義では相互参照的な関係, いわゆる循環的な定義であり, このような定義は Python では間違った定義です．しかし, MRO ではエラーではなく AYX が得られ, 一方の C3 MRO では

$$
\begin{aligned}
\mathcal{L}(Y) &= YX \\
\mathcal{L}(X) &= XY \\
\mathcal{L}(A) &= A + \mathcal{M}_{c3}(\mathcal{L}(Y), \mathcal{L}(X)) \\
&= A + \mathcal{M}_{c3}(YX, XY)
\end{aligned}
$$

と計算が進むものの $\mathcal{M}_{c3}(YX, XY)$ の処理で YX の Y が XY に含まれるために YX の処理が行えず, 今度は XY から X を取り出す処理に移ります．しかし, この X も前の YX に含まれるために XY からもクラスを取り出せずに作用素 $\mathcal{M}_{c3}$ はエラーを出力しなければなりません．このように間違った継承関係の図式が与えられていても C3 MRO では適切な処理が行えることを意味します．

3.11　名前空間とスコープ

3.11.1　名前と名前空間

■名前 (name):　オブジェクトの参照で用いられる識別子で, 次の EBNF を持ちます:

名前の BNF

名前 ::= [識別子 分離記号] 識別子

この名前への束縛 (binding) でオブジェクトと名前が結び付けられ, その結果, その名前の指示でオブジェクトへの参照が行われます. なお, 参照すべき名前が名前空間に存在しないときに送出される例外が「**NameError**」, 名前が名前空間に存在していても参照すべきオブジェクトが結び付けられていない変数を参照したときに送出される例外が「**UnboundLocalError**」です.

■名前空間 (name space): 統一的に名前を決定する手法です. Python は本体を小さくして必要に応じてモジュールで拡張する方式を採用していますが ここで複数のモジュールを読込む必要があったときに階層なしに名前をそのまま展開すると, モジュール A で定義した函数 func() とモジュール B で定義した函数 func() と同名の函数が存在するために「**名前の衝突**」と呼ばれる事態になります. ここで Python 上に展開する名前にモジュールに依存する識別子を付けてモジュール単位で区別するという機械的な方法にするとどうでしょうか? この方法で名前の衝突を避けられるだけではなく, オブジェクトの検索も階層構造が入ることで検索範囲が狭まるために探し易くなります. このように名前の不用意な衝突を避け, オブジェクトの検索を行う範囲を定める仕組みが名前空間です.

■ブロック (block): プログラムで一つの実行単位になる区画であり, モジュール, クラスと函数定義はブロックです. そして, Python のシェルを経由して対話的に入力された個々の命令もブロックです.

■コードブロック (code block): スクリプトファイル, スクリプト命令, 組込函数 eval() や exec() に引き渡した文字列, 函数 input() から読み取られて評価される Python の文で構成され, これらコードブロックは「**実行フレーム (execution frame)**」上で実行されます.

■スコープ (scope): 参照される名前の範囲のことで, Python のスコープにはモジュールの大域的なスコープと函数内部の局所的なスコープの二種類のみです. メソッドのコードブロックを含む拡張は行われません. その例としてリファレンスマニュアルでは生成子を一例として挙げています:

```
class A:
    a = 42
    b = list(a + i for i in range(10))
```

この例では名前 b に束縛する構築子 list() の引数が生成子 (generator) で, ここではリストの成員の型を for 節を使って表記するリストの内包表現による生成に対応します. そして, この生成子内部で名前 a への参照がありますが, 名前 a は構築子 list() のブロック外

部にあるためにスコープ外になり, 名前 a への束縛が行われていてもエラーになります:

```
>>> class A:
...     a=42
...     b=list(a+i for i in range(10))
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in A
  File "<stdin>", line 3, in <genexpr>
NameError: global name 'a' is not defined
```

名前がコードブロック内部で用いられているときに名前の参照で近傍のスコープを用いられます. ここで**ブロックの環境**とは, ある一つのコードブロック内で参照可能なスコープの全ての集合のことです.

名前があるブロック内部で束縛されているとき, その名前はそのブロックの局所変数になります. 名前がモジュールで束縛されているときは大域変数になります. そして, コードブロックで用いられていても, そのブロックで定義や束縛が行われていないときに, その変数は自由変数となります.

ここで名前の参照を行った際に, その名前に束縛されたオブジェクトがないと例外:NameError が, 局所変数で, 名前が束縛されていない変数の参照を行ったときに例外:UnboudLocalError が創出されます. 例外:UnboundLocalError は例外:NameError の派生クラスです. なお, del 文で指定された対象は, del 文の目的が対象の束縛の解除であるものの, 束縛済みのものと見做されます.

import 文や代入文は, クラスや函数定義, モージュールレベル内で行われます.

global 文で指定された名前がブロック内にあるとき, その名前は名前空間の最上層で束縛された名前の参照を行います.

3.12 例外

3.12.1 例外の概要

「**例外 (exception)**」はプログラムの処理時に何らかの原因で生じた「**異常**」とされる状況で, プログラムの本質的な欠陥, 開くべきファイルがないといった前提から外れた状況から生じた状況, さらには利用者側の理由 (受け取るはずのメールがまだ送られていない等) でプログラムを一時的に停止させて別の処理を実行させるために意図的に発生させる信号も含まれます. 意図的に発生させた例外は「**利用者定義例外 (user defined exception)**」と呼ばれ, その生成には raise 文が用いられます. また, 例外を分析して別の処理につなげる処理は「**例外処理**」と呼ばれ, Python では try 節で例外が生じ得る処

理, except 節で対応する例外への処理, finally 節で後処理 (clean up) を記載します．なお, finally 節は try 文に一つだけ try 文の末端に記載可能で, finally 節の処理は try 文の最後に必ず実行されます.

3.12.2　BaseException クラスについて

例外のクラスが BaseException クラスです．この BaseException クラスの組込の派生クラスを以下に示します:

- SystemExit: 函数 sys.exit() が送出する例外です．この例外が処理されなければスタックのトレースバックを全く表示することなくインタプリタが終了します．この関連値が通常の整数であればシステム終了ステータスを函数 exit() に渡して表示します．この値が None であれば終了ステータスは 0 になります．文字列のような他の型であれば, そのオブジェクトの値が表示されて終了ステータスが 1 になります. このクラスのインスタンスは属性 code を持ち, この値は終了ステータスまたはエラーメッセージ (既定値は None) に設定されています．なお, この例外は厳密に異常ではないために BaseException の派生クラスです．ここで函数 sys.exit() は後処理 (try 文の finally 節) が実行されるようにするため, またデバッガが制御不能になるリスクを冒さずにスクリプトを実行できるようにするために例外に翻訳されます．即座に終了する必要があるときに, たとえば, 函数 os.fork() を呼んだあとの子プロセス内で函数 os._exit() で実行できます．このクラスは BaseException を直接継承することで着実に呼出し元に伝わりってインタプリタを終了させせられます.
- KeyboardInterrupt: 利用者が $\boxed{\text{Control-C}}$ または $\boxed{\text{Delete}}$ を押したときに送出される例外です．割込の有無はインタプリタの実行中に定期的に調べられ, 組込函数 input() や raw_input() が利用者の入力を待っている間に割込みキーを押しても, この例外が送出されます．Exception クラスの例外を処理するコードに誤って捕捉されないように, このクラスは BaseException クラスを継承しています.
- GeneratorExit: generator 型のメソッド close() が呼び出されたときに送出される例外です． この例外は厳密な意味で異常ではないために, 一般の例外である Exception とは別のクラスです.
- Exception: 一般的な例外を表現します．このクラスの詳細は次の小節で述べます.

3.12.3　Exception クラスについて

Exception クラスは BaseException クラスの派生クラスです．利用者定義の例外クラスを構築するときは Exception クラスの派生クラスとして構築します.

Exception クラスの組込の派生クラスに StandardError, StopIteration と Warning があります.

StandardError クラスについて

StandardError クラスの派生クラスを以下に示しておきます:

Exception クラスの派生クラス

ArithmeticError	AssertionError	AttributeError	BufferError
EnvironmentError	EOFError	ImportError	LookupError
MemoryError	NameError	ReferenceError	RuntimeError
SyntaxError	SystemError	TypeError	ValueError

■ArithmeticError: 算術演算処理で送出される例学で, その派生クラスに OverflowError, ZeroDivisionError と FloatingPointError があります:

- OverflowError: 算術演算の結果が表現できない大きな値になったときに送出されますが, Python の整数処理で生じることはほとんどなく MemoryError が送出されるでしょう. C の浮動小数点演算の例外処理が標準化されていないために浮動小数点演算のほとんどが検査されません.

```
>>> try:
...     math.exp(10000)
... except OverflowError:
...     print "Too big!!"
...
Too big!!
```

- ZeroDivisionError: 除算や剰余で 0 による除算が発生したときに送出されます. 関連値はその演算における被演算子と演算子の型を示す文字列です:

```
>>> try:
...     1/0
... except ZeroDivisionError:
...     print "1/0!!!"
...
1/0!!!
```

- FloatingPointError: 利用している Python が '--with-fpectl' オプションを有効にしてコンパイルされているとき, あるいは pyconfig.h ファイルに WANT_SIGFPE_HANDLER が定義されているときに限って利用可能な例外で, プロセッサで IEEE 754 浮動小数点エラーが発生したときに出力されます. モ

ジュール fpectl はスレッドセーフではなく, 利用上の危険もあるため, 余程のことがない限り, この例外に関わることはないでしょう.

■BufferError: バッファ関連の処理に失敗したときに送出されます. 組込の派生クラスはありません.

■AssertionError: assert 文が失敗したときに送出されます. 組込の派生クラスはありません.

```
>>> x = 1
>>> try:
...     assert x>1
... except AssertionError:
...     print x
1
```

■AttributeError: 属性参照や代入が失敗したときに送出されます. なお, 参照しようとしたオブジェクトが属性値の参照や属性値の設定を提供していないときは TypeError が送出されます. 組込の派生クラスはありません.

```
>>> x = 1
>>> try:
...     x.test
... except AttributeError:
...     print x
1
```

■EnvironmentError: システム環境で生じる例外で, IOError と OSError がその組込派生クラスです. この型の例外はタプルで生成され, その長さが 2, あるいは 3 のタプルであれば第 1 成分はエラー番号に対応し, インスタンスの属性 errno で得られます. 第 2 成分は属性 strerror で得られ, その値はエラーに関連するメッセージです. タプルに第 3 成分があるときは, この第 3 成分が属性 filename で得られます. これらのタプルの成分は属性 args からも得られますが, 互換性のために属性 args 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 以外のときの属性 erno, strerror, filename の値は None です.

- IOError: 関数やメソッドによるファイルオブジェクト等の I/O 操作が「**ファイルが存在しない**」や「**ディスクに空き領域がない**」等の I/O に起因する理由で失敗したときに送出されます.
- OSError: 関数やメソッドがシステムに起因する異常を返したときに送出されます.

ここで属性 errno はモジュール errno に基づく数字のエラーコード, 属性 strerror は C の函数 perror() で表示されるような文字列です．この OSError クラスには VMSError と WindowsError の二つの組込例外の派生クラスがあります:

- ▲ VMSError: VMS でのみ利用可能な例外で VMS 特有のエラーが起こったときに送出されます.
- ▲ WindowsError: MS-Windows 特有のエラー, あるいは, エラー番号が errno 値に対応しないときに送出されます．winerrno, strerror の値は Windows プラットフォーム API の函数 GetLastError() と FormatMessage() の返却値から生成され, errno の値は数値で表現された winerror の値を基にファイル errno.h から対応する値を取り出したものです.

■EOFError:　ファイルの終端 (EOF) に到達したときに送出される例外です．なお, メソッド file.read() と file.readline() はこの例外ではなく空の文字列を返却します．組込の派生クラスはありません.

■ImportError:　import 文でモジュールが見つけられなかったときや from ... import 文で指定した名前を取り込めなかったときに送出されます．組込の派生クラスはありません.

■LookupError:　連想配列 (mapping) や列 (sequence) で要素の指定で用いた鍵や添字が範囲外のときに送出される例外で, メソッド codecs.lookup() で直接送出されることもあります．IndexError と KeyError が組込の派生クラスです:

- ▲ IndexError: 指定した添字がその範囲を超えているときに送出されます．なお, 添字が整数でないときは例外 TypeError です．また, スライスの添字で添字の範囲内に収まるように自動調整されるために IndexError は送出されません.
- ▲ KeyError: 連想配列の鍵が存在する鍵集合内に包含されなかったときに送出されます.

■MemoryError:　プログラムの処理の過程でメモリが不足したときに送出される例外で, 返却値はどの内部操作でメモリ不足が生じたかを示す文字列です．オブジェクトをいくつか消去すれば復旧可能なときもありますが, プログラムの暴走でも実行スタックの追跡結果が出力できるように, この例外が送出されます．組込の派生クラスはありません.

■NameError:　局所, 大域の双方で指定した名前が存在しなかったときに送出されます．関連値は見つからなかった名前を含むメッセージです．UnboundLocalError が組込の派生クラスです:

▲ UnboundLocalError: 値が束縛されていない函数, メソッド内の局所変数への参照を行ったときに送出されます. 組込の派生クラスはありません.

■ReferenceError:　メソッド weakref.proxy() で生成された弱参照 (weak reference) プロキシを使って塵収集 (GC) で回収されたあとのオブジェクト属性を参照しようとしたときに送出されます. 組込の派生クラスはありません.

■RuntimeError:　他のカテゴリに分類できない異常が検出されたときに送出されます. 関連値は何が問題であるのかをより詳細に示した文字列です. NotImplementedError が組込みの派生クラスです:

▲ NotImplementedError: 未実装であることを示す例外です. 利用者定義の基底クラスで抽象メソッドが派生クラスで上書きされることを要求するときに, この例外を送出しなくてはなりません. 組込の派生クラスはありません.

■SyntaxError:　Python の構文解析器が構文エラーを検出したときに送出される例外で, import 文, exec 文、組込函数 eval() と input(), 初期化スクリプトの読込と標準入力で対話的な処理でも生じます. このクラスのインスタンスは例外の詳細に簡単にアクセスできるようにするために属性 filename, lineno, offset, text があります. 例外インスタンスに対するメソッド str() はメッセージのみを返します. IndentationError が組込の派生クラスです:

▲ IndentationError: 字下の構文エラーに対応する基底クラスです. TabError が組込の派生クラスです:

♡ TabError: 字下げで TAB と Space を一貫した順序で並べていないときに送出されます. 組込の派生クラスはありません.

■SystemError:　インタプリタが軽度の内部エラーを発見したときに送出されます. 関連値は下位層の言葉でどのような問題があるのかを示す文字列です. Python の作者, インタプリタを保守している人にこの例外を報告してください. この際にインタプリタのバージョン (sys.version; 対話的セッションを開始した際にも出力されます), 正確なエラーメッセージ (例外の関連値), そして, 可能ならエラーを引き起こしたプログラムのソースコードを報告してください. 組込の派生クラスはありません.

■TypeError:　函数やメソッド等にて引き渡されたオブジェクトで型の不整合が生じたときに送出される例外です．関連値は詳細を述べた文字列です．組込の派生クラスはありません．

■ValueError:　組込の演算や函数で, 型は適合していても不適切な値を受け取ったときや IndexError のような詳細な例外では説明のできない状況で送出されます．UnicodeError クラスが組込みの派生クラスです:

- UnicodeError: 符号化/復号化で Unicode に関する異常が発生したときに送出されます．このクラスには異常を説明する次の属性があります:
  - encoding: 異常を送出したエンコーディングの名前．
  - reason: 異常を説明する文字列．
  - object: 符号化/復号化しようとしたオブジェクト．
  - start: オブジェクトの最初の無効なデータの添字．
  - end: オブジェクトの最後の無効なデータの添字の次の整数値．

  派生クラスとして次のクラスがあります:

  ▲ UnicodeEncodeError: エンコード中に Unicode 関連の異常が発生したときに送出されます．

  ▲ UnicodeDecodeError: デコード中に Unicode 関連の異常が発生したときに送出されます．

  ▲ UnicodeTranslateError: コード翻訳に Unicode 関連の異常が発生したときに送出されます．

3.12.4　StopIteration

それ以上要素がないことを知らせるためにイテレータ (iterator) のメソッド next() により送出されます．この状況は通常は異常とみなさないために StandardError ではなく Exception から派生します．

3.12.5　Warning

警告の基底クラスで, その派生クラスとして以下の警告があります:

- UserWarning: 利用者が構築したコードで生成される警告．
- DeprecationWarning: 廃止された機能に関する警告．

- PendingDeprecationWarning: 将来廃止される予定の機能に関する警告.
- SyntaxWarning: 曖昧な構文に関する警告の基底クラス.
- RuntimeWarning: ランタイムの挙動に関する警告.
- FutureWarning: 将来, 意味や構成に変更がある文に対する警告.
- ImportWarning: モジュールの読込の誤りと思われるものに関する警告.
- UnicodeWarning: Unicode に関連する警告.

3.13 Python プログラム作成の流儀

SageMath のプログラム作成の流儀は Python のプログラムの書き方に関連するプロセス PEP の PEP-8 と文書文字列 (docstring) に関連する情報 PEP の PEP-257 に従います. ここではまず Python コードの様式案内という表題の PEP-8 を紹介しましょう. この PEP は「**愚かな一貫性は小人物に憑いたおばけである**」を見出に持ち, プログラムというものは書かれる頻度以上に読まれる頻度が高いこと, このガイドラインの目的が, プログラムの可読性を高めて広範囲の Python プログラムに施策を一貫させることにあると明言されており, その内容は Python 以外の言語でも有用なものです. では, PEP-8 の骨子を以下に纏めておきましょう:

■一行の行数について: 一行の最大長は 79 文字とします.

■字下げについて: 字下げの一段は Space の 4 文字分とし, Space 以外の空白文字, 特に Tab の混在を推奨しません.

■空行の入れ方: トップレベルの函数とクラス定義の間は 2 行空け, クラス内部でのモジュール定義の間には一行空けます. 函数内部でも論理的な区分を明瞭にするために空行を用います.

■ファイルエンコーディングについて： ASCII, または Latin-1 エンコーディング (ISO-8859-1) が望ましく, Python 3 以降は UTF-8 が望ましいとされています. もし, 文字列に非 ASCII 文字列が含まれているときは “\x”, “\u”, “\U” 等のエンコーディングを示す記号を文字列の先頭に置いて文字コードを明示します.

■import 文の書き方: import 文で読込むパッケージは個別に読込むべきです. パッケージ内部の import 文は相対ではなく絶対パッケージパスを用べきです. これは名前空間を混乱させる原因を除外するためです. ここで悪い例を示しておきましょう:

悪い例

```
from test import *
```

この import 文の使い方ではモジュール test に含まれる函数や変数になんらの識別子が付かないために既存の名前と衝突が生じる恐れがあります．もし既存の名前を置換えることが目的であれば問題ありませんが，そうでなければ名前の衝突が生じないように 'import test' でそのまま読込を行うか，あるいは as 節を用いて識別子を変更すべきです:

良い例

```
import test as tst
```

この例では test に 'neko' という函数を定義していれば最初の import 文の例では Python 上で 'neko' という名前の函数を定義していれば test パッケージの函数 'neko' で書き換えが生じます．後者の import 文で読み込んだ場合は test パッケージで定義された函数 'neko' は 'tst.neko' という名前が与えられ，既存の函数 'neko' との名前の衝突が生じません．なお，ここでの識別子は import 文で as 節がなければパッケージの名前，as 節があればそこで指示された文字列です

■空白文字 SPACE の利用:　無駄な SPACE の利用を避けます．たとえば，演算子や式を合せるための SPACE の挿入を避けるべきです．ただし，可読性のために二項演算子の両端に SPACE を一つのみ入れたり，算術演算子の前後に SPACE を入れるべきです．ただし，記号 "=" をキーワードやパラメータの既定値として用いるときは前後に SPACE を入れないようにします．

■註釈の書き方:　ソースコードと矛盾する註釈は註釈がないとき以上に問題になります．そのためにソースコードを変更すれば註釈の更新も優先して行うべきです．なお，短い註釈では最後のピリオドを省略します．また英語を母国語としなくても記述したプログラムがさまざまな言語の利用者に読まれる可能性があれば英語で記述すべきです．

■文書文字列 (docstring) の書き方:　PEP-257 に準拠します．全ての公開モジュール，函数，クラス，モジュールには必ず文書文字列を def 節の直後に記載します．文書文字列が複数行にわたるときは最後の引用符 '"""' を単独の行に記載します．

■バージョンの記録:　ソースファイル内部に Subertsion, CVS, RCS 等のバージョン情報を持たせる必要があるときは以下のように記述すべきです:

```
__version__ = "$Revision$"
# $Source$
```

これらの行はモジュールの文書文字列よりもうしろで，他のソースコードよりも前に空行で前後を分けて記述します．

■命名規則:　推奨の命名規則がありますが, 既存のライブラリがそれと異なる書式で記載されていれば内部の一貫性を優先して命名規則に従う必要がありません. 代表的な命名方法を以下にまとめておきます:

命名規則

b	小文字一文字.
B	大文字一文字.
lowercase	小文字のみ.
lcs_w_underscores	小文字列を '_' で繋げたもの.
UPPERCASE	大文字のみ.
UCS_WITH_US	大文字列を '_' で繋げたもの.
CapitalizedWords	ラクダの瘤表記 (各語の先頭文字のみを大文字で結合.).
mixedCase	小文字と大文字の混合
CWs_With_Us	各語の先頭文字のみを大文字で '_' で結合.

なお, 避けるべき命名として, 小文字の 'l', 大文字の 'o', 大文字の 'I' を 1 文字の変数名にすることを挙げています. これらは '1' や '0' と混同し易いためです. その他の命名規則を以下に記しておきます:

- モジュール名: 記号 "_" を含まない小文字のみとします. これは, Python のモジュール名はファイル名に反映されるために OS のファイル名の制約, たとえば, 大文字小文字の区別をしないこと[*47]や長い名前は短縮されるといった制約を受ける可能性を排除するためです.
- クラス名:「ラクダの瘤表記」を用います. また, 内部のみで利用するクラス名の先頭には記号 "_" を追加します.
- 例外名: 例外がクラスであるためにクラスの命名規則を適用しますが, このときに 'Error' という語をうしろに付けます.
- 函数名: 小文字のみ, あるいは可読性のために記号 "_" で語を区切ります. mixedCase は互換性を保つことを目的とした利用に限定します.
- 大域変数名: 函数と同様の規則で命名します.
- 函数やメソッドの引数: インスタンスメソッドの第 1 引数は必ず self で, 同様にクラスメソッドの第 1 引数を必ず cls にします.

[*47] MS-Windows の FAT, VFAT や NTFS, macOS の HFS+ は大文字と小文字の区別を行わないファイルシステムです. 大文字と小文字が混入した状態でファイル名が見えているからといって OS がそれらを区別している訳ではありません!

- 定数: 全て大文字で記号 “_” を使って語を分離します.

■継承のための設計: 属性名の命名規則を次に列記しておきます:

- 公開属性の先頭に記号 “_” を付けません.
- 公開属性の名前が予約語と衝突するときは名前の最後に記号 “_” を付けます.
- 公開データ属性は属性の名前を公開することが最善です.
- 継承を意図したクラスにサブクラスからの利用を望まない属性があれば, その属性名の先頭に記号 “_” を付けて末尾に記号 “_” が付けられないかを考えましょう.

Python ではクラスやインスタンスの属性は特に保護されないために, 実質的に属性は Java の public の扱いになります. この属性の管理を行うためにはディスクリプタ等を利用する必要があります.

■プログラミングでの推奨策: 注意すべきことを挙げておきます:

- プログラミングでは Python の実装の欠点を引き出さないようにすべきです.
- None と比較を行うときは演算子 “is” や “is not” を用いるべきです. また, ‘if x is not None’ という条件を ‘if x’ と記述しないようにします.
- クラスを用いた例外は文字列を用いた例外より望ましい.
- 例外を発行するときは ‘raise ValueError(’message’)’ を利用します. 例外の引数が長いときや書式を整えるときは括弧を使った表記にすべきです.
- 例外を捕らえるときに ‘except:’ で受け取るのではなく, どの例外を捕らえるようにするかを明示すること. ‘except:’ を用いるのは例外処理がトレースの結果を表示したり, ログに保存するときとコードが何かの後片付けをさせる必要があるときの二つに限定すると良いでしょう.
- すべての try/except 節で try 節には最低限のコードを記述します. バクの隠蔽を避けるためです.
- string モジュールではなく文字列メソッドを用います. 文字列メソッドの方が高速で, UNICODE 文字列と同じ API を共有しているためです.
- 接頭辞, 接尾辞を調べるときに文字列のスライス処理は避け, 函数 startswith() や endswith() を用います.
- オブジェクト同士の型の比較では函数 instance() を用い, 函数 type() 等で型を直接比較しない. たとえば, オブジェクトが文字列であるかを調べるときに, そのオブジェクトが UNICODE 文字列であるかもしれません.
- 列の処理では空の列が ‘False’ であることを利用します.

- Boolean を使った条件分岐で演算子 “==” は不要です．Boolean の等値性の判断が余計に入る冗長な処理になります．

3.14 SageMath の様式

3.14.1 SageMath の流儀

SageMath では Python の PEP をそのまま適用しません． SageMath の流儀は **http://doc.sagemath.org/html/en/developer/coding_basics.html** に記載されています．ここではその骨子を簡単に纏めておきましょう:

SageMath の流儀の骨子

- プログラム内の字下げに Space を 4 文字を用いて Tab は使いません
- 函数名が全て小文字であれば文字 “_” を用いて切り分けます
- クラスや主要な函数名では「**らくだのこぶ記法 (CamelCase)**」を用います

字下げは条件分岐や反復処理等にて，その構造を視覚的に明瞭にするための構文の一部であり，この字下げの文字数は SPACE のみの 4 文字が推奨されています．函数名は小文字のみであれば ‘set_some_value’ のように記号 “_” を区切記号として使った文字列にするか，あるいは ‘SetSomeValue’ のように大文字を適宜利用する「**らくだのこぶ記法**」を採用します．クラスや主要な函数名も ‘PolynomialRing’ のように「**らくだのこぶ記法**」を用いますが，この基準は絶対的ではなく，それ以上に判り易さを重視します．そのために函数名を「**らくだのこぶ記法**」にするのではなく「**大文字**」を採用することも許容します．たとえば，SageMath の開発者マニュアルに記載されているように **Matrix_integer_dense.LL** という上記の指針から外れる命名もあります．

3.14.2 ファイル名やディレクトリ名に関する指針

SageMath のファイルやディレクトリ名にも指針があります．これはディレクトリ名が英語の複数形であったとしてもファイル名は単数形にします．たとえば環を定義するファイルはディレクトリ ‘rings’ に格納され，定義ファイルは ‘polynomial_ring.py’ と表記します．ただし，ディレクトリ名を英語で複数にする必要はありません．たとえば，多項式環に関連するファイルはディレクトリ ‘rings/polynomial’ に収納されています．

3.14.3 SageMath のライブラリに関する指針

SageMath のライブラリファイルのヘッド部分は次の書式とします:

```
"""
```

```
<一 行 の 概 要>
<Paragraph description>
...
AUTHORS:
- <貴 方 の 名 前> (年-月-日): initial version
- <修 正 者 の 名 前>(年-月-日): 短 か い 説 明
...
- <修 正 者 の 名 前>(年-月-日): 短 か い 説 明
...
<沢 山 の 例 題>
"""
#*****************************************************************************
#       Copyright (C) 20xx 貴 方 の 名 前 <貴 方 のe-mail>
#
#  Distributed under the terms of the GNU General Public License (GPL)
#  as published by the Free Software Foundation; either version 2 of
#  the License, or (at your option) any later version.
#                  http://www.gnu.org/licenses/
#*****************************************************************************1
```

Python ではプログラム内部に記載される文書は非常に重要視され, SageMath でも「**一つの例題は幾千の言葉に優る**」とあります.

3.14.4 文書文字列の利用について

SageMath の**全ての**函数は「**文書文字列 (docstring)**」を持たなければなりません. Python の文書文字列の書き方は PEP-257 で明示的に示されています. Python の文書文字列は, 文字列を二つの二重引用符 “"” で ‘""三毛猫""’ のように括った文字の列としての構造を持ち, 函数やモジュールの入出力, 例題や参照の解説を行うために **INPUT:** や **OUTPUT:**, **EXAMPLE:** や **SEEALSO:** といった見出を用い, それらの記述方法に従います. ところで, この文書文字列を MATLAB やその類似の言語のような文字列のままにしておくのは現在の強力な処理能力を持つ計算機の能力を考慮すると勿体無い話です. むしろ, テキストファイルとしても可読性が高くてそれを処理することで一層の情報や文書としての質を上げられないでしょうか? このことを可能にするのが組版指示 (マークアップ, markup) 言語です. 古くは TEX や近年では HTML がありますが, これらは処理を行う以前のテキストファイルは命令やタグを利用するためにとても可読性が高いものとは言えません. この点を改良した言語が Markdown や reStructuredText(reST 等と略記) で, これらはテキストファイルである程度の WYSIWYG を実現しています. また, Markdown については簡単な例を §1.2.2 に示しています.

第 4 章

SageMath を Python 環境として使おう

4.1 Python 環境として見た SageMath

SageMath は Python 上で構成された数学向けの統合環境であり, 処理言語そのものも Python です. さらに近年では Anaconda や Homebrew を使って SageMath のインストールさえもできます. そこで, この SageMath を Python 環境として使えないかという素朴な疑問が生じます. 標準の SageMath を Python 環境として使えなくもありませんが, Python を利用する上で最大の魅力は Python で利用できるパッケージが充実していることですが, 残念ながら SageMath は Python のありとあらゆるパッケージを掻き集めた環境ではありません. そのために Python では通常に用いられるパッケージで意外なものが収録されてないこともあります.

そのパッケージの一例として, Python の統計分析でなくてはならないデータフレームを定義するパッケージの Pandas がありません. さらに SageMath では Python の数体系とは別の SageMath 独自の数体系を導入しているために, Python の浮動小数点数と SageMath の浮動小数点数は別物になっています. そのために SageMath を Python 環境として利用するためには工夫が必要になります.

4.2 pip によるパッケージ管理

通常の Python 環境であれば, その環境構築で用いた conda(minconda) や pip 等のパッケージマネージャを用いて必要なパッケージの導入や更新, 不要なパッケージの削除といったパッケージ管理ができます.

実は SageMath も pip を使ったパッケージ管理が可能で, 不足するパッケージのインストールといったパッケージ管理が可能です. SageMath 環境で pip によるパッケージ管理

では, 当然, SageMath を構築している Python の環境下で行わなければなりません. 確実に行う方法は SageMath を起動して, 力行 (コンソールから利用している場合はプロンプト「sage:」が出ている状態) で pip の先頭に記号 “%” を付けて, pip の命令文を入力します. たとえば, pip でインストールされているパッケージが何かを見たければ,

```
sage: %pip list
```

と通常の命令文「pip install」の pip の先頭に記号 “%” を付けて入力します. このようにすることで pip を使ったパッケージ管理が行えます. この方法は SageMath 独自のもので, 通常の Python 環境ではできません.

以下に pip の代表的な命令 (オプション抜き) を挙げておきます. pip の詳細に関しては「pip documentation」[*1] 等の WEB で公開されている情報を参照して下さい:

pip の主な命令

命令	構文	概要
pip install	pip install ⟨パッケージ⟩	パッケージのインストール
pip uninstall	pip uninstall ⟨パッケージ⟩	パッケージの削除
pip show	pip show ⟨パッケージ⟩	パッケージの概要確認
pip list	pip list	インストールしたパッケージの一覧を表示
pip inspect	pip inspect	Python 環境に関する情報を JSON 形式で出力

では, SageMath に幾つかのパッケージをインストールしておきましょう. ここでは統計解析で必須とも言える DataFrame を定義するパッケージ Pandas, 統計解析ソフト GNU R 向けに提供されている様々な統計データのサンプルの rdatasets, それに加えて, Python の高速化の手法として注目されている Numba で遊んでみることにしましょう.

ここで, numba, pandas と rdatasets を pip を使ってインストールする必要がありますが, たとえば, pandas のインストールを行う場合は次のように入力します:

```
sage: %pip install pandas
```

こうすることで SageMath に含まれる Python 環境にパッケージ Pandas がインストールされます. ここで, 通常の Python 環境であればインストール後に早速, import 文等でパッケージの読込が行えますが, SageMath の環境下では, SageMath そのものが Python

[*1] https://pip.pypa.io/en/stable/user_guide/

上で記述されたアプリケーションであるために SageMath の再起動を行った上で新規にインストールしたパッケージが利用可能になります．ここでは「%pip install rdatasets」で GNU R のサンプルデータを「%pip install numba」で Numba のインストールを行っておきましょう．

4.3 Numba について

Python はさまざまな分野で用いられるようになり，近年は統計解析や AI 関連でも用いられるのが主流になってきています．しかし，Python 自体は決して高速な処理言語ではありません．この弱点について，Python 自体の改良で徐々に改善はしているものの，Julia のように C++ と同程度の処理能力には至っていないのが現状です．そこで，SageMath では Python の処理速度の遅さの克服のために Cython を用いています．ただし，Cython を利用するためには Python のコードを C++ に移植する程ではないにしろ，Cython 向けに書き直す必要があります．別の方法としてパッケージ Numba の利用が注目されています．Numba[*2]は JIT(Just In Time Compilor によって Numba に対応した Python の函数やクラス等を LLVM 上のオブジェクトとして動作させることで，C/C++ や FORTRAN と同程度の処理が可能になります．Cython と比較して，デコレータを記述する程度であったりと，書き換えの手間も多くはありません．ちなみに，NumPy の ndarray 型のオブジェクトの属性 dtype から内包されているデータの型が判ることを Numba は利用しています．逆に言えは，それ以外の型を扱うときは効果が期待できないか，Numba そのものが利用できない場合もあり，Python 環境を中心で考えるのであれば Cython の利用も検討することになるでしょう．この Numba Project は Anaconda の開発元である Anaconda, Inc. と The Gordon and Betty Moore Foundation が支援しており，Anaconda にも包含されていますが，SageMath のパッケージには含まれていません．また，この LLVM を採用した言語として前述の Julia があり，この Julia の処理は非常に高速で，C と大差のない速度で処理が可能です．Python のパッケージも利用可能なこともあって注目されている言語です．ただし，パッケージや文書が揃った Python と比べてまだ発展途上の段階にあります．

ここでは Numba のホームページにある例を利用して簡単な確認をしておきましょう．まず．Numba の函数の記述です:

```
@numba.jit(nopython=True)
def sum2d(arr):
    """
    二次元配列arrの成分の総和をfor文で計算するプログラム.
    行列の総和はNumPyのsum()を使えば高速に処理が行えるが,
```

[*2] https://numba.pydata.org

```
    処理言語のオーバーヘッドを見るためである.
    """
    M, N = arr.shape
    """
    SageMath上で動作させる場合，数構造がPythonのそれとは異なる
    ために明示的にNumPyのfloat64であることを示す．そのために
    '0.0'ではなく'0.0r'と記述している.
    """
    result = 0.0r
    for i in range(M):
        for j in range(N):
            result += arr[i,j]
    return result
```

この例ではデコレータ@numba.jit を使って Python の函数を Numba でコンパイルすることを指示しています．当然, Numba の JIT に対応したオブジェクトでなければ Numba のご利益はありません．函数引数の arr が何であるかは明示されていませんが, NumPy の ndarray 型を想定しており, この ndarray には dtype が内包する数値の型になることを Numba は利用しています 一般の場合は函数の引数の型を jit の引数として指示します ([63] の例を参照)．ここで注意すべきことは, SageMath 上で記入した数値の型は Python の数値の型と異なることで, そのために SageMath 上で Python の数値計算パッケージの利用では処理する数値が NumPy の数値であることを明示的に表記しておくことが必須になる場合があります．今回の例では,「result = 0.0r」という記述がありますが, そうでなければ SageMath の数構造 (§5.7 参照) がそのまま Numba に引き渡されるために「**TypingError**」が発生します．逆にこの記述のままで Python 環境で動作させると Python で用いない表記であるためにエラーになります.

この動作確認を次の例で行ってみましょう．この例では倍精度浮動小数点数を成分とする 2 次元配列の総和を, Numba, NumPy の函数 sum(), メソッド sum(), Python 本体の函数 sum() と for 文で計算します．このときの 2 次元配列は 1×1 から $3^{10} \times 3^{10}$ の範囲で計算し, 縦軸を列数, 横軸を行数, 高さを処理速度の対数値として結果を 3 次元的に可視化します．この 3 次元的な可視化では contour3d() を使ってコンター図表示させますが, このときに高さで色分けを matplotlib のモジュール cm から cm.jet を用います．また, 図の右側にその色分けを示すカラーバーを配置します.

```
eps = np.finfo(float).eps
rslt = np.zeros((N0, N0, 5))
for i in range(N0):
    M = 3 ** i
    for j in range(N0):
        N = 3 ** j
```

```
        a = np.random.rand(M, N)
        # 0. sum2d()の場合
        sum2d(a)
        "計測開始"
        start = time.process_time()
        sum2d(a)
        "処理時間を配列rsltに収納"
        rslt[i, j, 0] = time.process_time() - start
        start = time.process_time()
        # 1. NumPyのsum()を用いた場合
        np.sum(a)
        rslt[i, j, 1] = time.process_time() - start
        start = time.process_time()
        # 2. NumPyのsum()をメソッドとして用いた場合
        a.sum()
        rslt[i, j, 2] = time.process_time() - start
        start = time.process_time()
        # 3. Python純正のsum()を用いた場合
        sum(sum(a))
        rslt[i, j, 3] = time.process_time() - start
        s = 0
        # 4. 関数ではなく直接for文を用いた場合
        start = time.process_time()
        for m in range(M):
            for n in range(N):
                s += a[m, n]
        rslt[i, j, 4] = time.process_time() - start

from matplotlib import cm
T = ['SageMath: Numba + for', 'SageMath: Numpy sum as a function',
     'SageMath: Numpy sum as a method','SageMath: sum', 'SageMath:
        for']
x = np.arange(11)
y = np.arange(11)
for i in range(5):
    fig = plt.figure(figsize=(10,10))
    ax = plt.axes(projection='3d')
    X, Y = np.meshgrid(x, y)
    Z = np.log(rslt[..., i])
    mu = np.mean(Z)
    mx = np.max(Z)
    mn = np.min(Z)
    img1 = ax.contourf3D(X,Y,Z, cmap=plt.cm.jet, levels=20)
    ax.set_zlim([-13, 6])
```

```
plt.title(T[i]+": mean={:.2f}".format(mu)+", max={:.2f}".format
    (mx)+", min={:.2f}".format(mn))
plt.ylabel('Column ($3^x$)')
plt.xlabel('Row ($3^y$)')
plt.colorbar(img1)
plt.show()
```

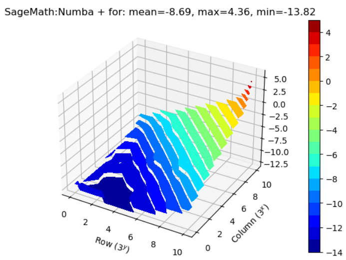


図 4.1　SageMath 上の Numba で for 文のみで総和

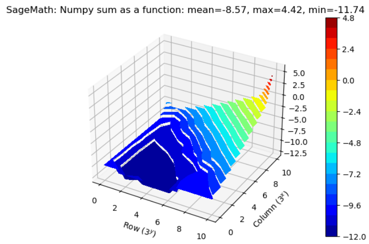


図 4.2　SageMath 上で NumPy の sum() を利用

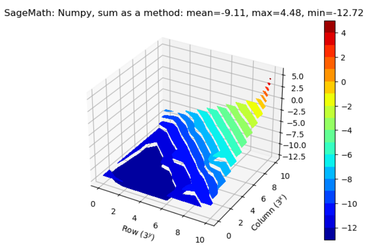


図 4.3　SageMath 上で NumPy の sum をメソッドとして利用

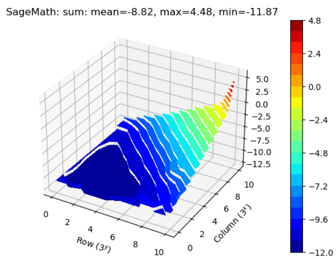


図 4.4　SageMath での函数 sum() を利用

ここでは SageMath 上で上記のプログラムを走らせた結果を確認しましょう．まず, 図 4.1 に Numba で for 文だけで行列の総和を計算した場合, 図 4.2 に NumPy の函数 sum() を呼出して計算した場合, 図 4.3 に行列オブジェクトのメソッドとして sum() を呼出した場合, 図 4.4 に SageMath の函数 sum() を呼出した場合, 最後に左図 4.5 に SageMath の for 文だけで行列の総和を計算した結果で, グラフの各軸は対数軸で, これらのグラフのタイトルに処理時間の平均値, 最大値と最小値が表示されていますが, この数値も対数軸での数値です．

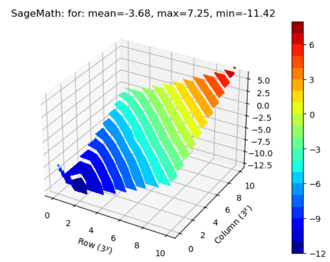


図 4.5　SageMath, for

グラフの形を見る限り, SageMath 上で for 文だけを使って総和を計算する図 4.5 の結果だけが他と異なった結果になっていることが判ります．さらにタイトルに表示されている計算処理時間を見ると NumPy の sum() と SageMath の sum() を使った 3 ケースでは挙動も非常に類似し, 平均値, 最大値, 最小値の結果も類似しています．このことから, SageMath の函数 sum() では NumPy の sum() が用いられていることが判ります．Numba 上の for 文は 100×100 以下の大きさの行列では NumPy よりも高速に処理ができているものの, それ以上の大きさになると NumPy がやや有利になることも判ります．

ところで, SageMath 上で for 文のみの場合はどうでしょうか? 図 4.5 は平均値も最大値も他の 3 ケースと比べて格段に大きなことが判ります．SageMath の for 文のみでは Numba を用いるかどうかで, このように大きな差異が出ており, このことからも Numba が処理の高速化に大きく役立っていると言えるでしょう．

では, 上記のプログラムを Python 上で動作させるとどうなるでしょうか? SageMath は Python 上で構築された環境であるとは言え, 様々なパッケージやライブラリを取り込んでいるために純粋な Python 環境とは別の結果が得られる可能性があります．そこで, Python 上で動作させるために函数 sum2d() 内部の註釈に書いたように, result の初期値設定を「0.0r」から Python の通常の書式の「0.0」に変更してプログラムを動作させた結果を示しておきましょう．また, タイトルの文字列も適宜変更しておきます．

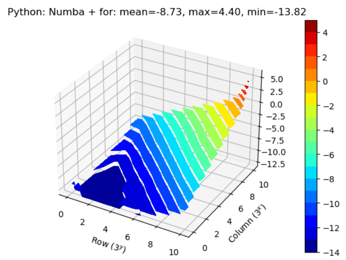


図 4.6　Python 上の Numba で for 文を利用

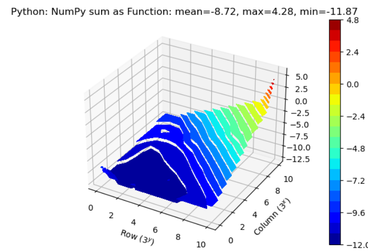


図 4.7　Python 上で NumPy の sum() を函数として利用

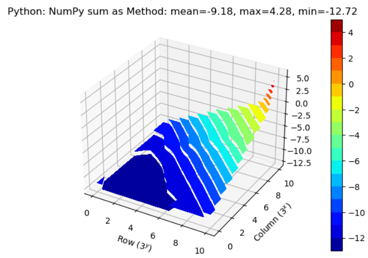


図 4.8　Python 上で NumPy の sum() をメソッドとして利用

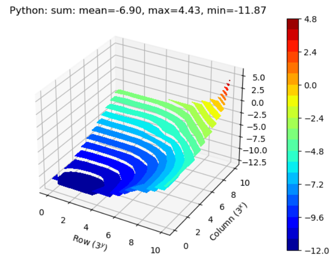


図 4.9　Python 上で函数 sum() を利用

SageMath 上では Numba と NumPy の sum() を函数として利用した場合とメソッドとして利用した場合, SageMath の函数 sum() を利用した場合の結果が類似した結果でしたが, Python の場合は函数 sum() を利用した場合に処理時間の平均値と最大値が Numba と二種類の NumPy の場合と比較して大きくなっています.

そして, Python の for 文だけで総和を計算した結果のグラフを図 4.10 に示しておきますが, こちらの傾向は SageMath の for 文のみと大差はありません. そして, 前述の Python の函数 sum() の結果は, for 文のみの場合と NumPy の場合の中間的な結果を示していると言えます.

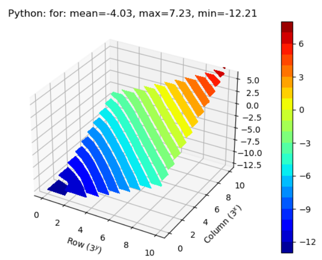


図 4.10　Python 上で for のみを利用

このように SageMath 上の函数 sum() と Python の函数 sum() の結果で違った傾向が出ていますが, SageMath の函数 sum() の結果が NumPy の結果に類似しているのに対して, Python の sum() の結果は SageMath や Python での for 文のみの結果にむしろ近い結果となっています. このことから, Python の函数 sum() は NumPy の函数 sum() のよ

うに数値行列の処理で最適化されたものではないと言えます.

また, Numba の結果もよくよく見ると, NumPy の各結果よりも行列のサイズが小さなものでは特に有効であり, 100×100 以上の大きさの行列に対しては NumPy よりも遅くなっていることが判ります. つまり, メモリの利用も含めて最適化された BLAS を使っている NumPy の利点が行列が大きくなるに従って現われていることも判ります.

これらの分布の傾向を別の見方で把握してみましょう. ここでは各結果の分散 σ^2 と平均 μ を計算し, 各結果を 2 次元平面上に点 $(-\frac{1}{2\sigma^2}, \frac{\mu}{\sigma^2})$ を描いてみましょう. ここで, 点 $(-\frac{1}{2\sigma^2}, \frac{\mu}{\sigma^2})$ を描く理由は, 分布を平均 μ, 分散 σ^2 の 1 次元の正規分布とすると, 領域 $\mathcal{X}$ での確率は $\frac{1}{\sqrt{2\pi\sigma^2}}\int_{\mathcal{X}} \exp\{-\frac{(x-\mu)^2}{2\sigma^2}\}dx$ で与えられますが, この正規分布の確率密度関数 $f(x) = \frac{1}{\sqrt{2\pi\sigma^2}}\exp\{-\frac{(x-\mu)^2}{2\sigma^2}\}$ の指数函数内部の多項式を展開すると

$$-\frac{1}{2\sigma^2}x^2 + \frac{\mu}{\sigma^2}x - \frac{\mu^2}{2\sigma^2}$$

が得られます. ここで x^2 と x の係数の対が $(-\frac{1}{2\sigma^2}, \frac{\mu}{\sigma^2})$ であり, これらの値で一意に正規分布が定まります. そのために分布の違いを把握する上で便利に使えます.

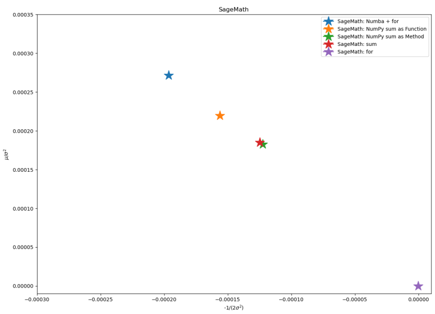


図 4.11　SageMath 上での結果を 2 次元平面上の点として可視化

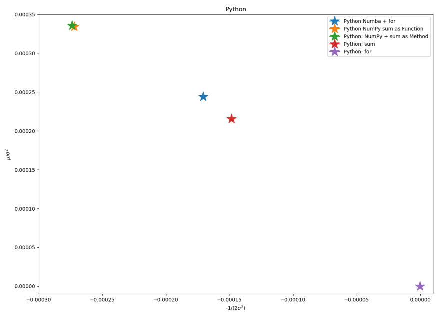


図 4.12　Python 上での結果を 2 次元平面上の点として可視化

これらのグラフからは, SageMath 上の Numba では分散が NumPy と比較して小さ目であるものの平均はやや大きいことが判り, SageMath の函数 sum() は NumPy の sum() をメソッドとしたものの結果と分散や平均がほぼ等しいことから, 浮動小数点数の計算で SageMath の函数 sum() で NumPy が用いられていると判断できます. そして, Python の函数 sum() は NumPy の sum() を使ったものとは何れも分布が異なり, むしろ, Numba に類似した分布であると言え, SageMath と Python の双方で for 文のみで行列の総和を計算する手法は共に分散が大きく, 行列のサイズが大きくなると NumPy 以上に急速に処理時間が増大していることが判ります.

これらの結果の可視化から, Numba による処理の高速化は SageMath でも Python の双方で有効であることに加え, SageMath は浮動小数点数の数値行列の計算で処理の高速化が図られた環境であると言えるでしょう.

4.4　アヤメデータを使った統計解析

次に GNU R のサンプルデータを使って統計的な分析を行ってみましょう. そのために前節の pip を使った手法で SageMath でも Python のパッケージ pandas で導入されたデータフレームと GNU R のサンプルデータをあらかじめインストールしておきましょう.

第 5 章

数学的対象の表現

5.1　SageMath の数学的対象の表現

SageMath は任意精度の数値計算を実現するために GMP 等の数値計算ライブラリを組込み, さらに多項式の表現で SymPy, 多項式の計算や基本的な解析では数式処理システムの Maxima, 可換環は Singular, 有限群は Gap, 数論は PARI, 処理速度を要求される倍精度浮動小数点数による数値行列計算は NumPy, 任意精度の数値演算では GMP と GNU MPFR ライブラリとさまざまなアプリケーションやライブラリを目的に応じて使い分けています. これらのアプリケーションやライブラリが実装している数学的対象や数学的構造を語る上で, それらの基礎になる整数や実数といった数が計算機内部でどのように実装されているかということは非常に大きな意味を持ちます. まず, 計算機内部で情報は '0' か '1' の bit で表現, つまり, 2 進数表現として符号化されます. しかし, 符号化の方法や計算機に実装されているメモリの制約から扱える数に上限と下限があり, それらを処理するときに CPU のレジスタの大きさ, さらにはデータの転送速度といったハードウェア上の制約もあります. したがって, 数の計算機内部の表現は可能な限りコンパクトで, 効率的な処理が行える構造が望ましいと言えます.

計算機内部で表現が容易な数は自然数です. 実際, 自然数は計算機内部で 2 進数で表現できるためです. そして, 自然数は多くの計算機言語で 'unsigned' の名称を持った符号を持たない整数型として実装されています. ところが, 日常生活では自然数では足らず, 負の数も必要ですが, この数は安易に 2 進数で表現できません. また, 1.1 のような小数点数を持つ実数も 2 進数で表現できません. そこで負の整数や実数を 2 進数で表現するための工夫が必要になります. このときに現実の制約として計算機のレジスタの bit 長が関わってじます. 現在の CPU のレジスタは 64 bit が中心で, かつては 8 bit,16 bit や 32 bit で, 計算機で扱える数の符号化もこのレジスタ長に関連する長さになります. たとえば, 整数では 32 bit のデータ長の 32 bit 整数, 64 bit データ長の 64 bit 整数があり, 実数の符号化では 32 bit データ長の単精度の浮動小数点数,64 bit データ長の倍精度浮動小数点数, 128 bit データの四倍精度浮動小数点数といったものがあります. ここで「n **bit 長**」とは 0 か 1 のいずれかの数のみが入れられる「n **個の箱の列**」と言い換えられますが, 64 bit の整数だからといって 64 個の箱が全て 1 の状態が整数 $2^{64}-1$ に対応しているとは限りません. なぜなら, 整数であれば整数としての, 実数であれば実数としての符号化を行わなければならないためで, どちらも自然数のような単純な 2 進数表現で済みません. 以降の節では整数の表現と実数の表現について解説しましょう.

5.2　計算機における整数の表現

5.2.1　補数を利用した負の整数表現

Python の int 型や long 型で負の整数も表現できていますが, それでは負の数をどのように内部で表現しているのでしょうか? たとえば -128 のような負の数には計算機内部で負の数であることを示すフラグを立ててしまえば良さそうですが, それよりも巧妙な方法があります. ここで正整数 a の負の整数 $-a$ は何よりも等式 $a+(-a)=0$ を充す数, つまり,「**元の数に足すと結果が 0 になる数**」です. ところで計算機内部では整数 a を一定の長さの長さの '0' と '1' の羅列の 2 進数で表現しなければなりません. そこで, 長さについては 64 bit, つまり, 64 個の 0 と 1 の列で考えます. まず, 0 はどのように表現すべきでしょうか？ これは 64 個の '0' で構成された 2 進数の列を 0 に対応させることが自然です. それから正整数は単純に 2 進数で表現し, その大きさを 64 個の箱に余裕で収まる範囲, 2^{63} より小さな数に制限します. これで式 $a+(-a)=0$ に現われる対象 a と 0 の計算機における表現ができました. では残りの正整数 a の負の正数 $-a$ はどう料理すべきでしょうか? 具体的に a を 2^{62} という整数で考えてみましょう. この数 2^{62} は $\boxed{0_{63}}\boxed{1_{62}}\boxed{0_{61}}\boxed{\dots}\boxed{0_0}$ で表現できます. ここで天下り的ですが, 正整数 a の 2 進数表現の各 bit を反転させた $\boxed{1_{63}}\boxed{0_{62}}\boxed{1_{60}}\boxed{\dots}\boxed{1_0}$ を考えます. この表現に対応する数は 2^{62} の「**1 の補数**」と呼ばれる数で, 元の数とその 1 の補数を足し合わせると $\boxed{1_{63}}\boxed{1_{62}}\boxed{1_{61}}\boxed{\dots}\boxed{1_0}$ と箱の中全てが 1 になる表現が得られます. この表現に 1 を加えると 2^{64}, すなわち $(1_{64})\boxed{0_{63}}\boxed{0_{62}}\boxed{0_{61}}\boxed{\dots}\boxed{0_0}$ が得られ, この箱の外に出た 1 bit を無視すれば, すなわち 2^{64} による剰余で考えれば, $\boxed{0_{63}}\boxed{0_{62}}\boxed{0_{61}}\boxed{\dots}\boxed{0_0}$, つまり 0 が得られます. このことは, 2^{62} の逆数 -2^{62} は 2^{62} の「**1 の補数**」に 1 を加えた数, つまり 2^{62} の「**2 の補数**」である $\boxed{1_{63}}\boxed{1_{62}}\boxed{0_{60}}\boxed{\dots}\boxed{0_0}$ で表現できることを意味します. この 2 の補数を利用する負の整数の表現では 64 番目の bit を正負の表現で利用するために正整数としては残りの 63 bit しか使えません. そのために 64 bit 長の整数は 64 番目の bit が '0' で他がすべて '1' になる数の符号化 $\boxed{0_{63}}\boxed{1_{62}}\boxed{1_{61}}\boxed{\dots}\boxed{1_0}$ に対応する正整数 $2^{63}-1=9223372036854775807$ が上限になります. しかし, その下限は 64 番目の bit のみが '1' で他が '0' となる符号化 $\boxed{1_{63}}\boxed{0_{62}}\boxed{0_{60}}\boxed{\dots}\boxed{0_0}$ が 2^{63} の「**1 の補数**」であるために 64 bit 整数の下限は $-2^{63}=-9223372036854775808$ になります. これを一般化すると n bit 長の整数は正整数が $n-1$ 個の bit を使って表現され, 上限が $2^{n-1}-1$, 下限は -2^{n-1} になります. このように自然数は「**神様がお創りになられた数**」[*1]だけあって計算機上でも「**自然**」に表現できますが,「**負の数というものは存在**

*1 自然数, 全部創ったのは神様, その他全部は人様の創作 (Die ganzen Zahlen hat der liebe Gott gemacht, alles andrere ist Meschenwerk.) - Leopold Kronecker

しない」[*2]ために補数を用いるという「**人工的な方法**」で表現しています．このように負の数も純虚数 i と同様に人間の想像上の産物です．

5.2.2　剰余類上の演算

このように計算機上で n bit 長の 2 進数を使うときは 2^n の剰余として整数を表現し，この 0 から 2^n-1 までの 2^n の剰余の整数の集合を $\mathbb{Z}_{(2^n)}$ と表記とします．計算機での整数の和 “+” や積 “×” はすべて $\mathbb{Z}_{(2^n)}$ で範囲で考えますが，計算結果がこの枠を超えるときは実装された計算機言語で異なる状況が生じます．多倍長精度の整数を持つ言語であれば計算結果が元のデータ長を超過すれば自動的にデータ長の延長が行われます．その結果，NumPy の配列で成分の整数型を ‘int64’ のように処理範囲を指定していれば，その枠内で処理が行われるために予期しない結果を得る可能性があります．実際，$n>2$ のとき 2^{n-2} と 2^2 は 0 ではありませんが，これらの積 2^n は剰余の集合 $\mathbb{Z}_{(2^n)}$ で 0 になります! このように $n>1$ のときに $a\times b=0$ となる 0 と異なる元 a, b, すなわち「**零因子**」と呼ばれる数が存在します．また，正整数同士の和が負の整数になることがあります．たとえば $2^{n-1}-1$ と 2 の和は $(2^{n-1}-1)+2=2^{n-1}+1$ ですが，この $2^{n-1}+1$ は $2^{n-1}-1$ の「**2 の補数**」であるために $-(2^{n-1}-1)$ になります! このことを Python 3 + NumPy で確認しておきましょう:

```
>>> import numpy as np
>>> a = np.array([2**62], dtype=np.int64)
>>> b = np.array([4],dtype=np.int64)
>>> print(a, b)
[4611686018427387904] [4]
>>> a*b
array([0])
>>> a = np.array([2**63-1], dtype=np.int64)
>>> a
array([9223372036854775807])
>>> a * a
array([1], dtype=int64)
>>> a + 1
array([-9223372036854775808])
```

ここで ‘np.array([2**62], dtype=np.int64)’ の意味は Python のリスト ‘[2**62]’ を NumPy の ndarray 型の配列で，その成分が 64 bit 長の整数 (int64 型) として格納するという意味です．ちなみに ‘array([2**63], dtype=int64)’ と入力すると「**大き過ぎる**」とエラーになりますが，それよりも絶対値が小さな値で int64 型の配列と して入力できてしま

[*2] De Morgan は負の数を認めなかったそうですが，その気持も理解できませんか?

えば, あとは 64 bit 長の整数の範囲内で処理されます. その結果, 0 と異なる正の数同士の積が 0 になったり, 正の数の和が負の数になるという一見して不可思議な現象が生じます. その理由は演算結果が 64 bit 長の整数で表現できる範囲を越えたためです. このように計算機上の整数演算は**計算機で表現可能な範囲内で処理が行われ, 入力や結果が範囲外になると, 自動的に型の変換が行われるシステムでなければエラーになるか, 範囲内の整数で置換えられます**. もし, 入力や結果が領域を超過する可能性があれば後述の「**浮動小数点数**」を用いるか, Python や SageMath のように多倍長整数計算が可能なツールで処理速度を度外視したプロトタイプを構築し, 動作と動作範囲を確認した上で正式なモデルを構築する必要性があります. 現在, ビックデータ処理が流行っていますが, 扱う数の範囲, 場合によっては配列の添字として現れる整数値の最大値も踏まえて, モデルの規模に応じた処理に関する考察が必須です. ここで注意すべきことの一つが扱う数値の大きさだけではなく, 処理すべきデータを格納する配列の添字の上限です. 実際, 超高層ビルやそれらを含む街, それに自動車等の空気の流れのシミュレーションで実際に問題になっており, 添字の範囲の拡張を行うといった対処が行われています.

5.3 IEEE 754 による実数の表現

5.3.1 浮動小数点数の概要

実数 $\mathbb{R}$ の濃度は $\aleph$ で, 整数 $\mathbb{Z}$ の濃度 $\aleph_0$ よりも各段に大きな数です. さらに実数 $\mathbb{R}$ よりも小さいとはいえ整数 $\mathbb{Z}$ の濃度 $\aleph_0$ も有限個ではありません. これらの無限個の数を計算機の有限な資源を使って表現するためになんらかの工夫が必要で, それが「**浮動小数点数**」と呼ばれる数の表現です. この浮動小数点数の考え方を例で解説しましょう. まず実数 -101.234 の浮動小数点数への符号化を考えます. この数は負の数のために符号は -1, 残りは絶対値 101.234 と分解できます. つぎに絶対値を整数の桁が一桁の小数と 10 の冪の積に分解しましょう. その結果, $101.234 = 1.01234 \times 10^2$ が得られ, そこで, 符号, 小数と冪の次数の 3 成分のタプル $(-1, 10^2, 1.01234)$ で -101.234 を置換えます. ちなみに小数の部分は「**仮数 (mantissa)**」と呼ばれます. ところで数の符号が -1 であれば 1,1 であれば 2 を対応させることにし, 10 の冪は次数だけ, 小数の部分を 10 進数の列 “101234” とすると自然数のタプル (1, 2, 101234) が得られます. この符号化で考える数を, その冪の次数と仮数の整数部分を 0 以上, 9 以下に限定しておけば, このタプルの成分を繋いだ数の列 “12101234” で数 -101.234 が符号化ができます. これが浮動小数点数への符号化の手法です. この符号化では先頭に数の符号情報, 基数 10 の冪の次数が配置されているために符号化した数の大小関係の判定が迅速に行えます. 実際, 最初に符号, 同じ符号なら冪の次数, 次数も等しければ仮数で比較するという処理で判断できます. また, この符号化は 10 進数である必要はありません. 正整数 b を使って浮動小数点数を 0 から $b-1$ まで

の数の列, つまり, b-進数の数でも構いません[*3]. この正整数 b を「**基数 (radix)**」と呼びます. 浮動小数点数に関しては IEEE 754 という国際規格があり, 符号小数点数そのものの規格化, それだけではなく, 浮動小数点数の演算も規格化されています.

5.3.2　IEEE 754 について

IEEE 754 は浮動小数点数の書式と演算等の処理を定める国際規格です. はじめに 2008 年に策定された IEEE 754-2008 が定める事柄を挙げておきます:

- 基数が 2 と 10 のときの浮動小数点数の書式
- 無限大 $\{-\inf, \inf\}$, 浮動小数点数例外 NaN として $\{\mathrm{qNaN}, \mathrm{sNaN}\}$
- 和, 差, 積, 商の四則演算, 平方根, および融合積和演算, 大小関係の処理
- 整数と浮動小数点数間の変換
- 異なる浮動小数点数間の変換
- 浮動小数点数と文字列としての外部表現との変換

このように IEEE 754 が定めるのは浮動小数点数の書式だけではなく, 例外 NaN に加えて四則演算や大小関係の処理や異なる型の数への変換を定めています. つぎに IEEE 754-2008 が想定している実数 $\mathbb{R}$ の符号化の段階を示しておきましょう:

実数の浮動小数点数への符号化の階層

第零層:	$\mathbb{R}$	実数そのもの
	$\downarrow$	
第一層:	$\mathbb{R} \cup \{-\infty, \infty\}$	無限大の追加で実数拡張
	$\downarrow$	
第二層	$\{-\infty, \ldots, -0\} \cup \{+0, \ldots, \infty\} \cup \{\mathrm{NaN}\}$	± 0 と例外 NaN の導入
	$\downarrow$	
第三層:	$(S, E, T) \cup \{-\inf, \inf\} \cup \{\mathrm{qNaN}, \mathrm{sNaN}\}$	実数を近似, NaN を qNaN, sNaN に分類
	$\downarrow$	
第四層:	$\underbrace{b_{(0)} b_{(1)} b_{(2)} \ldots b_{(n-1)}}_{n-\mathrm{bit}}$	2 進数表現として符号化

ここでは無限遠 $\{-\infty, \infty\}$ と例外 NaN(Not a Number) の導入による実数 $\mathbb{R}$ の拡張から符号化に至るまでの抽象化の段階を示しています. まず, 各階層にある矢 ($\downarrow$) が集合の元との対応関係を示し, 特に第二層から第三層の対応関係は多対一になります. なぜなら, 濃

[*3] GMP の多倍長浮動小数点数は仮数部の実体を別に持ち, 浮動小数点数側に仮数部の番地を指定すれば仮数部の長さの制約を外して任意精度浮動小数点数を表現しています.

度 $\aleph$ の実数 $\mathbb{R}$ を符号 (S), 指数部 (E) と仮数部 (T) から構成される固定されたデータ長のタプルへの射影は「**丸め**」で同じ浮動小数点数に異なる数が符号化されるためです. また, -0 と $+0$ は 0 の浮動小数点数への符号化で生じ, IEEE 754 で $0 = -0 = +0$ と規定されています. さらに第二層で実数 $\mathbb{R}$ を負の数と正の数に分けられ, 負の実数側は負の規格化数, 非規格化数, -0 と $-\infty$, 正の実数側が正の規格化数, 非規格化数, $+0$ と ∞ で構成されます. ここで規格化数, 非規格化数の違いは浮動小数点数で表現したときに後述の隠れ bit があるものが規格化数, そうでないものが非規格化数ですが, 0 周辺の数を表現するものが非規格化数とも言えます. また, NaN については, IEEE 754-1985 では NaN のみ, IEEE 754-2008 では qNaN(quiet NaN) と sNaN(signaling NaN) の二種類の NaN が規定されています. ちなみに qNaN が 0/0 や $\infty - \infty$ のような不定値になる演算から出力されて演算を通じて qNaN として処理され, sNaN が発生した時点で OS の例外として処理されるためにデバッグで用いられます. なお, Python では sNaN と qNaN を区別せずにひとまとめに NaN で処理しています.

5.3.3 浮動小数点数の書式

浮動小数点数は, 符号部, 仮数部と基数部の三部構成で, IEEE 754-2008 では基数を 2 あるいは 10 とする一定の長さの 2 進数です. この符号化の長さを「**符号長**」と呼び, 浮動小数点数で表現可能な数の領域, つまり, 精度が決定されます. ここで符号長が 32 bit 長の「**単精度 (single)**」, 符号長が 64 bit 長の「**倍精度 (double)**」, IEEE 754-2008 では基数が 10 の浮動小数点数に関して倍精度と四倍精度が規格化されています. ここでは基数 2 の話に限定して解説します.

符号化する実数について, その符号を $(-1)^0 = 1$ と $(-1)^1 = -1$ から正のときに 0, 負のときに 1 を対応させる部位を「**符号部 (sign)**」と呼んで記号 S で表現し, 符号化した数の先頭に配置します. つぎに与えられた数の絶対値 $|n|$ からガウス記号 [ ] で表現された $\log_2|n|$ を越えない最大の整数 $e = [\log_2|n|]$ を計算し, それから $\dfrac{|n|}{2^e}$ を 2 進数表示で小数点以下第 p-桁まで $d_0.d_1d_2\ldots d_p$ と計算します. この構成方法で $d_0 = 1$ となるために残りの小数点以下の $d_1d_2\ldots d_p$ の符号化を行い, この d_0 に配置される 1 のことを「**隠れ bit**」, あるいは「**暗黙の 1**」と呼びます. また, 残りの p 個の数字の列 $d_1\ldots d_p$ を「**仮数部 (mantissa, significand)**」と呼んで記号 T と表記します. ところで 2 の冪の次数 e には負数もあるため, 次数 e の符号化が正整数になるように「**下駄履き値 (bias)**」と呼ばれる正整数 β を加えた「**下駄履き表現**」を考えます. つまり, $\varepsilon \overset{\mathrm{Def.}}{=} e + \beta$ で定めた正整数 ε を 2 進数で表現し,「**指数部 (exponent)**」を格納する部位を記号 E と表記します. この符号化の状況を各精度毎にまとめておきます:

基数 2 のときの浮動小数点数の精度と各部の長さ

精度	符号長	仮数部長 (p)	指数部長 (w)	下駄履き値
単精度	32	23	8	127(=2^7-1)
倍精度	64	52	11	1027(=$2^{10}-1$)
四倍精度	128	112	15	16337(=$2^{14}-1$)

この表には載せていませんが符号部は 1 bit 長で, 下駄履き値は数の符号化で用いる指数部の上限 (emax) と下限 (emin) の和が 1 になるように設定するために $2^{\text{指数部長}-1}-1$ が下駄履き値 β の値で, 同時に指数部で表現できる整数の上限値 emax と一致し, 下限値 emin の値は $1-\beta$ になりす. なお, 指数部の表現する数の上限 emax や下限 emin の指数部を持つ浮動小数点数に inf, NaN や 0 が割り当てられます.

5.3.4　浮動小数点数を構成する各部位

IEEE 754 が定める浮動小数点数は一定の bit 長の符号部 (S), 指数部 (E), 仮数部 (T) の三部で構成される 2 進数列です. 符号部 S は 1 bit 長で精度と無関係に固定され, 指数部 E の bit 長を w, 仮数部 T の bit 長を p と表記します. このとき浮動小数点数の符号長は $w+p+1$ で, この値は各精度に対応する 32, 64, 128 のいずれかの値です. そして, 浮動小数点数に符号化した数 a は精度に対応した長さの 2 進数で, その桁数に対応するように番地を入れます. この番地は a の列の長さが n であれば, 数の列 a の左端を $a_{(n-1)}$, 右端を $a_{(0)}$ とします.

■**符号部** S:　1 bit 長で 2 進数 a での最大の番地の成分 $a_{(w+p)}$ に対応し, 符号化する数が正であれば 0, 負であれば 1 が設定されます.

■**指数部** E:　w bit 長で符号部に直後に配置されます. 2 進数 a の w 桁の部分 $a_{(w+p-1)}\dots a_{(p)}$ に情報が格納されます. この指数部 E が表現する整数 ε は, 本来の指数 e に「**下駄履き値** (bias)」と呼ばれる定数 β を加えた値で,「**下駄履き表示**」と呼ばれます. 下駄履き値 β は指数部の bit 長に依存するために浮動小数点数の精度で異なります. この下駄履き表示に対して $b_{(i)}=a_{(i+p)}$ $(0\le i\le w-1)$ のときに指数部で表現される整数 ε は $b_{(0)}\times 2^0+b_{(1)}\times 2^1+\cdots+b_{(w-1)}\times 2^{w-1}$ で与えられます.

■**仮数部** T:　p 桁の 2 進数 $a_{(p-1)}\dots a_{(0)}$ で表現されます. この桁数 p が浮動小数点の精度で, 倍精度浮動小数点数のときは $p=53$ です. また, w bit 長の指数部が表現する数 ε で正規化数と非正規化数に分類され, このことが仮想部の復号化に影響します:

規格化数と非規格化数の仮数部

種類	指数部が表現する数	指数部の復号化	仮数部の復号化
規格化数	$2^w - 2 \geq \varepsilon \geq 1$	$2^{\varepsilon-\beta}$	$1.d_{(p-1)} \dots d_{(1)} d_{(0)}$
非規格化数	$\varepsilon = 0$	$2^{1-\beta}$	$0.d_{(p-1)} \dots d_{(1)} d_{(0)}$
$\pm\inf$, NaN	$\varepsilon = 2^w - 1$		

指数部が表現する整数 ε が $2^w - 2 \geq \varepsilon \geq 1$ のときに仮数部の 2 進数の頭に「**暗黙の 1**」, あるいは「**隠れ bit**」と呼ばれる 1 が配置され, $\varepsilon = 0$ のときに暗黙の 1 は配置されません. この暗黙の 1 がある浮動小数点数を「**規格化浮動小数点数 (normal floatingpoint number)**」, 略して「**規格化数**」, 暗黙の 1 がない浮動小数点数を「**非規格化浮動小数点数 (subnormal floatingpoint number)**」, 略して「**非規格化数**」と呼びます. 規格化数では隠れ bit を加えた桁数 $p+1$ が実際の有効桁になるため, この $p+1$ が「**有効桁精度**」と呼ばれます. 一方の非規格化数は 0 周囲の微小な数の表現で, 非規格化数は最大でも p bit と有効精度以下の精度になるために規格化数から非規格化数の領域に移行することを「**段階的アンダーフロー**」, または「**漸近アンダーフロー**」と呼びます. ここで規格化数と非規格化数の復号化の書式を示しておきます:

規格化数と非規格化数の復号

規格化数 ($\varepsilon = 1$):	$(-1)^s \times 2^{-\beta} \times (1 + d_{(1)} \cdot 2^{-1} + \cdots + d_{(p-1)} \cdot 2^{1-p})$
	$= (-1)^s \times 2^{\mathrm{emin}+1} \times (1 + d_{(1)} \cdot 2^{-1} + \cdots + d_{(p-1)} \cdot 2^{1-p})$
非規格化数 ($\varepsilon = 0$):	$(-1)^s \times 2^{\mathrm{emin}} \times (0 + d_{(1)} \cdot 2^{-1} + \cdots + d_{(p-1)} \cdot 2^{1-p})$

非規格化数の復号では指数部の冪の次数 e が規格化数と同様の $-\beta$ ではなく $1-\beta = \mathrm{emin}$ であることに注意して下さい.

■零, 無限大と NaN の表現: 浮動小数点数の表現の多様性のために用いられるます:

特殊な数の定義

表現される数	ε の値	仮数部の値
0	0	0
$\{-\inf, \inf\}$	$2^w - 1$	0
$\{\mathrm{qNaN}, \mathrm{sNaN}\}$	$2^w - 1$	0 以外 (符号部は無関係)

指数部と仮数部を表現する整数 ε が 0 のとき, すなわち $(-1)^s \times 0$ で数 0 の浮動小数点数への符号化を定義します. このときに $s = 0$ であれば「**正の零**」と呼んで +0 と表記し, $s = 1$ であれば「**負の零**」と呼んで -0 と表記しますが, $+0 = -0$ であることが IEEE 754 で定められています. この符号付き零の規定は右極限や左極限を考慮する上で重要です. 演算では他の浮動小数点数と同様の演算が行え, 1/0 のような 0 による割算では無限

大 ∞ に対応する浮動小数点数の inf を返します.

w bit 長の指数部が表現する数 ε の取り得る範囲で, 規格化数と非規格化数が用いない $\varepsilon = 2^w - 1$ の領域は NaN $\{\mathrm{qNaN}, \mathrm{sNaN}\}$ と無限大 $\{-\inf, \inf\}$ 向けに確保されています. まず, 無限大 $\{-\infty, \infty\}$ は指数部が表現する数 ε が $2^w - 1$, すなわち, w bit 長の指数部がすべて 1, 仮数部がすべて 0 のときで, 符号部が 0 のときに「**正の無限大 (inf)**」, 符号部が 1 のときに「**負の無限大 (-inf)**」です. IEEE 754 では無限大と 0 を含む通常の浮動小数点数との演算が定義されています. NaN は 0^0 や $\infty - \infty > 0$ のような不定値が現われる演算を行ったときに生じる例外で, 仮数部が 0 でなく指数部の bit がすべて 1 です. IEEE 754-1985 では NaN のみ, IEEE 754-2008 で qNaN と sNaN の二つの NaN の扱いが規定されています. まず, 0/0, inf/inf や inf - inf のように不定値を返す式では例外として qNaN が返されます. この qNaN は演算を通して qNaN であるために, OS レベルの例外にはなりませんが, sNaN は不正な処理に対する NaN であるために OS レベルの例外として処理されるためにデバッグ用に使えます.

■浮動小数点数で表現できる範囲:　浮動小数点数は有界で有限個であるために表現できない実数があります. 具体的に倍精度浮動小数点数の範囲を示しておきます:

倍精度浮動小数点数で表現可能な領域

最大正規化数	最小正規化数	最小非正規化数
$\pm 1.7977 \times 10^{308}$	$\pm 2.2251 \times 10^{-308}$	$\pm 4.9407 \times 10^{-324}$

ここで浮動小数点数の上限が最大正規化数, 下限が最小非正規化数で, 最大正規化数よりも大の数を「**桁溢れ (overflow)**」, 逆に, 最小非正規化数よりも小の数を「**アンダーフロー (underflow)**」と呼びます. また, 仮数部を w bit 長, 指数部を p bit 長とする浮動小数点数で表現可能な数の集合を $\mathcal{F}_{w,p}$, 簡単に $\mathcal{F}$ と表記します.

■計算機イプシロン:　浮動小数点数 1.0 に最も近接する浮動小数点数との差として定義され, IEEE 754 で計算機イプシロンの値は $2^{-52} = 2.220446049250313 \times 10^{-16}$ です. この浮動小数点数は 0 と異なる微小な実数の表現で用いられ, NumPy は ‘finfo(float).eps’ に束縛された値から判ります[*4]. このことを NumPy を np として読込んだ 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)
```

*4 MATLAB で変数 eps, Scilab は変数 %eps に束縛された値です.

```
True
```

上の二つの確認は大雑把ですが, 計算機イプシロンの定義と矛盾しない結果が得られています. この計算機イプシロンは 0 による割算や対数軸グラフの生成等で 0 であることに起因するエラーで落ちて困る処理に利用できます.

■函数 ulp: 閉区間 $[-x_{\max}, x_{\max}]$ に包含される実数のみが浮動小数点数として表現され, その区間中の実数の符号化は基本的に近似値で, 実数と違って連続するのではなく離散的に配置されます. ここで ulp(Unit in the Last Place)*5という名前の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} になるために ulp は 1 になります. このことを Python 3 で確認しましょう:

```
>>> print('%16.0f\n' %(2.0**53))
9007199254740992

>>> print('%16.0f\n' %(2.0**53+1))
9007199254740992

>>> print('%16.0f\n' %(2.0**53+2))
9007199254740994
```

このように絶対値が 2^{53} を越える整数は倍精度浮動小数点数で一意に表現できないことを意味します. 逆に言えば絶対値が 2^{53} 以下の整数は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必要があるときは後述の BLAS 等の数値行列計算ライブラリを使った高速演算で対処できます.

5.3.5 浮動小数点の演算について

IEEE 754 は浮動小数点数の書式だけを定める規格ではありません. IEEE 754 は浮動小数点数の書式の他に通常の浮動小数点数に対する「**四則演算**」: (“+”, “−”, “∗”, “/”) と「**融合積和演算 (FMA)**」,「**平方根**」: (“$\sqrt{\ }$”) と「**剰余**」:(“%”),「**比較操作**」:(“>”, “≥”, “=”) といった演算, さらには無限大 $-\infty, \infty$ や「**NaN(Nnot a Number)**」に対する処理, さらには「**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**」も規格化しています.

ここで IEEE 754-2008 で新規に追加された「**融合積和演算 (fused multiply-add,**

*5 「ウルプ」と呼びます

FMA)」[*6]は演算 $(a,b,c) \mapsto a+(b \times c)$ のことです．FMA がなければ $b \times c$ を処理して a との和の演算と全体で二度演算を実行するために後述の丸めが 2 度生じる可能性がありますが, FMA が CPU に実装されていれば一度の処理で済むために誤差も小さく, より高速な処理が期待できます．このように IEEE 754-2008 では浮動小数点数の四則演算, 平方根と融合積和演算を定めているために IEEE 754-2008 の浮動小数点数を実装した計算機であれば計算結果が一致します．しかし, 三角函数や対数函数等の初等函数が規格化されていないために利用している数値計算ライブラリやハードウェア上の処理で結果が微妙に異なる可能性があります．

■**丸め/切捨**:　実数 x と $\hat{x} \in \mathcal{F}$ の対応付けを「**丸め**」, あるいは「**切捨**」と呼び, 次の 4 種類の操作が規定されています:

丸めと切捨

丸めの種類	**概要**
1. **上向きの丸め**	a 以上の浮動小数点数で最小のものを採用:
	$\rhd\, a \stackrel{def}{=} \min(\{\ x \in \mathcal{F}\ \wedge\ a \leq x\ \})$
2. **下向きの丸め**	a 以下の浮動小数点数で最大のものを採用:
	$\lhd\, a \stackrel{def}{=} \max(\{\ x \in \mathcal{F}\ \wedge\ a \geq x\ \})$
3. **最近値への丸め**	a に最も近い浮動小数点数を採用:
	$\circledcirc\, a$ と表記
4. **切捨**	絶対値が $\lvert a \rvert$ 以下で a に最も近い浮動小数点数を採用:
	$\unlhd\, a$ と表記

1. と 2. の丸めで実数 x に対応する浮動小数点数 $\tilde{x}$ との差の絶対値は $\mathrm{ulp}(x)$ 以下, 3. と 4. の操作による浮動小数点数からの距離は $\mathrm{ulp}(x)/2$ 以下になります．

■**丸めと切捨の性質**:　演算子 $\bigcirc : \mathbb{R} \to \mathcal{F}$ を演算子 “$\rhd$”, “$\lhd$”, “$\circledcirc$” か “$\unlhd$” 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)	$\circledcirc\,(-x) = -(\circledcirc\, x)$	
4)	$\lhd\,(-x) = -(\rhd\, x)$	
5)	$\rhd\,(-x) = -(\lhd\, x)$	
6)	$(\bigcirc x) \bullet (\bigcirc y) = \bigcirc\,(x \circ y)$	

性質 2) より, 丸めの演算子は「**大小関係の順序を保つ**」ことが判りますが, 比較の演算

[*6] Intel の x86_64 では Haswell 以降, AMD なら Bulldozer 以降の CPU が対応.

子 “$\leq$” を “$<$” で置き換えられません．実際, $x \in \mathcal{F}$ に対して $0 < |x - y| < \mathrm{ulp}(y)/2$ を充す実数 $y \in \mathbb{R}$ は $\circledcirc y = x$ になって性質 2) を充さないためです．また, 浮動小数点数は「**近似値**」であり, それらを使った演算結果も同様に近似値でなければなりません．すなわち, 丸めの演算子 “$\bigcirc$”$\in${“$\triangleright$”, “$\triangleleft$”, “$\circledcirc$”, “$\trianglelefteq$”} に対して実数上での演算子 “$\circ$”$\in${“$+$”,“$-$”,“$\times$”,“$/$”}と, それに対応する浮動小数点上の演算子 “$\bullet$”$\in${“$\oplus$”,“$\ominus$”, “$\otimes$”,“$\oslash$”} との間には条件 6) を満たすことが IEEE 754 で要求されています．つぎに浮動小数点数の応用として数値計算ライブラリの話を次の節で行うことにしましょう．

5.4　数値行列ライブラリについて

5.4.1　数式処理と数値行列処理

数式処理システムの多くがメモリ等の環境の制約があるにせよ, 利用者が必要とする精度で数値計算ができますが, 残念ながらその処理は低速です．なぜなら, 通常の数値計算で用いられる浮動小数点数は IEEE 754 に準拠し, CPU に四則演算等の処理が実装されている一方で, 数式処理システムの任意精度による数値計算はより長いビット長の浮動小数点数が用いられてデータ量と計算量の双方が増大していても直接的なハードウェアの支援がないためにどうしても処理に時間がかかります．ところで, 主要な数値計算の多くが数値行列処理で置換えられるため, 数値計算目的のソフトウェアは数値行列処理の効率化に工夫を凝らしています．これは計算を効率的に行うだけではなく, 行列成分のメモリ上の配置やキャッシュを用いたデータの再利用といったさまざまな工夫があります．このような処理は数値行列計算ライブラリのサブルーチンとして纏められており, 数値行列処理ソフトウェアの多くは数値行列処理ライブラリのサブルーチンに数値行列データを引き渡して結果を取り込むという手順を採用しています．

このようなソフトウェアで著名なものが The MathWorks, Inc. の MATLAB でです．MATLAB は本来, FORTRAN で記述された数値行列計算ライブラリ LINPACK や EISPACK[*7]を学生が容易に扱えることを目的とした教育向けのフリーソフトであり, そのために数値行列操作を行うソフトウェアの多くが MATLAB の行列操作を取り入れる大きな要因になりました．なお, 商用化された現在の MATLAB は Toolbox と呼ばれるライブラリ群を備えた数値行列計算ソフトウェアの標準的存在で, 特に制御系の解析では MATLAB+Simulink[*8]の組合せが事実上の標準です[*9]．この MATLAB の影響を強く受けた「**OSS**(Open Source Software)」のアプリケーションの代表として「**GNU Octave**」, INRIA の「**Scilab**」, C 風の「**Yorick**」を挙げておきましょう．まず, GNU

[*7] これらの数値行列計算ライブラリの後継が LAPACK です．

[*8] MATLAB で制御系のブロック線図の解析を目的にしていましたが, 現在は MATLAB の GUI 環境, 特に「**モデルベース開発**」の中核を担っています．

[*9] 構成は「MathWorks 製品一覧 - 製品構成図」(http://www.mathworks.co.jp/products/pfo) を参照．

Octave は GNU の MATLAB クローンと呼ばれ, MATLAB との高い互換性を持ちますが, 単なるクローンではなく, ファイル操作等で独自の改善点も加わっています. Scilab は MATLAB に類似し, その上, MATLAB の Simulink に類似した「**Xcos**」を持つ下手な売物を凌駕するシステムです. ちなみに EU はドイツ政府が提唱した Industory 4.0 に向けて戦略的な動きがみられますが, その中核の一つに Modelica 言語を使った物理現象や機構等のライブラリ化があります. ちなみに, この Modelica 言語は日本の「東ロボ君」プロジェクトで物理学の入試問題を読み取り, それを Modelica 言語に変換して解いています. SciLab では Xcos の modelica block に直接, Modelica 言語のモデルの記載が可能で, それを使って Xcos 上で解析ができます. それから GNU Octave や Scilab が機能を満載して重量級のアプリケーションになっている状況に対し, Yorick は逆に軽量言語で, 対話処理が可能な C と言えるほど C に類似しているものの, 扱う数値行列は多次元配列であり, さらにテンソルの処理に適したユニークな配列の添字処理が大きな特徴です [42]. これら MATLAB 風の行列操作が可能な言語をまとめて「**MATLAB 系言語**」と呼びます.

ところで, SageMath の母体である Python 本体の数値函数やリスト処理は MATLAB 系言語と比較して格段に貧弱ですが, この数値計算の弱点を強化するパッケージが NumPy です. この NumPy も MATLAB の影響を強く受けており, 数値配列や行列の処理で MATLAB と遜色のない機能と処理速度が可能です. また, NumPy を基盤に SciPy 等の科学技術計算パッケージや Matplotlib 等のグラフ表示パッケージが構築されており, これらのパッケージを利用すれば Python で MATLAB 本体と同程度の処理ができます. このことは Python で構築された SageMath でも MATLAB と遜色がない水準で数値行列や配列の処理が可能なことを意味します. ここで, MATLAB 系の言語や Python+NumPy では数値行列を数値計算ライブラリのサブルーチンに引き渡して結果を取り込むという手法で, そのような数値行列の処理に限れば C や FORTRAN と大差ない水準で処理ができます. しかし, 分岐, 反復といった処理言語で配列処理を行うと時間がかかります. この現象は「**処理言語のオーバーヘッド**」と呼ばれる MATLAB 系言語の大きな弱点で, 処理速度の低下は Python や MATLAB 系言語での配列の内部処理の手順が関係します. それに対して NumPy の函数を利用するときは配列ごと BLAS のサブルーチンに引き渡され, その処理結果も配列ごとに返却されるため, NUmPy の函数を利用する場合は C や FORTRAN と大差ない処理が可能になります. このことは数値配列の処理では C や FORTRAN と同様に BLAS の性能が処理時間に関係することを意味しするだけではなく, 数値配列の処理で配列がどのように BLAS で行われるかを多少とも知っておくと, より効率的な処理が可能になります.

5.4.2　BLAS の概要

数値計算処理では, その操作を行列演算に置換えられるものが沢山あります. たとえば連立一次方程式の解法で用いられるガウスの消去法は連立一次方程式の係数を行列として取り出し, その係数行列に対する機械的な処理で置き換えられます. このような数値行列に対する処理を最適化し, ライブラリ化して必要に応じて利用できるようにしておけばプログラムの手間もかかららず, 処理速度や計算精度も保証されるために何かと便利です. 前述のように数値行列処理システム MATLAB も学生に数値行列ライブラリを使わせることが目的でしたが, このときの数値計算ライブラリの後継が「**LAPACK(Linear Algebra PACKage)**」です. この LAPACK は線形連立方程式の求解などの数値行列処理の効率的な処理を目的とするライブラリで,「**BLAS(Basic Linear Algebra Subprograms)**」と呼ばれるルーチン群上に構築され, この BLAS の性能が LAPACK の性能に影響します. これら LAPACK と BLAS の公式標準実装は netlib [*10]で公開されていますが, この実装は FORTRAN で記述された仕様書と言える水準で, 各種計算機環境向けの細かな調整は利用する個人が行う必要があります. そのために各種計算機環境向けに最適化を行った BLAS が存在します. まず, OSS のものに「**OpenBLAS**」と「**ATLAS(Automatically Tuned aLgebrA Software)**」[*11], 商用のものには Intel の「**MKL(Math Kernel Library)**」と AMD の「**ACML(AMD Core Math Library)**」が代表的です. まず, OpenBLAS は高速処理で知られていた GotoBLAS2[15][*12]の後継で, SageMath や NumPy は OpenBLAS を利用するように設定されています. MKL は Intel の CPU 向けに最適化された BLAS で, Anaconda, Inc. の Anaconda に同梱されており, Anaconda 上の NumPy 等で使えます[*13]. そして, ATLAS は OpenBLAS が現われるまで広く活用された BLAS でした. これらの BLAS の大雑把な処理速度は MKL が全般的に優れ, その次が OpenBLAS, それから ATLAS, 最後に netlib の公式実装の順番になります.

5.4.3　BLAS の構成

BLAS と LAPACK のルーチンは「**精度 ＋ 行列の型 ＋ 処理**」で名付けられており, その名前から精度, 行列の性質と処理内容が判ります. そして, BLAS のルーチンは, その処

[*10] http://www.netlib.org/blas/
[*11] ATLAS と OpenBLAS は BSDL で配布されています.
[*12] 後藤和茂氏が開発・管理されていた BLAS で, 後藤氏が Microsoft に移籍されたために開発中止になっています.
[*13] 2012 年に GotoBLAS の開発者だった後藤氏が Intel に移籍され, チューニングに参加されているとのことです. また, 2015 年からはサポートなしで利用目的を問わず無償利用可能 (Community licensing for Intel Performance Libraries) です.

理内容から以下の三水準に分類できます:

BLAS サブルーチンの水準

- 第 1 水準: ベクトル単体やベクトル同士の演算
- 第 2 水準: 行列とベクトルの演算
- 第 3 水準: 行列同士の演算

これらの水準は BLAS 開発の歴史的な経緯があり, 1979 年にリリースされた最初の BLAS は 1970 年代のベクトル型計算機での効率的動作を目的に構築されたベクトル演算を中心とする第 1 水準でした. それから行列とベクトルの演算を行う第 2 水準 (1987), 行列同士の演算を行う第 3 水準 (1989) を追加と 10 年かけて拡張されます [50][53]. それから複素数の 1-ノルムと割算を計算するルーチンを第 0 水準が加わります. なお, BLAS は扱う行列の大きさが小さければ威力を発揮し難く, むしろ, 行列のサイズが大きくなるほど, その威力を発揮します.

BLAS の最適化では CPU の構造や動作が考慮されています. まず, 主メモリは容量が大きなものですが, 実際にデータが転送される時間 (レイテンシ) と単位時間でのデータ転送量 (バンド幅) の特性があまりよくないという傾向があります. そこで一度利用したデータや隣り合うデータは再利用される可能性が高いために主メモリよりも高速なメモリに利用したデータを一時的に保持します. これがキャッシュと呼ばれる仕組みで, このキャッシュの特性で重要な事項はバンド幅、レイテンシとキャッシュの大きさの順になります. なお, 近年の CPU はコアを増やして並列処理で効率向上を目指すため, 複数のコアがデータ転送の帯域を奪い合う問題が顕著になってきており, キャッシュにデータがある間に並列した演算で「**データの再利用**」を行うことが重要になります. また, FORTRAN の配列データは, 列データがメモリ上に連続して並び, C/C++ の配列は行データがメモリ上に連続して並びます. 前者を「**列優先 (Column major)**」, 後者を「**行優先 (Row major)**」と呼びます. ここで列優先の FORTRAN では行データへのアクセスがメモリ上を不連続に飛ぶために処理速度で致命的な悪化を招き, 行優先の C ではその逆になります. このように BLAS の最適化では言語の仕様に適合した効率的なデータ転送を考慮しなければなりません. ここで各水準の特徴を挙げておきましょう.

■**BLAS の第 1 水準**:　ベクトル演算が主要であるために計算量が他の水準と比較して少ないことが特徴です. 実際, ベクトルの大きさを N とするとデータ量と計算量には $O(N)$ と N に正比例する関係があり, その性能は CPU の理論的性能とメモリバンド幅に依存します. また, データの再利用がほとんどできないために, この水準のルーチンの性能向上はさほど期待できません.

■**BLAS の第 2 水準**:　行列とベクトルの演算が主要で, 計算量とデータ量の関係は $O(N^2)$ と N^2 の水準で正比例になるために第 1 水準以上に効果が期待できます. その性能は計算

機のメモリバンド幅に依存し, ベクトルのみにデータの再利用性があります.

■**BLAS の第 3 水準**: 行列同士の処理が主要で, 扱うデータ量は $O(N^2)$, 計算量が $O(N^3)$ と最多であり, その性能も CPU の理論性能値に依存し, $O(N)$ 回のデータの再利用性があるために, この面での性能向上も望めます.

5.4.4 ベクトルと行列の表記について

BLAS の引数表記では大文字のアルファベット A, B, $C, \ldots$ が行列, 小文字のアルファベット v, u, $w, \ldots$ が (列) ベクトル, 小文字ギリシァ文字 α, $\beta, \ldots$ がスカラーを表現し, 行列 A を格納する配列を a と小文字のアルファベットで, ベクトル v, u, w を格納する配列を x, y, z と表記します. それからベクトルや行列のスカラー積を $\alpha \cdot A$ や $\beta \cdot v$ と表記し, 行列 A と行列 B の積は $A\ B$, 行列 A とベクトル v の積を $A\ v$ と表記します. また ${}^tA$ を行列 A の転置, 行列 A の転置と複素共役 ${}^t\overline{A}$ を ${}^HA$ と表記することもあります.

5.4.5 配列への格納方法

ベクトルは 1 次元配列で扱い, 行列は 2 次元配列, あるいは行列の列, または行を繋いで 1 次元配列として格納する方法[*14]に加え, 行列の特性を生かしてメモリを節約しつつ効率的な処理が行える配列への収納の工夫があります. この収納方法に加えて行列の特性に合せた処理で効率向上が狙えます. ここで行列の収納方法には行列を 1 次元的, あるいは 2 次元的に素朴な方法で配列に転記した「**一般形式**」, 三角行列, 対称行列やエルミート行列の性質に合せて成分を規則的に取込んで 1 次元配列で表現する「**圧縮格納形式**」, 帯行列向けの「**帯格納形式**」の三種類があります. ここで BLAS は FORTRAN で記述されているために FORTRAN でメモリ効率が良くなるように行列を配列に格納します. なお, FORTRAN では「**列優先 (Column major)**」であるために行列の列単位で配列に格納し,「**行優先 (Row major)**」の C/C++ では行単位で配列へ格納します. なお, FORTRAN 以外で MATLAB, GNU Octave が列優先, C 以外では NumPy と Scilab が行優先です. ここで MATLAB 系言語であれば '[1 : 10]' で生成されるベクトルのサイズを調べると分かります. ところでこの行優先, 列優先は MATLAB 系言語や NumPy の配列データの格納方法と配列の表現にも密接に関係します. なお, ここでの解説では FORTRAN を念頭にしているために列優先で話を進めますが, 行優先であれば単純に列と行を入れ替えて読むことができます.

---

*14 MATLAB 系言語の多くでは行列を 1 次元配列とみなして処理ができます. また, NumPy の ndarray 型の多次元配列の実データも 1 次元的にメモリ上に連続して配置されています.

■一般形式:　この書式を利用するルーチンは命名規則 3 の “MM” の箇所が “GE” です. 一般形式は行列 A の i 行 j 列成分の A_{ij} をそのまま配列 a の成分 $a[i,j]$ に格納する場合と行列の列を繋いで 1 次元配列に収納する場合があり, 1 次元配列に格納する場合は $m \times n$-行列 A の i,j-成分 a_{ij} を 1 次元配列 a の $i + m \cdot (j-1)$ 成分に格納します.

■圧縮格納形式:　この書式を利用するルーチンは命名規則の “MM” の箇所の二文字目が “P” です.「**圧縮格納形式 (Packed storage)**」と呼ばれる格納方法は上下三角行列, エルミート行列や対角行列の 1 次元配列への格納で用いられます. 上三角行列で $i \leq j$ のときに A_{ij} の値を $a[i + j \cdot (j-1)/2]$ に格納し, 下三角行列であれば, $i \geq j$ のときに A_{ij} の値を $a[i + (2 \cdot n - j) \cdot (j-1)/2]$ に格納します. 具体的に 4×4 の上三角 U と下三角行列 L の格納状況を以下に示します:

行列の配列への格納方法

$$U : \begin{pmatrix} A_{11} & A_{12} & A_{13} & A_{14} \\ & A_{22} & A_{23} & A_{24} \\ & & A_{33} & A_{34} \\ 0 & & & A_{44} \end{pmatrix} \Rightarrow A_{11}\,\underbrace{A_{12}A_{22}}\,\underbrace{A_{13}A_{23}A_{33}}\,\underbrace{A_{14}A_{24}A_{34}A_{44}}$$

$$L : \begin{pmatrix} A_{11} & & & 0 \\ A_{21} & A_{22} & & \\ A_{31} & A_{32} & A_{33} & \\ A_{41} & A_{42} & A_{43} & A_{44} \end{pmatrix} \Rightarrow \underbrace{A_{11}A_{21}A_{31}A_{41}}\,\underbrace{A_{22}A_{32}A_{42}}\,\underbrace{A_{33}A_{43}}\,A_{44}$$

これらの例から上三角, あるいは下三角の列を並べて 1 次元配列に取り込む様子が分ります[*15]. この格納方法は対称行列やエルミート行列でも使えます. なぜなら対称行列は $A_{ij} = A_{ji}$, エルミート行列は $A_{ij} = \overline{A_{ji}}$ を充し, 上下三角行列と同様に行列の上三角領域があれば復元できるためです. 行列を圧縮格納形式で配列に格納したときに前述の命名規則 “MM” は対称行列であれば “SP”, エルミート行列であれば “HP” のルーチンを用います.

■帯格納形式:　帯行列の対角成分付近の帯の成分だけを配列に格納する方式です. 帯行列 $A \in \mathrm{M(m,n)}$ は k_l 行の劣対角成分 (対角成分の下側の成分) と k_u 列の優対角成分 (対角成分の上側の成分) を持つものとします:

[*15] 行優先であれば列ではなく行になります.

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

行列 A の成分 A_{ij} を $(k_u+k_l+1)\times n$ の大きさの配列 a に格納します．この収納の方法は netlib で公開されているルーチンのコメント中の記述が参考になります:

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

'matrix(I,J)' が行列 A の i,j 成分である A_{ij} に対応し，'A' が配列 a に対応します．配列 a への格納は列単位で行われ, 最初に行列の第一列先頭の成分 A_{11} が $a[(k_u+1),1]$, 以降の第一列の成分が続いて配列 a の 1 列目に収納されて $i<ku$ を充す i 列の先頭 A_{1i} が $a[k_u-i,i]$ に収められます．そして $i\geq k_u$ を充す i 列の先頭 A_{1i} が配列 a の i 列の先頭 $a[1,i]$ に格納されます．つまり, $i>k_u$ を充す配列 a の i 列はその先頭の $a[i-k_u,i]$ が一行目に配置されて収納, すなわち対角成分を挟んで上に行 k_u, 下に k_l 行と帯成分が収納されます．ここで, LAPACK マニュアルに記載されている $m=n=6$, $k_u=1$, $k_l=2$ の行列の例を示しておきます:

帯行列の格納方法

本来の行列 A	$\Rightarrow$	配列 a への格納状態
$\begin{pmatrix} A_{11} & A_{12} & 0 & 0 & 0 & 0 \\ A_{21} & A_{22} & A_{23} & 0 & 0 & 0 \\ A_{31} & A_{32} & A_{33} & A_{34} & 0 & 0 \\ 0 & A_{42} & A_{43} & A_{44} & A_{45} & 0 \\ 0 & 0 & A_{53} & A_{54} & A_{55} & A_{56} \\ 0 & 0 & 0 & A_{64} & A_{65} & A_{66} \end{pmatrix}$		$\begin{bmatrix} * & A_{12} & A_{23} & A_{34} & A_{45} & A_{56} \\ A_{11} & A_{22} & A_{33} & A_{44} & A_{55} & A_{66} \\ A_{21} & A_{32} & A_{43} & A_{54} & A_{65} & * \\ A_{31} & A_{42} & A_{53} & A_{64} & * & * \end{bmatrix}$

ここでの '$*$' は成分の配置が行われない箇所を示しており, 帯行列を列単位で対角成分を

積み重ねて格納する方法で本来の行列よりも小さな 2 次元配列に格納できます．この帯格納形式は一般形式と同様に列で並べた 1 次元配列に変換できます．また ‘$k_l = 0$’ であれば上三角行列, ‘$k_u = 0$’ ならば下三角行列になりますが, 格納方法は通常形式に一致します．そして, 通常形式の場合と同様に行列 A が対称行列やエルミート行列であれば, その行列の上三角成分, あるいは下三角成分を帯格納形式で格納できます．

5.4.6　ルーチンの命名規則

BLAS と LAPACK のルーチンの名前は「**PMMAAA**」の書式に限定されます．この名前は FORTRAN の函数名の制約に由来し, 次の意味があります:

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

これら **P**, **MM** と **AAA** を解説しましょう．

■P:　対象の精度, 対象が実数, あるいは複素数であるかを指示します:

精度を表現する文字

	単精度 (32bit)	倍精度 (64bit)	拡張倍精度 (128bit)
実数	S	D	Q
複素数	C	Z	X

■MM:　ルーチンが扱う行列の型 (かたち) を指示します．行列が対角行列なら「**D**」, 三角行列なら「**T**」, 対称行列なら「**S**」, エルミート行列なら「**H**」で, その他の一般の行列は「**G**」で指示され, これらの行列の性質に続いて行列の格納方式を示す文字が入ります．ここで一般形式なら「**G**」, 圧縮格納形式なら「**P**」, 帯格納形式なら「**B**」で, MM はこれらを組合せた文字列「**行列の性質 ＋ 格納形式**」です:

行列の型を示す二文字

BD	二重対角行列	DI	対角行列	GB	帯行列
GE	一般行列	GG	一般行列 (一般行列の対)	GT	一般三重対角行列
HB	エルミート帯行列	HE	エルミート行列	HP	エルミート行列 (圧縮格納形式)
HG	上 Hessenberg 行列	HS	上 Hessenberg 行列	OR	直交行列
OP	直交行列 (圧縮格納形式)	PB	正値対称/エルミート帯行列	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	対称帯行列
SP	対称行列 (圧縮格納形式)	SY	対称行列	TB	三重対角行列/帯行列
TG	三角行列	TP	三角行列 (圧縮格納形式)	TR	三角行列
TZ	台形行列	UN	ユニタリ行列	UP	ユニタリ行列 (圧縮格納形式)

■AAA: ルーチンの処理内容を指示します. 文字数は 2 文字, あるいは 3 文字で, BLAS の多くが 2 文字, LAPACK の多くが 3 文字です. BLAS では「**M**」が行列,「**V**」がベクトル,「**S**」が逆行列計算という意味があり, これらを組合せた名前になっています:

処理内容を示す文字列

MV	行列とベクトルの積
SV	逆行列とベクトルの積
MM	行列同士の積
SM	逆行列と行列の積
EV	固有値問題
QRF	QR 分解

5.4.7 ルーチンの引数について

引数のベクトル v, u を格納する配列をここでは x, y と表記し, これらの配列 x, y に関連する引数「**INCX**」と「**INCY** は配列 x, y の添字の増分を指示します. この本では x, y の増分を 'd_x', 'd_y' とも表記します. なお, これらの増分は 0 より大でなければなりま

せんが, GotoBLAS2 の解説 [15] によると INCX と INCY が 1 のときのみに最適化され, それ以外は処理速度が致命的に低下するとあります.

行列 A を格納する配列は小文字の a, 行列の行数を m, 列数を n と表記し, 正方行列では行数と列数を区別せずに m を用います. ここで行列 $A \in M(m,n)$ を格納する配列 a が行列 A の行数よりも行数が多くなることがあります. たとえば, 連立一次方程式の数値的解法で用いられるガウスの消去法では, 方程式の係数行列と定数項で構成されるベクトルを合わせた行列を処理します. この拡大した配列 a の行数を指示する引数として「**Leading Dimension**」があり, 行列 A の Leading Dimension を「**LDA**」と略記し, この本では L_A と表記します. 一般形式のときは行列 A を列で繋いで格納した 1 次元配列 a で行列 $A(i,j)$ 成分へのアクセスは 'a[i + j*LDA]' で行なわれ, 行列 A を配列 a の部分配列として包含するのでなければ LDA $= \max(1,m)$, つまり, LDA の値として「**行列 A の行数 m を指定**」になり, 行列 A が配列 a に「**帯行列格納**」されたときは, 対角成分数 k_u, 劣対角成分数 k_l の $m \times m$ 行列 A の格納先の配列 a の大きさが $(k_u + k_l + 1) \times m$ になるために引数 LDA として $k_u + k_l + 1$ を指定します.

行列の転置や転置共役を指示する行列変換のフラグは, ルーチンの引数として TRANS がありますが, ここは f と表記し, このフラグで指示される行列の作用素 op_f を以下に示しておきます:

作用素 op_f の挙動

フラグ f の値	$\mathrm{op}_f(A)$	概要
"N", "n"	A	無変換 (要するにそのまま)
"T", "t"	${}^tA$	転置
"C", "c"	${}^t\overline{A}$	転置 + 共役

フラグ f の値は上記の二重引用符 (") で括った文字列です.

つぎに三角行列が上三角行列か下三角行列を指示するフラグ「**UPLO**」があります. この本では f_U と表記し, この値と意味を以下にまとめておきます:

フラグ f_U(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(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

5.4.8　BLAS のルーチン

BLAS の第 1 水準のルーチンが行列の複製や回転, ベクトルの和やノルムの計算, 第 2 水準が行列とベクトルの演算, そして, 第 3 水準が行列同士の演算です. 第 1 水準のルーチンでは直接, 値が返却されますが, 第 2, 第 3 水準のルーチンの多くは計算結果がルーチンの引数末端のベクトルや行列に対応する配列に代入されます. 以降の解説では函数名の先頭の精度を示す箇所を表で「**型**」とし, 実数単精度, 実数倍精度, 複素数単精度, 複素数倍精度の順で並べ, 型以外の語幹に相当する箇所を「**ルーチン名**」と表示します.

5.4.9　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	$\lvert\mathfrak{Re}(x)\rvert_1 + \lvert\mathfrak{Im}(x)\rvert_1$
s	d	c	z	adiv	$x/y \quad x, y \in \mathbf{C}$

5.4.10　BLAS の第 1 水準のルーチン

行列やベクトルの複製, 二つのベクトル同士の和や内積, ベクトルのノルムを含みます. 計算量も少なく, データの再利用性もなく, 性能は CPU の理論性能値とメモリバンド幅に依存します:

第 1 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: ${}^t\overline{v}\ u$
		c	z	dotu	ベクトルの内積: ${}^t v\ u$
sd	d			sdot	ベクトルの内積: ${}^t\overline{v}\ u$
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	$\|$ Re(x) $\|_1$ + $\|$ Im(x) $\|_1$
is	id	ic	iz	amax	絶対値が最大の要素の添字を返却

5.4.11　BLAS の第 2 水準のルーチン

第 2 水準に行列とベクトルの積, 行列同士の和の処理です. 計算量もそれなりにあり, ベクトルに関してデータの再利用性があって性能はメモリバンド幅に依存します.

三角行列とベクトルの積を計算するルーチン

三角形行列 A の $\mathrm{op}_f(A)x$ や $\mathrm{op}_f(A)^{-1}x$ を計算するルーチンを以下に示します:

三角行列とベクトルの積

型				ルーチン	行列の種類
s	d	c	z	trmv	一般の三角行列
s	d	c	z	tbmv	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

ベクトルの積を意味します．三角形行列は対角成分の下側，あるいは上側が全て 0 になる正方行列で，ルーチンの引数は，この行列の形に関連する三種類のフラグ f_U, f, f_D に加え，行列 A とベクトル v を格納した配列と行列の行数に対応する n があります．また，圧縮形式以外の配列を利用するときの引数として $L = \max(1, n)$ を充す変数 L を必要とします．それから，ベクトル v に対応する配列の添字の増分も引数に含まれ，帯行列を処理するルーチンでは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) を指定
f	{N, n, T, t, C, c}	作用素 op_f のフラグ
f_D	{U, u, N, n}	単位三角行列 (U/u) かそれ以外 (N/n) を指定

ここで行列 A の対角成分が全て 1 の三角行列を「**単位三角行列 (unit triangular matrix)**」と呼びます．

$\alpha \cdot \mathrm{op}_f(A)v + \beta \cdot u$ を計算するルーチン

引数は行列の型による違いを除き，引数として行列の転置，共役転置を与える作用素 op_f のフラグ f，行列の行数と列数に対応する m, n，ベクトルに対応する配列 x の添字の増分 d_x とスカラ α, β，さらに配列 a の行数を指定する引数 L があります:

$\alpha \cdot \mathrm{op}_f(A)v + \beta \cdot u$ を処理するルーチン

型				ルーチン	行列
s	d	c	z	gemv	一般行列
s	d	c	z	gbmv	帯行列
		c	z	hemv	エルミート行列
		c	z	hbmv	エルミート帯行列
		c	z	hpmv	圧縮格納形式のエルミート行列
s	d	c	z	symv	対称行列
s	d			sbmv	対称帯行列
s	d			spmv	圧縮格納形式の対称行列

行列を生成するルーチン

第 2 水準では行ベクトルと列ベクトルの積から生成される行列と同じ大きさのベクトルの和を計算します．ここで「**階数 1 の更新 (rank-1 update)**」と「**階数 2 の更新**

(rank-2 update)」と呼ばれる処理があります. 階数 1 の更新が　$\alpha \cdot v^t \overline{u} + A \longmapsto A$ を行う処理, 階数 2 の更新が $\alpha \cdot v^t \overline{u} + A \mapsto A$ 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	$\alpha \cdot v\ {}^t\overline{v} + A$	$A = {}^t\overline{A}$
		c	z	hpr	$\alpha \cdot v\ {}^t\overline{v} + A$	$A = {}^t\overline{A}$, 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	$\alpha \cdot v\ {}^t\overline{u} + A$	$A \in M(m, n)$
		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	$\alpha \cdot v\ {}^t\overline{u} + \overline{\alpha} \cdot u\ {}^t\overline{v} + A$	$A = {}^t\overline{A}$
		c	z	hpr2	$\alpha \cdot v\ {}^t\overline{u} + \overline{\alpha} \cdot u\ {}^t\overline{v} + A$	$A = {}^t\overline{A}$, A: 圧縮格納形式

なお, 階数 2 のルーチンは名前の末尾に 2 が付いているために容易に判ります. 実際, her, hpr,syr,spr,gerc と geru が階数 1 の更新, her2, hpr2, syr2 と spr2 が階数 2 の更新を行うルーチンです.

第 3 水準のルーチン

行列同士の積を含む計算処理を行うために計算量が大きくてデータの再利用性も高く, その処理能力は CPU の理論性能値に依存します:

第 3 水準の BLAS ルーチン

型				ルーチン	処理式	フラグ
s	d	c	z	gemm	$\alpha{\cdot}\mathrm{op}_{\mathrm{f}_1}(\mathrm{A})\ \mathrm{op}_{\mathrm{f}_2}(\mathrm{B}){+}\beta{\cdot}\mathrm{C}$	
		c	z	hemm	$\alpha \cdot A\ B + \beta \cdot C$	$s \in \{"L","l"\}$
					$\alpha \cdot B\ A + \beta \cdot C$	$s \in \{"R","r"\}$
s	d	c	z	symm	$\alpha \cdot A\ B + \beta \cdot C$	$s \in \{"L","l"\}$
					$\alpha \cdot B\ A + \beta \cdot C$	$s \in \{"R","r"\}$
s	d	c	z	trmm	$\alpha \cdot \mathrm{op}_{\mathrm{f}}(\mathrm{A})\ \mathrm{B} + \beta \cdot \mathrm{C}$	$s \in \{"L","l"\}$
					$\alpha \cdot B\ \mathrm{op}_{\mathrm{f}}(\mathrm{A}) + \beta \cdot \mathrm{C}$	$s \in \{"R","r"\}$
s	d	c	z	trsm	$\mathrm{op}_{\mathrm{f}_1}(\mathrm{A})\ \mathrm{X} = \alpha \cdot \mathrm{B}$	$s \in \{"L","l"\}$
					$X\ \mathrm{op}_{\mathrm{f}_1}(\mathrm{A}) = \alpha \cdot \mathrm{B}$	$s \in \{"R","r"\}$
		c	z	herk	$\alpha \cdot A\ {}^t\overline{A} + \beta \cdot C$	$t \in \{"N","n"\}$
					$\alpha \cdot {}^t\overline{A}\ A + \beta \cdot C$	$t \in \{"C","c"\}$
s	d	c	z	syrk	$\alpha \cdot A\ {}^tA + \beta \cdot C$	$t \in \{"N","n"\}$
					$\alpha \cdot {}^tA\ A + \beta \cdot C$	$t \in \{"C","c"\}$
		c	z	her2k	$\alpha{\cdot}A\ {}^t\overline{B}{+}\overline{\alpha}{\cdot}B\ {}^t\overline{A}{+}\beta{\cdot}C$	$t \in \{"N","n"\}$
					$\alpha{\cdot}{}^t\overline{A}\ B{+}\overline{\alpha}{\cdot}{}^t\overline{B}\ A{+}\beta{\cdot}C$	$t \in \{"C","c"\}$
s	d	c	z	syr2k	$\alpha{\cdot}A\ {}^tB{+}\alpha{\cdot}B\ {}^tA{+}\beta{\cdot}C$	$t \in \{"N","n"\}$
					$\alpha{\cdot}{}^tA\ B{+}\alpha{\cdot}{}^tB\ A{+}\beta{\cdot}C$	$t \in \{"C","c"\}$

BLAS についてそのルーチンについて軽く触れましたが, BLAS は数値計算の根幹の一つです．この BLAS の上に MATLAB のような数値行列処理言語や次に解説する Python の数値計算の基本となる NumPy があります．そして, 効率的な数値計算を行うためには計算処理をどのように明瞭で簡易な行列演算に落し込んで BLAS のルーチンに引き渡すかが大きな課題の一つになります.

5.5 NumPy による数値計算

5.5.1 NumPy の概要

Python 本体には円周率 π やネイピア数といった主要な数学的定数, 三角函数, 指数函数といった初等函数が包含されておらず, パッケージで追加しなければなりません．ところで, Python の標準モジュール math で導入される初等函数は実数値のみであって, 基本的な複素数値函数はモジュール cmath を読込まなければならず, しかも, これらのモジュール math と cmath の双方で定義される函数は豊富ではありません．また機能的にも難があり, たとえば, 指数函数を実数と複素数で利用したいときはモジュール math と

モジュール cmath の函数 exp() を切替える必要があります．このような函数の機能の貧弱さに加え, Python のリスト処理は MATLAB 系言語と比較して機能的に貧弱なままで, 数を成分とするリストの和や積といった算術演算もままなりません．実際, 標準では演算子 “+” はリストの結合処理であって, 数値ベクトルの和を計算するものではありません．この状況を改善するのがパッケージ NumPy です．NumPy は主要な数学的定数, 初等函数に加え, 多次元配列とその操作を定義するパッケージであり, 科学技術数値計算ライブラリ SciPy,MATLAB と同等のグラフックス機能を付加する Matplotlib, データフレームを導入する Pandas などの多次元配列や数値計算を利用するさまざまなパッケージの基盤になっています．

Python/SageMath で Python のパッケージやモジュールの読込は import 文で行いますが．‘import numpy’ で読込を行うと NumPy で定義される定数, 函数の呼出は先頭に ‘numpy.’ を付ける必要があります．一般的に ‘import numpy as np’ で NumPy の読込み, ‘np.’ を接頭辞とすることが推奨されています[*16] また, SageMath のように巨大なシステムでは全く薦められませんが, Python 上の軽い計算で, 名前の衝突が生じる恐れがないときに ‘from numpy import *’ で読込むと NumPy で定義される函数や定数が接頭辞なしの名前で呼出ができます[*17]．ちなみにこの本では基本的に ‘import numpy as np’ であらかじめ読込を行なったものとし, NumPy のオブジェクトとしては名前のみ, 実例では接頭辞 “np.” を付けて説明します．つまり, NumPy の函数 sin() を指示するときは “sin()” と名前のみを表記し, 実例では “np.sin(x)” のように接頭辞 “np” を付けます．

5.5.2　NumPy の配列の基本

NumPy の配列 (ndarray 型) は多次元配列です．NumPy の ndarray 型の配列は均質的であり, 単精度, 倍精度, 四倍精度の実数, あるいは複素数の浮動小数点数, あるいは文字列をその成分にすることができます．この配列の生成はタプルやリストからも構築できます．なお, Excel の表のように文字列, 整数や実数といった雑多な型の列と, それらの列を名前で指示できる表を扱いたければ Pandas のデータフレームを用いるのが一般的です．また,「**構造化配列 (structured array)**」を用いて構築も NumPy の範囲内に限定するのであれば可能です．ところで, Python のリストは MATLAB 風の数値演算が行えなえないオブジェクトを保存する箱のようなものです．それに対して数値成分の ndarray 型の配列では四則演算を含めた計算ができます．ただし, MATLAB 系言語と異なり, 演算は成分単位の演算で, ベクトル・行列の積演算はメソッド dot() を用いるか, クラス ndarray の派生

[*16] Import conventions, http://www.scipy-lectures.org/intro/numpy/array_object.html#numpy-arrays

[*17] 薦めない理由に「**名前の衝突**」があります．後述するように SageMath では NumPy と異なる独自の数体系を導入するために, 一見して同じように見える数オブジェクトに関係する TypingError の原因になります．

クラスの matrix 型に変換してベクトル・行列の演算を処理します．この点で NumPy の多次元配列は MATLAB の行列とはやや異なる性格を持ちます.

クラス ndarray の配列の形やメモリ上の配置に関連する属性を以下に示します:

ndarray の属性

属性名	値	概要
ndim	整数	配列の次元
shape	整数のタプル	配列の形
size	整数	配列の成分総数
itemsize	整数	配列の成分の byte 数
strides	整数のタプル	各次元で移動に必要な byte 数
nbytes	整数	配列が占有する総 byte 数
data	buffer/memoryview	配列データのメモリ上の情報
flags	flagobj	配列のフラグ情報
dtype	文字列	配列の成分の型

クラス ndarray のインスタンスの大きさが属性 shape に反映され, 配列の次元は属性 shape のタプルの長さ, あるいは括弧 “[ ]” の深さに対応します．また, クラス ndarray のインスタンスの成分はすべて属性 dtype で指示される型を持ち, インスタンス化の時点で未指示であれば与えられた配列成分で和演算が閉じられる型に自動的に変換されます．ここで dtype として指定可能な値は Python 由来のものと NumPy で追加されたものの二種類があります．まず, 整数には符号付き (int) と符号なし (uint) があり, そのビット長として 8, 16, 32 と 64 bit があります. 実数を表現する浮動小数点数 (float) については 16, 32(単精度),64(倍精度) と 128 bit (四倍精度), 複素数を表現するの浮動小数点数 (cfloat) は実部と虚部の浮動小数点数の対になるために 32(単精度), 64 bit(倍精度) があります. これらの数値に加えて True, False のみの bool 型 (8 bit) があります. さらに Python 由来の数値型に整数の int, 浮動小数点数に float と cfloat がありますが[*18], その実質は int は NumPy の int32, float は float64, cfloat は complex128 になり, 配列成分の型の指定で配列の行すべての成分がこれらの型に合わされます．このことが属性 itemisize, strides, nbytes に関係します．まず, 属性 itemsize は整数値で, 配列の各成分の byte 数で表現した大きさに対応します．たとえば, 倍精度浮動小数点数の配列であれば倍精度浮動小数点数の符号長が 64 bit = 8 byte であるために属性 itemsize は 8 になります．そして, 属性 strides は各次元の移動に必要な byte 数を示します．それから属性 data が ndarray 型のオブジェクトの配列データのメモリ上の情報で, Python 2 であれば buffer 型で函数 str()

[*18] Python 2 であればさらに long があります.

で文字列に変換すると 1 次元データとして格納された配列データを 16 進数として見られますが, Python 3 ではより汎用性のある memoryview 型に変更されています．この属性 data が実際の配列データに対応し，ここでの ndarray 型オブジェクトのデータは属性 itemsize で指示される均質的なデータがメモリ上に連続して並んだ 1 次元的な構造で, 属性 shape で指示されるビューで取り出す仕様になっています. これらの属性に加えて配列のメモリ上の配置に関連するフラグもあります:

flags

C_CONTIGUOUS	Boolean	配列の格納方法の指定
F_CONTIGUOUS	Boolean	配列の格納方法の指定
OWNDATA	Boolean	配列データ保持方法の指定
WRITABLE	Boolean	書込可能性の指定
ALIGNED	Boolean	連続配置の指定
UPDATEIFCOPY	Boolean	指定不可能

これらのフラグで WRITABLE, ALIGNED と UPDATEIFCOPY はメソッド setflags() で変更兼兼可能で, C_CONTIGUOUS と F_CONTIGUOUS は配列の生成時に決定されるために配列の生成後は変更不可，これらのフラグは配列データの格納方法を指示し, C_CONTIGUOUS が C 流に行優先, F_CONTIGUOUS が FORTRAN 流に列優先です．このフラグ設定は構築子 ndarray() や特定の形の配列生成を行う函数や reshape() の様なメソッドにオプション order があれば設定できますが, NumPy は行優先が既定値です．WRITABLE は配列の書込禁止の指定ができるフラグです．残りの ALIGNED と UPDATEIFCOPY は操作することはありません.

　この他の配列の属性にビューと呼ばれる配列そのものを変更せずに配列の転置や 1 次元配列に変換した配列を返却する属性があります:

ndarray のビュー変更の属性

属性名	概要
T	配列の転置
real	配列の実部
imag	配列の虚部
flat	配列の 1 次元配列化
ctypes	配列の ctypes 型への変換

これらの属性はビューを返すだけで基のインスタンスに影響を及ぼしません．たとえば, a.T で配列 a の転置を返しますが, 配列 a の shape が a.T の影響で転置されることはあ

りません．また, ‘b = a.T’ で変数 b に配列 a の転置したビューを持つ ndarray 型のオブジェクトが生成されて変数 b に束縛されますが, 配列 a 自体にはその操作の影響が生じません．ちなみに配列 a の shape が $(n_0, n_1, \ldots, n_k)$ のときに転置 a.T の shape は $(n_k, \ldots, n_1, n_0)$, 配列の成分の対応関係は $\mathrm{a}[i_0, i_1, \ldots, i_k] \mapsto \mathrm{a.T}[i_k, \ldots, i_1, i_0]$ です．また, 配列 a の ‘a.flat’ は flatiter 型のオブジェクトで, 添字で値の参照は ndarray 型と同様にできますが, その名前を入力しても返却されるのは名前に束縛されたオブジェクトの情報であり, ndarray 型のオブジェクトと動作が異なります．そして, a.ctypes は外部函数ライブラリ ctypes 向けのデータ型になります．

ndarray 型のオブジェクトの生成は, Python のリストや括弧で括ったタプルを引数に構築子 array() を使って生成できます．ここで dtype で型を指示しなければ, 引数として与えたオブジェクトが数値だけであれば, 数値塔の考え方で全体がその数値の型に変換され, 文字列が含まれていれば, 全ての成分を文字列に変換して文字列の配列として自動的に生成されます．具体的には, 数値, あるいは文字の列を一重に括弧で括ったリストやタプルからは 1 次元配列が生成されます．この一重に括弧で括られたオブジェクトの列が「**行**」になります．そして, NumPy では行優先のために行の成分はメモリ上に連続して並びます．2 次元配列は 1 次元配列のリストやタプル, つまり, 行の列から構築されます．そして, ndarray 型のオブジェクトの次元は括弧の深さ (=属性 shape で返却されるタプルの長さ) と属性 ndim の値に対応します:

```
>>> import numpy as np
>>> a = np.array([1,2,3,4,5])
>>> a
array([1, 2, 3, 4, 5])
>>> a.shape
(5,)
>>> a.size
5
>>> b = np.array([[1], [2], [3], [4], [5]])
>>> b
array([[1],
       [2],
       [3],
       [4],
       [5]])
>>> b.shape
(5, 1)
>>> b.size
5
>>> c = np.transpose(b)
>>> c
```

```
array([[1, 2, 3, 4, 5]])
>>> c.shape
(1, 5)
>>> d = np.array([[1, 2, 3], [5, 4, 3]])
array([[1, 2, 3],
       [5, 4, 3]])
>>> d.ndim
2
```

ndarray 型の配列の生成では括弧 “[ ]” で括ったリストや括弧 “( )” で括ったタプルが使えます．ただし, タプルであるとは言え, ‘1,2,3’ のような「**式の列**」, すなわち,「**括弧のない式のタプル**」はエラーになります．なお, 多次元配列の定義でタプルとリストの混在は問題ありません．そして, これらの括弧の深さが配列の次元に対応します．たとえば, 行列は括弧 “[ ]” の深さが 2 層になるために 2 次元です．ちなみに NumPy の配列の次元は属性 ndim に割り当てられています．そして, 配列を 1 次元配列とみなしたときの長さは属性 size, 多次元配列の形は属性 shape で調べられます.

構築子 array() で属性 dtype を設定することで配列のデータ型を指示できます．ただし, 属性 dtype が未指定のときは, 成分として与えられたデータが全て整数であれれば整数, 実数表記があれば浮動小数点数と, 数値塔の考え方でその型が各成分に一律に与えられます:

```
>>> a = np.array([1], dtype=np.complex128)
>>> a
array([ 1.+0.j])
>>> type(a[0])
<class 'numpy.complex128'>
>>> b = np.array(range(24),dtype=np.double).reshape(4,3,2)
>>> b
array([[[  0.,   1.],
        [  2.,   3.],
        [  4.,   5.]],

       [[  6.,   7.],
        [  8.,   9.],
        [ 10.,  11.]],

       [[ 12.,  13.],
        [ 14.,  15.],
        [ 16.,  17.]],

       [[ 18.,  19.],
        [ 20.,  21.],
```

```
          [ 22.,   23.]]])
>>> b.ndim
3
>>> b.itemsize
8
>>> b.strides
(48, 16, 8)
>>> b[0,...]
array([[ 0.,  1.],
       [ 2.,  3.],
       [ 4.,  5.]])
>>> b[0,0,...]
array([ 0.,  1.])
```

この例では NumPy の complex128 型 (128=64+64, 複素数の倍精度) と double 型 (倍精度) の配列を生成しています．最初の配列 a は一成分のみの配列です．二番目の配列 b は $4 \times 3 \times 2$ の多次元配列です．この配列の型は倍精度であるために成分の大きさは属性 itemesize から 8 byte, 次元は属性 ndim から 3 次元であることが判ります．属性 strides は各次元の移動に必要な byte 数のタプルが収納されています．この例の配列 a では a[0,...] の長さが 6×8 より 48 byte, a[0,0,...] の長さが 2×8 より 16-byte, そして, a[0, 0, 0] と a[0, 0, 1] の長さが 1×8 より 8-byte になります．属性 data は Python 2 では buffer 型, Python 3 では memoryview 型に切替わっています．この memoryview 型は buffer 型よりも Python 言語のオーバヘッドを伴わないためにより効率的なアクセスが可能で, その上, より高い汎用性を持っています.

構築子 array() によるオブジェクトの生成では MATLAB 風の記号 “:” を利用した数列の生成や記号 “;” を利用した配列の行指示ができません．構築子 array() が配列のデータとして取り込める書式はリストかタプルに限定されます．また, ndarray 型のオブジェクトの四則演算は成分単位の演算で, ベクトルと行列の演算を行うために NumPy には matrix 型が用意されています．この matrix 型は ndarray 型の派生クラスでベクトルと行列の表現に用いられ, 演算も ndarray 型の成分単位の演算ではなくベクトル・行列演算に対応するように積演算 “*” と冪演算 “**” が上書きされています．この matrix 型のオブジェクトの生成は構築子 matrix() を用い, その際に MATLAB 風の行列表記も可能です:

```
>>> import numpy as np
>>> a = np.matrix([[1,2,3],[5,4,3]])
>>> a
matrix([[1, 2, 3],
        [5, 4, 3]])
>>> type(a)
<class 'numpy.matrixlib.defmatrix.matrix'>
```

```
>>> b = np.array([[1,2,3],[5,4,3]])
>>> b
array([[1, 2, 3],
       [5, 4, 3]])
>>> type(b)
<type 'numpy.ndarray'>
>>> c = np.matrix(b)
>>> type(c)
<class 'numpy.matrixli.defmatrix.matrix'>
>>> c
array([[1, 2, 3],
       [5, 4, 3]])
>>> b is c
False
>>> d = np.array(a)
>>> d
array([[1, 2, 3],
       [5, 4, 3]])
>>> type(d)
<type 'numpy.ndarray'>
>>> a.T
matrix([[1, 5],
        [2, 4],
        [3, 3]])
```

matrix 型のオブジェクトと ndarray 型のオブジェクトは構築子 matrix() と構築子 ndarray() を使って互いの型に変換できます. また, 構築子 matrix() による行列の定義を記号 “;” を使って MATLAB 風に処理できますが, Python のオブジェクトに記号 “;” を使う表記がないために行列全体を文字列にする必要があります:

```
>>> m1 = np.matrix('[1, 2, 3; 4, 5, 6]')
>>> m1
matrix([[1, 2, 3],
        [4, 5, 6]])
>>> m2 = np.matrix('1, 2, 3; 4, 5, 6')
>>> m2
matrix([[1, 2, 3],
        [4, 5, 6]])
>>> m3 = np.matrix('1 2 3; 4 5 6')
>>> m3
matrix([[1, 2, 3],
        [4, 5, 6]])
>>> m4 = np.matrix('1:3;4:6')
>>> m4
```

```
matrix([[13],
        [46]])
```

最初の 3 種類が MATLAB 風の行列定義の方法ですが問題なく生成できていますが最後の記号 “:” を用いる例だけが本来意図した行列の生成に失敗しています. このように MATLAB 系言語で可能なスライス書式を使った行列の生成で意図した通りの行列が生成できないこともあり, MATLAB 系言語に慣れた方は注意が必要です.

NumPy の多次元配列の生成では, 最初に 1 次元配列として生成し, それから多次元配列としての形式をメソッド reshap() で変更するという方法もあります:

```
>>> a = np.random.random((4,3,2))
>>> a
array([[[ 0.89448666,  0.12482228],
        [ 0.15401515,  0.17503209],
        [ 0.20679215,  0.14524648]],

       [[ 0.89211856,  0.83433077],
        [ 0.32175697,  0.34052149],
        [ 0.31405685,  0.99350275]],

       [[ 0.10147491,  0.79617256],
        [ 0.81234934,  0.2516962 ],
        [ 0.28776398,  0.66480801]],

       [[ 0.99904254,  0.97522365],
        [ 0.68656026,  0.53876777],
        [ 0.67829143,  0.40291106]]])
>>> id(a)
80768496L
>>> a = a.reshape((4,6))
>>> a
array([[ 0.89448666,  0.12482228,  0.15401515,  0.17503209,
    0.20679215,
         0.14524648],
       [ 0.89211856,  0.83433077,  0.32175697,  0.34052149,
           0.31405685,
         0.99350275],
       [ 0.10147491,  0.79617256,  0.81234934,  0.2516962 ,
           0.28776398,
         0.66480801],
       [ 0.99904254,  0.97522365,  0.68656026,  0.53876777,
           0.67829143,
         0.40291106]])
```

```
>>> id(a)
80525952L
>>>a.reshape((4,3,2))
array([[[ 0.89448666,  0.12482228],
        [ 0.15401515,  0.17503209],
        [ 0.20679215,  0.14524648]],

       [[ 0.89211856,  0.83433077],
        [ 0.32175697,  0.34052149],
        [ 0.31405685,  0.99350275]],

       [[ 0.10147491,  0.79617256],
        [ 0.81234934,  0.2516962 ],
        [ 0.28776398,  0.66480801]],

       [[ 0.99904254,  0.97522365],
        [ 0.68656026,  0.53876777],
        [ 0.67829143,  0.40291106]]])
>>> a.shape
(4L, 6L)
```

この例ではモジュール random の函数 random() を使って $4 \times 3 \times 2$ の乱数配列を生成し, メソッド reshape() で 4×6 の配列に変換していますが, このメソッド reshape() は配列そのものを弄るメソッドではなく, その見栄え (view) を変更するソッドです. ただし, その出力値を変数 a に束縛してしまえば, メソッド reshape() で指示した配列の形への変更ができます. nadarray 型のオブジェクトの配列データは, 既定値では配列の行をメモリ上に連続して配置されますが, オブジェクトの生成時の指定で列優先にできます.

```
>>> c1 = np.array(range(6)).reshape((2,3),order='C')
>>> f1 = np.array(range(6)).reshape((2,3),order='F')
>>> c1
array([[0, 1, 2],
       [3, 4, 5]])
>>> f1
array([[0, 2, 4],
       [1, 3, 5]])
>>> c1.flags
  C_CONTIGUOUS : True
  F_CONTIGUOUS : False
  OWNDATA : False
  WRITEABLE : True
  ALIGNED : True
  UPDATEIFCOPY : False
```

```
>>> f1.flags
  C_CONTIGUOUS : False
  F_CONTIGUOUS : True
  OWNDATA : False
  WRITEABLE : True
  ALIGNED : True
  UPDATEIFCOPY : False
>>> mc1 = np.matrix(c1)
>>> mf1 = np.matrix(f1)
>>> mc1
matrix([[0, 1, 2],
        [3, 4, 5]])
>>> mf1
matrix([[0, 2, 4],
        [1, 3, 5]]
>>> mc1.flags
  C_CONTIGUOUS : True
  F_CONTIGUOUS : False
  OWNDATA : True
  WRITEABLE : True
  ALIGNED : True
  UPDATEIFCOPY : False
>>> mf1.flags
  C_CONTIGUOUS : True
  F_CONTIGUOUS : False
  OWNDATA : True
  WRITEABLE : True
  ALIGNED : True
  UPDATEIFCOPY : False
```

この例ではメソッド reshape() で配列の shape を変更していますが, その際にオプション order に'C' で行優先にしたときと'F' を指定することで列優先にしたことで得られる配列が同じ shape であっても, 実体が異なることに注意して下さい．なお, 行優先のときは行, 列優先のときは列がメモリ内部で連続し, その結果, 行と列の優先を入れ替えた配列は, そのビューが転置の関係になります．この配列の行優先か列優先であるかは配列の flag からも判断可能です．なお, 配列の生成で order の指定を行わなければ配列は行優先として生成されます．ここで NumPy の行列型 mattix の構築子 matrix() で行優先と列優先の指定ができません．つまり, matrix 型のオブジェクトでは行のみがメモリ上連続して配置されることを意味します．

5.5.3　特定の配列の生成

NumPy でよく使われる配列を生成する函数があります．ちなみに, ここで紹介する函数と同名の函数が MATLAB 系言語にも用意されています:

特殊な形の配列を生成する函数

zeros(shape [, dtype, order])		
ones(shape [, dtype, order])		
diag(shape [, dtype])		
eye(整数 [, dtype)]		
tri(整数, 整数, 整数 [, dtype])		
shape	::=	"(" | "[" 整数列 "]" | ")"
整数列	::=	整数 | 整数列 "," 整数
dtype	::=	"dtype=" keyword
order	::=	"FORTRAN" | "fortran" |"F"|
		"C" | "c"

■zeros():　全ての成分が 1 の配列を生成します．dtype を未指定のときは倍精度浮動小数点数 (float64) の型の配列になります.

```
>>> np.ones([2,3])

array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
```

■ones():　全ての成分が 0 の配列を生成します．dtype が未指定のときは倍精度浮動小数点数 (float64) の型の配列になります.

```
>>> np.zeros([2,3])

array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
```

■diag():　第 1 引数が 1 次元配列のときは, 引数の配列成分を対角成分に持つ 2 次元配列を生成し, 第 1 引数が 2 次元配列のときは, その配列の対角成分に対応する 1 次元配列を構築します．返却される配列の型は対角成分として与えた行 (タプル, あるいはリスト) の

成分で自動的に定まります．この函数 diag() は配列の行優先, 列優先の並びの指定ができません.

```
>> np.diag([1,2,3])

array([[1, 0, 0],
       [0, 2, 0],
       [0, 0, 3]])
>>> np.diag([1,2,3],3)

array([[0, 0, 0, 1, 0, 0],
       [0, 0, 0, 0, 2, 0],
       [0, 0, 0, 0, 0, 3],
       [0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0]])
>>> np.diag([1,2,3],-2)

array([[0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0],
       [1, 0, 0, 0, 0],
       [0, 2, 0, 0, 0],
       [0, 0, 3, 0, 0]])

>>> A=np.random.randn(3,3)
>>> A

array([[-0.37655391, -0.80374079,  0.12192416],
       [-0.50526631,  1.400896  , -0.01749459],
       [-0.64377851, -1.88182683, -2.361904  ]])
>>> np.diag(A)
array([-0.37655391,  1.400896  , -2.361904  ])
```

■eye()　指定した大きさで対角成分のみが 1 の 2 次元配列を生成します．dtype が未指定のときは倍精度浮動小数点数 (float64) の型の 2 次元配列になります．この函数 eye() も行優先か列優先かの並びの指定ができません.

■tri(), triu(), tril():　2 次元配列で, 三角行列に対応する配列を生成．これらの函数も行優先か列優先かの成分の並びの指定ができません.

対角成分のみが 0 でない配列, 三角行列に対応する配列に関しては行か列の成分の指定ができませんが, これらの書式の行列に対して, BLAS には効率的な 1 次元配列への格納

方法があります．そのために指定をする必要がないと考えるべきでしょう．

5.5.4　その他の配列の生成方法

CSV 形式のファイルを読み込んで配列を生成する函数 loadtxt() や既存の配列の複製で新たに配列を生成する函数 copy() やバッファから配列を生成する函数 frombuffer() 等が用意されています．ここでは函数 loadtxt() と函数 copy() について簡単に解説しておきます．

■loadtxt():　CSV 形式のファイルを読み込んで配列を生成する函数です．一般的なヘッダや註釈が CSV 形式のデータが配置されているファイルに対して．函数 loadtxt() に読み飛ばすべきヘッダ行数をオプションの skiprows, 注釈行の先頭文字をオプションの comments にリストで与え, 実データの区切文字の指定をオプションの delimiter, 読み込むべき列の指定をオプションの usecols で与えることで容易に対処できます．さらに一つの配列ではなく, 列単位で出力させられ, この場合はオプションの unpack を既定値の True から False に変更します．ちなみに, ndarray 型の配列を書き出す函数が函数 savetxt() で, この函数では保存すべき一つの配列とその書式, 区切文字, 改行文字, ヘッダとフッタ, それと注釈行の指定ができます．なお, カラムのタグ付きで, 要するに Excel のシートのようにデータを読込みたければパッケージ Pandas が実現するデータフレームとして read_csv() を使って読込み, 書込はメソッド to_csv() を使うと良いでしょう．

■copy():　配列の複製を行う函数/メソッドです．Python では名前はオブジェクトを指示する名札のようなもので, ‘a = b’ は名前 b のオブジェクトを名前 a に複製するという意味になるとは限らず, 通常は名前 b が指示するオブジェクトに名前 a でも指示できるという意味です．すなわち, 名前 a で指示されるオブジェクトが変更可能なときに名前 a でメソッドや函数を使ってオブジェクトを変更すると, そのオブジェクト自体が変更されます．ところで, 名前 b はそのオブジェクトの別名であるために名前 b でオブジェクトを参照すると, さきほどの名前 a で変更を加えた値が表示されることになります．多くの言語では ‘a = b’ と指示したあとに変数 b の値を変更しても, 変数 a に束縛された値はもとのままであるのとは状況が異なります．そして, ndarray 型のオブジェクトは変更可能なオブジェクトであるために不用意に ndarray 型のオブジェクトに変更を加えると, このような意図しない置き換えが生じます．この状況を確実に避けるための一つの確実な方法として, あらかじめ函数 copy() で配列の複製を生成し, それから本体, あるいは複製のどちらか一方に手を加えるようにします．なお, オブジェクトが数値のような書き換えができないオブジェクトであれば, 名前 a の値を書換えても名前 a のオブジェクトが新しいオブジェクトへの指示として更新されるだけで, 名前 b と本来のオブジェクトの関係に変化がないために, 名前 a の値のみが更新され, 名前 b の値に変化はありません．

■**Python の内包表現**: NumPy で定義される函数は基本的に ndarray 型の配列の直接処理が可能です. たとえば, ndarray 型の配列からその正弦函数 sin の値を計算したければ, 'np.sin(x)' のように記述するだけで十分で, for 文や Python の内包表現を併用する必要はありません. ただし, Python 言語の内包表現には表現力があります. ここで Python の内包表現は '[2*i-1 for i in range(10)]' のように内部に for 節や if 節を伴うリストの生成方法で, この方法では Python のリストを生成し, それを ndarray 型のオブジェクトに変換する手順になります. そこで同じ ndarray 型の配列を最初は内包表現, for 文と NumPy の函数のみで生成したときの処理時間の比較をしてみましょう:

```
>>> import time
>>> import numpy as np
>>> def TEST1():
...     a = np.array([np.sin(i) for i in range(10000)])
...
>>> def TEST2():
...     a = []
...     for i in range(10000): a.append(np.sin(i))
...
>>> def TEST3():
...     a = np.sin(np.arange(10000))
...
>>> s1 = time.time();TEST1();print(time.time()-s1)
>>> s1 = time.time();TEST1();print(time.time()-s1)
0.0112931728363
>>> s1 = time.time();TEST2();print(time.time()-s1)
0.0159051418304
>>> s1 = time.time();TEST3();print(time.time()-s1)
0.0004589557647705078
```

このように内包表現は for 文と比較して処理時間が多少ましとは言え, NumPy の函数だけの処理と比較して非常に遅い処理であることが判ります. NumPy で定義される数値函数は ndarray 型の配列を直接処理できるために複雑な処理が必要とされない限り, for 文と同様に内包表現も可能な限り避けるべきです.

5.5.5 スライス表記による配列の生成と成分の取り出し

Python の列の添字は MATLAB 系の言語と異なり 0 から開始する点と, スライス表記で, 始点, 終点と刻幅の順で, スライス表記の末尾に刻幅を記載する点が異なります. また, MATLAB 系言語と異なりスライス表示が生成的な性格を持たないためにスライス表示は配列の生成では関係しません. あくまでも既存の列に対する部分列の取出のための添字の

表記方法です．つぎに NumPy の構築子 array() を使った ndarray 型の配列の生成と成分の取出の例を示しておきましょう:

```
>>> a = np.array([i*0.2 for i in range(0,11)])
>>> a
array([ 0. , 0.2, 0.4, 0.6, 0.8, 1. , 1.2, 1.4, 1.6, 1.8,
      2. ])
>>> a[0]
0.0
>>> a[-1]
2.0
>>> a[0:4]
array([ 0. , 0.2, 0.4, 0.6])
>>> a[4:]
array([ 0.8, 1. , 1.2, 1.4, 1.6, 1.8, 2. ])
>>> a[:]
array([ 0. , 0.2, 0.4, 0.6, 0.8, 1. , 1.2, 1.4, 1.6, 1.8,
      2. ])
>>> a[0:11:2]
array([ 0. , 0.4, 0.8, 1.2, 1.6, 2. ])
>>> a[:11:2]
array([ 0. , 0.4, 0.8, 1.2, 1.6, 2. ])
>>> a[0::2]
array([ 0. , 0.4, 0.8, 1.2, 1.6, 2. ])
>>> a[::2]
array([ 0. , 0.4, 0.8, 1.2, 1.6, 2. ])
```

Python で列の添字は 0 から開始しますが, 配列の末尾の成分は-1 で直接指示できる点は MATLAB 系言語と同様で, 以降, -2, -3,... と負数を指定すれば配列の末尾から順に要素が取り出せる点も同様です．また, スライス表記を使って配列の一部分を切り出すような MATLAB 系言語と同様の添字処理ができます．注意すべきことは配列が 0 から開始する点とスライス表記が $[n_0 : n_1]$ のときに末端 n_1 が MATLAB 系言語では含まれていても, Python では含まれない点, それと配列の増分の指定の仕方が MATLAB 系と異なり, $[n_0 : n_x : dn]$ と添字の増分 dn を末尾に記載します．これらの点は MATLAB 系の言語と Python 言語の間でプログラムを移植するときに重要です．次に 2 次元配列でスライス表記の例を示します:

```
>>> a = np.array(range(0,10)).reshape(2,5)
>>> a
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])
>>> a.shape
(2,5)
```

```
>>> a[:,1]
array([1, 6])
>>> a[:,2:4]
array([[2, 3],
       [7, 8]])
>>> a[0,:]
array([0, 1, 2, 3, 4])
>>> a[0:2,2:5]
array([[2, 3, 4],
       [7, 8, 9]])
>>> a[...]
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])
>>> a[...,1]
array([1, 6])
>>> a[1,...]
array([5, 6, 7, 8, 9])
```

と, これも MATLAB と同様です. ただし, ここで Ellipsis"..." を入れています. これは記号 ":" で始点と終点を入れなかったときと同様の「**省略**」の意味もありますが, この Ellipsis にはそれ以上に「**次元の省略**」の意味があります. たとえば, 'a[1, ...]' と 'a[..., 1]' は 'a[1, :]' と 'a[:, 1]' と同じ結果ですが, 'a[...]' は配列 a 全てを指示しています. このことは 3 次元以上の配列を操作すると一層, 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 $3 \times 2 \times 4$ の配列を生成しています．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できるでしょう．このように Ellipsis“...” は次元を含めた省略記号, 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．このことからも判るように NumPy は多次元配列を効率的に扱うために配列の添字機能が拡張されています．ちなみに数式処理システム Maple で演算子 “...” は数列を生成する演算子としての意味を持ち, SageMath では Ellipsis に類似 した演算子 “..” が同様の数列の生成機能を持ちます:

```
sage: [1..10]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
sage: type([1..10])
<type 'list'>
sage: type((1..10))
<type 'generator'>
```

この演算子は SageMath 独自のもので, Python+NumPy の機能ではありません．

5.5.6　NumPy の配列演算

MATLAB 系言語では, 加減算, 積, 商, 冪といった基本的なベクトルや行列演算は数式に近い表記です．NumPy を取り込んだ Python もこの点は同様ですが, 演算子の機能に関して微妙な違いがあります．これは NumPy の ndarray 型の多次元配列の演算が同じ shape の対象に対する成分単位の演算で, 冪乗の演算子が MATLAB 系の言語で用いられる演算子 “^” ではなく演算子 “**” を使う点です．ちなみに MATLAB 系言語で成分単位の演算は俗に「**dot 演算子**」と呼ばれる演算子群, 積 “.*”, 商 “./” と冪 “.^” を用います．また, NumPy でベクトルや行列の積を計算するメソッドとして dot() がありますが, こちらは演算子でないために使い勝手の良いものではありません．そこでクラス ndarray を

継承したクラス matrix を使い, その積 “*” と冪 “**” がベクトル・行列演算に対応した演算子になっています. このように配列の成分単位の演算が中心であれば ndarray 型, 行列・ベクトル演算が中心であれば matrix 型と使い分けができます.

なお, SageMath でベクトル・行列に対する積 “*” と冪 “^”,“**” は通常のベクトル・行列の演算であって, ndarray 型の各演算のように成分単位の演算ではありません. 逆に言えば, 成分単位の演算を行うためにはメソッド apply_map() の利用や NumPy の ndarray 型として計算を行うといった工夫が必要です. ただし, このメソッドの処理は for 文を使った処理であるために処理速度に難があります. ndarray 型に変換して NumPy の数値配列として処理できるような SageMath の行列にはメソッド apply_map() を利用すべきではなく, 可能な限り NumPy 側の処理で置換えるべきです.

5.5.7 配列のパターンマッチング処理

NumPy の ndarray 型配列でも MATLAB 系言語と同様の処理が可能です. この MATLAB の重要な処理の一つのパターンマッチング処理があります. これは配列の成分に対する条件式に合致する成分を, 真であれば 1 か True, 偽であれば 0 か False に置換える処理です. たとえば, 配列 a の 1 より大の成分を 1, 1 以下の成分を 0 とする配列が ‘a <1’ で容易に生成できます. この機能に関連して, 条件式に合致する成分を取り出す函数もあります. たとえば, MATLAB 系言語では函数 find() が条件に合致する添字を出力する函数 find() があり, NumPy では函数 where() が MATLAB の函数 find() に対応します:

```
sage: a = np.random.randn(5,5)
sage: a

array([[-0.30877266, -0.57845967,  1.44142409, -0.33045793,
    0.00515611],
       [-1.73495952, -0.60578728, -0.01251624,  1.39613281,
        0.24274784],
       [-1.87836239,  1.81042014, -1.64739157, -0.84597255,
        -0.34525553],
       [-2.21983147,  1.874392   , -2.80169616, -1.75605379,
        2.49597114],
       [-1.63498591,  0.21757873,  0.81173105, -0.38778725,
        -1.21050712]])
sage: a>1

array([[False, False,  True, False, False],
       [False, False, False,  True, False],
       [False,  True, False, False, False],
```

```
        [False,  True, False, False,  True],
        [False, False, False, False, False]], dtype=bool)
sage: np.where(a>1)
(array([0, 1, 2, 3, 3]), array([2, 3, 1, 1, 4]))
sage: a[np.where(a>1)]
array([ 1.44142409,  1.39613281,  1.81042014,  1.874392  ,
    2.49597114])
sage: a[a>1]
array([ 1.44142409,  1.39613281,  1.81042014,  1.874392  ,
    2.49597114])
sage: a[(lambda x: x>1)(a)]
array([ 1.44142409,  1.39613281,  1.81042014,  1.874392  ,
    2.49597114])
sage: 10*a*(a>1) - 3*(a<=1)

array([[ -3.        , -3.        , 14.4142409, -3.        , -3.
    ],
        [ -3.        , -3.        , -3.        , 13.96132813, -3.
           ],
        [ -3.        , 18.10420137, -3.        , -3.        , -3.
           ],
        [ -3.        , 18.74391995, -3.        , -3.        ,
           24.95971141],
        [ -3.        , -3.        , -3.        , -3.        , -3.
           ]])
```

この例では函数 randn() で 5×5 の乱数配列を生成し, 函数 where() を使って, この配列で 1 より大になる成分の添字を検出し, この出力を添字として引き渡すと対応する成分のリストが返されます. この条件式に合致する成分の返却は函数 where() を使わなくても配列の内包式, つまり, 配列の添字として条件式を与えることで返却されます. これは Python 独自の仕様です[*19] また, ‘a>1’ は各成分で 1 より大になれば True, そうでなければ False で置換えた配列を生成します. つまり, for 文を使って添字操作をしなくても, 各成分が一律で充すべき式を与えることで容易に式の判断結果の配列が得られます. さらに, ‘10*a*(a>1)-3*(a<0)’ がパターンマッチングを使った式で, 配列 a の成分で 1 より大の成分を 10 倍, 0 より商の成分を-3 倍し, それ以外の成分を 0 にするという意味になります. この表記によって成分に対する if 文と for 文を利用せずに配列の積と和で表現できています. MATLAB 系言語や NumPy ではこのような配列/行列に対するパターンマッチングと配列演算を上手く利用することで処理言語のオーバヘッドの低減が可能です. と

[*19] MATLAB 系言語で Yorick がさらに多次元配列処理に関して優れた添字処理を持っています. ちなみに Yorick で「ゴム添字 (rubber index)」に対応する添字処理が可能な言語は Python 程度です.

は言え, どうしても大きな配列に対して for 文を利用しなければならないことがあります. そのときはできるだけ大きな ndarray 配列同士の演算になるように処理する工夫を行って for 文で処理する回数を減らし, それで十分でなければ Cython や Numba の利用, あるいは Python ではなく Julia の利用を検討すべきです. ただし, 行列の規模が大きくなればなるほど, NumPy が利用する BLAS の効果が顕著になります.

5.5.8　演算処理速度の比較

MATLAB 系言語で積, 冪と函数式で成分単位の同じ結果になる演算で処理速度を比較すると積演算が最も速く, 冪演算は比較的重く, 函数式は冪の指数の大きさの影響を直接受けないために, ある程度大きな演算になると冪よりも函数演算の方が逆転する現象があります. この傾向を SageMath でも確認してみましょう. この検証では配列の 2 乗, 3 乗と 12 乗の計算を行い, その処理速度を計測します. 計算する配列は 1000 成分の 1 次元の乱数配列で, NumPy の函数 random.randn() で生成します. SageMath の行列は, この random.randn() の結果を構築子 matrix() で SageMath の matrix 型に変換します. ここで SageMath の浮動小数点数には任意精度のものと倍精度があるために双方の比較もしてみましょう. ただし, 任意精度とは言え, 既定値では仮数部が 53 bit と実質的に倍精度浮動小数点数と同じです. そのために精度を 256 bit にしたものも計測してみます. この計算では MacBook Pro(Core i5, 1.4GHz) を用い, ndarray 型の配列計算で処理する内容を以下に示します:

```
b = np.random.rand(1000)
b[1]
M=[]
t1=time.perf_counter();b*b;t2=time.perf_counter();
M.append(t2-t1);
t1=time.perf_counter();b**2;t2=time.perf_counter();
M.append(t2-t1);
t1=time.perf_counter();np.exp(2*np.log(b));t2=time.
    perf_counter();
M.append(t2-t1);
t1=time.perf_counter();b*b*b;t2=time.perf_counter();
M.append(t2-t1);
t1=time.perf_counter();b**3;t2=time.perf_counter();
M.append(t2-t1);
t1=time.perf_counter();np.exp(3*np.log(b));t2=time.
    perf_counter();
```

```
M.append(t2-t1);
t1=time.perf_counter();b*b*b*b*b*b*b*b*b*b*b*b;t2=time.
    perf_counter();
M.append(t2-t1);
t1=time.perf_counter();b**12;t2=time.perf_counter();
M.append(t2-t1);
t1=time.perf_counter();np.exp(12*np.log(b));t2=time.
    perf_counter();
M.append(t2-t1);
```

これらの式は ndarray 型の配列 b に対して行う積, 冪, 函数式で, 内容は 2 乗, 3 乗と 12 乗の三種類です. また, 比較のために SageMath の行列を使って同様の計算をさせます:

```
A = random_matrix(RR,100,100)
A[0,0]
M=[]
t1=time.perf_counter();A.elementwise_product(A);t2=time.
    perf_counter();
t1=time.perf_counter();A.apply_map(lambda x:x*x);t2=time
    .perf_counter();
M.append(t2-t1)
t1=time.perf_counter();A.apply_map(lambda x:x^2);t2=time
    .perf_counter();
M.append(t2-t1)
t1=time.perf_counter();A.apply_map(lambda x:exp(2*log(x
    )));
t2=time.perf_counter();M.append(t2-t1)
t1=time.perf_counter();A.apply_map(lambda x:x*x*x);t2=
    time.perf_counter();
M.append(t2-t1)
t1=time.perf_counter();A.apply_map(lambda x:x^3);t2=time
    .perf_counter();
M.append(t2-t1)
t1=time.perf_counter();A.apply_map(lambda x:exp(3*log(x
    )));
t2=time.perf_counter();M.append(t2-t1)
t1=time.perf_counter();A.apply_map(lambda x:x*x*x*x*x*x*
```

```
    x*x*x*x*x*x);
t2=time.perf_counter();M.append(t2-t1)
t1=time.perf_counter();A.apply_map(lambda x:x^12);t2=
    time.perf_counter();
M.append(t2-t1)
t1=time.perf_counter();A.apply_map(lambda x:exp(12*log(x
    )));
t2=time.perf_counter();M.append(t2-t1)
```

これらの例では SageMath の matrix 型のオブジェクトに対して成分単位の演算をメソッド apply_map() で実現させています．このメソッドの引数として無名函数を記述すれば, 各成分に対してその処理が実行されます．ただし, 積のみは成分単位の積演算を行うメソッド elementwise_prodoct() があります．そのために最初にメソッド elementwise_product() を用いた 2 乗の計算を追加しています．それから, matrix 型のオブジェクトの生成で, 成分を実数とするときに何も指定しなければ任意精度の浮動小数点数のクラス RR のインスタンスとして生成されます．なお, 任意精度とはいえ精度をあらかじめ指定しなければ 53 bit の浮動小数点数で計算されます．そのために構築子 RealField() に精度を指定して, その精度の行列として生成します．また, ndarray 型の配列は倍精度浮動小数点数 float64 を成分とする配列として生成しているために倍精度浮動小数点数のクラス RDF の行列の計算も追加しておきます:

配列/行列データの詳細

ndarray 型	10*np.abs(np.random.randn(10000))+eps
ndarray 型	10*np.abs(np.random.randn(100,100))+eps
matrix 型	random_matrix(RR,10000,1).apply_map(lambda x:abs(x))
matrix 型	random_matrix(RealField(256),10000,1).apply_map(lambda x:abs(x))
matrix 型	random_matrix(RDF,10000,1).apply_map(lambda x:abs(x)+eps)
matrix 型	random_matrix(RR,100,100).apply_map(lambda x:abs(x))
matrix 型	random_matrix(RealField(256),100,100).apply_map(lambda x:abs(x))
matrix 型	random_matrix(RDF,100,100).apply_map(lambda x:abs(x)+eps)

NumPy のインスタンスに対しては機械イプシロン eps(= np.finfo(float).eps) を成分に

加えておきます．その理由は log 函数の引数が常に 0 より大とすることで 0 のときの処理停止を避けるための工夫です．SageMath の多倍長浮動小数点数 RR でその心配が少ないために eps を加えてはいませんが, 倍精度浮動小数点数 RDF では NumPy と同じ理由で加えています．なお, eps は倍精度浮動小数点数であるために多倍長の RR のインスタンスとの演算結果は倍精度浮動小数点数 RDF になることに注意して下さい．では, 最初に ndarray 型の配列の shape が 10000 と 100×100 で比較した結果です:

NumPy: float64

式	10000-成分	100×100-成分
$x * x$	0.000080	0.000113
$x * *2$	0.000062	0.000118
$\exp(2\log x)$	0.000393	0.000619
$x * x * x$	0.000091	7.1e-05
$x * *3$	0.000101	0.000353
$\exp(3\log x)$	0.000102	0.000258
$\underbrace{x * \cdots * x}_{12}$	0.0000935	0.000156
$x * *12$	0.0000902	0.000284
$\exp(12\log x)$	0.0000948	0.000239

これらの結果は厳密ではありませんが, おおよその傾向の把握には十分と判断しています．配列の形に関しては 10000 の大きさの配列よりも 100×100 の大きさの配列の処理がやや速くできる傾向があり, 処理内容に関しては函数式よりも冪が比較的重い処理であり, 通常の積は計算量の増加に比例し, 函数は指数の大きさに比較的左右されずに処理ができているという常識的な結果が得られています. また, これらの結果から冪 “**” は単純な積ではなく, 函数式であることも判ります．これと同様の処理を SageMath で行った結果を示しておきましょう:

SageMath: 10000×1

式	RR	RealField(256)	RDF
elementwise_product()	0.003375	0.004154	0.003724
$x * x$	0.357358	0.295088	0.277982
xˆ2	0.292818	0.281530	0.323870
$\exp(2 \log x)$	0.275750	0.281344	0.287341
$x * x * x$	0.281486	0.316250	0.260350
xˆ3	0.337786	0.269038	0.259678
$\exp(3 \log x)$	0.299361	0.283587	0.268591
$\underbrace{x * \cdots * x}_{12}$	0.277474	0.268872	0.266026
xˆ12	0.275138	0.271303	0.263398
$\exp(12 \log x)$	0.276284	0.267231	0.263766

ここでの処理は 10000×1 の SageMath の数値行列に対して ndarray 型と類似の計算を行っています．全体的に多倍長 RR と倍精度浮動小数点数 RDF の処理速度はほぼ同じです．詳細を見ると, 積は冪や函数よりも処理が速いものの, 積の回数が増えると冪や函数が有利になる傾向が見られます．また, 処理時間が長い処理では浮動小数点数 RDF の処理が僅かに速い傾向が見られます．また,apply_map(lambda x:x*x) よりもメソッド elementwise_product() の方が 100 から 70 倍程度高速ですが, それでも ndarray 型の積演算と比較して 1/50 程度でしかありません．つぎに 100×100 行列で同じことをさせてみましょう:

SageMath: 100×100

式	RR	RealField(256)	RDF
elementwise_product()	0.003788	0.004377	0.003818
$x * x$	0.299374	0.301662	0.294724
xˆ2	0.284338	0.330068	0.277884
$\exp(2 \log x)$	0.283143	0.273411	0.294427
$x * x * x$	0.287572	0.278994	0.278699
xˆ3	0.338518	0.276039	0.277193
$\exp(3 \log x)$	0.299978	0.295962	0.273921
$\underbrace{x * \cdots * x}_{12}$	0.282424	0.319068	0.287519
xˆ12	0.293768	0.285631	0.282770
$\exp(12 \log x)$	0.340085	0.299616	0.278488

ndarray 型の演算と比べて SageMath の行列の処理時間は行列の形に依存する傾向が見られます．また，データ型が多倍長であろうが，浮動小数点数であろうがその処理時間の差は僅かです．これらの結果が得られた理由として考えられることがメソッド apply_map() の処理内容です．このメソッドの実体は実質的に for 文である Python の内包表現を使ったマッピング処理です．そのために Python 言語のオーバーヘッドの影響が，RR と RDF の違い以上に顕著であるために，差が見られないと考えられます．このように Python という処理言語が介入すると速度低下の大きな原因になります．

最後に行列計算を比較してみましょう．ここでは 100×100 の正方行列で，行列積を比較します．そのために成分単位の計算で用いた函数式を外して計測します．

SageMath: 100×100

式	ndarray	RR	RealField(256)	RDF
$x * x$	0.000732	0.516586	0.607733	0.001098
x^ 2	0.000748	0.552291	0.595384	0.000648
$x * x * x$	0.000710	1.028639	1.168670	0.000588
x^ 3	0.000628	1.021157	1.191532	0.002221
$\underbrace{x * \cdots * x}_{12}$	0.002527	5.530172	6.375411	0.002397
x^ 12	0.000959	2.032416	2.328560	0.000892

ここで倍精度と任意精度の浮動小数点数の差が明確に出ました．NumPy の ndarray 型と RDF 型がほぼ同等の処理速度になりますが，RR 型はこれらと格段の遅くなります．RR 型も既定値の精度 (prec=53) はより高精度 (prec=256) よりも幾分高速ですが，それでも RDF 型の 500 倍もの処理時間を要しています．

以上の結果から．数値計算では NumPy の利用が数値配列の成分単位や数値行列の演算処理で有利と言えます．とは言え，数値行列の処理では SageMath の倍精度浮動小数点数を用いる RDF 型であれば NumPy とほぼ同等の処理速度が期待できます．このように使い方次第で SageMath の数値計算は従来の数式処理の弱点を十二分に克服したものになっています．

5.6　Python の数の構成

5.6.1　Python における数オブジェクトの扱い

おおよその計算機の CPU には整数，実数といった数の表現とそれらの主要な演算は最初から実装されています．実際，自然数や整数は 2 進数で表現され，実数は浮動小数点数と呼ばれる 2 進数表現への符号化として近似され，さらに浮動小数点数はその四則演算，

大小関係やいくつかの演算, 丸めの処理も含めて IEEE 754 で規格化されています. そのために計算機言語で, 学習目的以外で数そのものと演算をフレーゲの「算術の基礎」や集合論のように最初から全てを構成する必要がなく, ハードウエアに実装されている機能をどのように効率良く使うかが問題になります. ただ, オブジェクト指向言語では扱う対象が全てオブジェクトで, それらの関係を利用して処理を行うという性格上, 数についても相互の関係を明確にしておく必要があります. たとえば, 整数, 有理数と実数には集合として包含関係があり, これらの算術では共通の演算子を用いています. そして個々の数はそれらの集合の成員, 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. そして, そのメソッドがどのクラスから派生したものであるかを捉えることで, それらが保持するメソッドの体系的な捉え方が可能になります. この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.

Python の数のクラス (Numeric Class) の定義は PEP-3141[*20]に沿って行われています. この PEP-3141 は PEP-3119[*21]で導入された「**抽象基底クラス (ABC, Abtract 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 2 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, これらの型は 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.

5.6.2 numbers モジュールを眺めてみよう

こと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:
```

[*20] 原文: http://www.python.org/dev/peps/pep-3141/
[*21] 原文: http://www.python.org/dev/peps/pep-3119/
日本語訳：http://mft.la.coocan.jp/script/python/pep-3119.html

```
NAME
    numbers - Abstract Base Classes (ABCs) for numbers, according
        to PEP 3141.

FILE
    /usr/local/Cellar/python/2.7.9/Frameworks/Python.framework/
        Versions/2.7/
    lib/python2.7/numbers.py

MODULE DOCS
    http://docs.python.org/library/numbers

DESCRIPTION
    TODO: Fill out more detailed documentation on the operators.

CLASSES
    __builtin__.object
        Number
            Complex
                Real
                    Rational
                        Integral
```

ここでは macOS 上の SageMath での函数 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-子) を示しています. この階層構造は「**数値塔 (Numerical tower)**」と呼ばれる Python の数オブジェクトの階層構造に対応します. ただし, 数値塔では整数, 有理数, 実数, 複素数と派生クラスを塔の上側に配置するために, この CLASSES での表記と逆になります. この数値塔で重要なことは塔の上側の型 (継承する側) とその土台の型 (継承される側) との間の演算で土台の型に「**自動変換 (暗黙の型変換, implicit conversion)**」が行われ, その逆は明示的に型変換を行わない限り生じません. たとえば, 整数の和, 積といった演算は整数内部で閉じ, 実数を表現する浮動小数点数へは型の変換が必要です. しかし, 整数と浮動小数点数の演算では自動的に浮動小数点数の演算として型の変更が行われ, 結果も浮動小数点数になるために型変換は不要です. この型の自動変換が「**暗黙の型変換**」です.

また, このオンラインヘルプの FILE の項目にモジュール numbers.py の在処が掲載さ

れています. そこで, ファイル numbers.py からクラス Number の定義を抜き出しておきましょう;

```
# Copyright 2007 Google, Inc. All Rights Reserved.
# Licensed to PSF under a Contributor Agreement.

"""Abstract Base Classes (ABCs) for numbers, according to PEP
    3141.

TODO: Fill out more detailed documentation on the operators."""

from __future__ import division
from abc import ABCMeta, abstractmethod, abstractproperty

__all__ = ["Number", "Complex", "Real", "Rational", "Integral"]

class Number(object):
    """All numbers inherit from this class.

    If you just want to check if an argument x is a number,
        without
    caring what kind, use isinstance(x, Number).
    """
    __metaclass__ = ABCMeta
    __slots__ = ()

    # Concrete numeric types must provide their own hash
        implementation
    __hash__ = None
```

クラス Number の定義では import 文を使ってモジュール abs からクラス ABCMeta 等の読込とクラス属性___metaclass___に 'ABCMeta' を割り当てています. これらの設定は Python の抽象基底クラスの定義で必須です. ところで, このクラス Number にはメソッドの設定がなく, クラス属性の設定だけです. すなわち, このクラス Number の役割は数という枠組を提供することが目的で, 数が何であり, それがどのような性質であるかを指定することではありません. 数の本質的な属性やメソッドは, クラス Number を継承するクラス Complex 等のサブクラスに記載されます. そこで今度は複素数を表現する抽象基底クラス Complex の numbers.py からの抜粋を載せておきましょう;

```
class Complex(Number):
    """Complex defines the operations that work on the builtin
        complex type.

    In short, those are: a conversion to complex, .real, .imag, +,
         -,
    *, /, abs(), .conjugate, ==, and !=.

    If it is given heterogenous arguments, and doesn't have
        special
```

```
    knowledge about them, it should fall back to the builtin
        complex
    type as described below.
    """

    __slots__ = ()

    @abstractmethod
    def __complex__(self):
        """Return a builtin complex instance. Called for complex(
            self)."""

    # Will be __bool__ in 3.0.
    def __nonzero__(self):
        """True if self != 0. Called for bool(self)."""
        return self != 0

    @abstractproperty
    def real(self):
        """Retrieve the real component of this number.

        This should subclass Real.
        """
        raise NotImplementedError

-- 略 --

    @abstractmethod
    def __eq__(self, other):
        """self == other"""
        raise NotImplementedError

    def __ne__(self, other):
        """self != other"""
        # The default __ne__ doesn't negate __eq__ until 3.0.
        return not (self == other)

Complex.register(complex)
```

抽象基底クラス Complex の定義からな特徴的なメソッドの抜粋をここに掲載していますが, このクラス Complex では複素数の四則演算を含む演算に関連するメソッドと同値性に関連するメソッド等がデコレータを使って定義されています．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を使って記述されます.

このモジュール numbers で定義されているクラスは Complex, Real, Rational, Integral で, これらの構成方法は実利的な側面があります. ここで図式的に示すためにクラス A がクラス B のサブクラスであることを $A \to B$ と表記します. 集合論の数の構成に従うのであれば $\mathbf{N} \to \mathbf{Z} \to \mathbf{Q} \to \mathbf{R} \to \mathbf{C}$ と「**塔**」の上側から順に数を定義し, 演算も順次拡張することで数値塔の継承順と逆の構成順になりますが, 現実にはこれらのオブジェクトとその中で閉じた演算がメソッドとして既に実装されています. たとえば, 複素数では複素数同士の演算はもちろん, 複素数と実数, 複素数と有理数, そして複素数と整数の演算は, 実数, 有理数と整数が複素数のサブクラスであることから問題なく複素数の領域で行えますが, たとえば, 二つの整数 $2, 3$ の積 2×3 の結果を $6 + 0i$ や $6.0e - 0$ と複素数や実数 (浮動小数点数) と整数よりも上の階のインスタンスとして返却する必要はありません. つまり, 数値塔の上層のインスタンスとの演算では上層のメソッドをそのまま継承, 同じ層のインスタンス同士の演算では同じ層で閉じるようメソッドを再定義, 下の層のインスタンスとの演算では, 結果がその層のインスタンスになるようにメソッドを再定義したものとなり, 結果に対する制約がそのままメソッドへの制約になる形で再定義されることになります. つまり, この塔構成はメソッドへの制約の入り方と密接に関連し, 数学上の数の成り立ちとは別のものです. そして, Python ではこれらの数と演算は, 組込のオブジェクトとして個別に用意されており, 抽象基底クラスを導入することで後付的に数オブジェクトとして継承関係も含めて再定義しています. もちろん, この構成は計算処理だけが目的であれば, あってもなくても良いものかもしれませんが, 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り, この抽象化は多項式等の数の代数的な拡張で威力を発揮します.

5.7 SageMath の数の構成

5.7.1 SageMath の数

SageMath は整数, 有理数, 実数と複素数といった数オブジェクトを Python の数オブジェクトからそのまま継承していません. この理由としては

- 整数が符号付き 32bit 整数と符号付き 64bit 長整数の異なるリテラルを持つオブジェクトに分かれている.
- 有理数が実装されていない.
- 浮動小数点数が倍精度に固定されていて任意精度ではない.
- 算術演算がそもそも効率の良いものではない.

といった点が挙げられます．最初の整数のリテラルの問題は Python 2 の問題で, Python 3 のそれではありませんが, さらに SageMath が Python で数学を表現する性格上, Python の数が妥当なものであるとは言い難いものです．これらの問題に対処するために GMP(GNU MP, GNU Multi Precision Arithmetic Library) と GMPFR(GNU Multiple Precision Floating-point Reliablity) を使って整数, 有理数, 実数と複素数とそれらの基本演算を定義し直しています[22]．そのために SageMath で数式を構成する数が Python に既存の数ではなく SageMath 独自の数オブジェクトである理由です.

5.7.2 GMP の概要

GMP は C で記述された整数, 有理数と浮動小数点数の任意精度で演算を行うための GPL のライブラリです．なお, ここでの任意精度とは計算機のメモリ等のハードウエ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る可能性, 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 が提供する以上の精度で可能な限り最速の数値計算を実現することを目的にしています．そのために GMP の計算最適化には Intel や AMD 等の主要な CPU 向けに最適化されたアセンブリコードが含まれています．また, GMP は基本的に四則演算を提供するものであり, 指数函数, 対数函数と三角函数の計算機能は含まれていません．そのために GMP を基盤に構築された MPFR ライブラリがあり, 複素函数向けには GNU MPC ライブラリがあります.

ここで計算対象が数値行列であっても, その計算精度が高精度であることが要求されず, むしろ, その処理速度が要求されるのであれば任意精度ではなく倍精度の浮動小数点数による数値計算を検討すべきで, その場合は最低でも SageMath で倍精度浮動小数点数に限定して数値計算を実行するか, 大規模行列計算, 処理速度が強く要求されるときは Python のパッケージの NumPy を直接利用した計算, あるいは Cython の利用を検討すべきです.

ここで GMP の定める数を以下に示します:

SageMath の数の構成に必要な GMP のクラス

mpz_t	符号付整数とその演算, および名前が mpz_で始まる約 150 の函数
mpq_t	mpz_t 型の整数から構成される有理数とその演算, および名前が mpq_で始まる約 35 の函数
mpf_t	浮動小数点数とその演算, および名前が mpf_で始まる約 70 の函数

[22] GMP を利用している主要なアプリケーションに商用の数式処理 *Mathematica* があります.

ここで GMP の有理数は mpz_t 型の整数対として表現されるために mpz_t 型の整数演算の函数も必要になります. ところで, SageMath は多倍長精度の浮動小数点数の表現では GMPFR を用います. この GMPFR は GMP 上に構築されたライブラリで, より多くの函数, 符号付き零, 無限大 inf と NaN を提供しています. この GMPFR の提供する浮動小数点数は, IEEE 754 による浮動小数点数が固定長の 2 進数として表現されているのに対し, 仮数部の実体が分離したデータとして表現されます;

GMP の浮動小数点数の表現

__mpfr__prec	:	仮数部の bit 長
__mpfr__sign	:	符号と仮数部における非零要素数
__mpfr__exp	:	指数部
__mpfr__d	:	仮数部へのポインタ

'type(1.0)' で調べると 'sage.rings.real-mpfr.RealLiteral' と型が返却されますが, ここでの 'mpfr' で分かるように GMPFR が定める浮動小数点数が SageMath の実数の正体です. なお, 浮動小数点数は本質的に近似の数で, GMP や GMPFR では浮動小数点数への丸めを行うための函数が用意されています. そのために自然数 **N**, 整数 **Z** や有理数 **Q**, さらには整数係数の多項式の解になる数の集合である代数的数と別扱いになります.

ここで Python に標準で実装された長整数による処理と GMP を使った処理はどの程度違うものでしょうか? Python での長整数の演算処理は, たとえば乗算でカラツバ法 (Karatsuba algorithm) が用いられていますが, 各種の計算機環境に最適化されたものではありません. 次の例[*23]で SageMath の整数が Python 標準の長整数よりも高速に処理されていることを確認しましょう. そのために次の函数をあらかじめ定義しておきます;

```
def factorial(n, stop=0):
    o = 1
    while n > stop:
        o *= n
        n -= 1
    return o

def choose(n, k):
    return factorial(n, stop=k) / factorial(n - k)
```

ここで定義した函数 factorial() と choose() は階乗と組合せの値を計算する整数値函数で, 演算も積と差が中心で, 函数 choose() に商がある函数です[*24]. これらの函数定義を

[*23] http://jasonstitt.com/c-extension-n-choose-k の記事:「GMP bignums vs. Python bignums: performance and code examples」の例です

[*24] $\mathrm{choose}(n,k) = \frac{n!}{k!(n-k)!}$

Python と SageMath にそのまま評価させます．SageMath に関してはその引数の整数が自動的に整数クラスの構築子 Integer() で初期化されるために, 内部の計算が Python の整数型ではなく, SageMath の整数で処理されるために効果が判り易くなっています．そして, Python と SageMath での処理速度の計測は以下で行います;

```
import time
start = time.perf_counter()
a = choose(50000, 50)
elapsed = (time.perf_counter()-start)
print elapsed
```

この函数 time.perf_counter() 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 に包含されており, 利用するためにはあらかじめ ‘import time’ でモジュール time を読み込んでおく必要があります．それから処理時間は変数 elapsed に束縛されるために, この変数 elapse の値で小さな方がより高速な処理が行われていると判断できます．さて, この式の比較は Jupyter を使っていれば kernel を SageMath や Python 2 に変更して函数定義と計測を行えば容易に行えます．この式の評価を CoCalc で確認すると, Python 2 で 1.943, SageMath で 0.594 という結果を得て SageMath の整数処理が 4 倍近く高速であることが判ります．つまり, この程度の計算処理でも GMP と MPFR で数を定義した SageMath が圧倒的に高速で, Python による数値計算で処理速度を要求するのであれば, 数値行列演算ライブラリのことだけでなく GMP の利用も考慮すべきです．また, 整数演算で倍精度の浮動小数点数の範囲内で済む演算であれば最適化された BLAS+Python で処理するのも一手でしょう．ただし, 数式の検証やアルゴリズムの検討といった段階で黙って SageMath を使うことが最も安全で効率が高い方法です.

SageMath はこの GMP のような C のライブラリを取り込むために Cython を用います．Cython は C と Python を継ぎ目なしに融合することができる処理系で, 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．ここで 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 コンパイラが C のコードに変換[*25]し, それをコンパイルして Python の拡張モジュールが得られ, その結果, C で記述したものと大差ない程度に高速化できます.

[*25] ソースの構文解析は Pyrex で行います.

5.7.3 整数の実装

SageMath の整数は GNU MP に由来する mpz_t 型のオブジェクトを Cython で SageMath にクラス Integer として組込まれています．このクラス Integer は GNU MP 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，構築子 Integer() の引数に Python の int 型や long 型，NumPy の整数オブジェクト，整数論向けの数式処理 PARI の整数を与えることで SageMath の整数 Integer 型に変換する機能もあります．構築子 Integer() は他に PRP-3127 の数に関するリテラルも変換可能です．SageMath の整数の定義は 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 の実質的な定義です．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．

5.7.4 有理数の実装

有理数は GNU MP の mpq_t 型を Cython でクラス RationalField として取り込んで実装しており，クラス RaionalField 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．その構築子 QQ() の第 1 引数に RationalField に変換すべきオブジェクトを記述します．実質的な定義は src/sage/ring/rational.pyx にクラス RationalField が sage.structure.element.FieldElement のサブクラスとして定義されています．

5.7.5 実数の実装

有限な資源しかない計算機で実数は近似される数です．効率的な処理を行うために計算機内部では浮動小数点数と呼ばれる書式で数が表現され，さらに 2 進数に符号化されています．通常の計算機言語では単精度と倍精度の 2 種類の浮動小数点数が用意されていますが，Python 本体で倍精度のみであり，NumPy を導入することで単精度，倍精度と四倍精度の浮動小数点数が利用可能になります．Python 上で構築された SageMath ではパッケージ NumPy に加えてライブラリ GMPFR の利用で範囲が定まっているにせよ，その範囲内で任意精度の浮動小数点数で実数が近似できます．倍精度以上の高精度の演算は誤差に左右され易い対象の解析に有利ですが，高精度を実現するためにデータ量が増加することとに加えて，単精度や倍精度浮動小数点数演算と違い CPU で直接処理ができないこともあって処理速度に難があります．逆に言えば，倍精度浮動小数点数は CPU で直接処理が可能なために計算速度では圧倒的に有利です．そのために浮動小数点数による数値計算では対象の特性を考慮した上で，任意精度のままで計算するか，倍精度の浮動小数点数で計算するかという判断，さらには NumPy を利用して数値行列として一括して計算する

か, GMP 等の数値計算ライブラリを用いて精度を保証して処理を行うかといった判断が必要になります．SageMath の倍精度浮動小数点数クラスは RealDoubleField_class, 略して RDF であり, 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．なお, クラス RealField は任意精度とは言え, 実際の演算ではその精度を指定して処理が行われます．通常は浮動小数点数の仮数部の bit 長が 53 と倍精度浮動小数点数と同じ精度が既定値として定められており, 計算結果も一致することが期待できますが, 倍精度浮動小数点数のクラス RDF と比較して処理速度は格段に劣ります．また, 任意精度とは言え実現できる浮動小数点数の範囲があり, sage.rings.real_mpfr の函数である mpfr_prec_max() で仮数部の最大値, mpfr_prec_min() で仮数部に最小値が調べられます:

```
sage: from sage.rings.real_mpfr import mpfr_prec_min,
    mpfr_prec_max
sage: rint(mpfr_prec_max(), mpfr_prec_min())
(2147483391, 1)
```

なお, RealField の精度の設定は ‘RealField(128)’ や ‘RealField(prec=128)’ でクラス RealField のオブジェクトのインスタンス化の際にできます．また, 構築子 RealField には精度に関連する prec の他に浮動小数点数の近似の方法, 丸めの方法の指定も可能です．

5.7.6　複素数の実装

複素数 $\mathbf{C}$ は実部と虚部の実数対 $\mathbf{R} \times \mathbf{Z}$ 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, $\mathbf{Z} - \{0\}$ 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 $\mathbf{Q}$ は $(\mathbf{Q}, +)$ が可換群, そして, $(\mathbf{Q} - \{0\}, *)$ も可換群であることから体になります．

■CommutavieAlgebraElement:　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 $f : R - \{0\} \to N$ 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.

5.8　数学的対象の実装

5.8.1　カテゴリー, ペアレントとエレメントの概要

SageMath のライブラリには数百万行を越えるコード, 文書やテストが実装されています. そこで, このライブラリを本棚のように関連するオブジェクトやアルゴリズムを体系化して整理し, 類似のオブジェクトは類似のふるまいをするように, それも信頼性があり, 無駄なダブリもなくしたいものです. そのための工夫として, SageMath では代数的構造を表現する数学的オブジェクトに対して「**カテゴリー, category**」が実装され, このカテゴリーの実装では群やベクトル空間といった数学上の概念を確固としたソフトウエア・エンジニアリングによるデザイン・パターンに流し込むことを目的にしています. そのための SageMath の手法は数式処理 AXIOM やその後継者 (Aldor 等) に加え, Magma の Parent/Element といった概念, Python のオブジェクトの扱いからも影響を受けています. SageMath のカテゴリーの哲学は, 効率的に概念を生かし, 最終的にソフトウエアの管理を簡単、高速な処理が行えるようにシステムに数学的な情報を構築し, この際に GNU MP の様な特定のライブラリに必要以上に依存しないようにすることです. ちなみに SageMath の数学的対象のクラスの階層では具体的な実装が重視されていますが, カテゴリーの実装では数学的な構造が重視されています.

次に SageMath はオブジェクト指向プログラミング言語の Python を基底とした数式処理システムであり, この Python はオブジェクト指向プログラミング言語でもクラスに基く言語です. そのために SageMath では数学的概念や数学的対象の集合をクラスとして表現し, それらに所属するもの, たとえば, 数値や多項式といったものは, 数学的概念や数学的対象の集合に対応するクラスのインスタンスとして捉えられますが, 逆に言えば, SageMath で扱われる数学的対象には, その対象が所属すべきクラスが存在することを意味します. SageMath では整数の ‘1’ や有理数の ‘1/2’, 多項式の ‘x + 1’ のようなある数学的概念に対応する集合の元のことを「**エレメント, element**」と呼び, これらのエレメントを包含する数学上の概念に対応するオブジェクトを SageMath では「**ペアレント, parent**」と呼びます. ペアレントとエレメントの違いは, ペアレントは一意に存在しますが, エレメントは一意であるとは限らないことです.

ここで具体的な例で説明しましょう. 整数 5 は整数環 $\mathbf{Z}$ の要素, 有理数 $\dfrac{1}{2}$ は有理体 $\mathbf{Q}$ の要素ですが,SageMath でリテラル ‘1’ と ‘1/2’ はそれぞれ整数環 ZZ と有理体 QQ のエレメントであり, ‘1’ のペアレントは ZZ, ‘1/2’ のパアレントは QQ になります. このペア

レント探しはメソッド/函数 parent() で調べられます:

```
sage: 1.parent()
Integer RIng
sage: 1 in ZZ
True
sage: parent(1/2)
Rational Ring
sage: 1/2 in QQ
True
sage: ZZ.parent()
<type 'sage.rings.integer_ring.IntegerRing_class'>
sage: QQ.parent()
<class 'sage.rings.rational_field.RationalField_with_category'>
```

ここでは演算子 “in” を使って集合の要素としてのインスタンスの帰属先の確認とメソッド parent() を使ってインスタンスの parent を調べています．ここで, ‘Integer Ring’ は整数環のクラス名で, ‘ZZ’ は SageMath にあらかじめ用意されているインスタンスで, ‘Rational Ring’ は有理数環のクラス名で, ‘QQ’ があらかじめ用意されているインスタンスです.

```
sage; ZZ
Integer Ring
sage: ZZ.parent()
<type 'sage.rings.integer_ring.IntegerRing_class'>
sage: QQ
Ratinal Field
sage: QQ.parent()
<class 'sage.rings.rational_field.RationalField_with_category'>
sage: IntegerRing() is IntegerRing()
True
sage: RatinalField() is RationalField()
True
```

ところで, 整数の集合は, 環や通常の順序での順序集合, 加法による半群といったように環の構造以外の数学的構造も入れられます．SageMath ではこれらの数学的構造を実装するインスタンスは, それらの概念を表現する異なるカテゴリーに帰属します．そして, SageMath のカテゴリーには対応する数学的構造からの包含写像に対応するように Python のクラスの継承関係があります．この継承関係での親クラスをオブジェクトの帰属先と同様に「**ペアレント**」と呼びます．ちなみに, SageMath ではこの親子関係を有向グラフとして可視化できます．以下で SageMath の全カテゴリーを頂点とし, カテゴリー間の継承関係を向き付けられた辺とする有向グラフを生成してみましょう:

```
sage: g = sage.categories.category.category_graph()
sage: g.show(vertex_shape=None, figsize=10)
```

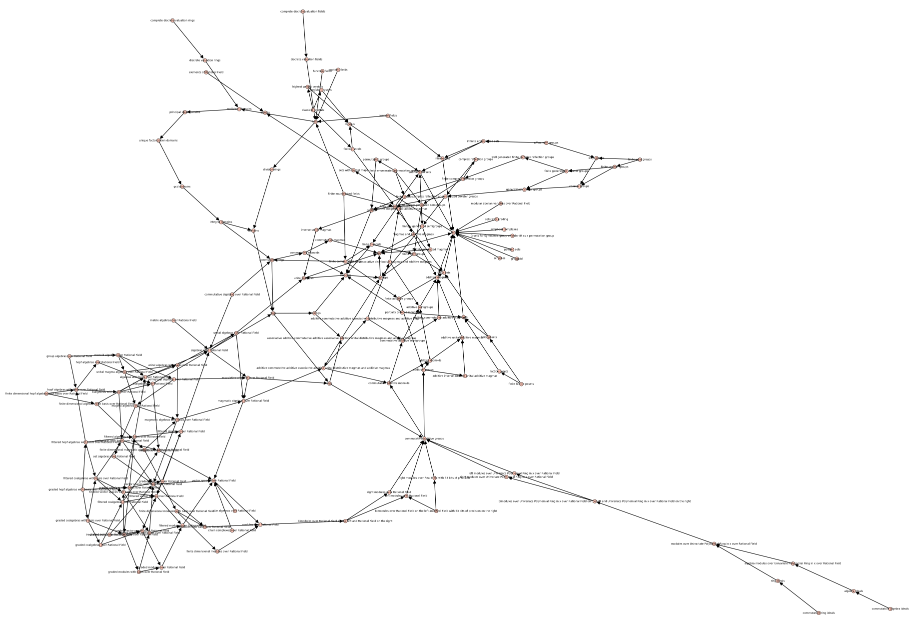


図 5.1 SageMath の全カテゴリーのグラフ

このように全カテゴリーの有向グラフは地下鉄の路線図のように混み入った様相を呈しています．この状況ではグラフがどようなものかが判り難いためにより根源的なカテゴリーの継承関係を確認することにしましょう．そこで, モノイドの圏に対応する Monoids の継承関係を見てみましょう:

```
sage: g = Monoids().category_graph()
sage: Monoids().super_categories()
[Category of semigroups, Category of unital magmas]
sage: Monoids().all_super_categories()
Category of monoids,
 Category of semigroups,
 Category of unital magmas,
 Category of magmas,
 Category of sets,
 Category of sets with partial maps,
 Category of objects]

sage: g.show(vertex_shape=None, figsize=10)
```

まず, カテゴリーの直接の継承関係はメソッド super_categories() を使って確認できます. カテゴリー Monoids は半群のカテゴリーである Semigroups と単位元を持つマグマのカテゴリーである UnitalMagmas の二つを継承していることが判ります. そして, そのカテゴリーの全ての継承関係を見たければメソッド all_super_categories() で調べられ, メソッド category_graph() を使ってその継承関係の可視化ができます.

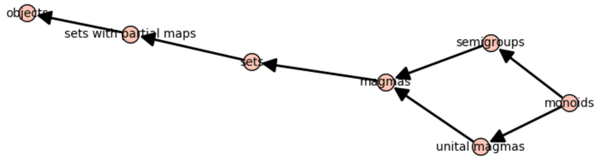


図 5.2　カテゴリー Monoids の parent のグラフ

この図に示すように, メソッド category_graph() で生成される継承関係のグラフでは, 各頂点がカテゴリーで, それらを結びつける辺の矢印の向きでカテゴリーのペアレントを指示します. この図では objects に矢印が向いていることから object が最上位のカテゴリーであり, それから順に SetWithPartialMaps, Sets, Magmas と継承関係があり, さらに Magmas は Semigroups と UnitalMagmas の parent であり, Monoids は Semigroups と UnitalMagmas の双方がペアレントであること, つまり, 多重継承があることが図からも読み取れます. なお, メソッド all_super_categorires() では §3.10 で解説した Python の C3 MRO を用いて継承関係の探索が行われます.

さて, エレメントとペアレントは SageMath のあるクラスのインスタンスですが, 大きな違いはエレメントはオブジェクトとしては複数存在し得ますが, ペアレントは単一であることです. たとえば, 整数環のインスタンスとして “ZZ” がありますが, 構築子 ‘IntegerRing()’ でインスタンス化したものは Python 上で全て同じオブジェクトになります. しかし, 整数環の元 ‘1’ は全て別のオブジェクトになります:

```
sage: Z1 = IntegerRing()
sage: Z2 = IntegerRing()
sage: Z1 is Z2
True
sage: Z1 is ZZ
True
sage: a = Z1(1)
sage: b = Z1(1)
sage: a is b
False
sage: a == b
True
sage: a
```

```
1
sage: a.parent()
Integer Ring
sage: b.parent()
Integer Ring
```

ここでは演算子 “is” でインスタンスの比較を行っています．この演算子 “is” はオブジェクトが配置された先頭の番地で等しいかどうかを判断するために, True が返却されれば同じオブジェクトであることが判ります．ここで結果から, 構築子 IntegerRing() でインスタンス化した Z1 と Z2, それと ZZ が同一のオブジェクトであることが判りますが, ‘Z1(1)’ でインスタンス化したオブジェクトで, 名前 a と b で指示されるものの値は整数値の 1 ですが, “is” の結果は False であることから別のオブジェクトであり, “==” の結果が True になることから値が同じであるものの異なるオブジェクトであることが判ります．そして, 名前 a のオブジェクトも名前 b のオブジェクトもそのペアレントは同じ ‘Integer Ring’ です．ここでの例で示すように数学的概念に対応する集合の元であるエレメントは同じ値であってもオブジェクトとしては異なることがありますが, そのエレメントが帰属する集合に対応するペアレントは唯一のオブジェクトになります.

SageMath の数学的対象での演算はペアレント上で定義されるために, 演算は同じペアレントに所属するオブジェクト同士で行われます．したがって, 異なるペアレントのエレメントとの演算を行うためには「**型強制 (coercion)**」によって同一のペアレントのエレメントに変換する必要があります．そのための写像, すなわち,「**マップ (map)**」を定義しなければなりません.

5.8.2　カテゴリー

カテゴリーは前述したように数学的概念を表現する Python のインスタンスです．このカテゴリーは, 一つは数学的概念に加え, それに付随する構造, 集合 (homeset), 射 (morphism), 函数的な構造 (functional construction), それと公理 (axiom) といった数学的な側面, もう一つはプログラムや文書, 類似する数学的構造に亘るテストといったプログラミング上の側面の双方を提供します.

5.8.3　parent, element, category

SageMath では数学的概念に対応する集合とその元の関係が parent-element の関係であり, 数学的概念そのものの実装が category であると言えます．ここではこれら parent, element, category について説明しておきましょう.

■parent: Python のインスタンスで, 数学的要素の集合だけではなく代数的な構造もモデル化するものが parent です. たとえば, 数学的要素の集合には, 整数の集合, 素数の集合, 二つの与えられたベクトル空間の線形写像の空間といったものがありますが, これらの集合にはさらに数学的構造もあります. 実際, 整数の集合には環の構造といった風にです. SageMath では整数の集合のような数学的対象が持つ代数的構造を parent として表現します. この際にある代数的構造を表現する SageMath 上のインスタンスはただ一つだけです. このことを確認してみましょう. ここでは整数環 $\mathbf{Z}$ に対応するインスタンス ZZ, 有理体 $\mathbf{Q}$ に対応するインスタンス QQ の parent を調べてみましょう.

では, ZZ や QQ が所属する parent は何でしょうか? 同様に parent で調べると今度は整数環や有理数体のクラスが表示されます. さきほどの '1' や '1/2' と異なり, ZZ や QQ のような集合にはメソッド categories() や category() といった, この集合に付随するカテゴリーに関する情報があります. 実際に ZZ で確認してみましょう:

```
object at 0x10a63bbd8>
sage: ZZ,categopry()
Join of Category of euclidean domains and Category of infinite
    enumerated sets
 and Category of metric spaces
sage: ZZ.categories()
[Join of Category of euclidean domains and Category of infinite
    enumerated sets
  and Category of metric spaces,
 Category of euclidean domains,
 Category of principal ideal domains,
 Category of unique factorization domains,
 Category of gcd domains,
 Category of integral domains,
 Category of domains,
 Category of commutative rings,
 Category of rings,
 Category of rngs,
 Category of semirings,
 Category of associative additive commutative additive associative
     additive
  unital distributive magmas and additive magmas,
 Category of additive commutative additive associative additive
     unital
  distributive magmas and additive magmas,
 Category of additive commutative additive associative distributive
      magmas
  and additive magmas,
```

```
 Category of additive associative distributive magmas and additive
     magmas,
 Category of distributive magmas and additive magmas,
 Category of magmas and additive magmas,
 Category of commutative monoids,
 Category of monoids,
 Category of semigroups,
 Category of commutative magmas,
 Category of unital magmas,
 Category of magmas,
 Category of commutative additive groups,
 Category of additive groups,
 Category of additive inverse additive unital additive magmas,
 Category of commutative additive monoids,
 Category of additive monoids,
 Category of additive unital additive magmas,
 Category of commutative additive semigroups,
 Category of additive commutative additive magmas,
 Category of additive semigroups,
 Category of additive magmas,
 Category of infinite enumerated sets,
 Category of enumerated sets,
 Category of infinite sets,
 Category of metric spaces,
 Category of topological spaces,
 Category of sets,
 Category of sets with partial maps,
 Category of objects]
```

メソッド category() ではインスタンス ZZ の代数的構造を特徴付けるユークリッド整域のカテゴリー, 無限可付番集合のカテゴリーと距離空間のカテゴリーの和集合であるという要約が表示され, メソッド categories() でそのカテゴリーの詳細が表示されます. このようにカテゴリーが数学的構造を決定付けます.

SageMath の標準的な構文で, 要素を生成するためには, それらの parent を生成しなければならず, その要素を定義するに十分な情報を parent は持っています. たとえば, 整数係数の変数が “x” の多項式の要素を生成したければ, その要素が所属する parent である整数係数の 1 変数多項式環を定義しなければなりません. 多項式環が生成されると, その要素を生成することができます:

```
sage: R = PolynomialRing(ZZ, name='x')
sage: R([1,2,3])
3*x^2 + 2*x + 1
```

実際に SageMath の幾つかの Category で遊んでみましょう.
ここでの例では有理数体 **Q** と整数環 **Z** について, ペアレントを調べたり, そのカテゴリを見ています. ペアレントを調べると, これらのカテゴリーが帰属する代数的構造を持つ対象に対応するクラスが表示されます.

5.8.4　SageMath 上で代数的構造の実装

SageMath の環は「**基盤 (base)**」や「**基礎環 (base ring)**」を持ちます. たとえば, SageMath の通常の環 R の商環はその基盤として R, その基礎環として 'R.base_ring()' を持ちます. ここで基盤となる環の宣言は容易で, 構築子の引数として基礎環を与えることでできます. たとえば, 1 変数の整係数の多項式の環を基礎として使いたければ, 'Field(ZZ['x']).base()' と入力します.

SageMath の型変換 (coercion) システムを使うために sage.structures.parent.Parent か sage.structures.element.Element のサブクラスとして実装することになります. そのために二項演算や大小関係を実現する Python の特殊メソッドの上書きではなく, それらのクラスに用意されたメソッドの上書きで対処します. これらのメソッドは和であれば特殊メソッド___add___から_add_と「_」が一つの名前になっています. また, オブジェクトの数学的性質はオブジェクトの category で宣言されます. そのためにオブジェクトの初期化で parent の category を宣言します. また, Element の呼出で属性を割り当てます. 型変換の実装ではメソッド___call___() の上書きではなくメソッド _element_constructor_() を用います. それから新たに構築する代数的構造で演算を行うときに, 写像を考慮しなければならないことになります. parent の代数的構造の算術演算子を利用するために, この写像に対応する型強制の宣言を行います. また, 必要であれば SageMath では型強制に必要な parent を自動的に生成します.

- 型強制システムを使うためには sage.structure.parent.Parent か sage.structure.element.Element のいずれかを使います.
- parent の構造を圏のオブジェクトに反映させる
- 要素クラスを使って parent 構造を提示
- 型強制を実装
- 型強制の宣言
- 実装した parent システム向けの構築函数を定義

Python の算術演算子に対応する特殊メソッド名は先頭と末尾に文字 "_" を二つ置いた名前になりますが, SageMath の算術演算子の名前は「**一文字の "_"**」を名前の先頭と末尾に置いたものです. これは公式の表示を与える特殊メソッド "_repr_()" も同様です.

商体 (fraction field) は整域 (integral domain) 上のみで定義可能です. そして, 定義し

た体の標数も追加して定義できます．この標数に陰ながら依存するものも存在します．

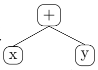
第 6 章

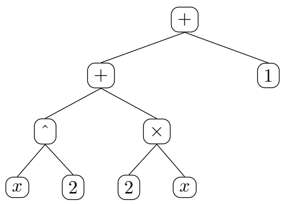
SageMath での多項式表現

6.1 計算機で式を表現するということ

数式処理は与えられた数式をそのままで処理することを目的にしています．この式の処理には, 方程式の解法や微分積分といった処理があり, 一見, 人工知能向けの処理に見えますが, 実際は代数的な計算処理で多くの処理ができます．しかし, 計算処理を行う以前に考えなければならないことがあります．それはどのように数式を計算機で表現するかということです．たとえば, $1+2$ や x^2+2x+1 といった式を' 1+2' 'x**2+2*x+1' といった文字の列として表現しても, 文字の列からは $1+2$ と 3, x^2+2x+1 と $(x+1)^2$ の同値性は出てきません．この同値性を確認するためには $1+2$ はその式を整数の和として評価することで, そして, $(x+1)^2$ は式の展開と項の整理を行えば x^2+2x+1 と等しいことが確認できます．と, このように数式の評価や展開や整理が必要になりますが, このときに和, 積や冪といった演算の性質を利用しなければなりません．そのため, 式を単なる文字列として考えるよりも, 式を構成する項を繋ぎ合せる演算子を鍵に, その式自体の成り立ちを含めた式の構造を考える必要があります．そして, この式の構造を踏まえた上で, 与えられた二つの数式が等しいものであるかどうかを判断するという処理を行わなければなりません．そのために式の構造をどのように表現するかが重要です．ここで, 二つの数式が等しいということを「**式の展開や整理, 等価な式の代入といった処理で互いに共通の式に変形できること**」と言い換え, 二つの式 A と式 B が互いに移り合えるときに '$A \sim B$' と記述しましょう．このときに二項関係 "$\sim$" は同値関係になります．実際, 最初の反射律については無問題, 対称律は式 A から B に変形可能なら, その逆操作から B から A に変形可能です．推移律は式 A から B への操作に続けて B から C への変換操作を行えば式 A から C に変換できます．以上から同値関係になることが判りました．さて, 同値類を考えるということは, 等しいものの集合から一つの対象を代表として選出して計算処理ができることを意味します．

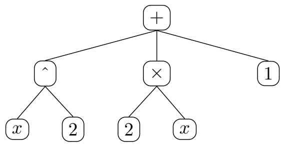
では, この同値関係 "$\sim$" の代表の選び方を幾つかの簡単な式で考えてみましょう．まず,

多項式 $x+y$ では演算子 “+” を中心に被演算子 x, y の入替が可能です．つぎに x/y であれば演算子 “/” を中心に被演算子の入替はできません．そして, $x*y$ であれば演算子 “+” のときと同様に左右の被演算子の入替ができます．これらの式では中央の演算子が異なるだけで, 左右の被演算子 x, y は同じですが, 商 “/” では被演算子 x, y の順番が問題になりました．このことから式の中核は演算子で, 被演算子の順番に大きな意味があると想像がつきます．そこで, 演算子を先頭に置き, 左右の被演算子を順番に並べたリストとして表現するとどうでしょうか? ここで $x+y$ を “+ x y” と表記すると式の範囲が分かり難いために “(+ x y)” と演算子と被演算子の列を括弧で括ります．この表記は LISP の S 式と同じ書式で, 演算子を前に置いた数式の表現を「**前置表現**」, あるいは「**ポーランド記法**」と呼びます．また, $x+y$ のように演算子が二つの被演算子の中に置かれた数式の表現を「**内挿表現**」, あるいは「**中置表現**」と呼びます．そして, 演算子を節, 被演算子を葉と見なせば「**木構造**」と呼ばれる構造が数式に導入できます．たとえば, $x+y$ は演算子が “+”, 被演算子が x, y となるために  と図示できます．そして, この木構造は式の構造の表現に優れており, S 式で表記した数式は容易に木構造表現に切替えられます．



たとえば, ‘x^2+2x+1’ を ‘$(x^2+2x)+1$ と書換えて S 式で記述すると ‘(+ (+ (ˆ x 2) (× 2 x)) 1)’ になり, この式の木構造は左に示すように S 式の括弧の階層に対応する階層を持ちます．このように数式は演算子を節, 被演算子を葉とする木構造が入り, この木構造は数式の括弧に合せた構造であり, 括弧の深さが木構造の深さに対応します．

ここで演算子の性質を反映させれば木構造それ自体がより簡易な表現になることがあります．ここで例に挙げた数式 ‘x^2+2x+1’ で x^2, $2x$ や 1 といった項を繋ぐ働きをしている演算子 “+” はどんな性質を持っているでしょうか？ まず, 最初に「**可換性**」‘$a+b=b+a$’ を充します．それから「**結合律**」‘$(a+b)+c=a+(b+c)$’ も充します．この結合律が成立することから演算の順序に関係する括弧を外して $a+b+c$ と記述できます．そして, S 式としては項を繋ぐ演算子 “+” を先頭に, 演算子 “+” で繋がれた被演算子を同水準に並べます．つまり, ‘$a+b+c$’ を ‘(+ a b c)’ と表記します．



その結果, 演算子 “+” については二項演算子として括弧で括る必要がなくなり, 足し合わされるものを平易に並べた S 式で表現されます．左に ‘x^2+2x+1’ の木構造を示すように, 木構造においても演算子 “+” で繋げられた項が同水準の葉として並べます．これと似た演算子に可換積 “$*$” かあり, 可換積 “$*$” も $x_1 * x_2 * \cdots * x_n$ を $(*\ x_1\ x_2\ \cdots\ x_n)$ と表記します．このように結合律を充す和や積といった演算では被演算子は二個に限定されません．そのために古株の数式処理 Maxima では複数の演算子を引数に取れる演算子 (narry 型演算子) と

して表現されています．そして，被演算子の入替ができない差と商は，逆元との和や積で置換えられます．つまり，$a-b$ は $a+(-b)$，a/b は $a*(1/b)$ と表記し，冪 a^b を $(\hat{}\ a\ b)$ で表現すると多項式は積や冪のみの部分式を和で結合したものとして表現できます．そして，木構造と S 式は 1 対 1 に対応が付きますが，和や積の被演算子が入れ替えられるということは，式の木構造が一意に定まらないことを意味します．実際，数式 $x+y+z$ は項の並び換えによって数式 $y+x+z$ や数式 $x+z+y$ 等の 6 通の並び方に対応する式があり，それらの式に対応する 6 個の木構造が存在します．そうすると展開された多項式であっても，二つの式が同じかどうかは，それらの項の和の順番，各項での積の順番について一々確認しなければなりません．これは流石に非効率です．ところが，この項や項内部の項や式がある規則で式の並びが一意に決っていれば，比較も式の頭から確認すれば良く，効率的な処理が可能になります．そのために項に順番，つまり，「**順序関係**」を入れます．具体的には，変数をアルファベット順に並べるようにすると式 $y+z+x$ は $x+y+z$ と一通りの式の表記になります．このように一意に式が表現されると，式の同値性の確認も，式の木構造が一致するかどうかを最初に判断し，その次に展開や同値な式の代入といった処理の順で確認できます．この順序関係は数学ではどのように定義されるべき概念でしょうか？次の節では順序について考えましょう．

6.2 順序について

「**順序 (order)**」は集合 S の二つの元に対して成立する関係 (二項関係と呼びます) です．そして，この関係 “$\geq_{\mathrm{order}}$” が以下の性質を充すときに「**順序**」，順序の入った集合 S を「**順序集合**」と呼びます:

順序の定義

反射律:	$x \geq_{\mathrm{order}} x$				
対称律:	$x \geq_{\mathrm{order}} y$	かつ	$y \geq_{\mathrm{order}} x$	$\Rightarrow$	$x = y$
推移律:	$x \geq_{\mathrm{order}} y$	かつ	$y \geq_{\mathrm{order}} z$	$\Rightarrow$	$x \geq_{\mathrm{order}} z$

ここで集合 S の順序 “$\geq_{\mathrm{order}}$” で集合の任意の元 x, y に対して ‘$x \geq_{\mathrm{order}} y$’，あるいは ‘$y \geq_{\mathrm{order}} x$’ のいずれかが成立するときに順序 “$\geq_{\mathrm{order}}$” を「**全順序**」，集合 S を「**全順序集合**」と呼びます．たとえば，集合を実数 $\mathbf{R}$ とするときに二項関係 “$\geq$” は全順序で集合 $\mathbf{R}$ は全順序集合になります．次に二項関係を包含関係 “$\supseteq$” とすると，この包含関係 “$\supseteq$” は集合 $\mathbf{R}$ の順序になりますが，今度は全順序にはなりません．実際，偶数全体の集合 A と奇数全体の集合 B を考えるとどうでしょうか？ 集合 A と集合 B の間には包含関係がありませんが，全順序となるためには任意の二つの元に対して包含関係がなければなりませんが，この包含関係のように任意の元 a, b に対して二項関係が成立するとは限らない順

序を「**半順序**」と呼びます．　この半順序関係を使えば式の構造がきちんと定義できます．まず, 関係 $<$ を持つ集合 T が「**木構造**」を持つとは, 関係 $\leq$ が集合 T にて半順序, この半順序 $\leq$ について最小限 $x_0 \in T$ が存在して, 任意の $x \in T$ に対して定まる部分集合 $T_x \stackrel{\text{def}}{=} \{y|x \leq y\}$ にて関係 $>$ が全順序となるときです．ここで最小限 x_0 のことを「**根**」と呼び, $x \leq y$ を充す $x, y \in T$ について $x \leq z \leq y$ を充す $y \in T$ が存在しないときに y を x の「**子**」, 逆に x を y の「**親**」と呼びます．そして木構造を持つ $(T, \leq)$ の集合 T の成員を「**節**」．$y \leq x$ となる $y \in T$ が存在しない $x \in T$ を「**葉**」と呼びます．

さて, 数式処理が扱う対象の集合は整数, 有理数といった一般の数に加え, 変数や函数を演算子を使って構築した式から構成され, このときに式は数, 変数や函数を演算子で結合した対象となります．そこで, 数は大小関係で順序関係を入れ, 変数や函数 (名) には辞書で採用されている方式で順序が入れられます．そして, 数と変数に対しては変数の方がより大きく, 変数と函数に対しては変数よりも函数が大きいと順序付けると変数の積で構成された項の順序が残ります．ここで考えている世界が 1 変数の世界であれば, 単項式 x^m と x^n との順序を冪の次数 m と n の大小関係 “$\geq$” で決めれば良いでしょう．では, 多変数の場合はどうすれば良いのでしょうか？ この場合, 変数に入れた順序で項を構成する変数の次数リストで比較します．たとえば, x, y, z の 3 変数多項式環 $K[x, y, z]$ で, 変数間の順番をアルファベット順で並べます．ここで $y\,z$ のように変数が抜けていれば, 抜けている変数の次数を 0 とします．すると, 項は $x^{i_1} y^{i_2} z^{i_3}$ の形式になります．これから長さ 3 の次数のリスト (i_1, i_2, i_3) が得られ, このリストと項は一対一に対応します．したがって, n 変数 $x_1, \ldots, x_n$ の単項式 A と B が与えられたときに n 個の変数 $x_1, \ldots, x_n$ の並びを固定します．次に, 単項式 A と B の中の変数 x_i が項に存在しない場合に x_i^0 を挿入します．こうすることで単項式 A と B を表現する長さが n の次数リスト $(\alpha_1, \ldots, \alpha_n)$ と $(\beta_1, \ldots, \beta_n)$ が一意に決ります．あとはこれらの次数リストに対して順序を入れてしまえば良いのです．ここで, 項の順序としては二つの項 x^α , x^β に対して ‘$x^\alpha > x^\beta$’ であれば, x^γ を両辺にかけた場合でも ‘$x^{\alpha+\gamma} > x^{\beta+\gamma}$’ を充すものの方が都合が良いですね．この項に対する性質 ‘$x > y \Leftrightarrow x \cdot a > y \cdot a$’ を充す順序を「**項順序**」と呼びます．では, 先程の例の二つの次数リスト $x^2\,y^2\,z (= (2, 2, 1))$ と $x\,y^2\,z^3 (= (1, 2, 3))$ が得られたとき, 順序はどのように入れれば良いでしょうか．単純に x の多項式と見れば, $x^2\,y^2\,z$ の方が大きく, 次に Z の多項式と見れば, $x\,y^2\,z^3$ の方が大きくなります．ところが ‘$x = y = z$’ として x の多項式として変形すると, $x\,y^2\,z^3$ の方が次数が大きくなり, 次数の大きさも含めて考えた方が良さそうです．このように項の順序は色々あります．そこで, 次に代表的な項の順序について説明しましょう．

6.2.1 さまざまな順序

項順序として代表的なものに「**辞書式順序**」,「**斉次辞書式順序**」,「**逆辞書式順序**」,「**逆斉次辞書式順序**」, この他にこれらの順序に変数の重みを加味したものといろいろあります. ここでは基本的な辞書式順序, 斉次辞書式順序, 逆辞書式順序と斉次逆辞書式順序について簡単に説明します. なお, 変数の並び順を $x_1, \ldots x_n$ とし, $x_1^{a_1}, \ldots x_n^{a_n}$ の次数リストを $a = (a_1, \ldots, a_n)$, $x_1^{b_1}, \ldots x_n^{b_n}$ の次数リストを $b = (b_1, \ldots, b_n)$ とし, さらに $|a| = a_1 + \cdots + a_n$ で次数リスト a に含まれる次数の総和を表記します. また, 順序の定義に含まれる関係 "$>$" は通常の整数環 $\mathbf{Z}$ の元に対する大小関係とします.

☆辞書式順序

辞書式順序 "$>_{\mathrm{lex}}$"

$$a >_{\mathrm{lex}} b \quad \Leftrightarrow \quad a_1 = b_1 \quad \ldots \quad a_i = b_i \text{ かつ } \quad a_{i+1} > b_{i+1}$$

この関係で定められる順序を「**辞書式順序**」と呼び, "$>_{\mathrm{lex}}$" と表記します. この辞書式順序では左側の変数の次数が大きなものが大きな項となります. たとえば, 二つの項 $x^2\,y^2\,z$ と $x\,y^2\,z^3$ の次数から構成される整数リスト $(2,2,1)$ と $(1,2,3)$ に対しては, 先頭の 2 と 1 を比較した段階で $2 > 1$ から '$(2,2,1) >_{\mathrm{lex}} (1,2,3)$', すなわち, '$x^2\,y^2\,z >_{\mathrm{lex}} x\,y^2\,z^3$' になります.

☆斉次辞書式順序

斉次辞書式順序 "$>_{\mathrm{glex}}$"

$$a >_{\mathrm{glex}} b \quad \Leftrightarrow \begin{cases} |a| > |b| \text{または} \\ a_1 = b_1, \ldots, a_i = b_i, a_{i+1} > b_{i+1} \end{cases}$$

この関係で定められる順序を「**斉次辞書式順序**」と呼び, '$>_{\mathrm{glex}}$" と表記します. 斉次辞書式順序では総次数で最初に項を比較し, 総次数が一致すれば項を辞書式順序で比較する二段式です. たとえば二つの項 $x^2\,y^2\,z$ と $x\,y^2\,z^3$ を表現する整数リスト $(2,2,1)$ と $(1,2,3)$ に対しては '$|x^2y^2z| = 5$' かつ '$|xy^2z^3| = 6$' となるために '$(1,2,3) >_{\mathrm{glex}} (2,2,1)$', すなわち '$x\,y^2\,z^3 >_{\mathrm{glex}} x^2\,y^2\,z$' と辞書式順序の逆の結果になります.

☆逆辞書式順序

逆辞書式順序 “$>_{\text{revlex}}$”

$$a >_{\text{revlex}} b \quad \Leftrightarrow \quad a_n = b_n \ \ldots \ a_i = b_i \quad \text{かつ} \quad a_{i-1} < b_{i-1}$$

この関係で定められる順序を「**逆辞書式順序**」と呼び, “$>_{\text{revlex}}$” と表記します．辞書式との違いは調べる方向と次数の小さい方を取ることで, これらが逆になっています．たとえば, 二つの項 $x^3y^2z^3$ と xy^2z^3 を表現する整数リスト $(3,2,3)$ と $(1,2,3)$ に対しては, このリストのうしろから比較を行います．そこで, 各リストのうしろから先に移動してリストの先頭の 3 と 1 に到達すると定義から ‘$(1,2,3) >_{\text{revlex}} (3,2,3)$’, すなわち, ‘$xy^2z^3 >_{\text{revlex}} x^3y^2z^3$’ を得ます．

☆斉次逆辞書式順序

斉次逆辞書式順序 “$>_{\text{grevlex}}$”

$$a >_{\text{grevlex}} b \quad \Leftrightarrow \quad \begin{cases} |a| > |b| \text{または} \\ a_n = b_n, \ldots, a_n = b_n, a_{i-1} < b_{i-1} \end{cases}$$

この関係で定められる順序を「**斉次逆辞書式順序**」と呼び, “$>_{\text{grevlex}}$” と表記します．この順序は項の総次数で最初に項を比較して 総次数が等しければ逆辞書式順序で項の順序を決める二段階の方式です．たとえば, 二つの項 x^2y^2z と xy^2z^3 の比較では, 総次数がそれぞれ 5 と 6 になるために ‘$xy^2z^3 >_{\text{grevlex}} x^2y^2z$’ になります．ちなみに逆辞書式であれば次数リストの右端の z の次数から見るために ‘$x^2y^2z >_{\text{revlex}} xy^2z^3$’ となって斉次逆辞書式順序と逆の結果になります．

6.3 SageMath 上の式の表現

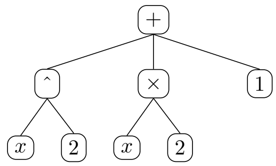
SageMath の式の表現について簡単に述べておきましょう．SageMath は多くの OSS のアプリケーション, パッケージを用いています．簡易的な数式処理が可能な Python のパッケージとして SymPy がありますが, SageMath はその SymPy で用いているメソッドとも微妙に異なり, この点では注意が必要になります．まず, 式の木構造の根幹になる演算子はメソッド operator() から, 被演算子はメソッド operands() から得られます:

```
sage: a = x^2 + 2*x + 1
sage: a.operator()
 <function add_vararg at 0x7ff7d2a5fa28>
sage: operator().__name__
 'add_vararg'
```

```
sage: a.operands()
 [x^2, 2*x, 1]
sage: a.operands()[0].operator().__name__
 'pow'
sage: a.operands()[1].operator().__name__
 'mul_vararg'
sage: a.operands()[1].operands()
 [x, 2]
```

ここでの例では多項式 x^2+2x+1 の構造を調べています．この式は S 式で表記すると ‘(+ (^ x 2) (* x 2) 1)’ になります．

この数式の樹形図を左図に示しておきますが, 和演算子 “+” は結合律を充すために複数の被演算子, 函数としては複数の引数を取れます．そのために x^2, $2x$ と 1 の 3 項を引数に持ちます．その結果, 和演算子 “+” の下に 3 個の項が並び, これらの項は辞書順序で並んでいます．SageMath では多項式のオブジェクトのメソッド operands() でこれらの項をリストとして取り出せます．この項の並びは樹形図と一致しています．また, 演算子の情報はメソッド operator() を使って取り出せます．ここでは式を括る演算子として ‘add_varag’ が返却されていることから, S 式で表現したときと同じ状況であることが判ります．次に $(x-2)^4/x^2$ でも構造を確認しておきましょう:




```
sage: b = (x - 2)^4 / x^2
sage: b.operator().__name__
 'mul_vararg'
sage: b.operands()
 [(x - 2)^4, x^(-2)]
sage: c = b.expand()
sage: c.operator().__name__
 'add_vararg'
sage: c.operands()
 [x^2, -8*x, -32/x, 16/x^2, 24]
sage: c.operands()[-2]
 16/x^2
sage: (c.operands()[-2]).operator().__name__
 'mul_vararg'
sage: (c.operands()[-2]).operands()
 [x^(-2), 16]
```

この式は $(x-2)^4$ を x^2 で割ったものですが, 前述のように商を逆数の積として表現すると $(x-2)^4x^{-2}$ と置換えられます．実際, SageMath で確認すると式の演算子が積であり, 被演算子が ‘[(x - 2)^4, x^(-2)]’ と書換えられていることが判ります．次にメソッ

ド expand() を使って式の展開を行った結果の式の演算子は和演算子であり, その引数は '[x^2, -8*x, -32/x, 16/x^2, 24]' であることが判ります. ここでうしろから二番目の項の '16/x^2' の演算子を確認すると'mul_vararg' より積演算子であることが判り, 実際の表示の商演算子 "/" でないことが判ります.

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．ちなみに 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包含されており, その中の組紐群を使って結び目/絡み目を処理ができますが, ここでは結び目/絡み目をガウス・コードと呼ばれるリストで表現し, このガウス・コードに対して結び目/絡み目の連結和と呼ばれる幾何学的な切った貼ったといった操作からモノイドと呼ばれる代数的構造を導入したり, 結び目/絡み目の射影図の交差点操作から得られる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します．ここでスケイン多項式の計算過程では膨大な中間データが発生するために, 簡易的な RDB として SQLite3 を用いることとし, これらの中間データの可視化も行えるようにしてみましょう.

7.2 結び目/絡み目とは

最初に結び目/絡み目が何であるか明瞭にしておきましょう．ここで, 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ているために, 結び目が何であるかを明確にしておきましょう．現実の結び目には「**蝶々結び**」等の色々な紐の結び方がありますが, 結び目を構成する紐が互いに交わらず, 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大きさの 3 次元空間内の図形です．そこで, 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し, 結び目を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．このように考える理由は実用上の問題で, 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, 紐をまっすぐにできるためです.

さて, この定義で結び目は 3 次元球 B^3 にどのように入っているのでしょうか? さすが

に結び目が自分自身で交差して複数個の穴の開いたドーナッツみたいな状態であることを考える人はいないでしょう．ちなみに, この図形は「**ブーケ (bouquet)**」と呼ばれる図形で, その穴の数で一意に分類できます．そのために自分自身が交差した状態は除外し, 自分自身が交差しない 1 次元球面 S^1 が 3 次元球に包含された状態, つまり, 1 次元球面の 3 次元球への「**埋め込み**」を考えます．このように「**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**」を考えたときに, 1 次元球が 1 個のときが「**結び目 (Knot)**」, 1 次元球が 2 個以上で, それらが埋め込まれた 3 次元球が互いに交差しないものが**絡み目 (Link)**」です．

これら結び目や絡み目には「**向き (orientation)**」が入れられます．実際, 絡み目の各成分と結び目は円で, 円には反時計回りや時計回りの二種類の向きが入れられるためです．各成分に向きが入った絡み目/結び目を「**向き付けられた絡み目/結び目**」と呼びます．

つぎに結び目/絡み目を構成するこれらの円がどのように 3 次元球に入っているかを調べるために「**結び目/絡み目の補空間**」と呼ばれる空間を考察します．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 $N(K)$ を抜いた空間で, 絡み目に沿って目に見えない小さな虫が果実を食べてしまった状態にたとえられますが, この補空間の境界には絡み目の環状近傍の境界でもあるチューブ状の境界と, 絡み目全体が収まっている 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．ところで, 2 次元球面は絡み目を収納する容れ物で, 絡み目とは直接, 無関係です．そこで, この 2 次元球面に 3 次元球を貼り合わせて, この境界を除去します．つまり, 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 が表現できますが, 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 次元球に分割できます．これは原点を中心とする 3 次元空間内の 3 次元球で, その表面の 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状況と同様です．そして, 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．このときに絡み目が 3 次元球 B_+^3 に入っていると考えます．この 3 次元球の貼り合わせには別の考え方があり, それは境界の球面を 1 点に潰すという考え方です．この 3 次元球面 S^3 は通常の 3 次元空間 $\mathbf{R}^3$ に無限遠点 ∞ を追加した空間として考え, この貼り合わせを「**1 点コンパクト化**」と呼びます．

7.3　正則射影図

ところで, 結び目/絡み目をそのまま 3 次元の対象として扱うことは容易ではありません．そこで, 結び目/絡み目を平面に射影して平面図形として考察します．この射影された結び目/絡み目の図を「**射影図**」と呼びます．この考え方を「**三葉結び目 (trefoil)**」と

呼ばれる結び目で SageMath を使って説明しておきましょう．まず, 結び目を描画する SageMath のスクリプトを示しておきます:

```
var('t,u,v')
a, b = 3, 1
x = (a + b*cos(u))*cos(v)
y = (a + b*cos(u))*sin(v)
z = b*sin(u)
T = parametric_plot3d([x,y,z],(u,0,2*pi),(v,0,2*pi),
    opacity=0.3, aspect_ratio=1)
c = (3*t,2*t)
s =[ _.subs(dict(zip((u, v), c))) for _ in (x, y, z) ]
K = parametric_plot(s, (t,0,2*pi), color='red',
    thickness=20,plot_points=200 )
sb =[ _.subs(dict(zip((u, v), c))) for _ in (x, y, -7) ]
Kb = parametric_plot(sb,(t,0,2*pi), color='blue',
    thickness=20,plot_points=200)
(K + Kb + T).show()
```

最初に変数 t, u, v を函数 var() を使って宣言し, それから 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座標式, 変数 a がトーラスの 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, それらを使って函数 parametric_plot3d() を使って描きます．このときに ‘opacity=0.3’ で曲面を半透明にします．トーラスは二つの円の直積: $S^1 \times S^1$ と同相で, ドーナッツの穴の周りを一周する「**緯線 (ロンジチュード, longitude)**」と, その緯線に直交しトーラスの腕 (断面が円) を一周する「**経線 (meridian)**」の二つの座標があります．ここで変数 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) 側の円の半径にそれぞれ対応します．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 回ほど回ることで表現され, 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で行っています．そして, 変数 s に割り当てた結び目の座標情報から函数 parametric_plot() で結び目を描画し, 同様に Z = -7 にある XY 平面への結び目の射影も描画します．これらのグラフィックス・オブジェクトを重ね合わせたオブジェクトを生成し, メソッド show() で表示したものが図 7.1 に示すグラフです:

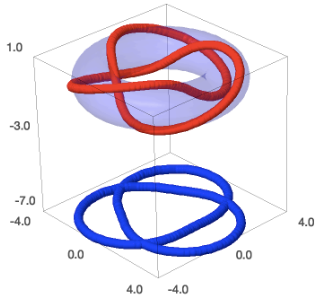


図 7.1　3 次元空間内の三葉結び目

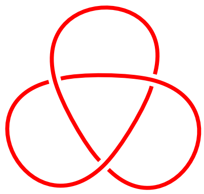


図 7.2　結び目の射影図

この図 7.1 のように絡み目/結び目を 3 次元空間内の対象として描くことは面倒ですが, これが平面図形ならまだ何とかなるでしょう. とはいえ, 図の下側のように平面に結び目を射影して平面図にすると今度は交差点でどの紐が上か下かという情報が欠落しているために良いものではありません. そこで, 図 7.2 のように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いてみます. こうすると平面図形でありながら十分な情報を持たせられます. この図を「**絡み目/結び目の射影図**」と呼びます. ところで, 絡み目の射影図は, その射影の方向で色々と形が異なります. 困るのは交差点で 2 本以上の紐が長々と重なったり, 3 本以上の紐が交差している射影図ですが, このような射影図に対しては, 問題のある交差点の付近で紐を少しだけ平行移動させてやれば二つの紐だけが交差して紐の向こう側にもう一方の紐が渡りきる「**横断的**」と呼ばれる状況の交差点だけにできます. このような横断的な二重点のみで構成される性格の良い射影図を「**正則射影図**」と呼びます. 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, ここで扱う結び目/絡み目はその正則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近似で必要な絡み目/結び目を「**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

ここまでで, 3 次元空間内の結び目/絡み目が扱えるようになりました. では, 与えられた二つの結び目/絡み目が「**同じである**」と言えるのはどのような状態にあるときでしょうか? これが三角形であれば互いに重なり合うための条件を充すことを示せられれば十分ですが, 結び目/絡み目は簡単ではありません. まず, 一方の絡み目の成分の総数が n であ

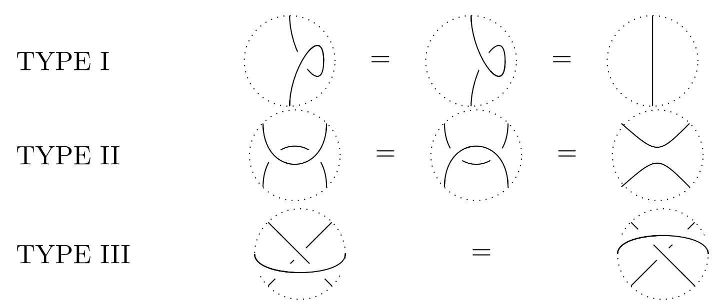
ればもう一方の成分の総数も n でなければなりません．それから絡み目をゴム紐でできたものと考えて 3 次元球面 S^3 内部で紐を切ったり, 紐を交差させずに変形させて互いの絡み目に移り合えるときに同じ絡み目と呼ぶことにしましょう．このことは二つの同じ成分数 n の絡み目 L_1 と L_2 の間に「**アンビエント・イソトピー (ambient isotopy)**」と呼ばれる連続写像 $F: S^3 \times [0,1] \to S^3 \times [0,1]$ が存在することに対応します:

アンビエント・イソトピーの性質

- $t \in [0,1]$ に対して $F_t(x)(\stackrel{\mathrm{Def}}{=} F(x,t))$ は 3 次元球面 S^3 の同相写像である
- $F_0: S^3 \to S^3$ 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が, このアンビエント・イソトピーの構築はどうすれば良いでしょうか．できれば 2 次元空間内で, それも局所的な図形の変形に置換えて考えたいものです．そこで正則射影図の局所的な操作の有限回の反復処理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

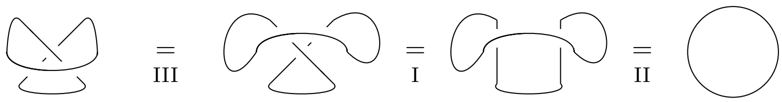


ライデマイスター移動はこれら 3 種類の正則射影図の局所的な変形操作の組み合わせから構成されます．最初の TYPE I は紐に捩れがあったときにまっすぐにしても同じ結び目/絡み目になるという操作, TYPE II は絡んでいない紐の交差点を解消する操作, 最後の TYPE III は交差点付近で絡まっていない紐を平行移動させる操作です．これら 3 種類

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・アイソトピーの存在が知られているため, ライデマイスター移動を使って結び目/絡み目の同値性が判断できます.

ここで, 3 次元球面 S^3 内部の結び目 K の XY 平面への射影図が $\{(x,y)\in S^3|x^2+y^2=1\}$ で与えられる結び目の正則射影図にライデマイスター移動で変形できるときに結び目 K を「**自明な結び目 (trivial knot)**」と呼びます. このライデマイスター移動で自明な結び目になる例を次に示しておきましょう:

ライデマイスター移動の例



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, 最後に 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ています. なお, 最後の TYPE II は TYPE I を左右の捩れに施す操作にも置き換えられます. この「**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**」から同成分数の二つの絡み目の間にライデマイスター移動が存在しないと証明できれば, これらが同値でないと主張できます. これで分類問題は一件落着のように思えますが, では「**ライデマイスター移動が存在しないこと**」はどうやって示せば良いのでしょうか? 知恵の輪よろしく, 試行錯誤の結果から「**ライデマイスター移動が見つからない**」ことと,「**ライデマイスター移動が存在しない**」ことは違います. そこで, 結び目/絡み目の固有の値を定め, それらの値の比較で判断したいものです. 当然, その固有の値はライデマイスター移動で不変で, 機械的な操作で計算できるものであるべきで, そのためには絡み目/結び目がどのようなものであるか数学的な特徴付けを行わなければなりません. この特徴付けを行うために「**群**」を結び目/絡み目に導入します.

7.5　絡み目に入る代数的構造

§2.6 で解説した半群や群といった数学的構造を持つ集合は整数や多項式といった数学的対象から構成された集合に限定されません. 実際, 演算子に相当する操作が集合にあれば, その性質に似合った数学的構造が導入できます. 実際, 絡み目/結び目の集合には「**連結和**」と呼ばれる新しい絡み目/結び目を作り出す操作があります. この操作は向き付けられた二つの結び目 K_1, K_2 の紐がまっすぐな箇所を選び, その箇所を取り外して双方の向きを保つように繋ぎ合わせて新しい結び目 $K_1\#K_2$ を生成する操作で, 絡み目であれば第一成分に対して結び目と同様の操作を行います.

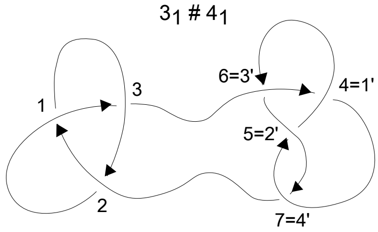


図 7.3　連結和の例

図 7.3 に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 $3_1\#4_1$ を示します．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 $K_1\#K_2$ で片方の K_1 を小さく潰し，K_2 上の好きな位置に紐の上を動かして $K_2\#K_1$ に変形できるためです．自明な結び目を 0 と表記するときに，この自明な結び目 O と任意の結び目 K との連結和は $K\#O = O\#K = K$ になり，(向き付けられた絡み目/結び目, $\#$) に自明な結び目を単位元 0 とする半群の構造が入ります．しかし，$K_1\#K_2 = 0$ となる結び目 K_1 と K_2 が存在したとしても，K_1 を小さく潰しても結び目が解けるのは K_2 が自明な結び目のときだけで，このことは K_1 についても同様であるために $K_1\#K_2 = 0$ になるのは双方が自明な結び目 0 のときに限られます．つまり，連結和は自明な結び目でなければ逆元を持ちません．連結和に関連して，「**非自明な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．あとで解説するように自明な結び目がスケイン多項式と呼ばれる結び目/絡み目の不変量で 1 になり，連結和で構成された結び目のスケイン多項式が，その連結和を構成する結び目の積になるために，素な結び目はスケイン多項式の既約性に関係し，さらには結び目多項式が 1 になる結び目が自明な結び目に限定されるかどうかという重要な問題があります．

この絡み目のような幾何学的対象にも演算が導入できますが，半群や群を分かり易く表記する方法はないでしょうか? 一つの方法に「**群の表示**」があります．この群の表示の説明のために語の集合の例で説明しましょう．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 (word)**」と呼びます) と空白 “”(1 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，式 $mike * neko$ は $mikeneko$ と演算子 “$*$” の左右の語を繋いだ文字列です．また $WWWWW\cdots$ のように同じ語 W が n 回続くときに W^n と表記します．ここで空白 “” を文字として考えると，この空白を語の左右に繋いでも元の語になるので空白 “” が演算 $*$ の単位元になることが判ります．そして，語 $W_1W_2W_3$ は $(W_1 * W_2) * W_3$ と $W_1 * (W_2 * W_3)$ の双方から構成されるために演算子 “$*$” 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 $\langle a, \ldots, z\rangle$ と記述します．なお, 記号 “...” は Python の拡張スライス構文で (次元も含めた) 省略を意味する Ellipsis というオブジェクトですが, ここでの表記は単純に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 $a_1, \ldots, a_n$ を連結して生成されるものも新規 A の文字列になります．この演算で生成される群を $\langle a_1, \ldots, a_n\rangle$ と表記し,「**自由群**」と呼びます．そして, $\langle a_1, \ldots, a_n\rangle$ の中の対象 a_1, ..., 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元を繋ぎ合せる処理と削除を持つ集合です．また, 積順序の入替は前後の文字の入替と一致し, 生成元が 2 個以上の自由群であれば, もとの語と別の語ができます．実際, 最初の語の例で ‘inu’ と ‘uni’ は別物です．このように積の順番が入れ替えられない群のことを非可換群と呼びます．このように生成元が 2 個以上の自由群は非可換群ですが, 生成元が一つだけのときのみ可換群になります．この「**可換**」という意味は群の生成元が $a_1, \ldots, a_n$ のときに任意の 生成元 a_i, a_j に対して $a_ia_j = a_ja_i$ という規則があることと言えます．このように群の生成元を使って, その群が持つ規則を表現する式を「**関係式)**」と呼びます．そして, 集合の表記にならって可換群であれば $\langle a_1, \ldots, a_n : a_ia_j = a_ja_i, i, j \in [1, \ldots, n]\rangle$ と群の表示ができます．一般的には群の生成元が $a_1, \ldots, a_n$, 関係式が $r_1, \ldots, r_m$ であれば群の表示は以下のようになります:

群の表示

$$\langle a_1, \ldots, a_n | r_1, \ldots, r_m\rangle$$

ここで関係式を変形して単位元 1 に等しい式で置き換えて 1 に等しいことを省略します．そのような式 $r_1, \ldots, r_m$ を「**関係子 (relator)**」と呼びます．　この関係子で群 G の表現を与えると群 G の生成元で生成される自由群 F_n から群 G への自然な写像 f が考えられます:

$$\begin{array}{cccccc} f & : & \langle a_1, \ldots, a_m\rangle & \to & \langle a_1, \ldots, a_m | r_1, \ldots, r_n\rangle & \\ & & \Cup & & \Cup & \\ & & a_i & \mapsto & a_i & i \in \{1, \ldots, m\} \end{array}$$

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 に写し, 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 $r_{i,i\in\{1,\ldots,m\}}$ 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, 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．ここで一般的に群 G_1 から群 G_2 への準同型写像 $f : G_1 \to G_2$ で群 G_2 の単位元 1 に写される群 G_1 の元の集合を $\mathrm{Ker}(f)$ と表記して準同型写像 f の「**核 (kernel)**」と呼びます．この核 $\mathrm{Ker}(f)$ は帰結群の議論で見たように群 G_1 の部分群になります．ここで見たように関係子を記述することは自由群からの自然な準同型写像の核を記載することに対応します．このように群の表示を与えることで群の具体的な形が見えてきます．これで絡み目/結び目を指示する群を表現する手段が得られました．つぎは絡み目/結び目の群の表示の具体的な話に移りましょう．

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 (foundamental 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/絡み目を組紐で表現することで「**組紐群**」と呼ばれる群が構築できます．これらの群は正則射影図から容易に構築できます．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

ここでは三葉結び目を使って絡み目/結び目の基本群の構成を説明します．図 7.4 が, この結び目に向きを入れた正則射影図です．結び目/絡み目群の構成では, このように結び目/絡み目を構成する各 1 次元球面に入れた向きを示すために正則射影図の道を矢印で表現した, 向き付けられた正則射影図を用い, この向き付けられた正則射影図から「**Wirtinger 表示**」や「**Dehn 表示**」と呼ばれる構成方法によって基本群が構築できます．ここでは Wirtinger 表示を解説しましょう．まず, 結び目/絡み目群の Wirtinger 表示を以下に示しておきます:

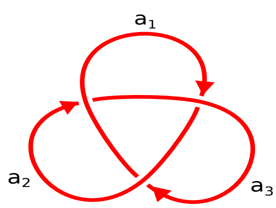


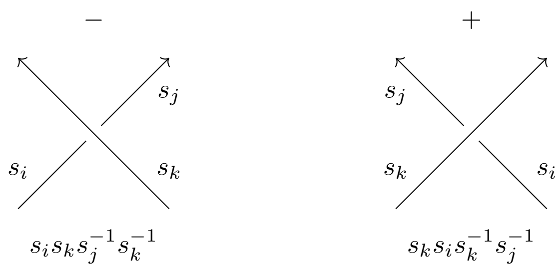
図 7.4　向き付けられた正則射影図

結び目/絡み目群の Wirtinger 表現

$$\text{結び目/絡み目群} = \langle \text{上道}_1, \ldots, \text{上道}_n | \text{関係子}_1, \ldots, \text{関係子}_n \rangle$$

ここで n は正則射影図の交点の数で, 関係子は各交差点で定まります. なお, 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. そして, Wirtinger 表示の関係子は次で与えられます:

Wirtinger 表示の関係子



各交差点の上にある $+1$ と -1 は「**交差点の符号**」で, 正則射影図 $\mathcal{D}$ の交差点 P_i の符号を $w(P_i)$, あるいは $\mathrm{sign}(P_i)$ と表記し, 交差点の符号の総和を「**捻れ**」と呼び, $w(\mathcal{P})$ と表記します. この基本群の元は空間内部の基点から出て基点に戻る向き付けられた閉曲線で, 単位元 1 は滑らかな 2 次元の円盤が空間内部で張れる基点付きの閉曲線 (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は機械的に行えますが, 二つの基本群の同一性の判断は簡単ではありません. そこでより視覚的に分かりやすい組紐群を使った表現についても解説しましょう.

7.6.3　組紐について

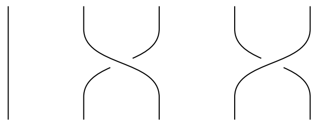
幾何学上の「**組紐 (Braid)**」は n 本の紐 (閉区間 $[0,1]$ と同相) の 3 次元球 B^3 への埋め込みです:

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.

図形的に組紐現は酒樽状の 3 次元球 $B^2 \times [0,1]$ の上下の蓋 ($B^2 \times 1$ と $B^2 \times 0$) に紐が取り付けられ, その紐は互いに交差せずに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みます. このことから組紐は以下に示す 3 成分から構成されます:

組紐の基本成分



例として図 7.5 に SageMath の BraidGroup モジュールを使って描いた 3 本の紐の組紐を示します．このように組紐の構成は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．組紐の定義にはこれとは別に配置空間 (configuration space) を使った定義もあります [47]．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, 紐に結び目を作っても書いませんが, タングルを構成する部品としては組紐の三つの基本成分に下の図で右から二番目の「**最小 (消滅)**」と右端の「**最大 (生成)**」の二つが加わるだけです:

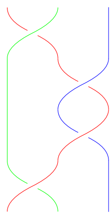
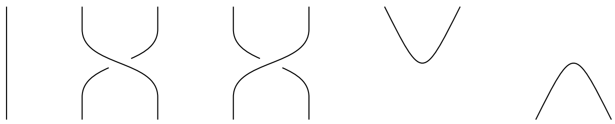


図 7.5　組紐の例

タングルの基本成分



組紐とタングルが同値であるとは結び目/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, すなわち, 二つの組紐に「**アンビエント・イソトピー (ambient isotopy)**」が存在するときです．さらに組紐は結び目/絡み目よりも視覚的にも明瞭な組紐群と呼ばれる群で表現できます．

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

$$\left\langle \sigma_1, \ldots, \sigma_{n-1} \;\middle|\; \begin{array}{l} \sigma_i\sigma_k = \sigma_k\sigma_i, (|i-k| \geq 2, i, k \in [1, n-1]), \\ \sigma_i\sigma_{i+1}\sigma_i = \sigma_{i+1}\sigma_i\sigma_{i+1}, (i \in [1, n-2]) \end{array} \right\rangle$$

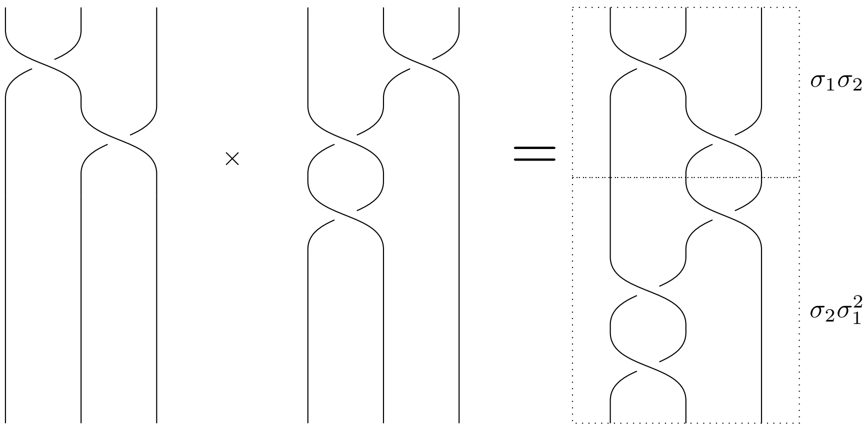
では生成元 σ_i は具体的にどのようなものでしょうか? つぎに σ_i と σ_i^{-1} と対応する組紐を示しておきましょう:

n 糸の組紐群の生成元

σ_i $\quad$ σ_i^{-1}

ここの図示のように σ_i は左から i 番目と $i+1$ 番目の紐を縦方向を Z 軸として反時計回りに 180 度回すという操作, σ_i^{-1} は紐を時計回りに 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. 具体的な例を示しておきましょう:

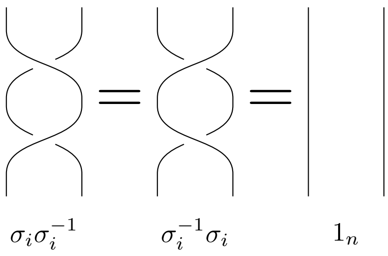
組紐群の積の例



$\sigma_1\sigma_2$ $\quad$ $\sigma_2\sigma_1^2$ $\quad$ $\sigma_1\sigma_2^2\sigma_1^2$

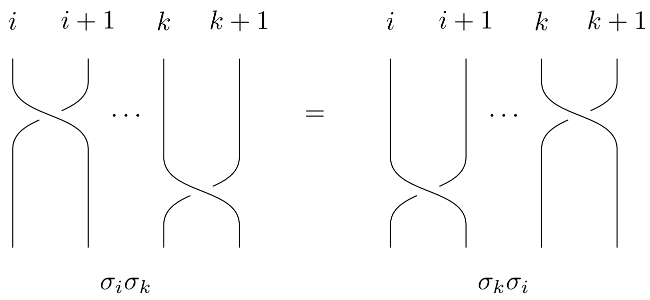
この例では $\sigma_1\sigma_2$ と $\sigma_2\sigma_1^2$ の積 $\sigma_1\sigma_2^2\sigma_1^2$ を示し, 右に示すように $\sigma_1\sigma_2$ を $\sigma_2\sigma_1^2$ の上に積み上げる方式が対応し, 積の順番では左側から右にかけて対応する組紐を上から下に繋ぐ恰好です. さて, ここで n 本の紐がまっすぐに垂れた組紐を n 糸の組紐の上に付けても下に付けても各紐の長さを調整すると元の n 糸の組紐に戻せます. つまり, このことはまっすぐに垂れた n 本の組紐が n 糸の組紐群の単位元 1_n であることを意味しています. また, $\sigma_i\sigma_i^{-1}$ と $\sigma_i^{-1}\sigma_i$ を描くと共に単位元 1_n と同じ組紐になります:

$\sigma_i\sigma_i^{-1}$ と $\sigma_i^{-1}\sigma_i$



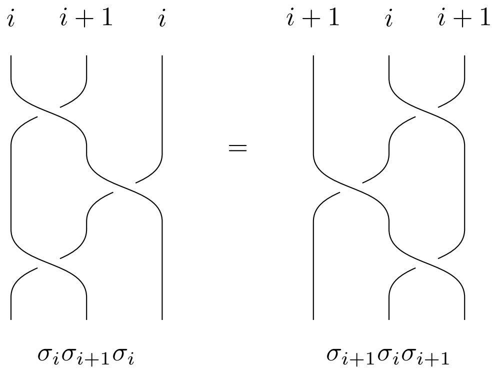
実際, 左側の状態の紐を引っ張って長さを調整すれば 2 本のまっすぐな紐になります. このことから σ_i^{-1} は表記どおりに σ_i の逆元です. また $a_1a_2\cdots a_n$ が与えられたとき, その逆元は $a_n^{-1}\cdots a_2^{-1}a_1^{-1}$ と語の順序を逆にして冪の正負を逆にした語で与えられ, 組紐を重ねるという操作がそのまま組紐を表現する項の積になります. そして, 群の表示には生成元の右側に二つの関係式: $\sigma_i\sigma_k = \sigma_k\sigma_i$ と $\sigma_i\sigma_{i+1}\sigma_i = \sigma_{i+1}\sigma_i\sigma_{i+1}$ があります. これらの関係式は図示するとその意味が明瞭になります. まず最初に $\sigma_i\sigma_k = \sigma_k\sigma_i, |i-k| \geq 2$ を図示しましょう:

$\sigma_i\sigma_k = \sigma_k\sigma_i$ の図示



この図から $|i-k| \geq 2$ であれば $\sigma_i\sigma_k$ と $\sigma_k\sigma_i$ は互いに影響しないために交差点を上下を動かせることに対応し, この移動で互いの組紐が得られるために, これらの組紐を同じものとみなせます. 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:

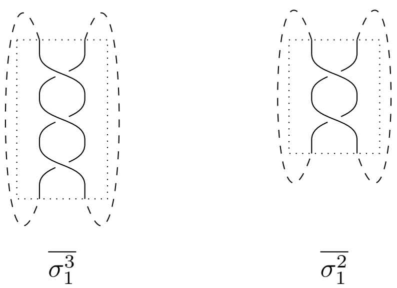
$\sigma_i\sigma_{i+1}\sigma_i = \sigma_{i+1}\sigma_i\sigma_{i+1}$ の図示



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, 結び目/絡み目が得られるライデマイスター移動の TYPE III の操作です. このように関係式を可視化することで図形的な解釈ができます.

7.6.5　組紐と結び目

閉じた組紐の例



この組紐の上下の端点を交互に縦に繋ぐ操作で得られる図形を「**(縦に) 閉じた組紐 (closed braid)**」と呼び, 組紐 b を縦に閉じて得られた絡み目を $\bar{b}$ と表記します. 例の左側の結び目は σ_1^3 から得られる「**三葉結び目**」, 右側の絡み目が σ_1^2 から得られる「**ホップ絡み目 (Hopf Link)**」です.

このように組紐から絡み目が得られましたが, その逆の「**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 [14]. 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固有の値, すなわち, 不変量の一つであることが知られています. そして, 二つの閉じた n 糸の組紐がアンビアント・イソトピーで移りあえるのは次のマルコフ移動 M I, M II を許容するときに限ることが知られています:

マルコフ移動

$$\begin{array}{lcccc} \text{M I.} & \overline{ab} & = & \overline{ba} & \\ \text{M II.} & \overline{a\sigma_n} & = & \overline{a\sigma_n^{-1}} & = & \overline{a} \end{array}$$

これも図示すると明快になります. 移動 M I は a を構成する生成元の閉じた組紐であることから a の上から順にまわして b の下に移せることに対応します. また移動 M II は最も右側の紐がライデマイスター移動の I と同じ状況に対応しますが, ただ, この場合は紐の数に変動が生じます.

7.7　組紐群と置換群

$n+1$ 糸の組紐の上蓋で紐の端点に左側から番号を $1, 2, \ldots, n$ と振り, 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 $1, 2, \ldots, n$ 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 が $\{1, 2, \ldots, n\}$ から $\{1, 2, \ldots, n\}$ への全単射写像で, $1, \ldots, n$ が $(i_1, \ldots, i_n)$ に対応するときに

$$\sigma = \begin{pmatrix} 1 & \ldots & n \\ i_1 & \ldots & i_n \end{pmatrix}$$

と表記します. このような $\{1, 2, \ldots, n\}$ から $\{1, 2, \ldots, n\}$ への全単射写像の集合に対し, その演算を函数の合成 $\circ$ 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, この写像は積を保つ準同型写像です. ただし, この準同型写像は「**回転方向の情報の欠落**」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の別物です.

この置換群から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 $k \in [1, \ldots, n]$ に対して $k \xrightarrow{\sigma} \sigma(k)$, $\sigma(k) \xrightarrow{\sigma} \sigma^2(k), \ldots \sigma^i(k) \xrightarrow{\sigma} \sigma^{i+1}(k) = k$ と k の置換 σ による像を追いかけることで次の置換:

$$\begin{pmatrix} k & \ldots & \sigma^{i-1}(k) \\ \sigma(k) & \ldots & \sigma^i(k) \end{pmatrix}$$

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 $(k, \sigma(k), \ldots, \sigma^i(k))$ と表記され, 任意の置換 $\sigma \in \mathfrak{S}_n$ は巡回置換の積として表現できます. 実際, $\sigma \in \mathfrak{S}_n$ に対

し, 1 から 1 に戻るまで置換 σ で写して巡回置換を求め, この巡回置換で現れなかった数に対して 1 のときと同様に置換 σ で写して巡回置換を求めます. この作業で全ての数が出た時点で置換 σ を構成する巡回置換が全て求められ, これらの巡回置換の積として置換 σ が得られます. さて, ここで組紐を組紐群で表現し, それを置換群への自然な写像で置換 σ に写されたとします. このときに巡回置換は 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, その軌跡が一つの円環を構成しています. 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環, すなわち, 絡み目の成分が現われます.

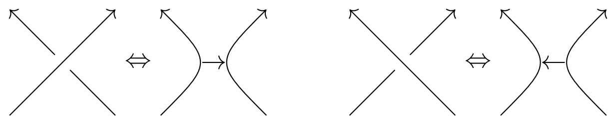
7.7.1　ザイフェルト曲面

結び目/絡み目の組紐への変形処理には不変量の計算で用いるさまざまな操作が含まれます. その中に「**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も含まれます. このザイフェルト曲面は, その境界が絡み目/結び目になる向き付け可能な曲面です. ここで「**向き付け可能**」という意味は曲面に「**裏と表がある**」という意味で,「**向き付られない曲面**」の代表格が帯に 180 度の捻りを入れて繋いでできる**メービウスの帯 (Möbius band)** があります. 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分類されます. この種数は「**ドーナッツの穴の数**」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ッツの表面です. 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. このサイフェルト曲面の構築も踏まえて, 絡み目/結び目の組紐表現の構築について解説しましょう.

■正則射影図の準備:　この処理では向き付けられた絡み目/結び目の正則射影図を利用します. 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で, 円盤を貼りつければザイフェルト曲面が得られます. ところで, 交点が正則射影図にあるときは, 各交点で次に述べる交点解消の処理を行います.

■交差点の解消:　正則射影図の各交差点を次の手順で書き直します:

交差点の付け替え



L_+　　L_{0+}　　L_-　　L_{0-}

交差点の解消で交差点に入る下道は交差点から出る上道に繋ぎ, 交差点に入る上道と交差

点から出る下道に繋ぎ替えます．この繋ぎ替えを行った交差点の性質を示すために矢印を使います．この矢印は素粒子が経路に沿って交叉を解消した「**交差点**」で別の軌道に乗り移る方向を示します．ここで上道と下道が絡み目の別成分の道であれば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施して正則射影図から交差点を消すと, 今度は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．これらの最終的に得られる円周を「**ザイフェルト円周**」, ザイフェルト円周とそれらを繋ぐ矢印の全体を「**ザイフェルト系**」と呼びます．なお, ザイフェルト系の各円周を繋ぐ矢印はそれらを繋ぐ橋として各交差点の符号に対応した捩れた帯の表現です．

実際に三葉結び目の正則射影図を使ってこれらの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, 右図で黒線で示す 2 個のザイフェルト円周と正則射影図の交差点に対応する 3 個の黒色の矢印を持つザイフェルト系が得られます．

■**ザイフェルト曲面の構成**:　ザイフェルト系を構成する各ザイフェルト円周には円盤を貼り付け, それから矢印に正負に対応する捻りを入れた帯で置換えると「**ザイフェルト曲面**」と呼ばれるが得られます．この曲面は向き付け可能な曲面で, その境界は正則射影図に対応する結び目/絡み目になります．ところで, この円盤の貼り方にも二通りの方法があります．まず, その一つが円盤を無限遠点を含まないように貼る方法です．もう一つの方法は無限遠点を含むように貼る方法です．ここで無限遠点を含まないように貼った曲面はホットケーキを数段重ねたような形になりますが, 無限遠点を含むように貼ると平面的な曲面になります．ただし, ザイフェルト系で最大の円周と同じ中心の円で, より半径の大きな円で曲面を切り出して, その円周を円盤で蓋をした曲面も構築できます．下の図 7.7 に二つのザイフェルト曲面を示しておきましょう:

図 7.7　結び目の円盤

左図が無限遠点を含む円盤を貼った結果, 右が無限遠点を含まない円盤のみを貼った結果, 得られたザイフェルト曲面です. 左側は平面上に配置されて孤立した円盤を帯で結び付けるかたちになりますが, ここに図示するように最も外側の円の半球を視点の裏側に膨れるように貼り付けたものとしても表現できます. 右側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帯で置換えて構築されます. 次に円盤を貼る前のザイフェイルト系から組紐表現の構築手順について解説しましょう.

■円の向きを揃える:　ザイフェルト円周を同じ中心点, 同じ向きに揃えます. この手順を以下に示します:

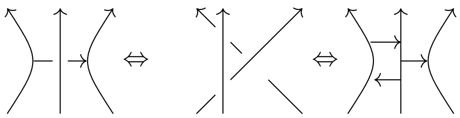
1. 基準円を一つ定めます. 選び方は複数の円の最も内側になっている円, あるいは線分が一番多く出ている円のいずれかです.
2. 基準円と線分で繋がっていない円があれば他の円や矢印と交差しないように基準円と向きと中心を合わせます. 基準円に矢印で繋り, 向きが一致する円は他の円や矢印と交差しないように中心点を揃えます. 逆向きの円があれば構築手順から基準円をその内部に包含しないため, その円に付随する矢印を円周に沿って移動させて矢印と交差点を持たない側の円弧を動かして基準円を包含するようにできます. このときにすでに基準円に合せた円から出ている矢印と交差が生じると 1 本の矢印を 3 本の矢印で置換します.

1. の操作を先程の結び目を使って説明しましょう. まず, ザイフェルト系から基準円を選択した様子を以下に示します:

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, そこから星印で示した点を中心とする基準になる円を選択しています. この場合は円周の向きが揃っていて中心点を揃えられるために最も中央の円周を選択しています. そのために 2. の操作が不要になっています. ところで, 2. の操作が必要な場合は, 円弧の移動によって交差に付替が生じ, その結果, 次に示す線分の張替えが必要になります:

線分と円弧の交差点での付け替え



L_{0+}+arc　　L_{+}+arc　　L'_{0+}

左図が L_{0+} に他の円が線分に交差した状況で, 本来は中心の絵の交差です.　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.

■閉じた組紐表現:　全ての円が同じ向きで共通の中心点に統合できれば, これらの円を繋ぐ矢印を円周に沿って移動させて纏め, それから矢印を本来の交差に戻します. これらの操作によって閉じた組紐表現が得られますが, 交差数が最小になる組紐を与えるものではありません. また, この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とザイフェルト曲面が得られますが, 閉じた n 糸の組紐から n 段重ねのホットケーキ状の曲面になります. もちろん, この曲面が最小種数を持つ曲面とは限りませんが, 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. これらの操作を先程の例に対して行ったものを図 7.9 に示しておきます.

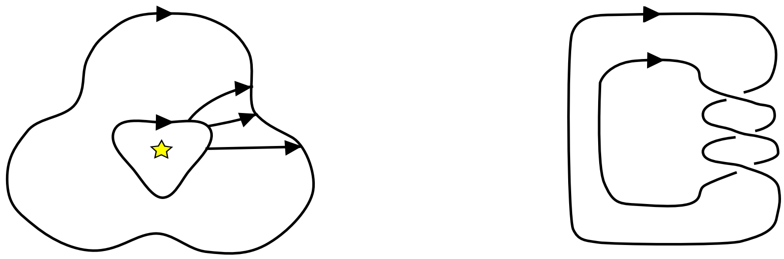


図 7.9　結び目の閉じた組紐としての表現

7.8　ガウス・コード

ここまでの表現は群表現を除いては視覚的なもので, 群表現を行うために人が正則射影図と取り組む必要があり, 計算機で絡み目/結び目を処理させるためには計算機で容易に処理できるデータで表現する必要があります. 絡み目/結び目の正則射影図の表現で「**ガウス・コード (Gauss code)**」と呼ばれる表記があります. このガウス・コードは絡み目の各成分の向きを入れ, 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結果が異なります. このガウス・コードの構成手順は, 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, つぎに各交差点に番号を重複がないように配置して絡み目上の各成分に基点を定めて成分の向きに沿って移動し, 交差点を通過するときに下道を通過すれば交差点番号に '-', 上道を通過すれば交差点番号だけとし, この操作を基点に戻るまで行います. この手順を三葉結び目で確認しましょう.

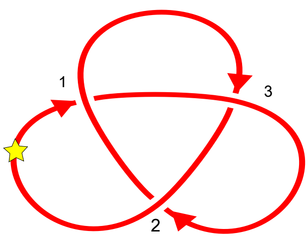


図 7.10　三葉結び目でのガウス・コードの生成

図 7.10 に示す向き付けられた結び目の基点☆からその向きに沿って一周するときに, 最初に出会う交差点 1 で下道になるために '-1', 次の交差点 3 で上道になるために '3', それから交差点 2 で下道になるために '-2' , 二度目に通過する交差点 1 で上道であるために '1', 交差点 3 では下道になるために '-3', 次の交差点 2 では上道になるために '2' と出発点の基点☆に戻るまでに通過する交差点の状況に対応する整数列 $-1, 3, -2, 1, -3, 2$ がこの結び目のガウス・コードです. ちなみに, この結び目のようにカウス・コードの成分の符号が交互に変化する結び目を「**交代結び目 (alternating**

knot)」と呼びます．また, 絡み目のガウス・コードは, 向き付けた絡み目の各成分に順序を入れ, その順序に従って成分毎に上記の方法で数列を構成するために絡み目のガウス・コードは各成分のガウス・コードに対応するリストを成分とするリストになります．また, 結び目も成分が一つだけの絡み目として解釈し, この解釈に基いた表現を採用します．すなわち, 三葉結び目のガウス・コードは一成分の絡み目としてリスト $[[-1, 3, -2, 1, -3, 2]]$ と表記します．

このガウス・コードに交差点の符号の情報が直接, 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．そこで, 各交差点の符号情報を追加した「**拡張ガウス・コード**」を考えます．拡張ガウス・コードは二種類の表記方法があり, 一つは交差点番号の前に 'p', 'm' 等の交差点の符号を表現する記号を付与する表記, もう一つは二度目に通過した時点の交差点の符号をその交差点番号に付ける表記です．計算機のデータとしては, 最初の方法は文字式のリスト, 二番目の方法は整数のリストになり, 計算機での処理は後者が単純です．なお, 復号化では交差点が同じ成分の道同士の交差で生じたものであれば一つの成分の参照で済みますが, 他の成分の交差であれば, 該当する成分の参照が必要になります．実際, 絡み目 L の二つの成分 K_i, K_j が交差していれば, これらの成分が横断的に交差するために K_i と K_j の交差点は必ず偶数個になります．ここで絡み目 L の成分順序で $K_i > K_j$ のときに K_i の交差点では道の上下関係に依存する符号, K_j の拡張ガウス・コードに交差点の符号が添付されます．復号化では交差点番号 p が一つだけ成分 K_i に現れなければ 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操作が必要になります．なお, ガウス・コードでは交差点の正負に意味を持たせているために, 1 以上の整数を交差点番号に用いなければなりません．参考までに図 7.10 の三葉結び目の拡張ガウス・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]] です．なお, 後者の拡張ガウス・コードは SageMath の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)」として提供されており, この本でも後者の整数列としての表記を採用します．

拡張ガウス・コードからは絡み目/結び目の交差点が容易に復元可能で, さらに Wirtinger 表現による基本群の計算ができます． 実際, 交差点番号が道に, その道が Wirtinger 表現での基本群の生成元に対応し, 基本群の関係子は n 個の交差点を有する正則射影図の $n-1$ 個の交差点から機械的に生成できます．この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. 正規化ガウス・コードの構築
4. 交差点情報の復元

絡み目の拡張ガウス・コードは絡み目の各成分に対応する成分リストから構成されており, 各成分の向きは成分リストを構成する交差点の順序で表現され, 絡み目の成分数 (濃度) n_c は拡張ガウス・コードの成分数が対応します. なお, 交差点における道の探索では成分リストをその両端を繋いだ円環として考察し, 円環の順方向はリストのときの順方向に一致させます. また, 交差点の符号を拡張ガウス・コードから読み取るとき, 最初の交差点の出現で上道か下道かが定まり, 二番目の出現で交差点の符号が決定されるために, 交差点の出現順序が重要になりますが, 絡み目の場合は交差点が二つの成分に分離するためにガウス・コードの成分に跨がる交差点の検索が必要になります. そこで, ガウス・コードの成分を区分する括弧 “[ ]” を外した平坦なリスト L_f を用いれば, 成分間での検索処理が削減できます. このリスト L_f を「**平坦化した拡張ガウス・コード**」と呼びます. このリスト L_f から交差点集合 S_c が得られます.

■**交差点の抽出**: 交差点リスト L_c を構成します. 一般的な方法は平坦化した拡張ガウス・コード L_f の先頭から順番に符号付き交差点番号 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 $|i|$ を交差点番号リスト L_c に追加します. ここでの交差点が絡み目群の Wirtinger 表現における生成元に対応します.

■**交差点の符号の抽出**: 交差点番号リスト L_c に対応する交差点の符号リスト L_s を構築します. ここでは計算効率の問題から平坦化した拡張ガウス・コード L_f を用います. 交差点番号リスト L_c から順番に交差点番号 i を取り出して L_f で二度目に番号 i に対応する数が出ると, 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. なお, 交差点リスト L_c と交差点符号リスト L_s は必ず対で処理しなければなりません. すなわち, L_c と L_s の第 i 成分をそれぞれ c_i, s_i とするときに, 交差点番号が c_i の交差点での符号は s_i でなければなりません.

■**正規化ガウス・コードの構築**: 拡張ガウス・コードから交差点の符号情報を持たない正規化ガウス・コードを構築します. 最初に拡張ガウス・コードの成分数のリスト n_s を生成しておきます. それから, 交差点リスト L_c の先端から順番に交差点番号 c_i を取り出して, この交差点番号に対応する成分が平坦化拡張ガウス・コード L_f での位置を検出します. ここで最初の出現は上道か下道を示す符号であるためにそのままとし, 二番目の出現を第一回目の L_f の符号を反転したもので置き換えます. このように平坦化拡張ガウス・コードの内容を正規化ガウス・コードの内容で置き換えます. ここで平坦化拡張ガウス・

コードは絡み目に対応する成分リストの括弧 “[ ]” を外す処理であって, 各成分ごとに並んでいるために, 予め生成しておいた成分数リスト n_s に従って切り分ければ本来のガウス・コード G_c が構築できます. この G_c をここでは特に「**正規化ガウス・コード**」と呼びます.

■自明な絡み目成分の除去: 正規化ガウス・コード G_c の成分は絡み目成分に対応するリストですが, その交差点番号が正のものだけ, あるいは負のものだけの絡み目の成分のリストは他の成分と本質的に絡み合っていない自明な成分です. このことは座標系を上手く設定すると他の成分をその座標系の Z 軸方向の負側に, その成分を Z 軸方向の正側に置いて平面 $Z = 0$ で綺麗に分離できるためです. そこで, 正規化ガウス・コードからこれらの成分を除去して番号を振り直します.

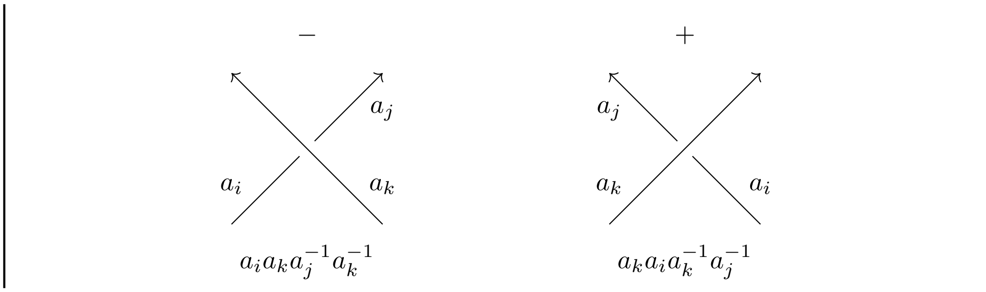
■交差点情報の復号: 絡み目群を Wiritinger 表現で構築するためには交差点情報を復号する必要があります. ここで復号すべき情報は上道と下道の配置状況です. この情報は正規化ガウス・コード G_c から交差点リスト L_c の成分順に復元します. 交差点リスト L_c の番号 i が平坦化ガウス・コード L_f に最初に現われたときの符号が道の交差状況, つまり, 交差点での上道と下道が何であるかという情報です. まず, 交差点番号 i に下道として入り込む道は道の定義から下道 a_i で確定していますが, 残りの交差点 i から出る下道と交差点 i を通過する上道の検出が問題になります. そこために $i \in L_c$ が正規化ガウス・コード G_c でどのように現われるかを考察しましょう:

- 負の整数 $-i$ の場合:
  道 a_i が交差点 i に潜り込むことを意味していますが, 交差点 i から出る道はこの交差点を出て次の負の交差点番号に出会うまで判りません. そこで正規化ガウス・コード G_c の $-i$ を含む成分リストを円環にし, 順方向で $-i$ に最も隣接する負の数 $-j$ を探します. そのような負の数 $-j$ はガウス・コードの生成方法から必ず存在し, a_j が交差点 i から出る道になります. 次に交差点 i を通過する上道 a_k の探索を行います. まず, i を包含する G_c の成分リストを探します. それから見付けた成分リストを円環にして, i から順方向に出発して i に最も近い負の数 $-k$ を探します. この負の数 $-k$ で定まる道 a_k が求める上道になります.
- 正の整数 i の場合:
  交差点 i を通過する上道がどの道に対応するかは, そ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. そのために交差点番号 i を包含する正規化ガウス・コード G_s の成分リストを円環にした状態で考えます. このときに交差点番号 i から出て順方向で i に最も隣接する負の数 $-k$ を探し, この上道 a_k が求める上道になります. 次に下道の検出は $-i$ を $-i$ を包含する正規化ガウス・コードを円環にして, 交差点 i から出る下道を $-i$ を基点に順方向で探します. この i に最も近接

する負の数 $-k$ で定まる道 a_j が求める下道になります.

このようにして求めた上道と下道の情報と交差点の符号の情報から交差点が次で復元できます:

番号 i の交差点での関係子



では, 図 7.10 の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. この処理で $[[-1,3,-2,-1,-3,-2]]$ が拡張ガウス・コードとして得られます.
2. 結び目の基本的な情報の収集:
   交差点集合 S_c は $\{1,2,3\}$, 交差点リスト L_c は $[1,2,3]$, 平坦化拡張ガウス・コード L_f は $[-1,3,-2,-1,-3,-2]$ です.
3. 交差点の符号と正規化ガウス・コードの計算;
   平坦化拡張ガウス・コード L_f をひっくり返して $[-2,-3,-1,-2,3,-1]$ とすると, 拡張ガウス・コードの構築方法から, このリストで最初に現れた数の符号が交差点の符号. したがって, 交差点符号リスト $L_s = [-1,-1,-1]$ が得られます. 次に平坦化拡張ガウス・コードで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化ガウス・コード $G_c = [[-1,3,-2,1,-3,2]]$ が得られます.
4. 交差点の復元:
   交差点番号を含む成分リストを円環にして考えます. まず, 交差点リスト $L_c = [1,2,3]$ から関係子の構築に必要な情報を取得します. 最初の交差点番号 1 に対しては成分リストで -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, 交差点番号 1 から出る下道は a_2, 次に数 1 を基点に順方向で最初に現れる負の数は -3. したがって, 交差点番号 1 を通過する上道は a_3. 以上から交差点に入る下道, 交差点から出る下道, 交差点での上道, 交差点の符号を

リストとして表現すると $[1,2,3,-1]$ が交差点番号 1 の情報になります. 次に交差点番号 2 では数 -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, 数 2 の順方向で最も近い負の数が -1 になるため上道は a_1. これらをリストで先程と同様に表記すると $[2,3,1,-1]$ になります. 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 つの交差点情報で十分です.

これらの交差点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

$$\langle a_1, a_2, a_3 \mid a_3a_1a_3^{-1}a_2^{-1}, a_2a_3a_2^{-1}a_1^{-1}\rangle$$

が得られます.

このようにガウス・コードの処理はリスト処理が中心で, あとは群表示のために交差点を復元するだけです. ここで述べた手続きを SageMath で表現してみましょう. まずは簡単なリスト処理を中心とした函数です:

```
def checkPlusOnly(L):
    """
    L: List
    与えられリストの全成分が0より大のときにTrue,
    それ以外はFalseを返す函数.
    """
    if True in [i<0 for i in L]: return False
    else: return True

def checkMinusOnly(L):
    """
    L: LIST
    与えられたリストの全成分が0より小のときにTrue,
    それ以外はFalseを返す函数.
    """
    if True in [i>0 for i in L]: return False
    else: return True

def getCrossings(GaussCode):
    """
    GaussCode: 拡張ガウス・コード. 平坦化されている必要性はない.
    与えられたガウス・コードから交差点のリストL_cを返す函数.
    """
    L_f = flatten(GaussCode)
    AL_f = [abs(i) for i in L_f]
    L_c =list(set(AL_f))
    return L_c
```

```
def getSignsOfCrossings(L_c, L_f):
    """
    L_c: 絡み目の交差点リスト
    L_f: 平坦化拡張ガウス・コード
    交差点リストの頂点に対応する交差点の符号(1 または−1)
    のリストL_sを生成
    """
    L_s = []
    RL_f = L_f[::-1]
    ARL_f = [abs(i) for i in RL_f]
    for i in L_c:
        n = ARL_f.index(i)
        L_s.append(sign(RL_f[n]))
    return L_s

def getGenerators(L_c):
    """
    L_c: 交差点リスト
    交差点リストから絡み目群の生成元リストを生成.
    """
    Generators = "a_"+str(L_c[0])
    for i in L_c[1:]: Generators += ", a_" + str(i)
    return(Generators)
```

函数 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. そして, 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. この函数では数値リストからその絶対値のリストを構築し, それから集合, そしてリストへと変換することで交差点リスト L_c を生成します. この函数 getCrossings() の中でリストの平坦化を SageMath の函数 flatten() を使って実行していますが, Python の内包表現を使っても平坦なリストの構築が容易にできます. 函数 getSignsOfCrossings() は交差点リスト L_c と平坦化拡張ガウス・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. ここで平坦化拡張ガウス・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のため, 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 が ARL_f で最初に現われる位置を検出して, その位置情報から RL_f の符号を求めて符号リスト 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, 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生成元を ‘a_i’ の書式にしています.

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):
    """
    ExGaussCode: 拡張ガウス・コード
    与えられた拡張ガウス・コードからガウス・コードG_c,
    交差点リストL_cと交差点の符号リストL_sを返却する函数.
    """
    G_c = []
    n_s = [len(i0) for i0 in ExGaussCode]
    L_c = getCrossings(ExGaussCode)
    L_f = flatten(ExGaussCode)
    L_s = getSignsOfCrossings(L_c, L_f)
    AL_f = [abs(i) for i in L_f]
    RAL_f = AL_f[::-1]
    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(j):
            L_t.append(L_f[0])
            L_f.pop(0)
        G_c.append(L_t)
    return [G_c, L_c, L_s]

def removeTrivialComponents(G_c, L_c, L_s):
    """"
    G_c: ガウス・コード
    L_c: 交差点リスト
    L_s: 交差点符号リスト
    自明な成分を取り除いたガウス・コード, 交差点リストと
    符号リストを返却する函数.
    """
    nG_c = []
    L_t = []
    for C in G_c:
        if checkPlusOnly(C): L_t.append(C)
        if checkMinusOnly(C): L_t.append([abs(i0) for i0 in 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):
    """
    G_c: ガウス・コード
    L_c: 交差点リスト
    L_s: 交差点符号リスト
    Wirtinger表現による関係子を生成するためのリストL_rを生成
    する函数.
    """
    L_r = []
    L_f = flatten(G_c)
    n_c = len(L_c[:-1])
    for h in range(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(-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_t.append(L_s[h])
        L_r.append(L_t)
    return L_r
```

函数 transGaussCode() は拡張ガウス・コードから正規化ガウス・コード G_c, 交差点リスト L_c とそれに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番号をガウス・コードから除去することで交差の解消を行っています. 函数

relatorOfCrossings() は正規化ガウス・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です:

```
def representationLinkGroup(Generators, L_c, L_r):
    """
    Generators: 自由群の生成元
    L_c: 交差点リスト
    L_r: 関係子の情報リスト
    Wirtinger表現による絡み目群を返却する函数.
    """
    Relators = []
    F = FreeGroup(Generators)
    for r in L_r:
        [i1, j1, k1, sgn] = r
        [i, j, k] = [L_c.index(i)+1 for i in [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):
    """
    ExGaussCode: 拡張ガウス・コード
    拡張ガウス・コードからガウス・コード, 交差点リスト, 交差点符号
        リストと
    Wirtinger表示による絡み目群を返却する函数.
    """
    G_c = []
    L_c = []
    L_s = []
    L_r = []
    Relators = []
    [G_c, L_c, L_s] = transGaussCode(ExGaussCode)
    [G_c, L_c, L_s] = removeTrivialComponents(G_c, L_c, L_s)
    Generators = getGenerators(L_c)
    F = FreeGroup(Generators)
    L_r = relatorsOfCrossings(G_c, L_c, L_s)
    G = representationLinkGroup(Generators, L_c, L_r)
    return [G_c, L_c, L_s, G]
```

SageMath で入力した多項式は何も指定しなければ通常の多項式環, すなわち, その積は

可換積として認識されて, さらに入力式の変数や項の並び換えと式の簡易化が自動的に実行されてしまいます. そのために群の関係子の本来の意味が失われることになります. このことを避けるために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. そこで自由群 F とその商群をあらかじめ定義してから, その自由群 F の中で関係子を定める必要があります. 関係子の計算は函数 representationLinkGroup() で処理しますが, 函数内部では自由群 F を定義し, その F の元として関係子をリスト形式で定めています. なお, 函数 FreeGroup() で生成元の指定は Python 流儀の 0 からではなく 1 から開始します. そのために交差点のリストと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, 絡み目群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 LinkGroup() はこれらの函数をまとめて拡張ガウス・コードから正規化ガウス・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.

これらの函数による計算例を示しましょう. 最初の結び目は図 7.10 に示す「**三葉結び目 (trefoil)**」で, この結び目のガウス・コードは [-1,3,-2,1,-3,2] で, 拡張ガウス・コードは [-1, -3, -2, -1, -3, -2] です. また, 二番目の結び目は「**8 の字結び目 (figure eight)**」と呼ばれる結び目です:

```
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, 2, 3],
 [-1, -1, -1],
 Finitely presented group < a_1, a_2, a_3 | a_1*a_3*a_2^-1*a_3^-1,
 a_2*a_1*a_3^-1*a_1^-1 >]
sage: EGC_FigureEight = [[-1, 3, -2, 4, 3, 1, 4, 2]]
sage: KG_FigureEight = LinkGroup(EGC_FigureEight)
sage: KG_FigureEight

[[[-1, 3, -2, 4, -3, 1, -4, 2]],
 [1, 2, 3, 4],
 [1, 1, 1, 1],
 Finitely presented group < a_1, a_2, a_3, a_4 | a_4*a_1*a_4^-1*a_2
     ^-1,
 a_1*a_2*a_1^-1*a_3^-1, a_2*a_3*a_2^-1*a_4^-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 LinkDiagram クラス

ここでは先程の単発的なプログラムを整理して, より一般的に絡み目のクラス LinkDiagram を定義しようと思います. ところで絡み目のクラス LinkDiagram はどのようなものであるべきでしょうか? このクラスは当然, 拡張ガウス・コードを与えるてインスタンス化されるべきで, 拡張ガウス・コードから正規化ガウス・コードや交差点符号等の情報を含むべきです. そして, 基本群も重要な属性として保持すべきでしょう. 以上から絡み目のクラスとして持つべき属性として

- 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

が挙げられます. 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化ガウス・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す. また, 絡み目は連結和等の二項演算があります. これもクラス LinkDiagram で表現しようと思います. このように演算といった数学的構造も導入したいため, SageMath の数学的オブジェクト SageObject のサブクラスとして定義します.

これらのことを踏まえて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定義しましょう. ここで天下り的ですが, このクラスに必要なパッケージを取り込む import 文を以下のものとします:

```
import sqlite3
import networkx as nx
import matplotlib.pyplot as plt
from sage.structure.element import RingElement
```

import 文記述は通常の Python プログラムと違いはありません. 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 とします. まず, SQLite3 は軽量 RDB で絡み目不変量の計算で現れる正則射影図の保存と管理のために用います. NetworkX はグラフ理論のためのパッケージで,

ザイフェルト系を有向グラフとして表現するために用います．そして, ザイフェルト系の可視化でパッケージ matplotlib の plot を用います．ここまでがクラス LinkDiagram を定義する上での準備になります．以下にクラス LinkDiagram の定義を機能に分割して記述しますが, ファイルや Jupyter notebook にはまとめて一つのファイルやセルに記述しなければなりません.

```
class LinkDiagram(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):
        rstr = ' GaussCodes:%s\n Crossings:%s\n'
        rstr = rstr + ' Signs:%s\n Components: %s\n'
        rstr = rstr + ' SeifertCircles:%s\n'
        sgc = str(self.GaussCodes)
        scr = str(self.Crossings)
        sgn = str(self.Signs)
        scm = str(self.Components)
        ssc = str(self.SeifertCircles)
        return rstr %(sgc, scr, sgn, scm, ssc)
```

‘class LinkDiagram(RingElement)’ でクラス RingElement をクラス LinkDiagram が継承することを指示し, これで RingElement が持つ代数的構造と付随するメソッドや属性が使えるようになります．ここに記載している二つのメソッドはインスタンスの初期化を行う特殊メソッド__init__() とインスタンスの表示を行うメソッド_repr_() です．まず, 特殊メソッド__init__() では拡張ガウス・コードを使って初期化を行うために

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 を記載します．ここで，'ExGaussCodes=None' と特殊メソッド___init___() の引数として記載すれば ExGaussCodes の既定値として None が設定され，この None を自明な結び目の値とします．それからメソッド内部で属性 ExGaussCodes の値で自明な結び目か，拡張ガウス・コードの処理を行うか分岐するようにします．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化カウス・コードと符号付き交差点のリスト，絡み目の成分数，ザイフェルト円周のリストを生成するようにします．このように属性 GaussCodes, Crossings, Components にインスタンス化で計算した値がそれぞれ割り当てられますが，属性 SQL は多項式不変量の処理で SQLite3 の RDB の情報を載せるために準備した属性のためにメソッド___init___() では指示しないことにします．

それからインスタンス名を指示したときに，とのような絡み目/結び目であるかを分かり易い書式で表示させたいものです．これを実現させるために特殊メソッド___repr___() があります．このメソッドはクラス LinkDiagram が継承する環のクラス RingElement にも当然あります．LinkDiagram クラスのインスタンス名を入力したときに表示させる内容としては，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としています．そして，指定した書式の文字列を return 文で返して対処します．

7.9.1 ガウス・コードの処理を行うメソッド

次にガウス・コードの処理を受け持つ Python プログラムに対応するクラス LinkDiagram のメソッドを纏めておきましょう．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，実際の利用で引数 "self" は記載しません．また，クラスやインスタンスの属性を書き直すメソッドに return 文は不要です:

```
    def checkPlusOnly(self, C):
        if True in [i<0 for i in  C]: return False
        else: return True

    def checkMinusOnly(self, C):
        if True in [i>0 for i in C]: return False
        else: return True

    def getCrossings(self):
        L_f = flatten(self.ExGaussCodes)
        AL_f = [abs(i) for i in L_f]
        self.Crossings =list(set(AL_f))

    def getSignsOfCrossings(self):
```

```
        self.Signs = []
        L_f = flatten(self.ExGaussCodes)
        RL_f = L_f[::-1]
        ARL_f = [abs(i) for i in  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 = [len(i0) for i0 in self.ExGaussCodes]
        self.getCrossings()
        Crossings = self.Crossings
        L_f = flatten(self.ExGaussCodes)
        self.getSignsOfCrossings()
        AL_f = [abs(i) for i in L_f]
        RAL_f = AL_f[::-1]
        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 removeTrivialComponents(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(C)
                if self.checkMinusOnly(C): L_t.append([abs(i) for i
                    in 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]
    nL_f = copy(L_f)
    L_c = self.Crossings
    L_s = self.Signs
    L_n = [len(i) for i in 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 += ", 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] = [self.Crossings.index(i)+1 for i in [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
```

上記のガウス・コードを処理するメソッドは先程の Python プログラムを基にしていますが, その中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 LinkDiagram に和や積といった演算に関わる代数的構造を入れることで, 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 $1, 2, \ldots, n$ を割り当て直し, メソッド removeTrivialComponents() 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．また, 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 に対する連結和等の演算 $L_1 \circ L_2$ の計算で生成されるリストが, これら二つの絡み目 L_1, L_2 の交差点の和集合になり, 双方の交差点の番号の衝突を避けるため, 第二引数 L_2 の交差点番号を L_1 の交差点数が n_1 のときに n_1 だけ移動させるメソッドです．これらのメソッドはリストの番号の振り直しでしかありませんが, ガウス・コードを観察することで初歩的な絡み目の同一性が判断できるようにする意図もあります．

7.9.2 導入すべき代数的構造

ここでクラス LinkDiagram はクラス RingElement を継承します．これは絡み目の代数的な構造に注目した結果です．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「**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, 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 次元平面に射影して二重点 (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・コードを基にして記述されています．だから, 扱う対象は拡張ガウス・コードであるべきです．これで対象が決まりました．次に正則射影図には連結和と直和の二つの演算があり, どちらにも「**和**」という言葉があります．だからといって全てを演算子 “+” で表現する訳にはいきません．では, 最初の連結和はどのように表現すべきでしょうか? まず, 絡み目の連結和は一つの成分での処理のために結び目の連結和が基底にあります．そして, 結び目の連結和 $K_1\#K_2$ は双方の起点で連結和の処理を行うのであれば最初に K_1 を回って次に K_2 を回るために単純にリストの結合として捉えられ, 単位元は空リストで表現できます．そして, 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, 絡み目 $L = \cup_{i=1}^{m} L_i$ を順序対 $\langle L_1, L_2, \ldots, L_m \rangle$ とするときに, 絡み目 L_1, L_2 の連結和は, その第 1 成分間の演算として定義すれば演算が一意に定まります．つまり, 絡み目 $L_1 = \langle L_1^1, \ldots, L_1^m \rangle$ と $L_2 = \langle L_2^1, \ldots, L_2^n \rangle$ の連結和を $L_1\#L_2 = \langle L_1^1\#L_2^1, L_1^2, \ldots, L_1^m, L_2^2, \ldots, L_2^n \rangle$ と定めます．この絡み目の連結和は成分の並び換えを許容すれば一意に定まりますが, 並び順を変項しなければ, 結び目同士, あるいは結び目と絡み目であれば可換, 2 成分以上の絡み目の同士の連結和は可換ではありません．実際, ガウス・コードが異なります．また, 連結和の単位元は自明な結び目で, この自明な結び目のガウス・コードは空リストのリスト ‘[[]]’ で表現されるために, この表現をそのままガウス・コードの単位元としても使えます．このように絡み目の連結和についてはガウス・コードの表現が一致しないかもしれないという難点があるものの, 絡み目自体は成分の並び換えて一致させることができるために, ここでは可換とし, 連結和の可換性から和 “+” で表記します．現実的には, 差の演算子 “-” に対応

する操作がないために, $(L_1+L_2)-(L_1+L_2)$ で悩むこともありません. そして, 絡み目のもう一つの演算である直和を

$$\langle L_1^1, L_2^1, \ldots, L_m^1\rangle \oplus \langle L_1^2, L_2^2, \ldots, L_n^2\rangle = \langle L_1^1, \ldots, L_m^1, L_1^2, \ldots, L_n^2\rangle$$

で定めます. この直和の単位元として空集合 $\emptyset$ があり, Python では 'None' が対応するでしょう. また, 絡み目の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. この直和をとりあえず積 "*" としましょう. また, 絡み目 L_1, L_2, L_3 で, L_2, L_3 の成分数が等しいときに $L_1*(L_2+L_3)$ は積を分配した結果の $L_1*L_2+L_1*L_3$ と異なります. つまり, 演算は二つ存在しますが, これらの演算では分配律を充たしません.

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, これに類似した数学上の対象には何があるでしょうか？ まず, 自然数 $\mathbf{N}$ がそうで, 式 '$1+1$' にある和 "+", 式 2×3" にある積 "×" の二つの演算があります. しかし, これらの演算は全て可換で, 絡み目の演算と異なります. では, 2 次正方行列の集合 $M(2)$ はどうでしょうか? こちらは和 "+" は可換で積 "·" は全ての元に対して可換でない点は絡み目と同様で, この 2 次の正方行列がモデルになりそうです. この 2 次の正方行列は環と呼ばれる数学的対象であり, このことが, 先程, 絡み目の連結和を和演算 "+", 直和を積演算 "*" で表現する正当性を補強し, さらには抽象基底クラス RingElement を継承することにした理由です. ただし, 絡み目のこれらの演算では結合律を充すものの分配律は充さず, この点で 2 次正方行列とは異なっています. この環構造の継承は演算それ自体の定義ではなく, その演算が持つ性質を利用するときに威力を発揮します. 実際, Python の和演算 "+" の定義は特殊メソッド ___add___(), 積演算 "*" の定義は特殊メソッド___mul___() の上書きで対処できますが, これらの二項演算子は結合律を充さないために $a+b+c$ や $a*b*c$ のような式はエラーになります. しかし, RingElement を継承すると, そのインスタンスの和や積を再定義するだけで, それらの演算が自動的に結合律を充します.

そこで, 連結和と直和をクラス LinkDiagram で定義しましょう. まず, 連結和は与えられた拡張ガウス・コード先頭の成分に対して行ない, 前述のようにリストの結合を基とします. 図 7.3 に三葉結び目と八の字結び目の連結和の例を示しますが, 連結和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あり, 和 "+" 右辺の被演算子の結び目に対して交差点番号の付け直しが必要になります. そこで, メソッド numberShift() で被演算子の番号の付け替えを行った上で拡張ガウス・コードを構築します. そして, 直和演算そのものは RingElement に用意された和演算のメソッド_add_() の上書きで対処します. ちなみに, このメソッドは二項演算子であるため, これら二つの被演算子に対応する引数が必要で, その一つはメソッドの定義で必要な self, もう一つが同一クラスの別インスタンスに対応する other で, self と other で指示されたインスタンスの ExGaussCodes で, そ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・コードから新たなインスタンスを生成して返します. これは直和も同様で, 直和の非可換性から RingElement の積演算を用い, メソッド__mul__() の上書きで対処します.

```
    def __add__(self, other):
        m = len(self.Crossings)
        n = len(other.Crossings)
        if m==0: return LinkDiagram(other.ExGaussCodes)
        if n==0: return LinkDiagram(self.ExGaussCodes)
        a0 = self.ExGaussCodes[0]
        b0 = []
        b = other.ExGaussCodes[0]
        for i in b:
            if i>0: i = i + m
            else: i = i - m
            b0.append(i)
        ExGaussCodes = [a0 + b0]
        L_f = [abs(i0) for i0 in flatten(ExGaussCodes)]
        mx = max(L_f)
        for C in self.ExGaussCodes[1:]:
            E_G = []
            for i in C:
                if not abs(i) in L_f: i = i + mx * (-1)^(i<0)
                E_G.append(i)
            ExGaussCodes.append(E_G)
        for C in other.ExGaussCodes[1:]:
            E_G = []
            for i in C:
                if i<0: i = i + mx * (-1)^(i<0)
                E_G.append(i)
            ExGaussCodes(E_G)
        return LinkDiagram(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 = [abs(i0) for i0 in 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 LinkDiagram(ExGaussCodes)


    def numberShift(self, n=None):
        if n is not None and n>0:
            ExGaussCodes = []
            m = min([abs(i0) for i0 in flatten(self.ExGaussCodes)])
            if n is None: bs = m -1
            else: bs = m - 1 - n
            for C in self.ExGaussCodes:
                D = []
                for i in C: D.append(i - sign(i) * bs)
                ExGaussCodes.append(D)
        else: ExGaussCodes = self.ExGaussCodes
        return ExGaussCodes
```

7.9.3 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 getComponent(self, n=None):
        ExGaussCode = []
        if n is not None:
            k = 0
            if n<self.Components:
                xgssc = self.ExGaussCodes[n]
                axgssc = [abs(i0) for i0 in xgssc]
                crssngs = list(set(axgssc))
                chck = [axgssc.count(x)==2 for x in axgssc]
                for i in chck:
                    if i: ExGaussCode.append(xgssc[k])
                    k = k + 1
        return LinkDiagram([ExGaussCode])

    def delComponent(self, n=None):
        dcrssng = []
        xgsscs = self.ExGaussCodes
```

```
        if n is None or n>self.Components:
            return LinkDiagram(self.ExGaussCodes)
        elif n-1 in range(self.Components):
            if n<self.Components:
                exgssc = xgsscs[n]
                xgsscs.pop(n-1)
                if len(exgssc)>0:
                    aexgssc = [abs(i0) for i0 in exgssc]
                    chck = [aexgssc.count(x)==1 for x in 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([-i for
                 i in x])
        else:
            xgausscode = [-i for i in self.ExGaussCodes[n]]
            xcrssngs = [abs(i) for i in xgausscode]
            crssngs = list(set(xcrssngs))
            cnt = [xcrssngs.count(i)==1 for i in crssngs]
            w = range(0,len(cnt))
            if sum(cnt)>0:
                crps = list(set([cnt[i]*crssngs[i] for i in 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([-i for i in  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 をオプションとし, 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.

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)
            gcs = []
            gcr = []
            sgn = sign(i)
            ai = abs(i)
            mi =-ai
            for C in GaussCodes:
```

```
                if not(i in C or -i in C): gcs.append(C)
                else: gcr.append(C)
            if len(gcr)==1:
                GaussCode = gcr[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(gcr)==2:
                    if ai in gcr[0]:
                        ga = gcr[1]
                        gb = gcr[0]
                    else:
                        ga = gcr[0]
                        gb = gcr[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 gcs: newGaussCodes.append(j)
            GaussCodes = newGaussCodes
        self.SeiferCircles = GaussCodes
```

メソッド seifert_circles() は各交差点で解消処理を行います．のちの不変量の計算では上道と下道の繋ぎ変えで向きを変更する処理が入りますが, 向きの逆転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リストです．番号は絡み目の各成分に与えた向きに従って並んでいます．

7.9.4 ザイフェルト系の可視化について

ザイフェルト系の可視化を行うメソッドを定義します．このメソッドはグラフの生成にパッケージ NetworkX, 生成したグラフの描画で matplotlib の pyplot から函数 plot() を利用します:

```
    def draw_seifert_circles(self, file=None, layout=None):
```

```
G = nx.MultiDiGraph()
G.clear()
scs = self.seifert_circles()
sgs = [self.Crossings[i]*self.Signs[i] for i in range(len(
    self.Signs))]
nodes = []
nodelist = []
fsc = []
edges = []
a = ''
b = ''
asc = 97
for C in scs:
    b = ''
    tmp = []
    for i in C:
        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=1, sign=0)
    if len(C)>0: a = chr(asc) + str(abs(C[0]))
    nodelist.append(tmp)
    edges.append((b,a))
    G.add_edge(b, a, weight=1, sign=0)
    asc = asc + 1
G.add_nodes_from(nodes)
for j in sgs:
    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)
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
```

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．ここでザイフェルト系は孤立したザイフェルト円周と, その円周上の点との間に矢印が入ったものです．このことからザイフェルト系を有向グラフとして定義します．都合の良いことに SageMath はグラフ理論向けのパッケージ NetworkX を標準で含んでます.

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)**」, 各交差点を繋ぐ道を「**辺 (edge)**」として定義します．また, ザイフェルト系では同じ交差点が二つの円周上に現われるために番号をそのまま節点にすると何を描いているのかが不明瞭になります．そこで, ザイフェルト系の円周単位で節点をグループ分けし, 節点名もグループに対応させます．たとえば, グループ a の交差点 i を節点 ai とするとプログラムでは函数 char() で整数からグループに対応する ASCII 文字を生成し, 交差点番号を函数 str() で文字列に変換してグループに対応する ASCII 文字のうしろに追加します．こうして定義した節点リストをメソッド add_nodes_from() でまとめてグラフの節点として定義し, nodelist にザイフェルト円周単位で節点リストとして保存します．一方のグラフの辺はグループ単位の処理が効率的なために個別にメソッド add_edge() で定義します．また, 辺の定義で重みを ‘weight’, 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ます．次に交差点も定義しましょう．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, ここでは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であれば-1, ザイフェルト円周を構成する辺であれば 1 を設定します. 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ます.

パッケージ NetworkX でグラフ可視化で節点配置をオプション layout で指定可能であり, レイアウトに circular_layout, shell_layout, spring_layout と special_layout の 4 通りがあます. 最初の circular は円周上に, shell は螺旋上に全ての節点を並べるために重みは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. とはいえ, 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, 絡み目の節点の良い配置が得られるかと思えば不可解な配置になったりと安定しません. この点は悩ましいことで, このメソッドでは spring, shell, circle を必要に応じて選択可能とし, 無指定であれば無難な 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もグループ分けを行いますが, このグループ分けした辺にメソッド draw_network_edges() で辺の線の太さ, 色, 形状を指定し, -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, ザイフェルト円周を構成する辺を黒の実線で表現します. なお, 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ることに注意が必要です.

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]
 Signs:[-1, -1, -1]
 SeifertCircles:[[3, 2, 1], [1, 3, 2]]
 Components: 1
 SeifertCircles:[[2, 1, 3], [2, 1, 3]]

sage: Duo_K3_1 = K3_1 + K3_11
sage: Trio_K3_1 = 3*K3_1
sage: Duo_k3_1
 GaussCodes:[[-1, 3, -2, 1, -3, 2, -4, 6, -5, 4, -6, 5]]
 Crossings:[1, 2, 3, 4, 5, 6]
 Signs:[-1, -1, -1, -1, -1, -1]
 Components: 1
 SeifertCircles:[[5, 1, 3, 2, 4, 6], [5, 4, 6], [2, 1, 3]]
 Components: 1

sage: Trio_K3_1
 GaussCodes:[[-1, 3, -2, 1, -3, 2, -4, 6, -5, 4, -6, 5, -7, 9, -8,
     7, -9, 8]]
```

```
 Crossings:[1, 2, 3, 4, 5, 6, 7, 8, 9]
 Signs:[−1, −1, −1, −1, −1, −1, −1, −1, −1]
 Components: 1
 SeifertCircles:[[8, 1, 3, 2, 4, 6, 5, 7, 9], [8, 7, 9], [5, 4, 6],
     [2, 1, 3]]
```

最初の load 文で LinkDiagram クラスの読込を行い, 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 7.11 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 7.12 に示しておきますが, このザイフェルト系は 4 個の円周で構成され, 特に交差点番号 6 を持つ円周では交差点 6 を一つだけ持つ円周が現われます. この図では交差点を置換える帯は負の交差であれば緑の点線, 正の交差であれば青の破線として表現しています. この例では三葉結び目側の交差点は全て +1 なので青の破線になりますが, 捩れの入った自明な結び目側は交差点の符号が全て-1 であるために緑の点線で表現されることになります. つぎに LinkDiagram クラスでの処理を見てみましょう:

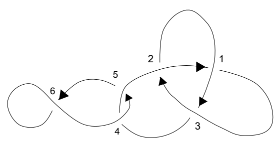


図 7.11 扱っている絡み目

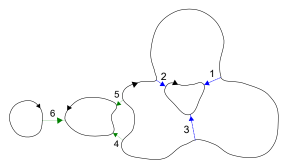


図 7.12 手書きのザイフェルト系

```
 GaussCodes:[[−1, 3, −2, 1, −3, −4, 5, 2], [4, −5, −6, 6]]
 Crossings:[1, 2, 3, 4, 5, 6]
 Signs:[1, 1, 1, −1, −1, −1]
 Components: 2
 SeifertCircles:[[6, 4, 5], [6], [4, 5, 2, 1, 3], [2, 1, 3]]
sage: g2 = d2.draw_seifert_circles(file="d2_test.png", layout="
    spring")
```

LinkDiagram クラスでは絡み目の拡張ガウス・コードをその引数として与えると絡み目に対応する LinkDiagram クラスのインスタンスを生成します. ここでインスタンス名

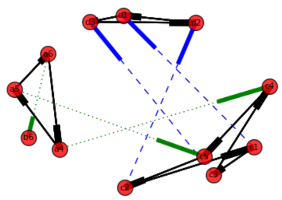


図 7.13　layout='spring' による描画

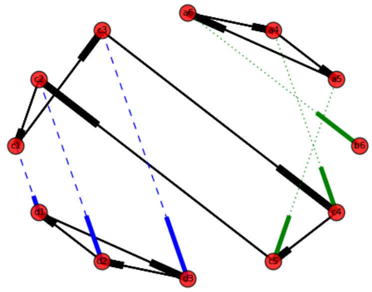


図 7.14　layout='circular' による描画

を入力するとガウス・コードやザイフェルト円周等の情報が表示されますが，この機能は RingElement クラスのメソッド__repr__() の上書きで実現しています．ザイフェルト円周の可視化はメソッド drawSeifertCircles() で行います．ここでは引数にファイル名と節点のレイアウトを指定しているために画像ファイルの出力と節点のレイアウトを行います．ここで出力した画像を図 7.13 と図 7.14 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 7.13 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 7.14 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単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．

7.9.5 語の問題

結び目/絡み目の基本群や組紐群の表示は機械的に計算することが容易ですが, 二つの結び目/絡み目の群が与えられたときに, それらか同じものかどうかの判別が難いという側面があります. これは群論で「**語の問題 (word problem)**」と呼ばれる問題です. ちなみに「**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	$\langle x_1,\ldots,x_n \mid r_1,\ldots,r_m\rangle$	$\Rightarrow$	$\langle x_1,\ldots,x_n \mid r_1,\ldots,r_m,u\rangle$
			$u\in R$
I′	$\langle x_1,\ldots,x_n \mid r_1,\ldots,r_m,u\rangle$	$\Rightarrow$	$\langle x_1,\ldots,x_n \mid r_1,\ldots,r_m\rangle$
			$u\in R$
II	$\langle x_1,\ldots,x_n \mid r_1,\ldots,r_m\rangle$	$\Rightarrow$	$\langle x_1,\ldots,x_n,y \mid r_1,\ldots,r_m,y\zeta^{-1}\rangle$
			$y\notin\{x_1,\ldots,x_n\}, u\in F$
II′	$\langle x_1,\ldots,x_n,y \mid r_1,\ldots,r_m,y\zeta^{-1}\rangle$	$\Rightarrow$	$\langle x_1,\ldots,x_n \mid r_1,\ldots,r_m\rangle$
			$y\notin\{x_1,\ldots,x_n\}, u\in F$

ここで群 F の表示を $\langle x_1,\ldots,x_n \mid r_1,\ldots,r_m\rangle$ とし, 関係子 $\{r_1,\ldots,r_m\}$ が自由群 $\langle x_1,\ldots,x_n\rangle$ 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 [28] の P.59 にある例題を使って, 二つの群の表示 $\langle x,y,z \mid xyz(yzx)^{-1}\rangle$ と $\langle x,y,a \mid xa(ax)^{-1}\rangle$ が同じ群の表示であることを確認してみましょう:

Tietze 変換の例

$$
\begin{aligned}
\langle x,y,z \mid xyz(yzx)^{-1}\rangle &\xrightarrow{\mathrm{II}} \langle x,y,z,a \mid xyz(yzx)^{-1}, a(yz)^{-1}\rangle\\
&\xrightarrow{\mathrm{I}} \langle x,y,z,a \mid xa(ax)^{-1}, a(yz)^{-1}, xyz(yzx)^{-1}\rangle\\
&\xrightarrow{\mathrm{I}'} \langle x,y,z,a \mid xa(ax)^{-1}, a(yz)^{-1}\rangle\\
&\xrightarrow{\mathrm{I}} \langle x,y,z,a \mid xa(ax)^{-1}, z(y^{-1}a)^{-1}, a(yz)^{-1}\rangle\\
&\xrightarrow{\mathrm{I}'} \langle x,y,z,a \mid xa(ax)^{-1}, z(y^{-1}a)^{-1}\rangle\\
&\xrightarrow{\mathrm{II}'} \langle x,y,a \mid xa(ax)^{-1}\rangle
\end{aligned}
$$

このように有限回の Tietze 変換の列によって二つの群の表示が同値であることが示せますが, この手法の難点は Tietze 変換の列を見付けられないことが違う群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変量が望まれる理由になります.

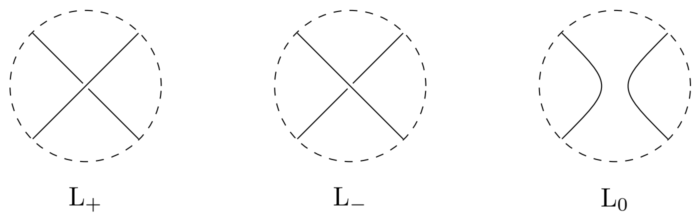
7.10　多項式不変量

このように群の表示から絡み目の同値性を判断することは簡単ではありません．そこで多少の情報が落ちても使い勝手の良い数学的対象を使って絡み目の分類を試みることになります．簡単なものに成分数や交差点数がありますが，これらは非力です．現実的には絡み目から多項式を求め，それらの多項式を使って判断します．これら多項式の計算方法は補空間の基本群を直接使う代数的な方法，ザイフェルト曲面から求める幾何学的な方法がありますが，「**スケイン関係式**」と呼ばれる正則射影図の一つの交差点での交差状況を変更することで得られた多項式間の関係式から求める手法があり，以降，このスケイン関係式を使った多項式の計算方法について述べます．

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:

三種類の交差



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)**」[*1]と呼びます．この関係式から得られる多項式は負の冪を許容する「**ローラン多項式 (Laurent polynomial**」です．そして，ライデマイスター移動で不変で，負の冪が出ないように，最小次数の項が定数項になるように正規化したローラン多項式をスケイン多項式と呼びます．スケイン多項式の代表には以下の多項式が挙げられます:

スケイン多項式

ジョーンズ多項式:	$t^{-1}V_{L_+}(t) - tV_{L_-}(t) = (t^{\frac{1}{2}} - t^{-\frac{1}{2}})V_{L_0}(t)$
コンウエイ多項式:	$\nabla_{L_+}(z) + \nabla_{L_-}(z) = z\nabla_{L_0}(z)$
アレキサンダー多項式:	$\Delta_{L_+}(t) + \Delta_{L_-}(t) = (t^{\frac{1}{2}} - t^{-\frac{1}{2}})\Delta_{L_0}(t)$
ホンフリー多項式:	$xP_{L_+}(x,t) - tP_{L_-}(x,t) = P_{L_0}(x,t)$

[*1] skein は縦糸の意味です．

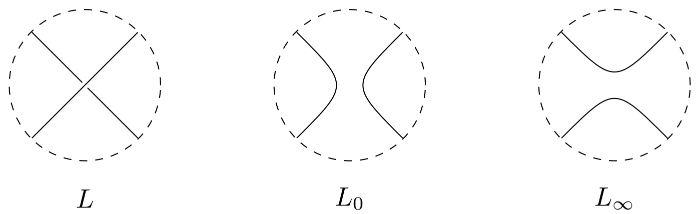
スケイン関係を用いて計算することから, 絡み目 L_1, L_2 の連結和 $L_1 + L_2$ のスケイン多項式 ww は L_1 と L_2 のスケイン多項式の積になることが判ります. そして, 自明な結び目なら 1 になりますが, 逆に 1 に等しいときに自明な結び目になるかどうかは別問題です. また, 変数の入替で互いの多項式に移り合えます. たとえば, コンウェイ多項式とアレキサンダー多項式は $z \leftrightarrow t^{\frac{1}{2}} - t^{-\frac{1}{2}}$ で互いに移り合う性質があります.

■ジョーンズ多項式:　作用素環の研究から得られた多項式で, やがてカウフマンのブラケット多項式で表現され, それからコンウエ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, 2018 年の時点でも非自明な結び目で多項式が 1 になるものが発見されていません. なお, SageMath の BraidGroup パッケージは, その関係からオプションの指定がなければカウフマンのブラケット多項式の流儀で表示され, オプションの指定があればスケイン形式から得られる多項式を出力します.

■アレキサンダー多項式:　古典的な結び目多項式で, 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する方法とザイフェルト曲面から計算する方法で計算されていました. 特に結び目群の表示から求める方法では, 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の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) アレクサンダー多項式が得られます. そのために計算機でも計算し易い多項式です. 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.

■カウフマンのブラケット多項式:　上述のスケイン関係式と別の交差関係を用いて構築されます. この多項式の計算で絡み目/結び目に向きを入れる必要はありません:

ブラケット多項式で用いる交差

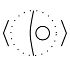
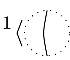


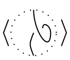
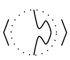
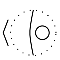
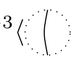
カウフマンのブラケット多項式は次の性質を充します:

カウフマンのブラケット多項式の性質

1. $\langle \mathrm{O} \rangle = 1$
2. $\langle L \sqcup \mathrm{O} \rangle = (-A^2 - A^{-2})\langle L \rangle$
3. $\langle L \rangle = A\langle L_0 \rangle + A^{-1}\langle L_\infty \rangle$

カウフマンのブラケット多項式はライデマイスター移動の TYPE II と TYPE III に関して変化がありませんが, TYPE I の移動については $A^{\pm 3}$ だけ違いが生じます. 実際に TYPE I の計算をしてみましょう:

$\langle$  $\rangle = A\langle$  $\rangle + A^{-1}\langle$  $\rangle = A(-A^2 - A^{-2})\langle$  $\rangle + A^{-1}\langle$  $\rangle = -A^3\langle$  $\rangle$

$\langle$  $\rangle = A\langle$  $\rangle + A^{-1}\langle$  $\rangle = A\langle$  $\rangle + A^{-1}(-A^2 - A^{-2})\langle$  $\rangle = -A^{-3}\langle$  $\rangle$

ここで  の交差点の符号が 1,  の交差点の符号が -1 と冪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:

$$J(L) \overset{\mathrm{Def.}}{=} (-A^{-3})^{-w(\mathcal{D}_L)}\langle \mathcal{D}_L \rangle$$

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

カウフマンのブラケット多項式の計算は機械的です. 実際, L から L_o, L_∞ への移行で交差が一つ解消され, こ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 $4 \times 2 = 8$ 個の正則射影図が得られ, 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

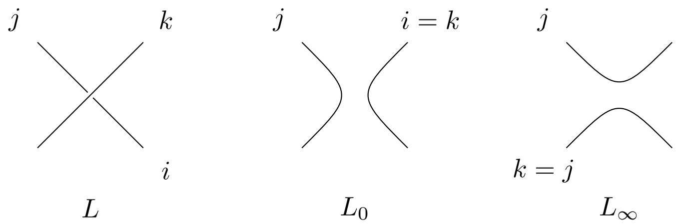
れるために交差点が n 個の正則射影図であれば n 段目で 2^n 個の互いに交差しない自明な結び目を成分とする絡み目の正則射影図が得られます．これらの自明な結び目をそれぞれ $-A^2-A^{-2}$ で置き換え，その段に至るまでの経路から A と A^{-1} を乗じて総和を計算すれば多項式が計算できます．このようにスケイン関係よりも非常に明瞭で機械的な処理ができます．

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み，さらに前述の LinkDiagram クラスのザイフェルト系を求めるメソッドが交差点の解消で道の向きを変更しないときの処理に対応しています．あとは残りの処理を構築すれば交差点の解消で生成される絡み目/結び目のガウス・コードが構築できるます．そこで，この多項式を計算するプログラムを LinkDiagram クラスのメソッドに定義しましょう．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ます．

そこで，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を「**Diagram**」, 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副産物で基本は Diagram 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．この交差点の解消で結び目が絡み目で置換えられる可能性があります．この状況を次の図で説明しましょう:

交差点の解消に伴う道の付け替え



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 を通過することがありません．

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絡み目の別成分であれば一つの成分に融合される処理になります. 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, このときに交差点 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, 右上から下道 a_i を通って交差点 i に戻るために L_0 の処理では左右に成分が分離します. ところが L_∞ の処理では a_i を通って上道 a_k の下半分を逆走して, つぎに下道 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 が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 で成分の変化は生じないものの L_∞ で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, 交差点の解消で上道を逆走することがあるために関連する交差点で符号の反転が生じます.

そこで, ここでは函数を交差点の状況に応じて二つに分けます. 一つは交差点が同一成分の道による場合で, もう一つが別成分との交差によって生じる交差点の場合の処理です. 上道と下道の融合が生じるために後者では成分が必ず一つ減りますが, 前者は成分が変化しないものと一つ増えるものの二種類があります. ここでは前者の同一成分の道による交差点の解消を行う函数 crossing_change_a() を示します:

```
def crossing_change_a(connectedDiagram):
    [GaussCodes, Crossings] = connectedDiagram
    GaussCode = GaussCodes[0]
    n = Crossings[0]
    s = sign(n)
    i = abs(n)
    sa = Crossings[1:]
    sb = copy(sa)
    x = []
    p0 = GaussCode.index(-i)
    p1 = GaussCode.index(i)
    "splitting"
    if p1>p0:
        a1 = GaussCode[p0+1:p1]
        a2 = GaussCode[p1+1:] + GaussCode[:p0]
    else:
        a1 = GaussCode[p0+1:] + GaussCode[:p1]
        a2 = GaussCode[p1+1:p0]
    "fusion"
    x = a1[::-1]
    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 = [abs(i0) for i0 in GaussCode]
    ordr = [abs(i0) for i0 in 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 toggke_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]
    sb = toggle_crossings(sb, x)
    pb =[[CodeA[0:p0] + x + CodeA[p0+1::]], sb]
    if s==1:
        z = [pa, pb]
    else:
        z = [pb, pa]
    return(z)
```

つぎにこれらを統括する交差の解消を行う函数 crossing_change() を示します．この函数では交差点がある一つの成分にあるかどうかは正則射影図を表現する対象 LinkDiagram の正規ガウス・コードに符号が異なった番号が含まれるかどうかで判別可能です．一つの正規ガウス・コードに正負の番号が含まれていれば函数 crossing_change_a() へ，正負のどちらか一方しか含まれないときは函数 crossing_change_b() で処理を行うようにします:

```
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ができます．そして, 函数 crossing_changes_a() と crossing_changes_b() が出力する Diagrams の第一成分が A を乗ずるものに, 第二成分が A^{-1} を乗ずるものに対応します．これは樹形図で左側に L_0, 右側に L_∞ を置く表記に意図的に合わせたためです．さらに Diagram のリスト Diagrams の添字を二進数で表示すると 添字 0 の箇所で A を乗じ, 添字 1 の箇所で A^{-1} を乗じることに対応していることが分かります．そして, 自明な結び目では交差点が存在しないために拡張, 正規ともに自明な結び目のガウス・コードは空リスト '[ ]' が対応します．このことから最後に函数 crossing_change() で得られたリスト Diagrams は空リストのみで構成され, 各 Diagram の添字を二進数表示したものが A と A^{-1} の積の情報で, 各 Diagram のガウス・コードの空リストの総数から 1 を減じたものが $-A^2 - A^{-2}$ の冪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 = []
    KBTK = []
    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
            KBTK.append((-A**2-A**(-2))**j)
        n = len(KBTK)
        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 + KBTK[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, 函数 range() が生成するリストの成分は Python の int 型であるため, そのままでは利用できません. 同一表示で型が異なっているのも厄介ですが, ここでは Integer() で型を int 型から Integer 型に変換して処理を行っています. SageMath では整数の利用でこのようなことが生じ易いので注意が必要です.

計算例を以下に示しておきましょう:

```
sage: K3_1 = LinkDiagram([[-1,3,-2,1,3,2]])
sage: Duo_K3_1 = K3_1 + K3_1
sage: Trio = K3_1 + K3_1 + K3_1
```

```
sage: Ans = map(factor ,map(kauffman_bracket , [K3_1, Duo_K3_1,
     Trio_K3_1]))
sage: for i in Ans:
....:       print(i)
....:
-(A^12 + A^4 - 1)/A^7
(A^12 + A^4 - 1)^2/A^14
-(A^12 + A^4 - 1)^3/A^21
```

と, ガウス・コードと交差点符号情報だけで結び目/絡み目に関連する多項式が計算できます. この計算では演算子の結合律が用いられており, 連結和によって多項式が連結和の成分の積になっていることが確認できます.

なお, この計算では交差点が n 個あれば実に $\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$ 個の絡み目が現われます. 交差点数が 3 個の三葉結び目でさえ 15 個の絡み目が現われるため, これらのデータをリストで保管することは賢明なことではありません. だから, ここでの函数も局所変数に一時的に蓄える以上のことはしていません. そこで SageMath に最初から包含されている RDBM の SQLite3 を利用しましょう. SQLite3 は軽量で, その利用のための前準備が不要の RDBM です. だからこそ, この計算処理で生じるデータを蓄えることに適しています. ここでは二つの表を使うことにしましょう. 一つは絡み目の名前, 絡み目の交差点数, 計算した絡み目のガウス・コードと交差点の情報, それと計算したカウフマンのブラケット多項式を属性とする表, もう一つの表をロルフセンの結び目表の名前, 消去した交差点番号, 樹形図での絡み目の位置, 絡み目のガウス・コードと交差点の情報を属性とする表とします. ここで前者の表が計算結果, 後者の表は計算過程に関するものになります. では SQLite3 で最初に DB と上記の表をあらかじめ生成しておきましょう. ここで SQLite3 を Python から利用するためのモジュール sqlite3 を読み込んでおく必要がありますが, SageMath にはパッケージに包含されているので import 文で読込むだけです.

それからデータベースを開いて表を生成する函数 connect_db() を次で定義します:

```
def connect_db(DB, tables):
    if len(tables)>0:
        cursor = sqlite3.connect(DB)
        sql = "SELECT * FROM sqlite_master WHERE TYPE=='table'"
        csr1 = cursor.execute(sql)
        ans = flatten(csr1.fetchall())
        for i in tables:
            tname = unicode(i[0])
            if not(tname in ans) and len(i)>2:
                sql0 = "CREATE TABLE " + tname + " "
                sql1 = "(" + unicode(i[1])
                for k in i[2:]:
```

```
                sql1 = sql1 + "," + unicode(k)
            sql1 = sql1 + ")"
            cursor.execute(sql0 + sql1)
        cursor.commit()
        return(1)
    else:
        return(0)
```

この函数では指定された DB に接続し, SQLIte3 の DB 単位に唯一作成される表 sqlite_master から表を検索し, リスト tables に同一名の表が無いか検索し, 表が存在しなければ表を生成する処理を行います. なお, cursor オブジェクト情報を一旦蓄え, メソッド commit() を実行することで初めて処理が確定されます. この commit を忘れていると cursor オブジェクトをメソッド close() で閉じた途端に情報が消えてしまうので注意が必要です.

```
def insert_diagrams2table(DBName, TableName, LinkName, Stage,
    Diagrams):
    cursor = sqlite3.connect(DBName)
    sql = "insert into " + TableName + " values (?, ?, ?, ?, ?)"
    m = 0
    for i in Diagrams:
        Position = number2position(m, Stage)
        m = m + 1
        GaussCodes = str(i[0])
        Crossings = str(i[1])
        cursor.execute(sql,(LinkName, int(Stage), Position,
            GaussCodes, Crossings))
    cursor.commit()
    cursor.close()

def insert_kauffman_bracket2table(DBName, TableName, LinkName,
    Diagrams, Polynomial):
    cursor = sqlite3.connect(DBName)
    sql = "insert into " + TableName + " values (?,?,?,?,?)"
    GaussCodes = str(Diagrams[0])
    Crossings = str(Diagrams[1])
    CrossingNumber = int(len(Diagrams[1]))
    KBP = str(Polynomial)
    cursor.execute(sql, (LinkName, CrossingNumber, GaussCodes,
        Crossings, KBP))
    cursor.commit()
    cursor.close()
```

まず, 位置を定める函数が number2position() です．この函数は引数として Diagrams の位置と交差点を消去する段階にそれぞれ対応する int 型の整数を取ります．この函数は位置情報として Diagrams での絡み目の情報の位置を 2 進数表記した文字列として返し, そのときの表示桁数を交差点を消去する段階を表現する数値が対応します．たとえば三葉結び目の段階は '0' で一つの交差点の消去を行うと段階は '1'．このときの L_0 が '0', L_∞ が '1' になり, 次の段階 '2' では L_0 派生のものが '00', '01', L_∞ 派生のものが '01' と '11' になるというあんばいです．この位置の情報が A と A^{-1} の積に対応します．実際, '0' が A, '1' が A^{-1} に対応します．この処理は函数 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()
sage: bf1 = cur1.execute("select * from diagrams")
sage: ans=bf1.fetchall()
....: for i in ans:
....:     print(i)
....:
(u'Trefoil', 0, u'', u'[[-1, 3, -2, 1, -3, 2]]', u'[1, 3, 2]')
(u'Trefoil', 1, u'0', u'[[3, -2], [-3, 2]]', u'[3, 2]')
(u'Trefoil', 1, u'1', u'[[-3, 2, -2, 3]]', u'[-3, -2]')
(u'Trefoil', 2, u'00', u'[[-2, 2]]', u'[2]')
(u'Trefoil', 2, u'01', u'[[-2, 2]]', u'[-2]')
(u'Trefoil', 2, u'10', u'[[-2, 2]]', u'[-2]')
(u'Trefoil', 2, u'11', u'[[2, -2], []]', u'[-2]')
(u'Trefoil', 3, u'000', u'[[], []]', u'[]')
(u'Trefoil', 3, u'001', u'[[]]', u'[]')
(u'Trefoil', 3, u'010', u'[[]]', u'[]')
(u'Trefoil', 3, u'011', u'[[], []]', u'[]')
(u'Trefoil', 3, u'100', u'[[]]', u'[]')
(u'Trefoil', 3, u'101', u'[[], []]', u'[]')
(u'Trefoil', 3, u'110', u'[[], []]', u'[]')
(u'Trefoil', 3, u'111', u'[[], [], []]', u'[]')
```

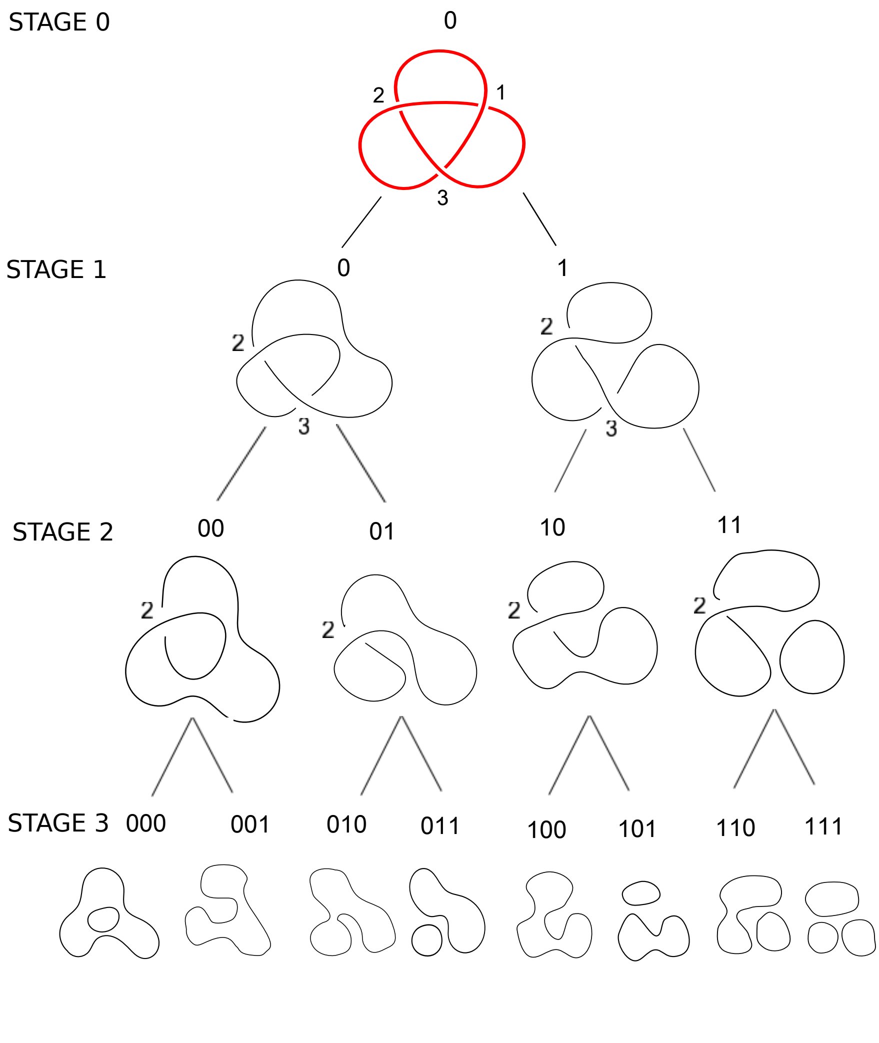


図 7.15　三葉結び目での処理

```
sage: bf1 = cur1.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 = cur1.execute("select * from kauffman_bracket_table")
sage: ans2 = bf2.fetchall()
sage: for i in ans2:
....:     print(i)
 (u'Trefoil', 2, u'[[[], []], []]', u'[[[], []]', u'-A^5 - 1/A^3 +
      1/A^7')
```

と, このようにデータの検索が行えるのです．ここでやっていることの詳細を解説すると, kauffman_bracket() では DB と LinkName にデータベースの情報と絡み目の名前を指定するとカウフマンのブラケットをデータベースに書き込みます．ここで kauffman_bracket() 関数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() 関数ではテーブルを初期化することはなく, 続々と書き込むだけです．

第 8 章

ベイス推定について

8.1　SageMath での統計処理

従来の数式処理システムは代数学的計算を重視していても, 数値計算に関しては独自の任意精度の数値計算を導入したものが多く, このことが仇になって, 数値計算が不得手なものが大半でした.

ところで SageMath は任意精度の数値計算を GNU gsl に任せ, 通常の倍精度の浮動小数点数計算では NumPy 等も利用するために, Python 環境でやれることの多くが実行可能です. もちろん, Python そのものは Julia のような高速な処理が可能な計算処理言語ではありませんが, Python の「糊言語」としての特性から, 外部のライブラリやパッケージを駆使することで実用的な計算処理が可能となっています. とは言え, SageMath の統計処理向けのパッケージは Probability モジュールが存在する程度です. このモジュールは GNU gsl ライブラリを利用し, 対象とする確率分布を RealDistribution, SphericalDistribution と GeneralDiscreteDistribution の三群に分類して提供している程度で, その他の細かな統計的手法のモジュールはありません. このこともあって, SageMath 上に各種の Python パッケージを導入して Python として利用することが現実的です.

このような使い方をする上で注意すべきこととして, 特に重要なことは SageMath の数体系と Python の数体系に違いがあることで, そのために SageMath で得られた数値が Python でそのまま使えないことも生じます. この状況は SageMath で数式処理を行いつつ, 数値計算では純粋な Python として SageMath を使っているときに顕著です. とは言え, 最初から SageMath を徹底して Python 環境として使い倒すのであれば, そのような注意はあまり必要はないでしょう. また, SageMath には GNU R とのインターフェィスもあるために. そちらを統計処理で専ら用いることもできます. ただし, この 文書では SageMath を Python 環境としても使うことに主眼を置いて解説します. その理由は, 2010 年頃までは統計処理は GNU R の独壇場でしたが, 2010 年以降になると徐々に Python が優勢になり, 現在, Python は一般的に統計処理, 特に AI 向けの言語として認

知されている現状があります．Python 自体はさほど処理速度が他の言語と比較して優れている訳でもありませんが, その平易な言語仕様に加えて糊言語としての優れた特性から, 様々なライブラリを統合していった結果, 統計解析や AI に必要とされる環境が構築できた点が大きいでしょう．

ところで, SageMath は Python を処理言語のベースとしており, しかも, 通常の Python よりも数値計算に関しては通常の Python よりも高速な処理が行えるような工夫がされています．そのような理由で, 統計分析で数式処理は SageMath, 数値計算は GNU GSL を用いたものが良ければ SageMath の数値計算, 膨大な Python のライブラリやノウハウを使った方が楽であるのなら, SageMath を Python として使い倒すという方向で解説したいと思う理由です．

この章では SageMath を使ったベイス推定の一例を示すことを目的としています．そのために確率とベイズ推定の初歩を最初にまとめておきます．

8.2 確率について

8.2.1 標本空間から構成される代数的空間

ここで最初に確率の概要について解説しておきます．確率はある事象が発生し得る可能性を評価, 比較するために数値的に表現したものです．この確率をある程度, きちんと定義するためには幾つかの事柄について説明しておく必要があります．

ここで考察する事象は, 小細工付きの骰子を使ったいかさま賭博のような, 何らかの意図に沿って偏った結果を出すような試行ではなく, 試行する側が得られた結果からどのような分析, モデルの構築を行うかは別として, その意図とは無関係な結果が得られる試行, すなわち,「**無作為試行 (random sample)**」による結果とします．このときに無作為試行で得られる個々の結果を「**標本 (sample)**」, 標本全体で構成される集合を「**標本空間 (sample space)**」といい, ここでは標本空間を「Ω」と表記します．そして,「**事象 (event)**」, または「**確率事象 (probability event)**」は標本を元とする集合であり, この事象に関して確率が計算されます．つまり, 事象は確率で語られるべき対象です．

ここで標本と事象の例を挙げておきましょう．まず, 無作為試行を「コインを二回投げる」とすれば, 標本は「裏裏」,「 裏表」,「表裏」,「表表」の 4 つあり, 標本空間 Ω は {裏裏, 裏表, 表裏, 表表} と 4 成分の集合になります．これに対して事象は標本空間の部分集合です．たとえば, コインを二度投げるときに「**1 回目が表になる**」という事象は{表裏, 表表}の 2 成分の標本空間 Ω の部分集合です．さらに「**一度でも表が出る**」という事象は {表裏, 裏表, 表表}の 3 成分の標本空間 Ω の部分集合になります．このコイン投げのように取り得る値が 2 値であり, 各試行が互いに影響することがない独立な試行であるときに, この試行で得られる列を「**ベルヌーイ試行列 (bernoulli trial)**」といいます．

また, 特殊な事象として「**何事も生じない事象**」として「**空事象 (empty event)**」を事象に含めます. このように事象全体は標本空間という集合の部分集合で構成された集合, つまり,「**集合族 (family of sets)**」であり, この全ての事象の集まりを「**事象族 (event family)**」といい,「$\mathcal{F}$」と表記します.

ところで, 標本空間 Ω の事象 A が生じ得ないことに対応する補集合 $A^c = \Omega \setminus A$ も事象であること, そして, 二つの事象 A, B の集合和 $A \cup B$ も事象であるべきことは極めて自然な要請です. そこで, 事象 $A \in \mathcal{F}$ の補集合 A^c を事象 A の「**余事象 (complementary event)**」といい, 事象 $A, B \in \mathcal{F}$ の和集合 $A \cup B$ を「**和事象 (sum event)**」といいます. この和事象 $A \cup B$ は事象 A か事象 B の何れかが生じる事象, すなわち, 事象 A が生じる場合, 事象 B が生じる場合, それと事象 A と事象 B が同時に生じる場合に対応します. さらに $A, B \in \mathcal{F}$ に対して $A \cup B \in \mathcal{F}$ であることは, 集合和 $\cup$ という事象族 $\mathcal{F}$ における二項演算が集合族 $\mathcal{F}$ で閉じていることを意味しています. これらの要請から新たに f-集合体という概念を導入します. すなわち,「**f-集合体 (f-field)**」, あるいは「**有限加法的集合族**」は以下に性質を充す集合族のことです:

f-集合体

集合 X の集合族 $\mathcal{F}$ が f-集合体であるとは以下の条件を充すときである:

$\mathcal{F}1$: $\emptyset \in \mathcal{F}, X \in \mathcal{F}$
$\mathcal{F}2$: $A \in \mathcal{F}$ であれば $A^c \in \mathcal{F}$
$\mathcal{F}3$: $A, B \in \mathcal{F}$ のときに $A \cup B \in \mathcal{F}$

ここで f-集合体である事象族 $\mathcal{F}$ に含まれる有限個の事象列 $\{A_i\}_{i \in \{1,2,\dots,n\}}$ に対しては, それらの和事象について $\displaystyle\bigcup_{i=1}^{n} A_i \in \mathcal{F}$ であることが条件 $\mathcal{F}3$ を繰り返して適用することで判ります. また, f-集合体では集合積 $\cap$ についても演算が閉じていることが判ります. 実際, $A, B \in \mathcal{F}$ であれば条件 $\mathcal{F}1$, $\mathcal{F}2$ から $A^c \cup B^c \in \mathcal{F}$ であり, さらに条件 $\mathcal{F}1$ から $(A^c \cup B^c)^c \in \mathcal{F}$ となります. ここでド・モルガンの法則[*1] によって $(A^c \cup B^c)^c = A \cap B$, 以上から $A, B \in \mathcal{F}$ であれば $A \cap B \in \mathcal{F}$, また, 積事象 $\displaystyle\bigcap_{i=1}^{n} A_i$ についても同様にド・モルガンの法則を使って $\displaystyle\left(\bigcup_{i=1}^{n} A_i^c\right)^c$ と書けるためです. ここで二つの事象 A, B の集合積 $A \cap B$ を「**積事象 (product event)**」といいます. この事象は事象 A と事象 B が同時に生じる事象に対応します.

さらに事象 $A, B \in \mathcal{F}$ が $A \cap B = \emptyset$ のときに A, B は「**排反 (disjoint)**」であるといい, 事象 B を「**事象 A の排反事象 (exclusive event)**」といいます. そして, 互いに排

[*1] $(A \cup B)^c = A^c \cap B^c$

反な事象の列 $\{A_i\}_{i\in\{1,2,\dots,n\}}$, すなわち, $i\neq j$ のときに $A_i\cap A_j=\emptyset$ となる事象列を「**排反事象列**」といいます．なお, $\{A_i\}_{i\in\{1,2,\dots,n\}}$ が排反事象列のときに限って, その和事象 $\displaystyle\bigcup_{i=1}^{n} A_i$ を $A_1+\cdots+A_n$, あるいは $\displaystyle\sum_{i=1}^{n} A_i$ と表記します.

この f-集合体の例として, 有限集合 X の全ての部分集合を元とする集合 X の冪集合 2^X , 全体集合 Ω と空集合 $\emptyset$ で構成された集合族 $\{\Omega,\emptyset\}$ といったものを挙げておきます. また, この章で確率を考察する上で, その土台となる事象族 $\mathcal{F}$ は少なくとも f-集合体であるとします.

8.2.2　初等確率モデル

ここで有限個の成分を持つ標本空間 Ω の事象族 $\mathcal{F}$ 上で事象 $A\in\mathcal{F}$ の確率 $P(A)$ を考えてみましょう．まず, 確率を 0 から 1 の範囲に限定し, 生じ得ない事象の確率を 0, 必ず生じ得る事象に対して確率は 1 を返すものとします．さらに二つの事象 $A,B\in\mathcal{F}$ について, もしも, $P(A)>P(B)$ であれば, 事象 A の方が事象 B よりも生じ易い事象であると判断できるでしょう．また, 互いに排他的な事象 $A,B\in\mathcal{F}$ については, そのどちらかが生じる事象 $A+B(=A\cup B)$ の確率が事象 A の確率と事象 B の確率の和になるべきでしょう.

これらの要請から事象族 $\mathcal{F}$ 上の確率 P は以下の性質を充すべきです:

事象族 $\mathcal{F}$ 上の確率 P の性質

P1. 任意の $A\in\mathcal{F}$ に対して $0\leq P(A)\leq 1$. 特に $P(\emptyset)=0, P(\Omega)=1$

P2. $A,B\in\mathcal{F}$ が $A\cap B=\emptyset$ であるときに $P(A\cup B)=P(A)+P(B)$

ここで性質 P2: $P(A+B)=P(A)+P(B)$ は事象 A,B が互いに排反である場合に $A\cup B=A+B$ となることに対応し, さらには有限個の排反事象列 $\{A_i\}_{i\in\{1,2,\dots,n\}}$ に対しては性質 P2 を繰り返して使うことで $P\big(\displaystyle\sum_{i=1}^{n} A_i\big)=\sum_{i=1}^{n} P(A_i)$ であることも判ります．ちなみに, この有限個の排反事象列に対する確率の性質は「**有限加法性**」と呼ばれる性質です．さらに性質 P1, P2 と $\Omega=A\cup A^c$ であることから事象 A の余事象 A^c の確率が $P(A^c)=1-P(A)$ であることも容易に判ります．そして, 二つの事象 $A,B\in\mathcal{F}$ が $A\subset B$ であれば $B=(B\setminus A)+A$ であることから, $P(B)=P(A)+P(B\setminus A)$ が得られ, さらに確率 P の非負性から $P(A)\leq P(B)$, このことから $A\subset B$ であれば $P(A)\leq P(B)$ であることが判ります.

これら標本空間 Ω, 事象族 $\mathcal{F}$ と確率 P の組 $(\Omega,\mathcal{F},P)$ を「**初等確率モデル**」と呼びます．ここで確率モデルの例を幾つか挙げておきましょう.

■骰子の目:　最初の例は骰子を一度投げて出た目を標本とする事例の確率モデルです．まず, 標本空間 Ω は $\{1,2,3,4,5,6\}$ であり, その事象族 $\mathcal{F}$ は標本空間 $\Omega=\{1,2,3,4,5,6\}$ の全部分集合 $\mathcal{P}(\Omega)$ になります．このときに事象 A を「**1 が出る**」とすれば $A=\{1\}$ です．

次に, この骰子はどの目が出る確率も等しいとすると, 各事象が排反事象であることから $P(\Omega)=P\big(\{1,2,3,4,5,6\}\big)=\sum_{i\in\{1,2,3,4,5,6\}}P\big(\{i\}\big)=1$, 以上から i の目が出る確率 $P(\{i\})$ が $\dfrac{1}{6}$ であることが判ります．また, $A=\{1\}$ の余事象 A^c である「**1 以外の目が出る**」は $A^c=\{2,3,4,5,6\}$ であり, このことから確率 $P(A^c)$ は $\displaystyle\sum_{i\in\{2,3,4,5,6\}}P(\{i\})=\frac{5}{6}$ になります．そして, 事象 $B=\{2\}$, すなわち,「**2 の目が出る**」のとき, 事象 $A\cup B=\{1,2\}$ は「**1 か 2 のどちらかの目が出る**」という事象に対応し, このときの確率は $P\big(\{1,2\}\big)=P\big(\{1\}\big)+P\big(\{2\}\big)=2\cdot\dfrac{1}{6}=\dfrac{1}{3}$ になります．

■歪んだコイン投げ: つぎは歪んだコインを投げて裏が出るか, 表が出るか試行を行ったときの確率モデルとします．　ここでコインの表が出る確率を p $(1\leq p\leq 1)$ とすると裏が出る確率は $q=1-p$ となります．　ここでの試行は, このコインを使って三回投げることにします．　すると標本空間 Ω は $\{$表表表, 表表裏, 表裏表, 裏表表, 表裏裏, 裏表裏, 裏裏表, 裏裏裏$\}$ の $2^3=8$ 成分の集合になります．　ここで「**表が一度でも出る**」という事象 A は全てが裏の場合を除く $\{$表表表, 表表裏, 表裏表, 裏表表, 表裏裏, 裏表裏, 裏裏表$\}$ であり, その確率 $P(A)$ は $P\big(\{\text{表表表}\}\big)+P\big(\{\text{表表裏}\}\big)+P\big(\{\text{表裏表}\}\big)+P\big(\{\text{裏表表}\}\big)+P\big(\{\text{表裏裏}\}\big)+P\big(\{\text{裏表裏}\}\big)+P\big(\{\text{裏裏表}\}\big)$ で与えられます．そして, 表しか出ない確率は p^3, 表が二度だけ出る確率は $3p^2q$, 表が一度だけ出る確率は $3pq^2$ となることも判ります．これらのことから表が一度でも出る確率 $P(A)$ は $p^3+3p^2q+3pq^2=(p+q)^3-q^3=1-q^3$ で与えられることが判ります．また, 事象 A の余事象が A^c であり, この事象は「**表が一度も出ない事象**」であることから $P(A^c)=q^3$, $P(A)=1-P(A^c)$ より同様の結果が得られます．次に「表」に 1,「裏」に 0 を対応させます．すると, 標本空間 Ω は $\{(1,1,1),(1,1,0),(1,0,1),(0,1,1),(1,0,0),(0,1,0),(0,0,1),(0,0,0)\}$ と 3 次元空間に配置された立方体の頂点に対応し, 標本空間 Ω から $\mathbb{R}^3$ への写像 X が考えられます．

■自然数の出る確率:　ここまでの例は全て標本数が有限の例でしたが無限の場合も考えられます．たとえば, 標本空間を自然数 $\mathbb{N}$, 事象族 $\mathcal{F}$ を $\mathbb{N}$ の部分集合全体 $\mathfrak{P}(\mathbb{N})$, そして, 自然数 n が出る確率を $\dfrac{1}{2^{n+1}}$ とします．このときに $P(\Omega)$ は

$$P(\Omega)=\sum_{i=0}^{\infty}\frac{1}{2^{n+1}}=\frac{1}{2}\cdot\sum_{i=0}^{\infty}\frac{1}{2^n}=\frac{1}{2}\cdot\lim_{n\to\infty}\left\{\frac{1-\left(\frac{1}{2}\right)^n}{1-\frac{1}{2}}\right\}=1$$

となるために確率モデルになります．ただし, 標本空間 Ω は自然数 $\mathbb{N}$ であり, いままで考察していた有限集合と異なり可算無限の集合です．

8.3　条件付確率と独立な確率

事象 A を前提として, あるいは事象 A の何らかの情報を引き継いで事象 B が生じる確率を「**条件 A の下での事象 B の条件付確率**」といい, $P(B|A)$, または $P_A(B)$ と表記します．この条件付確率については以下の式が成立します:

$$P(A \cap B) = P(A)P(B|A)$$

この式から条件 A の下での事象 B の条件付確率 $P(B|A)$ は確率 $P(A)$ と $P(A \cap B)$ を使って $P(B|A) = \dfrac{P(A \cap B)}{P(A)}$ で計算できます．また, $P(A \cap B) = P(A)P(B)$ でもあり, この場合の $P_B(A)$ は「**原因の確率**」に対応することになります．つまり, 事象 A を「ある病気に対応したワクチンを接種する人」, 事象 B を「その病気に罹患しない人」とすると, 事象 $A \cap B$ は「ワクチンを接種していて, かつ, その病気に罹患しない人」に対応し, 確率 $P_A(B)$ は「ワクチンを接種した人が, その病気に罹患しない確率」になります．つまり, A という条件の下で B が生じる確率が $P_A(B)$ です．一方で, 確率 $P_B(A)$ は「その病気に罹患していない人で, ワクチンを接種した人の確率」となり, $P_B(A)$ は事象 B となる原因の確率に対応します．

また, 全事象 Ω が排反な事象の列 $\{E_i\}_{i \in \{1,2,\ldots\}}$ の和として書けること, すなわち, $\Omega = \displaystyle\sum_i E_i$ であるときにベイズの定理と呼ばれる次の式が成立します:

ベイズ (Bayes) の定理

事象族 $\mathcal{F}$ の事象 A と互いに排反な事象の列 $\{E_i\}_{i \in \{1,2,\ldots\}}$ について, $\Omega = \sum_i E_i$ であるときに以下が成立する:

$$P_A(E_i) = \frac{P(E_i)P_{E_i}(A)}{\sum_j P(E_j)P_{E_j}(A)}$$

実際, 右辺の分母は $\displaystyle\sum_j P(E_j)P_{E_j}(A) = \sum_j P(E_j) \cdot \frac{P(E_j \cap A)}{P(E_j)} = \sum_j P(E_j \cap A)$, ここで, $\Omega = \displaystyle\sum_j E_j$ であることから $\displaystyle\sum_j P(E_j \cap A) = P(\Omega \cap A) = P(A)$, 右辺の分子は条件付確率の性質から $P(E_i)P_{E_i}(A) = P(A \cap E_i)$, ところで, $P(A)P_A(E_i) = P(A \cap E_i)$ より, $P(A)P_A P(E_i) = P(E_i)P_{E_i}(A)$, したがって, $P_A(E_i) = \dfrac{P(E_i)P_{E_i}(A)}{P(A)} = \dfrac{P(E_i)P_{E_i}(A)}{\sum_j P(E_j)P_{E_j}(A)}$ が得られます．

この定理によって $P(A)$ と条件 $P_{E_i}(A)$ の双方が既知であれば, 原因の確率 $P_A(E_i)$ が計算できることを意味します.

次に事象 $A, B \in \mathcal{F}$ が「**独立 (independent)**」であるとは, 互いの事象に影響を及ぼさないことですが, 確率 P を用いて $P(A \cup B) = P(A)P(B)$ を充すときと定義します. 事象 A, B が独立であれば, 条件付確率 $P_B(A), P_A(B)$ はそれぞれ $P(A), P(B)$ に等しくなります.

8.3.1　σ-集合体と可測函数, ルベーグ積分初歩

集合族 $\mathcal{F}$ が f-集合体であれば $\displaystyle\bigcup_{i=1}^{n} A_i \in \mathcal{F}$ は保証されますが, 集合族 $\mathcal{F}$ の濃度が可算無限となったときに $\displaystyle\bigcup_{i=1}^{\infty} A_i \in \mathcal{F}$ であることは保証されません. 同様に確率 P に関して, 事象族 $\mathcal{F}$ の有限個の排反集合列 $\{A_i\}_{i\in\{1,2,\ldots,n\}}$ に対して有限加法性: $P\bigl(\displaystyle\sum_{i=1}^{n} A_i\bigr) = \displaystyle\sum_{i=1}^{n} P(A_i)$ が保証されますが, $n \to \infty$ のときに $P\bigl(\displaystyle\sum_{i=1}^{\infty} A_i\bigr) = \displaystyle\sum_{i=1}^{\infty} P(A_i)$ であることも保証されません.

このように f-集合体 $\mathcal{F}$ 上で定めた初等的確率モデルでは, 可算無限個の排他事象列の集合和が事象族 $\mathcal{F}$ に属するのかどうか, また属したとしても, 今度は各事象の確率の和になるかどうかが判りません. そこで可算無限個の和集合も集合族に属し, さらには排他事象列に対しても加法性が保証されるようにします. この f-集合体の加法性を可算無限, すなち, $n \to \infty$ に拡大した性質は「**σ-加法的**」と呼ばれる性質で, この性質を追加した以下の性質を持つ f-集合体を「**σ-集合体 (σ-field**」と呼びます:

σ-集合体

集合 X の f-集合 $\mathcal{F}$ が σ-集合体であるとは以下の条件を充すときである:

$\mathcal{B}1$: $\emptyset \in \mathcal{F}, X \in \mathcal{F}$
$\mathcal{B}2$: $A \in \mathcal{F}$ であれば $A^c \in \mathcal{F}$
$\mathcal{B}3$: $A_i \in \mathcal{F}(i \in \{1, 2, \ldots\})$ のときに $\displaystyle\bigcup_{i=1}^{\infty} A_i \in \mathcal{F}$

この性質 $\mathcal{B}3$ によって, σ-集合体であれば可算無限個の集合和については有限の場合と同様に考えてよいことになります. ただし, 何でもかんでも集合和で取り込んだものについては保証がありません.

この σ-集合体の例としては, 集合 X の部分集合全体 $\mathfrak{P}(X)$ があります. $\mathfrak{P}(X)$ は**冪集合**」と呼ばれ, 2^X とも表記されます. この 冪集合 2^X は X の σ-集合体の中で (包含関係について) 最大の σ-集合体になります. 逆に集合 X を含む最小の σ-集合体は $\{X, \emptyset\}$ で与

えられます．さらに X の部分集合 A を含む最小の σ-集合体は $\{X, \emptyset, A, A^C\}$ で与えられます．ここで表記として f-集合体 $\mathcal{S}$ を含む最小の σ-集合体を $\sigma[\mathcal{S}]$ と表記することにします．ここで, 閉集合全体の集合を $\mathcal{C}$ と表記しましょう．すると, 閉集合は開集合の補集合であるために $\sigma[\mathcal{O}] = \sigma[\mathcal{C}]$ となります.

さて, 最も重要な σ-集合族として「**ボレル (Borel) 集合体**」があります．このボレル集合体は $\mathbb{R}^d$ の全ての開集合[*2]の集合 $\mathcal{O}$ を含む σ-集合体の中で最小のもの, すなわち, $\sigma[\mathcal{O}]$ として定義され, これを $\mathcal{B}(\mathbb{R}^d)$ と表記します．このボレル集合族 $\mathcal{B}(\mathbb{R}^d)$ は d 次元の半開区間 $(a_i, b_1] \times \cdots \times (a_d, b_d] \subset \mathbb{R}^d$ 全体の集合族 $\mathcal{J}^d$ からも構成可能です．このことは $\sigma[\mathcal{J}^d] \subset \sigma[\mathcal{O}]$ と $\sigma[\mathcal{J}^d] \supset \sigma[\mathcal{O}]$ であることが示せれば十分です．まず, $J = (a_1, b_1] \times \cdots \times (a_d, b_d]$ に対して $J_n = \left(a_1, b_1 + \dfrac{1}{n}\right) \times \cdots \times \left(a_d, b_d + \dfrac{1}{n}\right)$ とすると, $J = \displaystyle\bigcap_{n=1}^{\infty} J_n \in \mathcal{B}(\mathbb{R}^d)$ となるために　$\sigma[\mathcal{J}^d] \subset \mathcal{B}(\mathbb{R}^d)$　が判ります．逆の $\sigma[\mathcal{J}^d] \supset \sigma[\mathcal{O}]$ を示すためには任意の開集合 $U \in \mathcal{O}$ がある半開区間の列 $\{J_k\}_{k\in\{1,2,\ldots\}}$ を使って $U = \displaystyle\bigcup_{k=1}^{\infty} J_k$ であることが示せられれば十分です．そこで, 半開区間の集合 $\mathcal{J}_r = \{(a_1, b_1] \times \cdots \times (a_d, b_d] : a_i, b_i \in \mathbb{Q}(i \in \mathbb{N})\}$ を考えます．この $\mathcal{J}_r$ の濃度は有理数と同じ濃度であるために可算無限個 $(= \aleph_0)$ になります．ここで開集合 U が与えられたときに, その任意の元 x に対して $x \in J^{(x)} \subset U$ を充す $\mathcal{J}_r$ の元が取れるために $U = \displaystyle\bigcup_{x\in G} J^{(x)}$ であることが判ります．ここで開集合 U の濃度は実数濃度 $(= \aleph)$ ですが, $\mathcal{J}_r$ の濃度は可算無限個であるために $U = \displaystyle\bigcup_{k=1}^{\infty} J_k$ となるように $\mathcal{J}_r$ の可算個列 $\{J_k\}_{k\in\{1,2,\ldots\}}$ が選べます．ここで, $\sigma[\mathcal{J}^d]$ の σ-加法性から $U \in \sigma[\mathcal{J}^d]$ となり, 以上から $\sigma[\mathcal{O}] \subset \sigma[\mathcal{J}^d]$ が得られ, これらの結果から $\sigma[\mathcal{J}^d] = \sigma[\mathcal{O}] = \mathcal{B}(\mathbb{R}^d]$ であることが判ります.

ここで紹介した σ-集合体, ボレル集合体 $\mathcal{B}(\mathbb{R}^d)$ はルベーグ (Lebesgue) 積分の基盤となります．ここでルベーグ積分がどのようなものかを述べる前に, ルベーグ積分が可能となる函数のことに触れておきます．ルベーグ積分が可能な「**可測函数 (measurable function)**」と呼ばれる以下で定義される函数です:

可測函数

$\mathcal{F}$ が $X(\subset \mathbb{R}^n)$ 上の σ-集合体であり, 函数 $f : X \longrightarrow \mathbb{R}$ が任意の $a \in \mathbb{R}$ に対して $\{x \in X : f(x) < a\} \in \mathcal{F}$ を充すときに, 函数 f のことを「**可測函数 (measurable function)**」といいます.

つぎに, ルベーグ積分に直接関係する函数が「**単純函数 (simple function)**」という函数です．この函数を定義するために X の σ-集合体 $\mathcal{F}$ の元 A に対して「**指示函数**

[*2] 開区間の n 個の積 $(x_{11}, x_{12}) \times (x_{21}, x_{22}) \times \cdots (x_{d1}, x_{d2})$ の集合和として表現できる $\mathbb{R}^d$ の集合

(indicator function)」I_A を次で定めます:

$$I_A = \begin{cases} 1, & (x \in A) \\ 0, & (x \notin A) \end{cases}$$

このように指示函数は x がその集合 A に所属するかどうかを 0 や 1 を返すことから判定できる函数です．次に集合 X を $\mathcal{F}$ の集合の列 $\{A_i\}_{i\in\{1,2,,\ldots\}}$ で切り分けます．つまり，$X = \displaystyle\bigcup_{i=1}^{\infty} A_i$ で，$i \neq j$ のときに $A_i \cap A_j = \emptyset$ となっています．このような函数を定義する上で σ-集合体 $\mathcal{F}$ は不可欠です．さて，この集合列 $\{A_i\}_{i\in\{1,2,\ldots\}}$ を使って函数 $f(x)$ を次で定めます:

$$f(x) = \sum_{i=1}^{\infty} a_i \cdot I_{A_i}(x), (-\infty < a_i < \infty)$$

このような形の有界な函数を「**単純函数 (simple function)**」といいます．この函数 $f(x)$ は値が有界 $(f(x) \in (-\infty, \infty))$ な可測函数であり，さらには任意の非負可測函数 $f(x)$ に対して，ある単純函数の列 $\{f_n\}_{n\in\{1,2,\ldots\}}$ で，任意の $x \in X$ に対しては $\{f_n(x)\}_{n\in\{1,2,\ldots\}}$ が単調非減少で，しかも，$f(x)$ に収束するものが存在します．

実際，各 $n \in \{1, 2, \ldots\}$ に対して

$$f_n = \sum_{i=1}^{n\cdot 2^n} \frac{i-1}{2^n} \cdot I_{[(i-1)\cdot 2^{-n} \leq f < i\cdot 2^{-n}]} + n \cdot I_{[f\geq n]}$$

とおきます．すると，f_{n+1} と f_n の関係は，まず，$f(x) \leq n$ の範囲では f_{n+1} の方がより細かく分割を行ってより大きな指示函数の係数になる可能性があるために $f_n \leq f_{n+1}$，さらには $|f_n - f(x)| < \dfrac{1}{2^n}$ であることも判ります．したがって，$\{f_n(x)\}$ は各 $x \in X$ で減少することのない増加列です．また，$f(x) \geq n$ であれば $f_n(x) = n$ となり，$\{f_n(x)\}_{n\in\{1,2,,\ldots\}}$ は単調非減少で，さらに $f(x)$ に収束することが判ります．このことから非負可測函数は単調函数で下側から近似ができることを意味します．

ここで，非負を外した一般の可測函数 f については函数 $\overline{f}$ を $\overline{X} = \{x \in X : f(x) \geq 0\}$ で函数 f と一致し，それ以外の領域では 0 になる函数，函数 $\underline{f}$ を $\underline{X} = \{x \in X : f(x) \leq 0\}$ で函数 f に一致し，それ以外の領域では 0 になる函数の二つの函数に分割して考えます．すると，これらの函数は可測函数で，$f = \overline{f} - \underline{f}$ となります．さらに，$\overline{f}$ は非負可測函数，$-\underline{f}$ も非負可測函数となるために，$\overline{f}$ と $\underline{f}$ の双方が単純函数を使って近似可能．このことから可測函数 f は単純函数を使って近似できることが判ります．

ここで図 8.1 に単純函数 $f_2 = \displaystyle\sum_{i=1}^{8} \frac{i-1}{4} \cdot I_{[\frac{i-1}{4} \leq f < \frac{i}{4}]} + 2 \cdot I_{[f \geq 2]}$ の構築の様子を示しておきます:

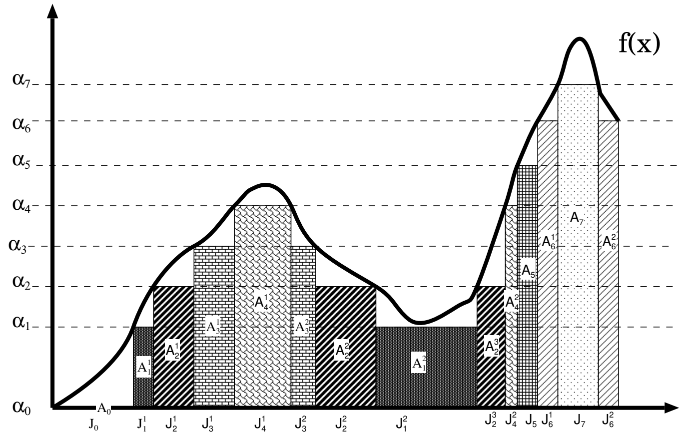


図 8.1　単純函数による函数の近似

この図で示すように単純函数の構築は Y 軸方向に函数 $f(x)$ を切って, その高さに対応する短冊に分割する処理に対応しています. つまり, $J_k \subset X = \mathbb{R}$ を指示函数 $I_{[\frac{k}{4} \leq f < \frac{k+1}{4}]}$ が 1 となる X の領域とすると, その領域上で f_2 は高さ $\alpha_k = \frac{k}{4}$ の互いに隣接しない短冊 A_k^j の集合 A_k に対応します. たとえば, 図で高さ $\alpha_2 = \frac{2}{4}$ に対応する単純函数は $\alpha_2 \cdot I_{J_2^1} + \alpha_2 \cdot I_{J_2^2} + \alpha_2 \cdot I_{J_2^3}$ で, 図に示す短冊 A_2^1, A_2^2, A_3^2 に対応します. そして, 単純函数 f_n は $n \to \infty$ で f に収束しますが, ここで $n \to \infty$ という操作は縦軸を微小に細かく分割することに対応し, それによって無数の短冊が現われることになり, この高さ α_k の短冊の集合の面積は半開区間 J_k^j の長さ l_k^j が計算できれば $\alpha_k l_k^j$ で与えられ, これらの短冊の集合の面積の和として単純函数 f_n の面積 S_n が計算できます. そして, $n \to \infty$ のときに $f_n \to f$ であることから, fn の面積 S_n の極限 $\lim_{n \to \infty} f_n$ が函数 f の面積 S に対応します. ここで短冊の底部の長さ (体積) を与える集合函数 μ がルベーグ積分で「**測度**」と呼ばれる函数です.

図 8.1 に示す単純函数の構築では, Y 座標側の高さで短冊を取っています. ちなみにリーマン積分が X 軸方向で一定の幅で短冊を取るのに対して, そのためにルベーグ積分はリーマン積分ができない函数であっても, その函数に収束する単純函数列が存在すれば良く, そこには被積分函数の連続性といった話はありません. 函数 $f()$ に収束する単純函数さえ構築できれば良いのです.

さらに $n \to \infty$ とすることから X を可算個の集合に切り刻むことになり, その切り刻んだ $\{J_i = \displaystyle\sum_{j=\{1,2,\cdots\}} J_i^j\}_{i \in \{1,2,\ldots\}}$ の和は X であるべきで, この要請は σ-集合族の σ 加

法性に対応します．さらには測度 μ に関しても $\mu\bigl(\sum_{i=1}^{\infty} A_i\bigr) = \sum_{i=1}^{n} \mu(A_i)$ であるべきです．このようにルベーグ積分で重要な測度ですが，この測度は集合から実数値を対応させる函数，すなわち，集合函数の一つであり，より正確には，集合 X 上の σ-集合体 $\mathcal{F}$ に対して以下で定められる函数です:

測度の定義

集合函数 $\mu : \mathcal{F} \to [0, \infty)$ が σ-集合体 $\mathcal{F}$ 上の測度になるのは以下の条件を充すときである:

$\mathcal{M}1$: $\mu(\emptyset) = 0$
$\mathcal{M}2$: 任意の $A \in \mathcal{F}$ に対して $\mu(A) \geq 0$
$\mathcal{M}3$: $A_i \in \mathcal{F} (i \in \{1, 2, \ldots\})$ が $A_i \cap A_j = \emptyset (i \neq j)$ のときに
$$\mu\Bigl(\sum_{i=1}^{\infty} A_i\Bigr) = \sum_{i=1}^{\infty} \mu(A_i)$$

ここで $\mathcal{M}1$ から空集合 $\emptyset$ に対しては 0 を返却しますが，逆に集合 A の値が 0 になるからといって $A = \emptyset$ となるとは限らないことに注意が必要です．そのため，二つの可測函数 $f, g : X \to \mathbb{R}$ について集合 $\{x \in X : f(x) \neq g(x)\}$ の測度が 0 になる場合は「$f = g$ *a.e.*」と記述し，「**ほとんど至る所 (almost everywhere) で** $f = g$」といいます．また，ここでの測度函数は，区間 $[0, 1]$ の大きさが 0 よりも大であること以上の条件はありません．特に d 次元の区間 $J = [a_1, b_1] \times \cdots [a_d, b_d]$ の測度が $|J| = \prod_{i=1}^{d} (b_i - a_i)$ で与えられる測度を「**ルベーグ測度**」といいます．このルベーグ測度は $\mathcal{B}(\mathbb{R}^n)$ 上で存在することが「**ルベーグ測度の存在定理**」として知られています:

ルベーグ測度の存在定理

以下の性質を充す $\mathcal{B}(\mathbb{R}^d)$ 上の集合函数 λ が一意に存在する:

1. $\forall x \in \mathcal{B}(\mathbb{R}^d)$ に対して $0 \leq \lambda(x) \leq \infty$, 特に $\lambda(\emptyset) = 0$
2. $\mathcal{B}(\mathbb{R}^d)$ の列 $\{E_i\}_{i \in \{1,2,\ldots\}}$ が $E_i \cap E_j = \emptyset (i \neq j)$ のときに
$$\lambda\left(\bigcup_{i=1}^{\infty} E_i\right) = \bigcup_{i=1}^{\infty} \lambda(E_i)$$
3. $A \subset \mathbb{R}^d$ に対してルベーグ外測度 λ^* を $\lambda^*(A) = \inf\left\{\sum_{i=1}^{\infty} |J_i| : \sum_{i=1}^{\infty} |J_i| \supset A\right\}$ で定めるときに，$\forall E \in \mathcal{B}(\mathbb{R}^d)$ に対して $\lambda(E) = \lambda^*(E)$．特に d-次元開区間 J に対しては $\lambda(J) = |J|$．

ここでルベーグ外測度 λ^* は $\mathcal{B}(\mathbb{R}^d)$ の元 A を覆って A の体積を測る測度で, A が d-次元開区間 $J = (a_1, b_1) \times \cdots \times (a_d, b_d)$ であるときは $\lambda^*(J) = \prod_{i=1}^{d}(b_i - a_i)$ を充します.

さて, 測度 $m : \mathcal{F} \to \mathbb{R}$ を使うと単純函数 $g(x) = \sum_{i=1} a_i \cdot I_{A_i}(x)$ の面積は $\sum_{i=1} a_i \cdot m(A_i)$ で計算できます. この面積を $\int_X g(x)m(dx)$, $\int_X g(x)dm(x)$, あるいは $\int_X g(x)dm$ と表記します. さらに非負可測函数 f は単純函数の非減少列 $\{f_n\}_{i=\{1,2,\ldots\}}$ で近似できるために

$$\int_X f(x)m(dx) = \lim_{n\to\infty} \int_X f_n(x)m(dx)$$

でその積分値を定めることができます. そして, 測度がルベーグ測度 λ のときに $\int_X f(x)\lambda(dx)$ を $\int_X f(x)dx$ と表記します.

8.3.2　確率モデル, 確率変数

さて, ここで述べた測度は初等確率モデルの確率と類似しています. 測度と確率は共に集合族から実数への集合函数であり, 共に排反事象に関しては加法性があります. ただし, 初等確率モデルが測度であるためには σ-加法性を追加する必要があり, 測度で確率を表現するためには全事象 Ω に対して $P(\Omega) = 1$ であることを追加する必要があります. これらのことを留意して, 測度を使って確率を再定義することにしましょう:

確率測度としての定義

σ-集合体である事象族 $\mathcal{F}$ 上の非負集合函数 P が確率測度になるのは以下の条件を充す場合です:

$\mathcal{P}1$: 任意の $A \in \mathcal{F}$ に対して $0 \leq P(A) \leq 1$. 特に $P(\emptyset) = 0, P(\Omega) = 1$
$\mathcal{P}2$: $\mathcal{F}$ の排反事象列 $\{A_i\}_{i\in\{1,2,\ldots\}}$ に対して $P\Big(\sum_{i\in\{1,2,\ldots\}} A_i\Big) = \sum_{i=\{1,2,\ldots,\}} P(A_i)$

これら標本空間 Ω, σ-集合体である事象族 $\mathcal{F}$, この事象族 $\mathcal{F}$ 上の確率測度 P の組を「**確率モデル**」, あるいは「**確率空間**」といいます.

つぎに重要な概念として「**確率変数 (random variable)**」があります. この確率変数は標本空間 Ω から実数空間 $\mathbb{R}^1$ への可測函数として定義し, 確率変数 $X : \Omega \longrightarrow \mathbb{R}$ の「**値域 (range)**」を「$\mathcal{R}_X$」と表記します. それから, d 個の確率変数 $X_1, X_2, \ldots, X_d$ を組にして Ω から $\mathbb{R}^d$ への写像 $X = (X_1, X_2, \ldots, X_d)$ を考えるときに, この函数 $X : \Omega \to \mathbb{R}^d$ を「**確率変数ベクトル**」と呼びます.

確率変数 (確率ベクトル) を導入することによって, 事象をユークリッド空間 $\mathbb{R}^n$ の点と見做すことが可能になるだけでなく, 後述の平均や分散等の確率モデルを特徴付ける諸量が計算可能になるといった利点があります. 確率変数の例を幾つかあげておきましょう. 最初に骰子を一度振って出た目を見るという試行にて, 事象は「1 の目が出る」,「2 の目が出る」,...,「6 の目が出る」であり, このときに確率変数 $X:\Omega\to\mathbb{R}$ としては各事象の骰子の目を対応させれば良いでしょう. また, コインを一度投げるという試行であれば標本空間 Ω は $\{$裏, 表$\}$ なので, 確率変数 $X:\Omega\to\mathbb{R}$ として「裏」であれば 0,「表」であれば 1 を対応させるようにすれば良いでしょう. さらにコインを n 回投げて, 表が k 回出る確率をモデル化するのであれば,「k 回表が出る」という事象 A_k は x_i を i 回目に表か出た場合は 1, 裏が出た場合には 0 とするときに $A_k=\left\{(x_1,\ldots,x_n):\sum_{i=1}^{k}x_i=k\right\}$ となります. そこで, 確率変数 X を $X(A_k)=k$ とすればよいでしょう. ところが, この確率変数は k 回目に表が出る確率を調べるためには不向きです. 何故なら, 和には順序が反映されないためです. このように確率変数 X は適当に定めて良いものではなく, 確率モデルそのものを決定付けることもあるために, モデルに適合したものを定義しなければなりません. また, その値にしてもコイン投げで「裏」に $-\infty$,「表」に ∞ を対応させるようなことをやってしまえば, 次の説明する平均や分散の計算に意味がなくなります. すなわち, 確率変数の値域はその必要性がなければ, 有限の範囲でおおよその値が決定されるようなものであるべきです. また, 確率変数 X を導入することで, 任意の $a\in\mathcal{R}_X$ に対して $[X=a]=\{\omega\in\Omega:X(\omega)=a\}\in\mathcal{F}$ を充すべきで, さらに言えば, $\mathbb{R}$ のボレル集合族 $\mathcal{B}(R)$ に対して, 確率変数 X の値域 $\mathcal{R}_X$ の任意の部分集合 E は $E\in\mathcal{B}(R)$ であり, また, $[X\in E]=\{\omega\in\Omega:X(\omega)\in E\}\in\mathcal{F}$ を充すべきです. これは X が $\mathcal{F}$ 上の可測函数であることから充します. この議論は確率変数ベクトル $X:\Omega\to\mathbb{R}^d$ が定義されたときの確率モデル $(\Omega,\mathcal{F},P)$ に対しても同様であり, 確率測度と確率変数 (あるいは確率ベクトル) を用いて可測函数 μ_X を以下で定めます:

分布の定義

確率変数 (確率ベクトル)$X:\Omega\to\mathcal{R}_X$ が与えられたときに非負可測函数 $\mu_X:\mathcal{R}_X\to\mathcal{F}$ を次で定義します:

$$E\subset\mathcal{R}_X\text{に対して}\mu_X(E)=P\big(X\in E\big)$$

そして, この μ_X を「X **の分布 (distribution)**」といいます: X が確率変数のときに $\mathcal{R}_X\in\mathcal{B}\big(\mathbb{R}\big)$, X が d 次元の確率ベクトルであれば $\mathcal{R}_X\in\mathcal{B}\big(\mathbb{R}^d\big)$ となります.

なお, 確率変数 X の値域 $\mathcal{R}_X$ が可算集合の場合, 分布 μ_X を「**離散分布 (discrete distribution)**」といい, さらに分布 $\mu_X(A)$ が非負可測函数 $\rho:\mathcal{R}_X\to[0,\infty)$ を

使って $\mu_X(A) = \int_A \rho(x)dx$ とかけるときに分布 μ_X を「**連続分布 (continuous distribution)**」, そして, 函数 ρ を「**確率密度函数 (probability density function)**」といいます. そして, 確率変数 (確率ベクトル)X を定めたとき, 集合 $B \subset \mathcal{R}_X$ に対する確率 $P(X \in B) = P(\{\omega \in \Omega : X(\omega) \in B\})$ は, 離散分布に対しては $\mu_X(B) = P\big(X \in B\big) = \sum_{\omega \in [X \in B]} P(\omega)$, 連続分布に対しては $P\big(X \in B\big) = \int_B \rho(x)dx$ で計算できます. なお, 確率測度 P を用いると $\mu_X(B) = \int_{[X \in B]} P(d\omega)$ とも記述できます.

8.3.3　平均と分散

確率変数 X を導入することで, 事象の「**平均**」や「**散らばり具合**」といった分布の性質を数値的に評価できるようになります. これらは「**平均 (mean)**」と「**分散 (variance)**」と呼ばれる数値であり, 離散分布であれば次の式で定められます:

離散分布 μ_X による平均 $E[X]$, 分散 $V[X]$

$$E[X] = \sum_{a \in \mathcal{R}_X} a\mu_X(a) = \sum_{a \in \mathcal{R}_X} aP(X = a) = \sum_{\omega \in \Omega} X(\omega)P(\omega)$$
$$V[X] = E[X^2 - E[X]^2] = \sum_{a \in \mathcal{R}_X} (a - E[X])^2 \mu_X(a) = \sum_{\omega \in \Omega} (X(\omega) - E[X])^2 P(\omega)$$

さらに可測函数 $f : \mathbb{R}_X \to \mathbb{R}$ に対しては次で確率 p による平均, すなわち,「**期待値 (expectation)**」を定義します.

離散分布 μ_X による f の期待値 $E[f]$

$$E[f] = \sum_{a \in \mathcal{R}_X} f(a)\mu_X(a) = \sum_{a \in \mathcal{R}_X} f(a)P(X = a) = \sum_{\omega \in \Omega} f(X(\omega)) \cdot P(\omega)$$

これが連続分布に対しては積分を使って平均, 分散と期待値が定義されます:

連続分布 μ_X による分散 $V[X]$, 平均 $E[X]$ と期待値 $E[f]$

$$V[X] = E[X^2 - E[X]^2] = \int_{\mathcal{R}_X} (x - E[X])^2 \rho(x) dx = \int_\Omega (X(\omega) - E[X])^2 P(d\omega)$$
$$E[X] = \int_{\mathcal{R}_X} x\rho(x)dx = \int_\Omega X(\omega)P(d\omega)$$
$$E[f] = \int_{\mathcal{R}_X} f(x)\rho(x)dx = \int_\Omega f(X(\omega))P(d\omega)$$

離散と連続分布の違いは $\mathcal{R}_X$ が可算個であるか, 非可算であるかの違いであるため, ここでは可算個の和であることを強調する目的以外では, 平均や分散で積分記号を用いた表記で記述することにします. また, 期待値 $E[f]$ は, どの確率分布を用いているかを明瞭にするために確率変数を添えて $E_X[f]$ と適宜, 記述することにします. また, 分散の平方根 $\sigma[X] = \sqrt{E[X]}$ のことを「**標準偏差 (standard deviation)**」と呼びます. 以降, 単純に σ と記載したときは分布の標準偏差, σ^2 と記載したときは分布の分散を意味します.

ここで幾つかの代表的な分布の平均値と分散を計算しておきましょう.

■二項分布: 金貨投げのように,「表」か「裏」のように二つの事象しかないときに, 長さ N の事象の列 $X_1, X_2, \ldots, X_N$ のことを「**長さ N のベルヌーイ列**」といいます. ここである事象の生じる確率が p のときに, 長さ N のベルヌーイ列で, その事象が r 回生じる確率は $\dfrac{N!}{r!(N-r)!}p^r(1-p)^{N-r} = {}_N\mathrm{C}_r p^r(1-p)^{N-r}$ で与えられ, この確率を $B_{N,p}(r)$ と表記し, この $B_{N,p}(r)$ が従う分布を二項分布といいます.

この二項分布の平均 $E[r]$ は $\displaystyle\sum_{r=1}^{N} rB_{N,p}(r) = Np$ で与えられます. これを確認するために多項式 $(p+q)^N$ の展開式:$(p+q)^N = \displaystyle\sum_{r=1}^{N} {}_N\mathrm{C}_r p^r q^{N-r}$ を利用します. まず, この展開式の両辺を p で偏微分します:

$$N(p+q)^{N-1} = \sum_{r=1}^{N} r\,{}_N\mathrm{C}_i p^{r-1} q^{N-r}$$

この式の両辺に p をかけると

$$Np(p+q)^{N-1} = \sum_{r=1}^{N} r\,{}_N\mathrm{C}_r p^r q^{N-r}$$

ここで, 式の右辺が二項分布の平均 $E[r]$ で, さらに式の左辺の $p+q$ が 1 であることから $E[r] = Np$ が得られます.

さらに二項分布 $B_{N,p}(r)$ の分散は Npq で与えられます．このことも平均の場合と同様に示せます．ここで分散 $E\left[(r-E[r])^2\right] = E[r^2] - (E[r])^2$ であることから，平均 $E[r]$ は判っているので $E[r^2]$ を計算しなければなりません．ここで平均 $E[r]$ を求めるときに使った $Np(p+q)^{N-1} = \sum_{r=1}^{N} r\,{}_N\mathrm{C}_i p^r q^{N-r}$ を p で偏微分します:

$$N(p+q)^{N-1} + N(N-1)p(p+q)^{N-2} = \sum_{r=1}^{N} r^2\,{}_N\mathrm{C}_r p^{r-1} q^{N-r}$$

次に式の両辺に p をかけて $p+q=1$ であることを用いると，$E[r^2] = Np + N(N-1)p^2$ となります．最後に $E[r] = Np$ であることから

$$E[(r-E[r])^2] = Np + N(N-1)p^2 - N^2p^2 = Np(1-p) = Npq$$

が得られます．

■**ポアソン分布**:　$P_\lambda(r) = \dfrac{e^{-\lambda}\lambda^r}{r!}$ で定義される，各変数 λ, r が自然数 $\mathbb{N}$ の値を取る分布をポアソン (Poisson) 分布といいます．ちなみに指数函数の級数展開式 $e^x = \sum_{r=0}^{\infty} \dfrac{x^r}{r!}$ から $\sum_{r=0}^{\infty} \dfrac{e^{-\lambda}\lambda^r}{r!} = 1$ であることが容易に示せるために $P_\lambda(r)$ が確率分布であることが判ります．それから，このポアソン分布 $P_\lambda(r)$ の平均は λ，分散も λ で与えられます．このことはトリッキーですが次で確認できます．まず，e^x の冪級数展開式を x で微分します:

$$e^x = \sum_{r=1}^{\infty} \lambda \frac{rx^{r-1}}{r!}$$

それから両辺に λ をかけて $x = \lambda$ とします:

$$\lambda e^\lambda = \sum_{r=1}^{\infty} \frac{r\lambda^r}{r!} = \sum_{r=0}^{\infty} \frac{r\lambda^r}{r!}$$

この式の両辺を e^λ で割ってしまうとポアソン分布の平均 $E[r]$ が

$$E[r] = \sum_{r=0}^{\infty} r\left(\frac{e^{-\lambda}\lambda^r}{r!}\right) = \lambda$$

であたえられることが判ります．

次にポアソン分布 $P_\lambda(r)$ の分散 $E[r^2]$ も λ になることを確認しましょう．そのために $\lambda e^\lambda = \sum_{r=0}^{\infty} r\left(\dfrac{\lambda^r}{r!}\right)$ を λ で偏微分すると以下の等式を得ます:

$$\lambda e^\lambda + e^\lambda = \sum_{r=1}^{\infty} r^2\left(\frac{\lambda^{r-1}}{r!}\right)$$

この式の両辺に λ をかけて, e^{λ} で割ってしまいます:

$$\lambda^2 + \lambda = \sum_{r=0}^{\infty} r^2 \left(\frac{e^{-\lambda}\lambda^r}{r!} \right) = E[r^2]$$

このように $E[r^2] = \lambda^2 + \lambda$ であることが判ります. ところで, 分散は $E[r^2] - (E[r])^2$ で計算できることから, $E[r^2]$ と $E[r]$ の結果を合せると

$$E\left[(r - E[r])^2 \right] = \lambda^2 + \lambda - \lambda^2 = \lambda$$

が得られます.

ポアソン分布 $P_\lambda(r)$ と二項分布 $B_{N,p}(r)$ には面白い関係があります. これは二項分布 $B_{N,p}(r)$ について, $Np = \lambda$ として λ の値を固定したままで $N \to \infty$ とするとポアソン分布 $P_\lambda(r)$ に近付くという性質です. このことは $B_{N,p}(r)$ を直接計算することで確認できます. まず, λ を固定したときに $Np = \lambda$ という関係を入れていることから $p = \dfrac{\lambda}{N}$ となります. このことから $B_{N,p}(r)$ は次のように展開できます:

$$B_{N,p}(r) = {}_N\mathrm{C}_r p^r q^{N-r} = \frac{N!}{r!(N-r)!} \left(\frac{\lambda}{N} \right)^r \left(1 - \frac{\lambda}{N} \right)^{N-r}$$
$$= \frac{\lambda^r}{r!} \left(\frac{N}{N} \right) \left(\frac{N-1}{N} \right) \left(\frac{N-2}{N} \right) \cdots \left(\frac{N-r-2}{N} \right) \left(\frac{N-r-1}{N} \right) \left(1 - \frac{\lambda}{N} \right)^{-r} \left(1 - \frac{\lambda}{N} \right)^N$$

ここで, $\dfrac{N}{N}, \dfrac{N-1}{N}, \ldots, \dfrac{N-r-1}{N}$ や $\left(1 - \dfrac{\lambda}{N} \right)^r$ は $N \to \infty$ のときに全て 1 に収束します. 残りの $\left(1 - \dfrac{\lambda}{N} \right)^N$ については

$$\left(1 - \frac{\lambda}{N} \right)^N = \left(\left(1 - \frac{\lambda}{N} \right)^{-\frac{N}{\lambda}} \right)^{-r}$$

であり, さらに, $\displaystyle\lim_{\nu\to\infty} \left(1 + \frac{1}{\nu} \right)^{\nu} = e$ であることから,

$$\lim_{N\to\infty} \left(\left(1 - \frac{\lambda}{N} \right)^{-\frac{N}{\lambda}} \right)^{-r} = e^{-r}$$

を得ます. このことから $\displaystyle\lim_{N\to\infty} B_{N,p}(r) = \frac{\lambda^r}{r!} e^{-r} = P_\lambda(r)$ が得られ, N が無限大になると二項分布 $B_{N,p}(r)$ がポアソン分布 $P_\lambda(r)$ に近付くことが判りました.

■ペテルスブルクの賭:　金貨を投げて表が出る確率を $\frac{1}{2}$ とします．ここで n 回投げてはじめて表が出たときに報酬として 2^n 円を受け取るとします．このときの期待値はどうなるでしょうか?

このペテルスブルクの賭で n 回目にはじめて表が出る確率が $\frac{1}{2^n}$ で与えられることから, その期待値は $\sum_{i=1}^{\infty} 2^i \cdot \frac{1}{2^i}$ となりますが, この式は明らかに発散します.

8.3.4　マルコフ, チェビシェフの不等式

平均と確率に関しては, マルコフ (Markov) の不等式, チェビシェフの不等式と呼ばれる関係が成立します:

マルコフの不等式とチェビシェフの不等式

確率変数 X に対して確率 P の期待値 $E[X]$ が有限であるときに以下の二つの不等式が成立します:

- マルコフの不等式:
  確率変数 X が非負の確率変数であれば, 任意の実数 $c > 0$ に対して以下の不等式が成立します:
  $$P[X \geq c] \leq \frac{E[X]}{c}$$
- チェビチェフの不等式:
  分散 $V[X] < \infty$ のときに, 任意の実数 $c > 0$ に対して以下の不等式が成立します:
  $$P\Big[X - E[X] \geq c\Big] \leq \frac{V[X]}{c^2}$$

マルコフの不等式は容易に証明ができます．実際, 期待値が有限で確率変数 X が非負のときに, 任意の $c > 0$ に対して指示函数 $I_{X \geq c}$ を考えると,

$$E[X] = E\Big[\{X \geq c\} \cup \{X < c\}\Big] \geq E\Big[\{X \geq c\}\Big] \geq E[c \cdot I_{X \geq c}] \geq cE[I_{X \geq c}] = cP[X \geq c]$$

以上から, マルコフの不等式 $P[X \geq c] \leq \frac{E[X]}{c}$ が得られます．また, チェビチェフの不等式も同様に示すことができます.

まず, $P\Big[X - E[X] \geq c\Big] = P\Big[(X - E[X])^2 \geq c^2\Big]$ ですが, $X - E[X]$, $\Big(X - E[X]\Big)^2$

も確率変数になります．ここでマルコフの不等式を用いると

$$P\Big[(X-E[X])^2 \geq c^2\Big] \leq \frac{E\Big[(X-E[X])^2\Big]}{c^2} = \frac{V[X]}{c^2}$$

が得られます．これらの不等式は大雑把ではありますが, 負の値を取らない確率変数に関して, その確率変数の領域における確率と, 確率変数領域全体での平均や分散との関係を示す便利な公式となっています.

8.3.5 モーメント

つぎにモーメントを導入しておきましょう:

モーメントと中心モーメント

実数 $q \geq 1$ に対して q 次のモーメント (moment) $m_q(X)$ と q 次の中心モーメント $m_q^{(c)}$ を以下で定めます:

$$m_q[X] = E\left[X^q\right]$$
$$m_q^{(c)}[X] = E\left[(X-E[X])^q\right]$$

$q=2$ のときの中心モーメント $m_2^{(c)}[X]$ は分散と一致します．また, $q=3$ のときの中心モーメント $m_3^{(c)}[X]$ は平均 $E[X]$ を中心としたときの対称性を示す量になります．実際, 1 次元の分布で考えると, そのヒストグラムが左右対称になるものであれば 0 になり, 負であれば, その分布のヒストグラムは平均 $E[X]$ よりも小さい左側に裾野を長く引き, 正であれば逆に平均 $E[X]$ よりも大きな右側にそのヒストグラムの裾野が長く引く傾向が判ります．そして, 分布の $m_3^{(c)}[Y]$ を標準偏差 σ の 3 乗で割った値 $\gamma_1 = \dfrac{m_3^{(c)}[Y]}{\sigma^3}$ を「**歪度 (skewness)**」といいます.

また, 4 次の中心モーメントは分布の尖り具合を示す量として用いられ, 2 次の中心モーメントの 2 乗 (=標準偏差 σ の 4 乗) で割った値 $\gamma_2 = \dfrac{m_4^{(c)}}{\left(m_2^{(c)}\right)^2}$ を「**尖度 (kurtosis)**」といいます．ちなみに正規分布であれば $\gamma_2 = 3$ となることから, 正規分布と比較したときの尖り具合を示すために, 尖度の定義を $\gamma_2 = \dfrac{m_4^{(c)}}{\left(m_2^{(c)}\right)^2} - 3$ で定めることもあります．実際, SciPy の Stats パッケージの尖度はこの 3 を引いた定義式になっています.

このモーメントに関連する関数として「**確率母函数**」と「**モーメント母函数**」があります．最初の確率母函数は離散的確率に対して定義され, モーメント母函数は連続的分布に対して定義されます:

確率母函数, モーメント母函数

確率変数 X が非負整数である離散的分布に対して「**確率母函数 (probability generating function)**」を以下で定めます:

$$G_X(t) = E_X[t^X] = \sum_{i \in \{1,2,\ldots\}} t^i \cdot P(i)$$

また, 連続的分布に対しては「**モーメント母函数 (moment generating function)**」を以下で定めます:

$$M_X(t) = E_X[e^{tX}]$$

確率母函数の特徴は, その n 次の微分から確率 $P(n)$ が計算できる点です. すなわち, 確率母函数 $G_X(t)$ の n 階導函数 $G_X^{(n)}(t)$ に関しては

$$G_X^{(n)}(t) = \frac{d^n G_x(t)}{dt^n} = \sum_{i \geq n} E[i(i-1)\cdots(i-n+1)t^{i-n}]$$

であることから $\dfrac{G_X^{(n)}(0)}{n!} = P(X=n)$ であり, このことが「確率母函数」の名前の由来となっています.　同様にモーメント母函数についても, その n 階微分が

それから重要な量として「**エントロピー**」があります

確率 p のエントロピー

$$S[X] = -\int_{\mathcal{R}_X} p(x) \log p(x) dx$$

このエントロピーは情報理論では「**情報量**」ともいわれる数値です. エントロピーは確率 p が 0 の場合と 1 の場合は共に 0 になります. つまり, 何も生じないことと, 必ず生じることといった決定的なことに関しては, そのような確率から得られる情報は皆無であることを意味します.

8.3.6　推定量

ここでは独立な試行に基づく推定量の話をします．まず，確率変数 X が従う確率 P に対して，それと同じ分布を持つ独立な確率変数の列 $X_1, X_2, \ldots$ が与えられているとします．このような分布を「**独立同分布 (independent and identical distribution(iid)**」といいます．そして，これらの確率変数の実現値としての標本を $x_1, x_2, \ldots$ と表記することにします．このときに平均や分散といった基本的な統計量の推測値は，これらの iid な確率変数を用いて表現されていて，実現値 $x_1, x_2, \ldots$ が与えられると即座に計算ができるようになっていとします．このときに真の統計量の値 θ からの乖離具合を測る量として「**バイアス (bias)**」があります．まず，Z を推定量とするときにバイアスは Z の平均 $E[Z]$ と真の値 θ の差として表現されます:

バイアスと不偏推定量

Z を統計量の推測量，θ を真の統計量とするときに

$$b = Z - \theta$$

を「**バイアス**」と言います．
なお，バイアスの平均値 $E[b] = E[Z] - \theta$ が 0 になる統計量 Z のことを「**不偏推定量**」と言います．

不偏推定量の例として，独立な試行に基くときに確率変数 X の平均値の推測値としての算術平均 $E_n = \dfrac{1}{n}\displaystyle\sum_{i=1}^{n} X_i$ を挙げておきます．実際，バイアスを計算すると，

$$E[E_n] - E[X] = \frac{1}{n}\sum_{i=1}^{n} E[X_i] - E[X] = E[X] - E[X] = 0$$

となることから判ります．また，平均値の推定値として，標本数 n ではなく $n-1$ で割る流儀があります．このときのバイアスは $E\Big[\dfrac{1}{n-1}\displaystyle\sum_{i=1}^{n} X_i\Big] - E[X] = \dfrac{1}{n}E[X]$，つまり，$O\big(\frac{1}{n}\big)$ のオーダーのバイアスを持つことになりますが，$n \to \infty$ でバイアスは 0，つまり，$E[X]$ に収束します．このように標本数 $n \to \infty$ でバイアスが 0 となる推定量のことを「**一致推定量**」と言います．ちなみに，この考えている分布が iid のときに E_n については以下のことが成立します:

E_n の性質

- 不偏性: $E[En] = E[X]$
- 分散の収束性: $V[E_n] = \dfrac{V[X]}{n}$

不偏性についてはバイアスの計算で示せています. 分散の収束性については, $X_1, X_2, \ldots$ が iid であるために $i \neq j$ のときに $V[aX_i + bX_j] = a^2V[X_i] + b^2V[X_j]$ が成立するため,

$$V[E_n] = \frac{1}{n^2}\sum_{i=1}^{n} V[X_i] = \frac{1}{n}V[X]$$

が得られます. ここで不偏性は独立した試行から得られた標本による算術平均 E_n は $n \to \infty$ で $E[X]$ に収束するという「**大数の法則**」に対応し, 分散の収束性は「**中心極限定理**」に関連します:

大数の弱法則

確率分布列 $\{X_i\}$ が独立同分布で, それらの分散が有限とし, 分散 $V[X] = \sigma^2$, 平均 $E[X] = \mu$, $E_n = \sum_{i=1}^{n} X_i$ とするときに以下が成立します:

- 任意の $\varepsilon > 0$ に対して

$$\lim_{n\to\infty} P\Big(|E_n - \mu| > \varepsilon\Big) = 0$$

- 実数 $\mathbb{R}$ 上で有界な可測函数 $f(x)$ が $x = \mu$ で連続であれば

$$\lim_{n\to\infty} E\Big[f(E_n)\Big] = f(\mu)$$

中心極限定理

確率分布列 $\{X_i\}$ が独立同分布で, それらの分散が有限とし, 分散 $V[X] = \sigma^2$, 平均 $E[X] = \mu$, $E_n = \frac{\sum_{i=1}^{n} X_i}{n}$ とすると, 任意の実数 $a < b$ に対して以下が成立します:

$$\lim_{n\to\infty} P\Big(a \leq \frac{E_n - \mu}{\sqrt{\frac{\sigma^2}{n}}} \leq b\Big) = \frac{1}{\sqrt{2\pi}}\int_a^b \exp\left\{\frac{x^2}{2}\right\} dx$$

8.3.7　乱数について

ここでは乱数について簡単に述べることにします. 統計的処理はデータがあってのことで, 立派な統計モデルが出来上がっていたとしても, 想定した分布に従った標本データがなければ, その統計モデルの正しさの検証は勿論, 後述する統計的推測も満足に行えません. では, その標本データをどうやって生成するかが問題になります. 二項分布が必要だからといって, コイン投げを 10 万回, 必要になる度に行うという訳にはいきません. そこで効率的なしかも. どのような分布に従うかで一々, 分布を切替えるのも大変なことです. ところが, 一様分布に従う乱数が生成できさえすれば, それを利用して, 他の分布に従う乱数を構成することが可能なことが知られています. そこで, 一様分布に従う乱数さえ生成できればよいことになります. では, 一様分布を生成するためにはどうすれば良いのでしょうか？

以下から半世紀程昔の本等を見ると, 数値計算による方法と放射線やダイオードで生じるノイズを利用する方法が挙げられています. ところで, 最近の統計に関する書籍を見ると物理現象に起因する物理乱数を使うことを取り上げた本はまず見当りません. それは数値計算による乱数の生成が劇的に向上したことを意味しています.

ところで, 数値的に生成される乱数は実のところ「**乱数みたいな数**」である「**疑似乱数 (pseudo randomnumbers)**」です. 実のところ,「**乱数**」に関しては実用的な定義がないために与えられた数列が乱数であるかどうか証明ができません. だからと言って, 闇雲に数値を出すだけで良いというものでもありません. たとえば, 数値的な疑似乱数生成で広く用いられている手法に線形合同法がありますが, パラメータの設定次第では, 一見すると出鱈目に並んでいるようでも, ある程度の長さで見ると, その周期性や偏り具合が丸見えになることもあります. そのために乱数列の「**統計的な筋の良さ**」を検証するために, 統計的仮説検定によってある事象を表現するために生成した数列から有限列を取り出し, 何らかの経験的な統計量を求めておいて, 真の分布から理論的に計算される統計量との乖離が許容範囲内に収まっていることを確認することもあります.

このように疑似乱数は上述の統計的な筋の良さも, もちろん, 要求されますが, それ以上に使い勝手の良さも重要です. たとえば, 数列の生成で外部の自然現象等のデータに依存するために, 別途, 外部データが必要であるとか, 計算時間が異様に長いといった乱数は幾ら計算機の処理が高速であったとしても使い勝手の面で良いものではありません. また, 計算機だけで, 余計な外部データ不要で高速に生成できたとしても, 再現性が皆無であれば, それはそれで乱数としては良いものであったとしても, 計算結果の評価・検証を行う上では不便極まりないものになってしまいます. そのため, 乱数の生成では, 計算機だけで速く効率的に計算できて, 再現性のあるものが良いという非常に都合の良いものであるべきです.

ここで幾つかの疑似乱数生成の手法を紹介しておきます．何れも合同式 ($a \equiv b \mod b$ をベースにした手法で, そのために周期が存在します．つまり, x_m を使って x_{m+1} を生成してゆくことで整数列 $\{x_i\}_{i=1,...,}$ を得ますが, ここで, 数列の一部を並べると

$$\begin{aligned} x_m, x_{m+1}, \ldots, x_{m+n-1} &= x_{m+n}, x_{m+n+1}, \ldots . x_{m+2n-1} \\ &= x_{n+2n}, x_{m+2n+1} \cdots, x_{m+3n-1} \\ &= \cdots \end{aligned}$$

のように長さ n の数列の反復になるときに, この数列の周期が n であるといいます．疑似乱数の生成では, この周期 n ができるだけ長いものが生成できるものが良いといえます．このようなことを踏まえて幾つかの手法を紹介しましょう．なお, この節は [21] を参照しており, 詳細はそちらを参照して下さい．

■線形合同法 (linear congruetial generator):　レーマー法 (Lehmer generator) とも呼ばれる手法で, 自然数 a, c, M に対して合同式 $x_{i+1} \equiv ax_i + c(\mod M)$ で自然数 x_i から自然数 x_{i+1} を生成し, こうして得られた数列 $\{x_i\}$ を疑似乱数とする手法です．この式で生成される自然数は $\{0, 1, \ldots, M-1\}$ に含まれるために周期が高々 M であり, この方法で生成される疑似乱数は最初の x_1 の与え方で一意に定まります．ちなみに, この x_1 のことを「**シード (seed)**」と呼びます．疑似乱数を用いたプログラムの検証で「**シードを固定する**」ということの意味は初期値を定めることで, 疑似乱数として同一の数列を使うことと同義です．

この方法で何時でも周期が M になるとは限らず, 最大周期 M を実現するためには以下の条件を充たし, また, そのときに限ることが知られています:

線形合同法で最大周期を得るための条件

- c と M は互いに素である
- M の約数である素数 p について $a \equiv 1(\mod p)$
- $M \equiv 0(\mod 4)$ のときに $a \equiv 1(\mod 4)$

この方式では自然数を生成するだけですが, $M > 2$ のときに得られた数列 $\{x_i\}_{\{i=1,2,...\}}$ の各 x_i を M で割ることで得られる数列 $\left\{\frac{x_i}{M}\right\}_{\{i=1,2,...\}}$ に変換することで区間 $[0, 1)$ の疑似乱数が得られます．以降で紹介する乱数の生成も同様のことを行って実数値の疑似乱数を計算します．

この線形合同法は仕組が単純なために広く用いられますが, 実のところ, 生成される乱数はある超平面上に乗っており, 非常に疎な集合になります．　ここで周期 n の乱数列 $x_1, x_2, \ldots, x_n$ について, この列から生成される n 個の乱数の組 $(x_1, x_2), (x_2, x_3), \ldots, (x_{n-1}, x_n)$ を 2 次元平面上に配置するとこの 2 次元平面では整数の組は格子状に並んでおり, n^2 個の点がある訳ですが, 周期 n の乱数からは n 個の点しか現われないことになります．実際, 合同式 $x_{i+1} \equiv 5x_i + 3(\mod 17)$ で乱数を生成して,

各 (x_i, x_{i+1}) を 2 次元平面にプロットしてみましょう:

```
import numpy as np
from matplotlib import pyplot as plt
ANS = []
x1 = 0
while True:
    ans = solve_mod([x == 5*Integer(x1) + 3],17)
    x1 = ans[0][0]
    if x1 in ANS:  break
    else:
        ANS.append(x1)

plt.scatter(ANS[:-1],ANS[1:], marker="*")
plt.text(0, ANS[0],str(1))
plt.scatter([0],[ANS[0]], marker="*")
for _i in np.arange(len(ANS)-1):
    plt.text(ANS[_i], ANS[_i+1],str(_i+2))
plt.title(r"$x_{i+1}\equiv 5 x_{i} + 3 (\mathrm{mod}\ 17), x_1 = 0$
    ")
plt.xlabel(r"$x_{i}$")
plt.ylabel(r"$x_{i+1}$")
plt.show()
```

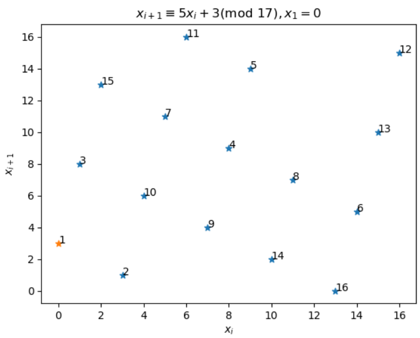


図 8.2 (x_{i-1}, x_i) の配置の様子

ここでは SageMath を SageMath と Python を混合させて利用しています．まず, 最初に Python のパッケージの読込を import 文で行っていますが, while 文の中で合同式を解くために SageMath の函数 solve_mod() を用いています．それからグラフ表示では, matplotlib を利用しています．基本的なことは Python の場合と同じですが, ここで注意することは合同式で x_{i+1} を計算するために必要な右辺の変数 x_i の初期値として入力した $x0$ の整数値を Integer() で SageMath の整数に変換していることです．この処理がなければ函数 solve_mod() に引き渡された引数の数値の型が SageMath のものと別個のものであるためにエラーが発生します．さて, 左図 8.2 には

生成した乱数 $x_1, x_2, \ldots, x_{16}$ を 2 次元平面上での点 (x_i, x_{i+1}) として描いています．ここで, 図での各点に割り当てられている番号は生成順を示す番号です．このように番号はバラバラになっていますが, 各点は複数の直線上に綺麗に並んでいます．何故なら, x_i, x_{i+1} はある自然数 k に対して $x_{i+1} - 5x_i - 3 - 17k = 0$ という直線上に載るためです．このように線形合同法では格子点上に生成した数が並ぶ「**結晶構造**」と呼ばれる分布の偏りが現われます．また, この例では mod 17 であるために $0, 1, \ldots, 16$ の 17 個の整数が現われますが, そもそも格子点自体は $17^2 = 289$ 個ある訳で, 非常に疎であることも実感できます．

ちなみに乱数性の尺度として「**高次均等分布性**」という尺度があります．これは計算機で生成する乱数の周期と数の表現に対応するビット長に着目したものです．まず, 乱数の周期が n のときに乱数列 $\{x_i\}_{i=1,2,\ldots]}$ はその部分列 $x_1, x_2, \ldots, x_n$ の繰返しになります．このときに各乱数 x_i が w ビット長の数であるとし, これら x_i の上位 $v < w$ ビットを取り出して, $x_i^{(v)}$ と表記します．ここで定義から $x_i^{(w)} = x_i$ を充します．そして, 乱数の周期が n のときに乱数列 $x_1, x_2, \ldots, x_n$ を使って自然数 $d < n$ に対して d 成分の n 個の組を以下のように定めます:

$$
\begin{gathered}
(x_1^{(v)}, x_2^{(v)}, \ldots, x_d^{(v)}), (x_2^{(v)}, x_3^{(v)}, \ldots, x_{d+1}^{(v)}), \\
(x_3^{(v)}, x_4^{(v)}, \ldots, x_{d+2}^{(v)}), \ldots, (x_n^{(v)}, x_{n+1}^{(v)}, \ldots, x_{n+d-1}^{(v)})
\end{gathered}
$$

先程の合同式の例では $d = 2, x_i^{(v)} = x_i$ としており, 各 (x_i, x_{i+1}) は全て異なった平面上の点として現われています．ただ, この手法で生成した d 次元空間内の点は超平面上の点として現われるため, 「**乱数**」としては物足りないものといえるでしょう．さらには M を偶数にしたときに偶数と奇数が必ず交互に現われるという性質もあります．実際, 漸化式を $x_i \equiv 5x_{i-1} + 3 (\mathrm{mod}\ 42)$ で計算してみましょう:

```
ANS = []
x1 = 0
while True:
    ans = solve_mod([x == 5*Integer(x1) + 3],42)
    x1 = ans[0][0]
    if x1 in ANS:  break
    else:
        ANS.append(x1)
```

この処理の結果は ANS = [3, 18, 9, 6, 33, 0] と, 奇数と偶数が交互に現われる結果になります．

何れにせよ, この線形合同法は単純な割には合同式を解かねばならず, 周期を大きくするために M も大きくするしかなく, 得られる数列も結晶構造を持ち, さらに M が偶数で

あれば奇数と偶数が交互に現われるといった性質があるために, 様々な改良方法が考案されています.

■高階漸化式法:　線形合同法の改良として, 周期を長くして, さらには高次均等分布性を改良する手法として, 漸化式をより複雑にすることを考えます. もちろん, 計算性も良好なものでなければ利用し難くなるので, k 階の線形漸化式を利用することにします. そこで, $a_1, a_2, \ldots, a_k$ を整数とし, さらに $a_1 \neq 0$ であるとします. それから十分に大きな自然数 M に関して次の線形漸化式を使って数列を生成することにします:

$$x_{i+k} = a_1 x_{i+k-1} + a_2 x_{i+k-2} + \ldots + a_k x_1 \ (\mathrm{mod} M)$$

この漸化式からも判るように, シードとして x_1 だけが必要だった線形合同法と異なり, 高階漸化式法では最初に x_k を定めるために k 個のシード $x_1, x_2, \ldots, x_{k-1}$ が必要です. この手法で可能な周期は最大 $M^k - 1$ と拡大され, 闇雲に M を大きくしなくても k を調整することでも周期を大きくすることが可能になります. とは言え, 合同式を解く必要があることと, 結晶構造も引き続き持っています.

■トーズワース法 (Tausworthe generator)　高階漸化式法を実際の計算機に実装する際に, 限られた計算資源を活用する上で, 計算効率を重視した実装方法があります. 実際, 現在の計算機で数値は内部で 2 進数で扱われているために $M = 2$ とすることが自然であり, さらには高階漸化式法の係数 a_i についても多くの係数が 0 となるようにしておくと計算量を減らせます.

ここで述べる「**トーズワース法 (Tausworthe generator)**」は $p > q$ となる自然数 p, q に対する漸化式 $x_{i+p} \equiv x_{i+q} + x_i (\mathrm{mod} 2)$ を用いて疑似乱数を生成します. なお, 排他的論理和「XOR(演算子 $\oplus$ で表記)」を用いることで漸化式は $x_{i+p} = x_{i+q} \oplus x_i$ で書換えられます. 特に自然数を w ビットの 2 進数 (=w 桁の 2 進数) として表現する計算機であれば, 自然数 x は $\{0, 1\}$ 成分のみの w 次元のベクトル $[x_{(1)}, x_{(2)}, \ldots, x_{(w)}]$ として表現され, 演算はベクトルの各段単位で行うことになります. さらに, 集合 $\{0, 1\}$ に積と和演算を入れたガロア体 $\mathbb{GF}(2)$ で考えるのであれば排他的論理和はガロア体 $\mathbb{GF}(2)$ 上の和演算として解釈できます. このガロア体 $\mathbb{GF}(2)$ では和演算「"$+$"」と差演算「"$-$"」は同じ結果が得られる演算になります.

このトーズワース法で最大周期は $2^p - 1$ になることが知られており, 前述の線形合同法等と比べても長い周期の疑似乱数列が容易に生成できます. ただし, 実数の疑似乱数生成では線形合同法のように M で割れば済むものではありません. そこで, トーズワース法で区間 $[0, 1]$ に属する実数の疑似乱数生成では, $x_{(1)} x_{(2)} \cdots x_{(w)}$ を整数 x の w 桁の 2 進

数表現とするときに, $\displaystyle\sum_{i=1}^{w} \frac{x_{(i)}}{2^i}$ で $[0,1]$ に包含される疑似乱数を返すことにします. このように 2 進数を数の内部表現として用いている計算機にとっては扱い易い表現になっています.

では, 具体的な計算例を示しておきましょう. x_{i+p} を求める $x_{i+q} \oplus x_i$ の計算で自然数の対 (x_{i+q}, x_i) として $(17, 19)$ が与えられたときには, まず, $(17, 19)$ を 2 進数表現に変換して $(10001_2, 10011_2)$ が得られます. ここで, 10001_2 の右側の添字の 2 は 2 進数表現であることを示しています. このときに排他的論理和 $17 \oplus 19$ は 2 進数表現の桁単位で実行されるために $17 \oplus 19 = 10001_2 \oplus 10011_2 = 00010_2$, 以上から $17 \oplus 19 = 2$ となります. このように整数を生成するために合同式を解く必要もなく, 二つの自然数を計算機内部のままの 2 進数で各ビット単位で排他的論理和を行えば良いために計算機での処理に適した定式化になっていることが判ります.

では, $p, q = (5, 3)$ として, トーズワース法で疑似乱数を生成してみましょう. このトーズワース法では $p > q$ のときに p 個のシードが必要で, ここでは $(p, q) = (5, 3)$ であるために, シードとして $x_1, x_2, \ldots, x_5$ が必要になります. そこで, $(x_1, x_2, x_3, x_4, x_5) = (1, 7, 3, 2, 11)$ で疑似乱数を計算してみます:

```
ANS = []
xs = np.zeros((2^5-1), dtype=np.int64)
xs[0:5] = np.array([1,7,3,2,11])
for _i in range(2^5-6):
    xs[_i+5] = xs[_i+3]^^xs[_i]
plt.scatter(xs[:4],xs[1:5], marker="o", color="red")
plt.scatter(xs[4:-1],xs[5:], marker="*")
for _i in np.arange(len(xs)-1):
    plt.text(xs[_i], xs[_i+1],str(_i+1))
plt.title(r"$x_{i+5}=x_{i+3}\oplus x_{i}$")
plt.xlabel(r"$x_{i-1}$")
plt.ylabel(r"$x_i$")
plt.show()
```

ここで示すプログラムでは最初に「xs=np.zeros((2^5-1), dtype=np.int64)」で NumPy の長さ 31 の 64 ビット整数配列 xs を定義し, その配列の先頭からシードとして 1, 7, 3, 2, 11 を代入しています. ここで配列の長さとして,「2^5-1」という SageMath の冪演算を用いています. Python でこの演算子「^」は排他的論理和の演算子となりますが, SageMath の式であるために冪演算子の意味になります. さらに for 文内での漸化式では SageMath の排他的論理和の演算子「^^」を用います. このように SageMath を Python として用いるときには演算子に注意する必要があります.

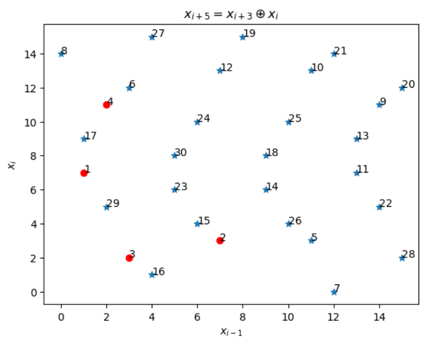


図 8.3　トーズワース法による (x_{i-1}, x_i) の配置の様子

for 節で漸化式を計算したあとは，Python の Matplotlib を用いてグラフ表示を行っています．その結果を図 8.3 に示しますが，ここで (x_i, x_{i+1}) の配置は，図 8.2 に示す線形合同法のものと比較して，直線上ではなく「l **くの字**」状に分布しており，線形合同法のように露骨な結晶構造を示さなくなっています．このように式が簡単な割には密で，周期も $2^5-1 = 31$ と長くなっており，その上，高次均等分布性も向上しているといっても良いでしょう．しかし，シードの与え方に依存することが多く，様々な統計的検定で棄却されることが多いといった弱点があります．

■**メルセンヌ・ツイスター法**:　トーズワース法の改良として，ビット間の情報を混ぜ合わせることで高次均等分布性や統計的検定で棄却され難くすることを目的として，トーズワース法の漸化式の中に行列を使って捻る操作を加えた「**twister GFSR(TGFSR)**」があります．漸化式は $x_{i+p} = x_{i+q} \oplus x_i A$ と記述されます．ここで扱う数値が w ビットであれば A は $\{0,1\}$ 成分の $w \times w$ の正方行列でになり，この行列が「**捻り (twist)**」になります．ところで，この行列は複雑なものにすると計算に時間を要することになりかねないために「**同伴行列 (companion matrix)**」と呼ばれる疎行列が用いられます:

$$A = \begin{pmatrix} & 1 & & & \\ & & 1 & & \\ & & & \ddots & \\ & & & & 1 \\ a_1 & a_2 & \dots & & a_w \end{pmatrix}$$

この同伴行列 A を用いて漸化式を書換えてみましょう．まず，w ビットの自然数 x_i を長さ w のベクトル $[x_{i,(1)}, x_{i,(2)}, \dots, x_{i,(w)}]$ として書換えると，$x_i A = [0, x_{i,(1)}, x_{i,(2)}, \dots, x_{i,(w-1)}] \oplus x_{i,(w)}[a_1, a_2, \dots, a_w]$，ここで $[0, x_{i,(1)}, x_{i,(2)}, \cdots, x_{i,(w-1)}]$ は 自然数 x_i を 2 進数表現に対応するベクトル $[x_{i,(1)}, x_{i,(2)}, \dots, x_{i,(w)}]$ の成分を右に一つシフトしたもので，$x_{i,(w)}a$ は $x_{i,(w)} = 0$ であれば 0，$x_{i,(w)} = 1$ であれば a になります．このように TGFSR での疑似乱数生成のための漸化式は排他的論理和と 2 進数表現の右シフトだけで構成され，数値の内部処理が 2 進数の計算機との相性が非常に良いもの

であることが判ります.

では実際に TGFSR で疑似乱数を計算してみましょう. まず, $w = 5$ として, $w \times w$ 正方行列 A を以下で定めます:

$$A = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 \end{pmatrix}$$

このときにベクトル a は $a = [1, 0, 0, 0, 1] = 10001_2 = 33$ を与えることにします. 以下に TGFSR のプログラムを示します：

```
ANS = []
xs = np.zeros((2^5-1), dtype=np.int64)
xs[0:5] = np.array([1,7,3,2,11])
for _i in range(2^5-6):
    xs[_i+5] = xs[_i+3]^^((xs[_i]>>1)^^(33*(xs[_i]%2)))

plt.scatter(xs[:4],xs[1:5], marker="o", color="red")
plt.scatter(xs[4:-1],xs[5:], marker="*")
for _i in np.arange(len(xs)-1):
    plt.text(xs[_i], xs[_i+1],str(_i+1))
plt.title(r"$x_{i+5}=x_{i+3}\oplus ((x_{i}>>1)\oplus(x_{i,(w)} \
    cdot a^T ))$")
plt.xlabel(r"$x_{i-1}$")
plt.ylabel(r"$x_i$")
plt.show()
```

まず,　トーズワース法のプログラムで漸化式は $x_{i+p} = x_{i+q} \oplus x_i$ でしたが, TGFSR では x_i の項を x_i の右シフトと $x_{i,(w)}$ と a の積の排他的論理和で置換えます. SageMath では x_i の右シフト $[0, x_{i,(1)}, x_{i,(2)}, \ldots, x_{i,(w-1)}]$ を演算子「>>」を用いて「$x_i >> 1$」とします.　また, $x_{i,(w)}$ は x_i を 2 で割ったときの剰余に対応するために「$x_i\%2$」とします.

これらによって, 行列 A による捻り x_iA は「$(x_i >> 1)$^^$(x_i\%2)$」で実装できることになります.

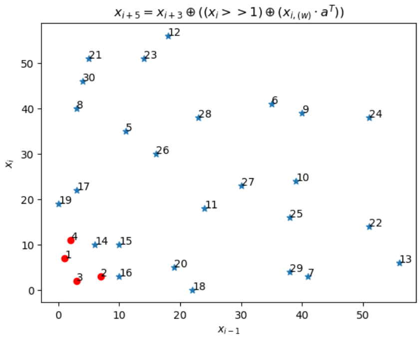


図 8.4　TGFSR 法による (x_{i-1}, x_i) の配置の様子

図 8.4 に結果を示しますが, TGFSR では線形合同法のような結晶構造もなく, トーズワース法で顕著だった「くの字状」の分布が弱くなり,「くの字状」の分布に捻りを入れたような状況になっていることが判ります.

さらに, トーズワース法の最大周期は $2^p - 1$ でしたが, TGFSR では $2^{wp} - 1$ にすることが可能です．そのためには I_w を $w \times w$ の単位行列とするときに, 行列 A の特性多項式 $\varphi(t) = \det(tI_w - A)$ について, $\varphi(t^p + t^q)$ が次数 wp の原始多項式なることが, 最大周期 $2^{wp} - 1$ となるための必要十分条件になります.

この TGFSR を更に改良した手法がメルセンヌ・ツイスター法です．メルセンヌ・ツイスター法では TGFSR の行列演算 $x_i A$ で捻りを入れる箇所を工夫しています．まず, x_i と x_{i+1} を 2 進数化することで得られたベクトル $[x_{i,(1)}, x_{i,(2)}, \ldots, x_{i,(w)}]$,$[x_{i+1,(1)}, x_{i+1,(2)}, \ldots, x_{i+1,(w)}]$ について, x_i 側からは先頭の $w - r$ 成分を, x_{i+1} 側からは後 r 成分を取り出して, 新しいベクトル $[x_{i,(1)}, \ldots, x_{i,(w-r)}, x_{i+1,(w-r+1)}, \ldots, x_{i+1,(w)}]$ を構成し, それから同伴行列 A を作用させます. つまり, 漸化式としては

$$x_{i+p} = x_{i+q} \oplus \left([x_{i,(1)}, \ldots, x_{i,(w-r)}, x_{i+1,(w-r+1)}, \ldots, x_{i+1,(w)}]A\right)$$

となります. こうすることで, x_i の下位 r ビットが次の x_{i+p} の生成に反映されなくなり, その結果として達成可能な周期が最大 $2^{wp-r} - 1$ にすることができます. そして, 自然数 r について, $0 \leq r \leq w - 1$ の範囲で $2^{wp-r} - 1$ が素数になるものを選択します. ちなみに $2^n - 1$ となる素数のことを「**メルセンヌ数 (Mersenne number)**」といいます.

メルセンヌ・ツイスター法は $(w, p, r) = (32, 623, 31)$ とすることで, メルセンヌ数である素数 $2^{wp-r} - 1 = 2^{19937} - 1$ を最大周期とする疑似乱数の実現に成功しています. また, 高次元均等分布性に関しても, $w = 32$, すなわち, 32 ビット精度で 623 次元まで均等分布しており, 32 ビット以下の w ビットでも良好な均等分布性を持っています. さらには, 多くの統計的検定に対しても棄却されないという, 非常に良い性質があり, その上, 高速な演算処理が可能でさえあります.

8.3.8 与えられた分布に従う乱数の生成

上述の各方法で一様分布に従う乱数が生成できることが判りました．では, 一様分布以外の正規分布やポアソン分布といった様々な分布に従った乱数はどのように計算すれば良いのでしょうか? ここでは一様分布に従う疑似乱数から指示した分布に従う疑似乱数を構築する幾つかの手法について述べることとします.

■逆函数法: 名前のとおりに累積分布函数の逆函数を用いる手法です．この手法は逆函数の構築が容易な分布に対しては有効ですが, 正規分布のように逆函数の計算が近似函数の計算になる場合にはあまり有効な手法とは言えません.

まず, 累積分布函数の定義を行いましょう．まず,　確率 $P(X)$ が離散分布のときに確率変数 X の取り得る値を $[\ldots,\xi_0,\xi_1,\ldots]\subset\mathbb{R},\xi_i<\xi_{i+1}$, $X=\xi_i$ となる確率 $P\left[X=\xi_i\right]$ を p_i とします．このときに累積分布函数を以下で定めます:

$$F_X(x)=P\left[X\leq x\right]=\sum_{\xi_i\leq x}p_i$$

この函数 $F(X)$ は単調増加函数です．そこで逆函数 $F^{-1}(Y)$ を以下で定めます:

$$F_X^{-1}(y)=\inf\left\{x:F_X(x)\geq y\right\}$$

また, 確率 $P(X)$ が連続分布のときに累積分布函数 $F_X(x)$ は確率密度函数を f_X とするときに

$$F_X(x)=P\left[X\leq x\right]=\int_{-\infty}^{x}f_X\left(t\right)dt$$

で定められます．この累積分布函数 $F_X(x)$ は単調増加函数であるために逆函数 $F_X^{-1}(y)$ が必ず存在しますが, 離散分布の場合と異なり, 逆函数の計算は一般的に簡単ではありません.

累積確分布数 F_X の逆函数 $F^{-1}(Y)$ を用いて確率分布 $P(X)$ が従う分布の乱数を生成する方法が「**逆函数法**」です.

逆函数法の原理

> U を閉区間 $[0,1]$ の一様分布の確率変数とし, 確率変数 X に対して累積確率函数 F_X とその逆函数 F_X^{-1} が定まるときに確率変数 W を $W = F_X^{-1}(U)$ で定義します. このときに確率変数 W の累積分布函数は F_X になります.

実際, 確率変数 W の累積分布函数は,

$$P\left[W \leq w\right] = P\left[F_X^{-1}(U) \leq w\right] = P\left[U \leq F_X(w)\right] = F_X(w)$$

となるためです. このことからあらかじめ一様分布の標本を計算しておき, その標本を累積分布函数の逆函数で写しさえすれば, 必要とする確率分布の標本が得られることになります. そのために逆函数をどのように構成するかが要です. 確率変数 X が離散的分布の場合, $F_X^{-1}(u)$ の計算は, まず, $W = [\ldots, \xi_i, \xi_{i+1}, \ldots]$, ここで $i, j \in \mathbb{Z}$ に対して $i > j$ であれば $\xi_i > \xi_j$ とします. このときに $P\left[W = \xi_i\right] = P\left[W \leq \xi_i\right] - P\left[W \leq \xi_{i-1}\right]$ となります. ここで $W = F_X^{-1}(U)$ であることから,

$$\begin{aligned} P\left[W = \xi_i\right] &= P\left[F_X^{-1}(U) \leq \xi_i\right] - P\left[F_X^{-1}(U) \leq \xi_{i-1}\right] \\ &= P\left[U \leq F_X(\xi_i)\right] - P\left[U \leq F_X(\xi_{i-1})\right] \\ &= P\left[F_X^{-1}(\xi_{i-1}) < U \leq F_X(\xi_i)\right] \end{aligned}$$

したがって, $P\left[F_X^{-1}(\xi_{i-1}) < U \leq F_X(\xi_i)\right]$ を充す $i \in \mathbb{Z}$ を求め, それから対応する ξ_i を標本として返せばよいことになります.

ところで, 連続分布の場合は逆函数 F_X^{-1} を求められれば良いのですが, 簡単に計算できるものであるとは限りません. 実際, 平均 $\mu = 0$, 標準偏差 $\sigma = 1$ の正規分布の場合, その確率密度函数 $f_X(x)$ は $\dfrac{1}{\sqrt{2\pi}} \exp\left\{-\dfrac{x^2}{2}\right\}$ で与えられ, その累積分布函数 $F_X(x)$ は

$$F_X(x) = \frac{1}{2}\left[1 + \mathrm{erf}\left\{\frac{x}{\sqrt{2}}\right\}\right]$$

と誤差函数 erf を使って記述できます. したがって, その逆函数 $F_X^{-1}(w)$ は $x = \sqrt{2}\,\mathrm{erf}^{-1}(2u-1)$ で与えられることになりますが, ここで問題になることが誤差函数の逆函数 erf^{-1} の計算で, この計算は数値的な近似計算になるために誤差が付き纏います. さらにベイズ推測で用いられる事後分布は $\dfrac{p(x)}{Z}$ といった形式で, 通常, Z の詳細は判らない場合もあります. そのような場合には逆函数法では手も足も出ないということになります.

■**変換法**:　逆累積確率函数を用いずに, 一様分布に従う乱数に対して「**変換操作**」を行うことで必要とする確率分布に従う標本を生成する手法です. たとえば, 正規分布に対しては「**ボックス-ミューラー法 (Box-Muller's method)**」があります.

まず, U_1, U_2 を区間 $[0,1]$ での一様分布に従う分布とします. このときに二つの確率変数 X_1, X_2 をそれぞれ

$$
\begin{aligned}
X_1 &= \left(\sqrt{-2\log U_1}\right)\cos(2\pi U_1) \\
X_2 &= \left(\sqrt{-2\log U_2}\right)\sin(2\pi U_2)
\end{aligned}
$$

で定めると, X_1, X_2 は互いに独立で, 正規分布 $\mathcal{N}(0,1)$ に従う確率変数になります. このことを利用して正規分布 $\mathcal{N}(0,1)$ の標本を計算する手法がボックス-ミューラー法です.

つまり, 区間 $[0,1]$ での一様分布に従う確率変数 U_1, U_2 に対し, 標本として, u_1, u_2 を取り出します. そこから標本 x_1, x_2 を以下の式で計算すれば良いことになります:

$$
\begin{aligned}
x_1 &= \left(\sqrt{-2\log u_1}\right)\cos(2\pi u_1) \\
x_2 &= \left(\sqrt{-2\log u_2}\right)\sin(2\pi u_2)
\end{aligned}
$$

なお, 三角函数の計算が負担になる場合は, この三角函数の部分をより計算し易い式に置換えた「**マーサグリアの極座標法 (Marsaglia polar method)**」で計算できます.

■**混合分布**:　確率が複数の確率の線形和となる分布です. このときに累積分布函数は $\sum_i a_i F_i(x)$ と記述され, 係数の総和については $\sum_i a_i = 1$ となるために, 各係数 a_i が確率であると考えられます. そこで, 確率 a_i に従って累積分布函数 F_i を選択して, その F_i に従う標本を生成することで混合分布全体の標本を生成するという手法が採用できます. たとえば, 一般の離散的分布で構成される混合分布であれば, 個々の分布は逆函数法を使い, 正規分布で構成される混合分布であればボックス-ミューラー法を用いるといった塩梅です.

■**受理破却法**:　逆函数法や変換法と比べて汎用性の高い手法です. 実際, 確率密度函数 f が各点で計算可能であっても累積分布函数 F やその逆函数 F^{-1} の計算が難しい場合でも対処可能です.

■**マルコフ連鎖モンテカルロ法**:　後述のベイズ推測で用いられる事後確率では,

8.4 ベイズ推測について

8.4.1 統計的推測と実現可能性

事象族 $\mathcal{F}$ に対して確率変数 $X : \mathcal{F} \to \mathbb{R}^d$ が定義されているとき, 各事象は $\mathbb{R}^d$ 内の点に対応付けられます. また, 確率変数 X がある確率分布 $\mathcal{D}$ に従うときに $X \sim \mathcal{D}$ と表記します. ここで, $X \sim \mathcal{D}$ であるときに n 個の標本 $x_1, x_2, \ldots, x_n$ を取り出したとします. これらの標本 $x^n = (x_1, \ldots, x_n)$ を使って, 次の標本 x_{n+1} が何になるのか予測ができないでしょうか? たとえ, n 個の標本が与えられたとしても, その事象が従う確率分布 $q(X)$ が判らないことが通常ですが, 日常生活では, さまざまな局面で推測を当たり前のように行っています. たとえば, 天体の運動に無知であっても, 日の出や日没, 季節があることを当たり前のように考えています. さらにコイン投げを 10 回行って全てが表であれば次も表だろうと推測したり, 職場への所要時間をその日の天候で推測するのはいつものことです. そして, 事象を深く観察した上で推測した方が, 下駄を飛すような適当な推測よりも信頼できることは良く経験することでもあります.

では, 与えられた標本を手掛かりに, できるだけ信頼できて, しかも, 効率的な推測ができないものでしょうか. また, そういった推測に対して「**推測の妥当性**」を数値的に把握するなり, 統計的な保証があるものにできないものでしょうか? そこで統計的手法を用いて与えられた標本から真の分布 $q(x)$ を推測してみましょう. この手法は「**統計的推測**」, あるいは「**統計的学習**」と呼ばれる手法になります.

ここでの推測では, まず, 確率分布の傾向がどのようなものであるかを考え, その分布がより適正な分布になるように調整するためのパラメータ $\theta \in \Theta \subset \mathbb{R}^d$ を持った確率 $p(x|\theta)$ を考え, この $p(x|theta)$ を「**確率モデル**」といいます. そして, 確率モデル $p(x|\theta)$ のパラメータ θ を上手く調整して, $p(x|\theta_0)$ が真の分布 $q(x)$ と一致してしまえば, その期待値を計算すれば数値的な判断が行えるでしょう. とはいえ, 問題になるのはパラメータ θ をどのように調整するか, そして, 幾つかの標本しかない状態で, 確率モデルが真の分布と一致しているかどうかを, 何で, どうやって判断するかという難問がありますが, ここではこれらのことには目をつぶって議論を進めましょう.

まず, 真の分布 $q(x)$ に対し, パラメータの集合 $\Theta \subset \mathbb{R}^d$ を持つ確率モデル $p(x|\theta)$ が与えられたとします. ここで確率モデルが実情に則さない下手なモデルであれば, どのようにパラメータ $\theta \in \Theta$ を調整しても真の分布に一致させることはできないでしょう. すなわち, 確率モデルには真の分布にパラメータを調整して一致させられるものと, どのように調整しても一致させられないものの二通りが存在します. ここでパラメータ $\theta \in \Theta$ を調整して真の分布 $q(x)$ に一致させられる確率モデル $p(x|\theta)$ を「**実現可能なモデル**」といい, そうでないモデルを「**実現可能でないモデル**」といいます. さらにパラメータの集合

$\Theta \subset \mathbb{R}^d$ について,「**真のパラメータの集合**」を以下で定めます:

真のパラメータの集合

真の分布 $q(x)$ の確率モデル $p(x|\theta)$ が実現可能なモデルであるときに,

$$\Theta_{00} = \left\{\theta_0 \in \Theta : 全ての x \in \mathcal{X} に対して q(x) = p(x|\theta_0)\right\}$$

を真のパラメータの集合といいます.

ここで実現可能なモデル $p(x|\theta)$ の真のパラメータの集合 Θ_{00} は一点のみとは限りません. また, $p(x|\theta)$ をパラメータ θ で微分することを考えれば, Θ_{00} 上での微分がそれぞれ異った挙動を示す可能性も当然考えられます. このことから $p(x|\theta)$ の微分構造も考慮する必要が出てくるかもしれません. それに加えて最も重要なことは, 二つの分布 p, q が与えられたときに, それらが同じ分布であるかどうかをどのように判断すればよいでしょうか?

8.4.2　ダイバージェンス

前節で述べたように, 事象族 $\mathcal{F}$ 上の二つの確率分布 p, q が与えられたときに, それらが同じ分布であるかどうかはどのように判断すればよいでしょうか. 一つの方法として, 沢山の標本を取り出して, その度数分布が似ているかどうかで判断するという方法が挙げられますが, この方法はあまりにも安易な方法です. 実際, 類似の度合いをどのように決定するかが明瞭ではありませんね.

これが空間上の二点であれば「**距離**」を計算して, それらの違いを数値的に把握できます. この距離は次の公理を充す函数として考えられます:

距離の公理

集合 X 上で定義された函数 $d : X \times X \to \mathbb{R}$ で以下の性質を充すものを距離といいます:

- 任意の $x_1, x_2 \in X$ に対して $d(x_1, x_2) \geq 0$ を充す (正値性)
- $d(x_1, x_2) = 0$ であれば $x_1 = x_2$ であり, そのときに限る
- $d(x_1, x_2) = d(x_2, x_1)$ を充す (対称性)
- 任意の $x_1, x_2, x_3 \in X$ に対して, $d(x_1, x_2) \leq d(x_1, x_3) + d(x_3, x_2)$ を充す (三角不等式)

ここでは確率分布の違いを知りたいため,「**距離のようなもの**」で与えられた確率分布が同じかそうでないかが判断できないでしょうか? そこで, 距離の公理での集合 S を集合族 $\mathcal{F}$ 上の「**確率分布全体の空間** $\mathcal{D}$」で考えることにしましょう. そして, 真の分布 $q \in \mathcal{D}$ に対して分布 $p_1, p_2 \in \mathcal{D}$ が与えられたときに, q と一致するのは p_1 と p_2 のどちらかであるか, あるいは, より q に近い近い分布として, p_1, p_2 のどちらかを選択すべきかを判断す

ることを目的とします．ここで対象となる分布が正規分布といった正体がよく判っている分布に限定するのであれば，その分布を定義するパラメータの空間で，それらの「ユークリッド距離」を計算すればよいでしょう．ところが，分布は別に一種類だけある訳ではなく，それどころか，既知であるとは言え，さまざまな分布が混合したものかもしれません．そこで，二つの確率分布の近さを数値的に評価するために擬似的な距離として，「**ダイバージェンス (divergence)**」，「**分離度**」があります:

ダイバージェンス (分離度)

集合 $X \subset \mathbb{R}^n$ 上で定義された函数 $D: X \times X \to \mathbb{R}$ で以下の性質を充すものをダイバージェンスといいます:

- 任意の $q_1, q_2 \in X$ に対して $D(q_1, q_2) \geq 0$ を充す (正値性)
- $D(q_1, q_2) = 0$ であれば $q_1 = q_2$ であり，そのときに限る
- $x_0, x_1 \in X$ について，x_1 が $\mathbb{R}^n$ の座標で $x_1 = x_0 + dx$ と微少量 $dx \in \mathbb{R}^n$ を使って表現できるときに $D(x_0, x_0 + dx) = \dfrac{1}{2}\displaystyle\sum_{j=1}^{n} g_{ij} dx_j$ となる．ここで $G = (g_{ij})_{i,j \in \{1,2,\ldots,n\}}$ は正値対称行列である．

ダイバージェンスは距離と違って，対称性と三角不等式を充すとは限りませんが，二つの対象の離れ具合を示す量を返す性質の良い函数です．ここでは「**カルバック・ライブラの距離**」と呼ばれるダイバージェンスを導入しましょう．このカルバック・ライブラの距離は「**カルバック・ライブラのダイバージェンス**」，あるいは短く「**KL-ダイバージェンス**」とも呼ばれる二つの確率分布 $p(x), q(x)$ に関するダイバージェンスです:

カルバック・ライブラの距離

$\mathcal{X} \subset \mathbb{R}^b$ 上の二つの確率分布 $p(x), q(x)$ に対してカルバック・ライブラ (Kullback-Leiber) を次で定めます：

$$D(p||q) = \int p(x) \log \left\{ \frac{p(x)}{q(x)} \right\} dx$$

この式を書換えると次の式が得られます:

$$D(p||q) = \int p(x) \log p(x) dx - \int p(x) \log q(x) dx$$

ここで第一項の $\displaystyle\int p(x) \log p(x) dx$ は確率 p のエントロピーの逆数で「**ネゲントロピー**」とも呼ばれます．また，第二項は $\log q(x)$ の確率 p による平均であり，KL-ダイバージェンスはネゲントロピーと確率 q の確率 p による平均の差になっています．このこともあってカルバック・ライブラの距離は「**相対エントロピー**」，あるいは「**カルバック・ライブラ**

の情報量」ともいいます.

8.4.3　最尤法と事後分布最大化法

事象族 $\mathcal{F}$ 上の二つの確率分布 p, q が与えられたときに, これらの確率分布が同じものであるかどうかは, カルバック・ライブラのダイバージェンス $D(p||q)$ を計算して, その値が 0 になるかどうかで判断できます. とはいえ, 真の分布 $q(x)$ の確率モデル $p(x|\theta)$ を定めたとしても, 真の確率分布 $q(x)$ が何であるかが判らなければ KL-ダイバージェンス $D(q||p)$ が計算できないため, 残念ながらダイバージェンスを使って $q(x) = p(x|\theta)$ かどうかの判断ができません. これで振出に戻るように見えますが, 実はまだやれることがあります.

KL-ダイバージェンス $D(q||p) = \displaystyle\int q(x)\log q(x)dx - \int q(x)\log p(x|\theta)dx$ の確率分布 q のエントロピーの項は一定であり, $-\displaystyle\int q(x)\log p(x|\theta)dx$ だけパラメータ θ を動かすことで変化し, さらに, この項が最小値を取るときに $D(q||p)$ も最小値を取ります. ここで n 個の標本 $x^n = (x_1, \ldots, x_n)$ を取り出したときに, これらの標本を取り出す確率が全て同程度の $\dfrac{1}{n}$ に近いと考えることは至って自然な仮定です. そうすると $-\displaystyle\int q(x)\log p(x|\theta)dx$ は $-\displaystyle\sum_{i=1}^{n}\frac{1}{n}\cdot\log p(x_i|\theta)$ で近似できます. この式の符号を反転させて, $\dfrac{1}{n}\displaystyle\sum_{i=1}^{n}\log p(x_i|\theta)$ が最大値を取るようにパラメータ θ_0 を計算してしまえば, この θ_0 に対して KL-ダイバージェンス $D(q||p)$ は最小値を取ることになります. そうすると, この最小値を与えるパラメータ $\theta_0 \in \Theta$ を使った確率モデル $p(x|\theta_0)$ が真の分布 $q(x)$ により近く, 場合によっては真の分布 $q(x)$ と一致しているかもしれません. このことから $\displaystyle\sum_{i=1}^{n}\log p(x_i|\theta)$ が最大値を取る $\theta_0 \in \Theta$ で定まる確率分布 $p(x|\theta_0)$ はより良い $q(x)$ の確率モデルの一つとして期待できます.　この $\displaystyle\sum_{i=1}^{n}\log p(x_i|\theta)$ を「**対数尤度函数 (logarithm likelihood function)**」, そして, $\displaystyle\prod_{i=1}^{n}p(x_i|\theta)$ を「**尤度函数 (likelihood function)**」といい, 尤度函数が最大値を取る変数 θ_{ML} を求めて $p(x|\theta_{\mathrm{ML}})$ を真の分布 $q(x)$ の推測とする手法を「**最尤法**」といいます.

この方法の他にも確率モデル $p(x|\theta)$ のパラメータ $\theta \in \Theta$ の決定手法が考えられます. まず, 最初にパラメータ $\theta \in \Theta$ の分布 $\varphi(\theta)$ を考えます. このパラメータの分布を「**事前分布 (prior distribution)**」といいます. さらに特徴的なこととして, n 個の標本を取り出して, 事前分布 $\varphi(\theta)$ を使って「**事後分布 (posterior distribution)**」と呼ばれるパラメータ $\theta \in \Theta$ の分布を以下の式で計算します:

$$p(\theta|x^n) = \frac{\varphi(\theta)}{Z_n}\prod_{i=1}^{n}p(x_i|\theta)$$

ここで $Z_n = \displaystyle\int \varphi(\theta)\prod_{i=1}^{n} p(x_i)d\theta$ は $p(\theta|x^n)$ が確率分布になるための調整項ですが, この Z_n はベイズ推定で重要な意味があります.

さて, ここでは標本 $x^n = (x_1, \ldots, x_n)$ が実現しているために, 事後分布 $p(\theta|x^n)$ が最大になっていると考えることは至って自然です. そこで事後分布が最大になるようにパラメータ $\theta \in \Theta$ の調整を行います. このときに対数函数は単調増加函数であるために $\log p(\theta|x^n)$ の最大値を求めること, すなわち, $\displaystyle\sum_{i=1}^{n} \log p(x_i|\theta) + \log\varphi(\theta)$ の最大値を与えるパラメータ $\theta_{\mathrm{MAP}} \in \Theta$ を求めることに対応し, $\hat{p}(x) = p(x|\theta_{\mathrm{MAP}})$ を真の分布 $q(x)$ の推測とします. このようにパラメータ $\theta_{\mathrm{MAP}} \in \Theta$ を求める手法を「**事後確率最大化推測**」といいます.

このように真の分布 $q(x)$ が判らなくても, それなりの妥当性で真の分布の二通りの推測が行えています. ところで最尤法で求めた θ_{ML} と事後確率最大化推測で求めた θ_{MAP} にはどの様な違いや関係があるのでしょうか?

まず, θ_{ML} は対数尤度函数 $\displaystyle\sum_{i=1}^{n} \log p(x_i|\theta)$ が最大値になるパラメータであり, θ_{MAP} は $\displaystyle\sum_{i=1}^{n} \log p(x_i|\theta) + \log\varphi(\theta)$ が最大値となるパラメータです. ここで, 事後確率最大化推測で用いられる式は最尤法の式に事前確率 $\varphi(\theta)$ の項 $\log\varphi(\theta)$ が和として入っています. この項は最尤法である標本に適合し過ぎて汎用性を持たなくなる「**過学習**」と呼ばれる状態に陥いり難くするための調整項としての働きをしています.

8.4.4 SageMath による最尤法と事後確率最大化推測

ここでは簡単な例で最尤法と事後確率最大化推測の違いを確認してみましょう. ある工場では製品の製造で, 正規分布に従う指標を用いて製造管理を行っているとします. ここで正規分布 $\mathcal{N}(\mu, \sigma^2)$ は平均 μ と標準偏差 σ(分散 σ^2) に対して, その確率密度函数が $\dfrac{1}{\sqrt{2\pi\sigma^2}}\exp\left\{-\dfrac{(x-\mu)^2}{2\sigma^2}\right\}$ で一意に定まる分布です. この工場では, 品質管理に用いる指標の分布の平均が μ_0, 分散が 1 であることが判っており, 実習に来た学生に平均 μ_0 を推測させることにしました. このときに学生には分散が 1 であり, 指標の分布が正規分布 $\mathcal{N}(\mu_0, 1)$ に従うことは伝えており, 分布の推測のために n 個の標本 $x^n = (x_1, \ldots, x_n)$ が提供されているとします.

このときの対数尤度函数は

$$\begin{aligned}\sum_{i=1}^{n}\log p(x_i|\mu) &= -\sum_{i=1}^{n}(x_i-\mu)^2-\frac{n}{2}\log(2\pi)\\ &= -n\mu^2+2\left(\sum_{i=1}^{n}x_i\right)\mu-\sum_{i=1}^{n}x_i^2-\frac{n}{2}\log(2\pi)\end{aligned}$$

と μ の二次式になります．この対数尤度函数が最大になる μ_{ML} はこの尤度函数の μ による微分が 0 になることから $-2n\mu+2\sum_{i=1}^{n}x_i=0$. 以上から $\mu_{\mathrm{ML}}=\dfrac{\sum_{i=1}^{n}x_i}{n}$ を得ます. つまり, 最尤法からは標本の平均を正規分布の平均 μ とするという至極, 尤もな結果になります．つぎに確率モデルのパラメータ μ の分布を平均 0, 分散 1 の正規分布 $\mathcal{N}(0,1)$ としたとしましょう．最尤法と同じ n 個の標本 $x^n=(x_1,\ldots,x_n)$ が与えられたときの事後確率は次で与えられます:

$$\varphi(\mu)\left(\prod_{i=1}^{n}p(x_i|\mu)\right) = \frac{1}{\left(\sqrt{2\pi}\right)^{n+1}}\exp\left\{-\frac{1}{2}\left(\sum_{i=1}^{n}(x_i-\mu)^2+\mu^2\right)\right\}$$

この事後確率を最大にする μ_{MAP} は最尤法の場合と同様に次の対数尤度函数の微分の零点になります:

$$\begin{aligned}\sum_{i=1}^{n}\log p(x_i|\mu)+\log\varphi(\mu) &= -\frac{1}{2}\left\{\sum_{i=1}^{n}(x_i-\mu)^2+\mu^2\right\}-\frac{n+1}{2}\log(2\pi)\\ &= -\frac{1}{2}\left\{(n+1)\mu^2-2\left(\sum_{1=1}^{n}x_i\right)\mu+\sum_{i=1}^{n}x_i^2\right\}-\frac{n+1}{2}\log(2\pi)\end{aligned}$$

この式の微分の零点が最適な μ_{MAP} となるために, $\mu_{\mathrm{MAP}}=\dfrac{\sum_{i=1}^{n}x_i}{n+1}$ が得られます.

この計算を SageMath でもやってみましょう．n 個の標本に対応する確率変数 X_i を最初に定義することにしますが, 方針としては最初に標本の個数を定めて, その個数に応じた確率変数を定義するようにします. ここで SageMath では変数の宣言では var() を用いますが, 指示された個数の X_1,X_2,... の変数を宣言するために工夫が必要になります. たとえば, $n=5$ の場合には次のようにします:

```
sage: var(μ)
μ
```

```
sage: n = 5
sage: Xs = [var('X_%d' %_i) for _i in np.arange(1, n+1)]
sage: Xs
[X_1, X_2, X_3, X_4, X_5]
```

それから真の確率 $q(x)$, 確率モデル $p(x|\mu)$ と事前分布 $\varphi(\mu)$ を SageMath の函数として定義しておきましょう:

```
sage: q(x) = 1/sqrt(2*pi) * exp( -(x - μ)^2/2 )
sage: p(x, mu) = 1/sqrt(2*pi) * exp( -(x-μ)^2/2)
sage: phi(mu) =  1/sqrt(2*pi) * exp( -μ^2/2 )
```

Python 流に "def" を用いた函数定義もできますが, こちらの場合は SageMath 側で函数の微分や積分といった処理が行えなくなるため, SageMath 上で数式処理を行う対象であれば, ここでの函数定義の方式で函数を定義しなければなりません.

まずは n 個の標本 $x_1, \ldots, x_n$ に対し, 対応する確率モデルのリストを生成して, リストの成分の積を計算することで, 標本 $x_1, \ldots, x_n$ が得られる確率を計算しましょう. まず, 確率モデルのリストの生成では函数 map() を, リストの積を計算する SageMath の函数として "prod()" を使うことにします. この確率を使って対数尤度も計算しましょう:

```
sage: PosP = prod([p(_xi,mu) for _xi in Xs])
sage: PosP
1/8*sqrt(2)*e^(-1/2*(X_1 - μ)^2 - 1/2*(X_2 - μ)^2 - 1/2*(X_3 -
     μ)^2 - 1/2*(X_4 - μ)^2 - 1/2*(X_5 - μ)^2)/pi^(5/2)
sage: LF = log(PosP)
sage: LF
log(1/8*sqrt(2)*e^(-1/2*(X_1 - μ)^2 - 1/2*(X_2 - μ)^2 - 1/2*(
    X_3 - μ)^2 - 1/2*(X_4 - μ)^2 - 1/2*(X_5 - μ)^2)/pi^(5/2))
```

ここで示すように対数尤度は展開されてはいません. SageMath では入力した式を自動的に展開するとは限らず, その展開の方法をメソッドで指示したり, 函数を使って処理する必要があります. まず, オブジェクトに付随するメソッドとして何があるかは函数 dir() を用いると利用可能なメソッドの一覧が表示されます. ここではメソッド expand_log() を用います:

```
sage: LF.expand_log()
-1/2*(X_1 - μ)^2 - 1/2*(X_2 - μ)^2 - 1/2*(X_3 - μ)^2 - 1/2*(
    X_4 - μ)^2 - 1/2*(X_5 - μ)^2 - 5/2*log(2) - 5/2*log(pi)
```

そして, この対数尤度を変数 μ で微分し, その零点を求めましょう. ここで式の微分は diff() を用い, 一次方程式の解法では solve() を用います:

```
sage: solve(diff(LF.expand_log(), μ),[μ])
[μ == 1/5*X_1 + 1/5*X_2 + 1/5*X_3 + 1/5*X_4 + 1/5*X_5]
```

次に事後分布を計算してみましょう．ここでは真の分布を $q(x) = \dfrac{1}{\sqrt{2\pi}} \exp\left\{-\dfrac{(x-10)^2}{2}\right\}$ とし，事前分布を $\varphi(\theta) = \dfrac{1}{\sqrt{2\pi}} \exp\left\{-\dfrac{\theta^2}{2}\right\}$ として計算してみましょう．それから標本を具体的に取得しておきましょう．SageMath には probability パッケージがあるために，それを用いることにします．まず，パッケージの読込ですが，ここでは「from sage.all import *」で一切合切を読込むことにしておきます．それから 20 個の標本を取った場合と 100 個の標本を取った場合の事後確率最大化で求めたモデルのパラメータ μ_{MAP} を比較してみましょう．ここでは最初にモジュールの読込と真の確率分布 q を定義しておきましょう:

```
sage: from sage.probablity.probability_distribution import *
sage: q = RealDistribution('gaussian', 1)
sage: Xs = [q.get_random_element()+10 for _ in range(10)]
sage: Xs
[8.909879349672984,
10.704921426766283,
10.648381995353455,
9.335836199117868,
10.229285897789657,
8.134844580911487,
8.978435970138483,
6.4771937137908795,
10.039780433652219,
8.119373231883221]
```

ここで真の確率分布 q を RealDistribution() を用いて定義しています．この RealDistribution() の第 1 引数は確率分布名を指示し，第 2 引数がその分布のパラメータになります．ここでの真の確率分布は正規分布としており，正規分布のパラメータは分布の平均:μ と偏差:σ で一意に定められますが，平均が μ の正規分布は原点まわりの正規分布を μ に平行移動したものと考えられるために，本質的なパラメータは偏差 σ のみで，RealDistribution で定義する際も，第 1 引数に確率分布名として「gaussian」，第 2 引数として偏差 σ を数値で指示します．これで定めた分布からの標本の取り出しはメソッド get_random_element() で行い，この例では 10 個の標本をリスト Xs に収納しています．

つぎに μ_{ML} と μ_{MAP} の挙動を見てみましょう．ここでは指定した確率分布 q と自然数 n に対して，$4, 8, \ldots, 2^{n+1}$ 個の成分を持つ標本の列を生成する函数 getSamples() を定義し，この函数を使って μ_{ML} と μ_{MAP} の挙動を観察してみましょう:

```
sage: def getSamples(q, n):
....:     Samples = []
....:     for _i in range(n):
....:         Samples.append([q.get_random_element()+10 for _ in
```

```
    range(2^(n+1))])
....:       return Samples
....:
sage: from matplotlib import pyplot as plt
sage: XSs = getSamples(q, 13)
sage: plt.plot([2^(_i+1) for _i in range(len(XSs))], [sum(_i)/len(
    _i) for _i in
....: XSs], label="$\mu_{ML}$", marker="*", color="green")
....: plt.plot([2^(_i+1) for _i in range(len(XSs))], [sum(_i)/(len(
    _i)+1) for _i
....:  in XSs], label="$\mu_{MAP}$", marker="+", color="blue")
....: plt.axhline([10], linestyle="--", color="red")
....: plt.xlabel("# samples")
....: plt.ylabel("$\mu$")
....: plt.xscale("log")
....: plt.legend()
....: plt.show()
[<matplotlib.lines.Line2D object at 0x1623ec410>]
[<matplotlib.lines.Line2D object at 0x173b2d220>]
<matplotlib.lines.Line2D object at 0x173b59880>
Text(0.5, 0, '# samples')
Text(0, 0.5, '$\\mu$')
<matplotlib.legend.Legend object at 0x173e76e10>
```

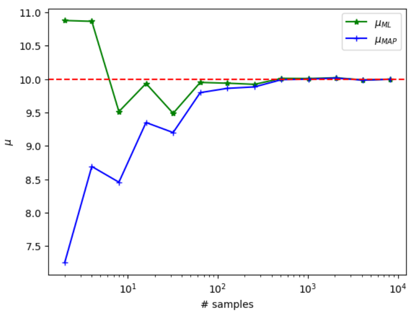


図 8.5　$\mu_{\rm ML}$ と $\mu_{\rm MAP}$ の挙動

ここでは最初に函数 getSamples() の定義を行っています．SageMath でのプログラム上の函数は Python の函数と同様です．ここではグラフ表示を行うために Python のモジュール matplotlib から pyplot を読込み, そのモジュール名を「plt」で置き換えています．SageMath で追加されたグラフ表示機能は数式処理の結果得られた式の可視化が主で, 数値計算の結果は SageMath のベースである Python の matplotlib 等の数値計算の結果を可視化するためのモジュールを用いることになります．

それから getSamples() を使って 13 成分の標本のリストを生成し, 以降で左図 8.5 に示すグラフを生成しています．このグラフからも判るように, 標本数 $\to \infty$ で $\mu_{\rm ML} = \mu_{\rm MAP}$ となることと, $\mu_{\rm ML}$ と比較して,$\mu_{\rm MAP}$ の方が $\mu_{\rm ML}$ よりも過剰に適合することもなく, ゆっくりと真の分布の平均値 $\mu = 10$ に接近することです．

8.4.5　最尤法と事後確率最大化推測の関係

つぎに最尤法と事後確率最大化推測の関係を調べてみましょう．ここで天下り的ですが，事後確率に新しいパラメータ $\beta \in (0, \infty)$ を次の式で導入します:

$$p(\theta|x^n) = \frac{\varphi(\theta)}{Z_n(\beta)} \left(\prod_{i=1}^{n} p(x_i|\theta) \right)^{\beta}$$

この式の $Z_n(\beta) = \displaystyle\int \varphi(\theta) \left(\prod_{i=1}^{n} p(x_i|\theta) \right)^{\beta} d\theta$ は $p(\theta|x^n)$ が確率になるように正規化を行うための調整項で「**分配函数**」といい，特に $\beta = 1$ のときを「**周辺尤度 (marginal likelihood)**」といいます．このように事後確率に正実数 β を導入した理由は，その対数尤度を考えると判ります．

まず，この事後確率の対数尤度は

$$\log p(\theta|x^n) = \beta \left\{ \sum_{i=1}^{n} \log p(x_i|\theta) + \frac{\log \varphi(\theta)}{\beta} \right\} - \log Z_n(\beta)$$

で与えられます．ここで β を固定して，パラメータ $\theta \in \Theta$ を動かすことにしましょう．このときに対数尤度函数を最大にする $\theta_0 \in \Theta$ は $\displaystyle\sum_{i=1}^{n} \log p(x_i|\theta) + \frac{\log \varphi(\theta)}{\beta}$ を最大にしますが，この式で $\beta = 1$ のときは前述の事後確率最大推測そのもの，$\beta \to \infty$ のときは最尤法そのものの式です．つまり，事後分布 $\displaystyle\frac{\varphi(\theta)}{Z_n(\beta)} \left(\prod_{i=1}^{n} p(x_i|\theta) \right)^{\beta}$ のパラメータ $\beta \in (0, \infty)$ は事象族 $\mathcal{F}$ 上の確率分布の集合における曲線のパラメータになっており，この確率分布の集合内の曲線は $\beta = 1$ のときが最尤法による最適解 θ_{ML}，終点の $\beta = \infty$ が事後確率最大化推測による最適化 θ_{MAP} と二つの手法の最適解を繋いでいる曲線です．このように二つの手法の最適解を通る曲線であるために，この曲線上に真の分布 $q(x)$ と一致しなくても，より良い性質の確率分布が得られることを期待しているとも言えます．これに対して最尤法と事後確率最大化推測はパラメータの集合 Θ 内の点を一つだけ選出する手法であり，このような推定方法を「**点推定 (point estimation)**」といいます．

8.4.6　ベイズ推定

最尤法が尤度函数を最大にするパラメータ θ を用いて確率モデルを決定する手法であり，事後確率最大推測は確率モデル $p(x|\theta)$ のパラメータ $\theta \in \Theta$ の分布 $\varphi(\theta)$ を導入し，周辺尤度を最大にするように θ を選択する手法です．事後確率最大推測で事前分布を導入することで，最尤法と比べて，特殊な状況に過剰に適合した結果，一般性に欠ける推測を行う

こと, すなわち, 過学習と呼ばれる現象がある程度, 避けられます.

ところで, 考察する事象から得られる標本には, その事象固有の分布に関連するバラツキがあり, さらに標本自体についても, その標本の指標となる値に導出に関するバラツキ, たとえば, 測定器に関する誤差, 観察者の特性に関わるバラツキ等がある筈です. そこで, 事後分布による確率モデルの平均を取ることで, パラメータ $\theta \in \Theta$ を消去し, これを予測分布とするとどうでしょうか? この手法では, 事前分布を標本を使って調整したものが事後分布であり, この事後分布を使って確率モデルを平均化したものが予測分布となっており, 最初の最尤法や事後確率最大化と比べると計算量は格段に増えますが, 間違ったモデルにしなければ各段での調整が可能で, 安定したそこそこの精度の予測ができそうです. この事後分布で平均を取る手法が「**ベイズ推定 (Bayes estimation)**」です.

ここでは最初に幾つかの用語を纏めて定義しておきましょう. まず, パラメータ $\theta \in \Theta \subset \mathbb{R}^d$ を使った条件付確率 $p(x|\theta)$ とそのパラメータの集合 $\Theta \subset \mathbb{R}^d$ 上でのパラメータ $\theta \in \Theta$ に関する確率 $\phi(\theta)$ の二つの確率を必要とします. ここで最初の条件付確率 p を「**確率モデル**」, そして, 確率 φ を「**事前分布 (prior distribution)**」といいます. ここで確率モデル $p(x|\theta)$ と事前分布 $\varphi(\theta)$ の組合せで真の分布 $q(x)$ を推測します. そのためには真の分布 q から取り出した標本が必要になります. ここで, $\mathbb{R}^N$ の n 個の標本を $x^n = (x_1, x_2, \ldots, x_n)$ と表記します. これらの標本が互いに独立であれば, 標本 $x^n = (x_1 \ldots, x_n)$ は確率分布 $q(x^n) = \prod_{i=1}^{n} q(x_i)$ の確率変数 $X^n = (X_1, \ldots, X_n) : \mathcal{F}^n \to \left(\mathbb{R}^N\right)^n$ の像, すなわち, 実現値として考えられます. 以降, n 個のサンプルを表現する確率変数を $X^n = (X_1, \ldots, X_n)$ と表記することにします.

次に確率モデル $p(x|\theta)$, 事前分布 $\varphi(\theta)$ と標本 x^n から逆温度 β を使った事後分布を定義します:

逆温度 β の事後分布

確率モデル p とそのパラメータの事前分布 φ が与えられたとき, 正実数 $\beta \in (0, \infty)$ に対して, 以下の式で逆温度 β の「**事後分布 (posterior distribution)**」$p(\theta|X^n)$ を定めます:

$$p(\theta|X^n) = \frac{1}{Z_n(\beta)}\varphi(\theta)\Big(\prod_{i=1}^{n} p(X^n|\theta)\Big)^{\beta}$$

ここで, $Z_n(\beta)$ は分配函数といい, $\int_W \varphi(w)\Big(\prod_{i=1}^{n} p(X^n|\theta)\Big)^{\beta} d\theta$ で与えられます.

この事後分布による確率モデル $p(x|\theta)$ の平均 $p^*(x)$ を予測分布といいます:

予測分布

$$p^*(x) = E_w[p(x|\theta)] = \int_W p(x|w)p(\theta|X^n)dw$$

この事後分布による確率モデル $p(x|\theta)$ の平均を「**予測分布 (prediction distribution)**」といいます.

ベイズ推測では, この予測分布 $p^*(x)$ が真の分布 $q(x)$ を近似するものと考えます. 予測分布 $q(x)$ は事前分布 $\varphi(\theta)$ と確率モデル $p(x|\theta)$ の他には標本 X^n のみに依存します. ここで, 事前分布と確率モデルの対 (φ, p) の選択が妥当であれば, あとは標本数を増やせばより真の分布に近い分布になり, そうでなければ, 標本数を増やしても一向に真の分布との隔が埋まらないままでしょう.

このことを確認してみましょう. ここで扱う確率分布を「**指数型**」と呼ばれるものに限定します. 一般的に指数型の確率分布はその確率モデルが

$$p(x|\theta) = \exp\left\{C(x) + \sum_{i=1}^{d} F_i(x) \cdot \theta^i - \phi(\theta)\right\}$$

の形式の確率分布です. ここで函数 $F_i(\theta), \phi(\theta)$ は $\mathcal{X}$ から $\mathbb{R}$ への函数であり, 指数型確率分布の代表例としては正規分布が挙げられます. たとえば, 1 次元の正規分布の密度函数は $\dfrac{1}{\sqrt{2\pi\sigma^2}} \exp\left\{-\dfrac{(x-\mu)^2}{2\sigma^2}\right\}$ で与えられます. ここで μ が分布の平均, σ が分布の偏差です. この式の指数部を展開して整理しておきましょう:

$$\frac{1}{\sqrt{2\pi\sigma^2}} \exp\left\{-\frac{(x-\mu)^2}{2\sigma^2}\right\} = \exp\left\{-\frac{1}{2\sigma^2}x^2 + \frac{\mu}{\sigma^2}x - \left(\frac{\mu^2}{2\sigma^2} + \frac{1}{2}\log\{2\pi\sigma^2\}\right)\right\}$$

ここで, $w = (\theta^1, \theta^2) = (\frac{1}{2\sigma^2}, \frac{\mu}{\sigma^2})$ とおくと,

$$\phi(w) = \frac{\mu^2}{2\sigma^2} + \frac{1}{2}\log\left\{2\pi\sigma^2\right\} = \frac{(\theta^2)^2}{4\theta^1} + \frac{1}{2}\log\frac{\pi}{\theta^1}$$

さらに $F_1(x) = -x^2, F_2(x) = x$ とおくと, 最終的に

$$p(x|w) = \exp\left\{F_1(x)\theta^1 + F_2(x)\theta^2 - \phi(w)\right\}$$

が得られるために, 1 次元の正規分布が指数型であることが判ります.

8.5　マルコフ連鎖モンテカルロ法 (MCMC) について

付録 A

手引 (ポルフュリオス)

A.1　概要

ここではテュロスのポルフュリオス (Πορφύριοσ) のエイサゴーゲー (Εἰσαγωγή) の翻訳を載せます. このエイサゴーゲーはアリストテレス の論理学 (特に「範疇論」) の入門書として書かれたもので, その題名の εἰσαγωγή はギリシャ語で「手引 (Introduction)」を意味し, この文書の一節から中世の「普遍論争」が発生したことでも知られています.

エイサゴーゲーはギリシャ語で記述された哲学の入門書で, 西ヨーロッパではラテン語への翻訳「イサゴーゲー (Isagoge)」[48] で知られています. ラテン語への翻訳は「ギリシャ語を理解する最後のローマ人」と呼ばれ,「哲学の慰み」等の著作やギリシャ語の哲学書の幾つかをラテン語に翻訳したことでも知られるボエティウス (Boethius) が行なっています. また, ビザンツ帝国ではその要約が知られ, シリア語, アルメニア語やアラビア語にも翻訳されています. そしてこの本は哲学を学ぶ上で最初に読むべき本とされ, 19 世紀末でも中近東では教科書として使われていたといいます.

このエイサゴーゲーの英語への入手し易い翻訳はオーエン (Owen) がラテン語訳から翻訳したものとバーンズ (Barnes)[45] がギリシャ語文献から翻訳したものの二つが代表的でしょう. オーエンの訳は 19 世紀半ばの翻訳で, WEB 等で公開[*1]されていたり, アリストテレスのオルガノンと一緒に併せて廉価で売られていたりします. こちらの章立てはイサゴーゲーと同一で, エイサゴーゲーがアリストテレスの「範疇論」の入門書であるという立場で翻訳されたものです. それに対してバーンズの翻訳はエイサゴーゲーがアリストテレスの「範疇論」だけに限定した入門書ではなく, むしろアリストテレスの論理学 (オルガノン) への入門書として捉えており, それに加えてギリシャ語文献の解釈や歴史的背景を含む詳細な解説, それに加えて原文のギリシャ語と英語の単語の対照もあって, より深く調べるためにはバーンズの翻訳が良いでしょう. なお, ここでの訳はバーンズの訳と註

[*1] http://www.ccel.org/ccel/pearse/morefathers/ files/porphyry__isagogue__02__translation.htm

釈を中心にオーエンの訳も必要に応じて参照して翻訳しています．

著者ポルフュリオスについて現代に伝わっている情報は意外にありません．まず，ポルフュリオスはフェニキュアのテュロスで 234 年に生まれ，シリア語で王という意味で Malcus と名付けられています．このことからのちにギリシャ語で「王」という意味のバシレウス (Βασιλεύσ)[*2] と呼ばれていたようです．さらに出身地のテュロスが紫色の染料 (Tyrian Purple) の生産で有名でしたが，この紫色がローマ皇帝の色であったこともあって「**ポルフュリオス**」という渾名[*3]が付けられています．ポルフュリオスはアテナに留学し，そこでは生き字引，歩く博物館と呼ばれたロンギノス (Λογγινόσ)[*4]に修辞学，数学と哲学を学んでいます．ちなみに，ポルフュリオスという渾名を付けたのがこのロンギノスのようです．それから 263 年にローマに行って新プラトン主義[*5]の創始者として知られるプロティノス (Πλωτῖνοσ, Plotinus) の一門に入り，プロティノスから大きな影響を受けます．そして 268 年にプロティノスの勧めもあってシチリア島に行き，270 年にプロティノスが死んだためにローマに戻って哲学を教え，301 年にはプロティノスの著作「エネアデス (Enneads)」の編纂を行っています．それから北アフリカに行ったらしく，それからマルセラ (Marcella) という女性と結婚したことがポルフュリオスが妻のマルセラに宛てた手紙 [57] [*6]から伺えますが，彼が何時，何処で死んだといった伝承が残っておらず不明です．なお，ポルフュリオスは哲学の一派を作ったり，指導者であったこともなかったようで，エイサゴーゲーの他にも幾つかの著作が残されていますが，エイサゴーゲーほど後世に影響を与えたものは他にありません．

新プラトン主義の創始者とされるプロティノスから強く影響を受けたことからも判るようにポルフュリオスは新プラトン主義の哲学者です．その彼が記述した手引書であるエイサゴーゲーが (新プラトン主義の) 哲学を学ぶ上で入門書として位置付けられたということは，後世のアリストテレスの哲学の受容に大きな影響を与えることになります．まず，アリストテレスは彼の著作である「形而上学」を見て判るようにプラトンのイデア論に批判的ですが，ポルフュリオスはこのエイサゴーゲーでアリストテレスの哲学を新プラトン主義と無理のない形で結合させるという荒業をやってのけています[*7]．まず，ポリピュリ

[*2] 東ローマ帝国では皇帝を意味します．

[*3] バーンズの本 [45] の表紙が紫色なのもこの洒落でしょう．

[*4] パルミラ王国の女王ゼノビア (Zenobia) の側近として仕え，ローマ帝国によるパルミラ陥落後に，ゼノビアから責任を全て押し付けられて処刑されています．

[*5] プラトンのイデア論を継承しつつ，絶対的な一者を想定し，万物はその一者からの流出として捉える「流出論」を特徴とする哲学思想です．

[*6] マルセラはポルフュリオスと結婚した時点で 5 人の娘と 2 人の息子の子持で，その子供の幾人かは幼ないものの，他は結婚適齢期を迎えていたようです．その手紙によると結婚した理由も，ポルフュリオスが子供が欲しい訳でも妻に世話してもらって楽をしたかった訳でもなかったと述べてますが，その手紙を読み進めてゆくと，どうやら妻もそれなりに哲学の愛好家であったことが判ります．

[*7] 詳細については「カテゴリー論」[2] の註を参照して下さい．このようなことができたのもポリピュリオスが師のプロティノスよりもアリストテレス寄りの考えを持っていたことも関係するようです．

オスはプラトンとアリストテレスが思想的に対立するものではなく調和的な関係にあるとし, プラトンを理解するためにアリストテレスを学ぶという基本方針を定めました. このことからポルフュリオス以後の新プラトン主義の哲学教育はエイサゴーゲーから教育を開始することになります. この点はプラトンの著作にもともと宗教的な側面があることに加え, アリストテレスの著作が「難解」であることは「秘儀の漏洩を防ぐ」ためと考えられたこと[*8], それと新プラトン主義のライバルであるストア派[*9]と対抗するためにアリストテレスの著作を使って体系化する必要があったと考えられています. このような経緯もあってアリストテレスの著作や注釈は新プラトン主義的な解釈等の夾雑物を含むことになります. なお, 西ヨーロッパでアリストテレスの著作は前述のボエティウスによるラテン語訳の「範疇論」と「命題論」のみが伝わった程度ですが, 一方の中東ではギリシャ語からシリア語へと, さらにアラビア語へと翻訳されます. その過程でアラビア哲学の基礎を作ったキンディー (al-Kindi) がさらにアリストテレスの哲学と新プラトン主義の思想と融合さます. 結局, 12 世紀になってイブン・ルシュド (ibn rušhd, ラテン語名: アベロエス (Averroes), または「注釈者」の名前で西ヨーロッパで 知られています. たとえば, トマス・アクイナスの「形而上学序説」では単に「注釈者」と呼んでいます.) が「**純正アリストテレス**」を唱え, その夾雑物を排除しようとするまで新プラトン主義の影響下での解釈が続くことになります. そして, イブン・ルシュドのアリストテレスの注釈を基に中世ヨーロッパの神学とその下僕としてのスコラ哲学が完成 (トマス・アクイナス (Thomas Aquinas) の「神学大全」等) されることになります.

このエイサゴーゲーをポルフュリオスが著述することになった契機はポルフュリオスがシチリア島に行ってしばらくローマを留守にしたことです. このシチリア島滞在については次の伝承があります. 前述のようにローマでプロティノスと暮らすうちにポリピュリオスは自殺したくなる程の憂鬱に陥ってしまいます. そこでプロティノスの勧めもあって保養のためにシチリア島に行ったという伝承です. これとは別の伝承ではエトナ山の火焔の調査のためとも伝えられています[*10]. いずれにせよ彼はしばらくローマを留守にしていたことになりますが, ちょうどその頃, ローマでの弟子のクリュサオリオス (Chrysaorius, ローマの元老院の議員) がアリストテレスの著作を読み始めたもののさっぱり判らないために, ポリピュリオスにローマに戻って彼を指導するか, それができないなら手引を書いて送って欲しいとポルフュリオスに依頼します. そこでポリピュリオスはシチリア島で

[*8] ある意味で深読みのし過ぎですが.

[*9] ヘレニズム哲学の一学派でキティオンのゼノン (Ζήνων) が創始者. アカデメイア学派, 逍遥学派, エピクロス派と並ぶ四大学派の一つ. 五賢帝の一人のマルクス・アウレリウス・アントニヌスも信奉者の一人であったように, 古代ローマ共和制末期から帝政期初期にかけて最も有力な哲学学派でした. ストア学派の論理学は断片しか伝わっていませんが, アリストテレスの名辞論理学と異なる命題論理学です.

[*10] このアンモニオス (Ἀμμώνιοσό ῾Ερμείυ, Ammonius) による伝承はエトナ山の火口に飛び込んで自殺したといわれる哲学者エンペドクレス (Ἐμπεδοςλήσ, Empedcles) や, その噴火を船で観察したというプラトンを思い起こすものとなっています.

268-270 年の間に「エイサゴーゲー」を記述し, これをクリュサオリオスに送付したと伝えられています.

エイサゴーゲーはアリストテレスの論理学の短い入門書 (A4 で 20 ページ未満) ですが, その後世への影響は非常に大きなものがあります. その理由に, 西ヨーロッパと古代のヘレニズム文明には大きな断絶があり, イサゴーゲーはこの断絶を超えて西ヨーロッパに伝わった数少ないヘレニズム哲学の文献の一つであるためです. もともと質実剛健の言葉が似合う古代ローマも五賢帝の時代にはギリシャ文明が帝国を席巻し, 知識人はラテン語だけではなくギリシャ語も当然のように使っていました. ところが, 時代が下ってエイサゴーゲーのラテン語訳を作成したボエティウスの時代になるとローマ帝国の本体はすでに東に移動してギリシャ化し, 一方の西ローマ帝国の故地は蛮族が割拠し, 言語的にも文化的にも東西に分断されつつありました. そして, 旧西ローマ帝国の領域ではギリシャ語を理解する者が少数派になった状況下でボエティウスがエイサゴーゲーを含む幾つかのギリシャ語で記述された哲学文献のラテン語への翻訳や, エイサゴーゲーの二つの注釈も著述し, これらの著作が西ヨーロッパに残されたヘレニズム哲学の基本的文献として西ヨーロッパの哲学が発展することになります. なお, ボエティウスは「範疇論」と「命題論」の他のアリストテレスの著作もラテン語に翻訳していますが, これらの翻訳は失なわれ, アリストテレスの本格的な受容はアラビア経由になります. エイサゴーゲーが西ヨーロッパに残った理由としては, 新プラトン主義の哲学を学ぶ上で, まず最初に読むべき本とされていたこと, それに加えてコンパクトな入門書であったことが挙げられるでしょう. このエイサゴーゲーの後世への影響としては前述のようにアリストテレスがプラトンと調和するという立場に加え, その第一章で類, 種, 差異, 特有性と偶有性という 5 つの事象を述べて, 以降の章ではそれらの解説と比較を行っていますが, この類, 種, 差異, 特有性と偶有性という分類が後世では一般的になっていること, それと類を種に分解する過程から「**ポルフュリオスの樹 (Arbor Porphyrianae)**」と呼ばれる系統学で用いられる図式が導びかれています. ちなみに類と種の関係をポルフュリオスは上位の類と下位の類と階層的に捉えています. また正確には前述のボエティウスの第一注釈によるものですが, 中世スコラ哲学の歴史的な論争で著名な「**普遍論争**」[*11]の火種になっています.

[*11] 「**普遍が存在するか?**」という問に関する論争で, 存在するという立場は「**実在論**」, そうでないという立場を「**唯名論**」という大雑把な分類ができます. もっとも, 実際はこう単純に切り分けられるものではなかったようです (詳細は [39] を参照.

A.2 エイサゴーゲー翻訳

A.2.1 はじめに

必要なこととして, クリュサオリオス (Chrysarorius)[*12]よ, まず, アリストテレス (Ἀριστοτέλησ, Aristotle) の述定[*13]について学ぶためには, 類 (γένοσ, genus) [*14] が何で, 差異 (διαφορά, difference) が何であり, 種 (εἶδοσ, species)[*15]が何であり, 特有性 (属性, ἴδιον, property) と偶有性 (付帯性, συμβεβηκόσ, accident) が何であるかを知っておくことがあります[*16] - そしてまた定義の論証, それと一般的な (類の種への) 分割や証明に関わる事象にとっても, これらの研究が有用なために - 私は, あなたに要所を押さえて説明するときに, 手短に入門の形式をとって, 古の賢者達[*17]が述べたことを, より深い探究を排除し, そして (あなたにとって) 適切でより単純になるようにおさらいをしようと思います[*18]. だから, 類と種については- それらが存在するものかどうか, それらが実際にそのままの思考にだけ依存するものなのかどうか, もし, それらが存在するのであれば, それらは物体 (σώμα, body) を持つものなのか, それとも非物体のもの (ἀσώματοσ, incorporeal) なのか, そして, それらは離在可能 (χωριστόσ, separable) なものなのか, あるいは明瞭に知覚できるものの中にあって, それらに関わって存在するものなのか - こういったことの議論を私は避けようと思います[*19]. というのも, このような事象は非常に深

[*12] ポルフュリオスの弟子. ローマの元老院の高位の議員だったらしく, 彼がアリストテレスの著作を読むための手引をシチリア島に滞在していたポルフュリオスに求め, それに応じて書かれた文書がこの「手引」です.

[*13] オーエンはここを「Category」と訳していますが, バーンズは「Predication」と訳しており, 翻訳の立場の違いが出ています.

[*14] バーンズ [45] によると γένος は「族」,「種類」や「型」といった意味で, 後述の「種」と訳語があてられている εἶδος も「型」, [族」,「種類」とほぼ同じ意味で, これらは混合して記述されることが多い言葉です.

[*15] εἶδος は一般的に形相 (form) と訳され, ものの形を意味します. ボエティウスはラテン語で種に対応する意味のものを species と形相の意味に対応するものを forma と分けて訳しており, これが英語の species と form に対応しています. ちなみにプラトンはイデアをἰδέα と εἶδος の双方を用い, アリストテレスは εἶδος を用いています.

[*16] この類, 種, 差異, 特有性と偶有性の 5 項目はポルフュリオスによります. なお, アリストテレスはトピカ (Τόποι, Topics) で定義, 特有性, 類, 偶有性の四つを挙げていますが, 後世ではポルフュリオスによるこの五つの事項が用いられています.

[*17] プラトン, アリストテレスや逍遥学派の哲学者達のことです. ちなみにその当時のモダンな賢者達はストア派の哲学者や師のプロティノスになるでしょう.

[*18] バーンズ [45] は以上の記述からエイサゴーゲーがアリストテレスの論理学全般の入門書だと主張しています. 実際, そう考えた方が文脈上良いように思えます

[*19] ボエティウスのイサゴーゲー第二注解 [35] でこの一節を取り上げたことが契機になって中世ヨーロッパで「**普遍論争 (the problem of universe)**」が生じる原因になった歴史的に意義のある一節です. なお, ここで実在を明言しないということはポルフュリオスが正真正銘の新プラトン主義者であったとは言えない側面を見せているように思えます. というのもプラトン主義者であればイデアの非実在を言う筈がないためです.

淵で, 他の物事やより広範囲の探求を必要とするためです. ここで私はあなたに古の賢人達 - 中でも殊に逍遥学派の人々 (Περίπατοσ, Peripatetic) - が論理学の視点からどのように, 類や種等を我々以前に扱ったかを示したいと思います.

A.2.2　類について

類 (γένοσ, genus) も種 (εἶδοσ, species) も, 実のところ, 一通りの方法でそう呼ばれている訳ではありません. 事実, 我々は一つの事象や互いにとにかく関係する人々のあつまりを類と呼びます. ヘーラクレース一族 (Ἡραλείδαι, ヘーラクレイダイ) という類は, この意味で, ある一つの事象 - 要するにヘーラクレース - との彼等の関係からそう呼ばれ, 互いにとにかく関係する多数の人々は, 他の類と相互に区別するために, 彼 (ヘーラクレース) に由縁のある婚姻関係から彼等の名前を持っています. また, 別の意味で我々が類と呼ぶものは, 各人の出生の起源, それは先祖であったり, 生まれた場所であったりします. その意味で, 我々はオレステース (Ὀρέστησ, Orestes) はタンタロス (Ταντaλόσ, Tantalos) からの類[*20], ヒュロス (Ὕλλοσ, Hyllus) はヘーラクレースからの類と言えます[*21]; それから再び, ピンダロス (Πινδαροσ, Pindar) は類としてテーバイ人[*22], プラトン (Πλάτων, Plato) はアテーナイ人[*23]と言えます - と, このように祖国は各人の出生の, ちょうど, 父親のような, 起源の一種になります. この意味付けは理解し易いものでしょう; と申すのも, 我々がヘーラクレース一族と呼ぶのはヘーラクレースの類からの子孫, ケクロプス一族 (Ceropids, ケクロダイ) はケクロプス王 (Κέκροφ, Cecrops) [*24]と彼等の血族からです. まず第一に, 各人の出生の起源が類と名付けられます; そのあとで, 単一の起源 (たとえば, ヘーラクレース) に由来する人々の多数, それを区別し, その他から切り分けることで我々はヘーラクレース一族のあつまり全体を類と呼びます. また, 別の観点で我々が類と呼ぶのは, その下に種が整理され, 先行する事象との疑いない類似性のためです; というのも, そのような類はそれの下にある事象にとって一種の起源で, さらに大多数はその下の全てを包含させられているためです.

このように類というものはその三つの方法で呼ばれています; そして, 第三のもの, それが (逍遥学派の) 哲学者たち向けの理論です. その理論の下描きをすると, 彼等は類が ‘それが何であるか?’ に対する回答で, 種において異なる幾つかの事象を述定 (κατηγορεῖν)[*25]

[*20] オレステースはミュケーナイの王アガメムノーンの息子, タンタロスはリュディア王でオレステースの先祖になります.

[*21] ヒュロスはヘラクレスの息子です.

[*22] ピンダロスはテーバイ生まれの詩人です.

[*23] プラトンは言うまでもなく哲学者のプラトンです.

[*24] ケクロプス王はアテーナイの伝説的な初代の王です.

[*25] 範疇 (カテゴリー) の由来で, カテゴリーとは述語付けの分類です. この κατηγορεῖν という用語は本来, 責を負わせるという法律用語でアリストテレスが哲学に導入した言葉です.「述語付け」と言えますが, ここでは「カテゴリー論」[2] の註にしたがって「述定」と訳します.

していることだと語ることで, それを表現しています[*26]; たとえば, 動物がそうです.

述定ということについては, あるものはただ一つの事象 - いわゆる個体 (たとえば, ソクラテスや ‘この人’ や ‘あのもの’) が語られ[*27], またあるものは幾つかの事象 - いわゆる類, 種, 差異, 特有性や偶有性 (これらは何かを固有ではなく共通で保持するもの) が語られます. たとえば, 動物は類であり; 人間は種であり; 理性的であるということは差異であり; 笑うことができるということは特有性であり; そして, 白色, 黒色, 座っているということは偶有性です[*28].

類は幾つかの事象を述定するものであって, ただ一つの事象だけを述定するものとは異なります. また, それら (類) は幾つかの事象を述定するもの - 種とも異なります. なぜなら種は, たとえそれらが幾つかの事象を述定していたとしても, 種ではなく数で異なる事象[*29]を述定しているためです. だから人間は種であるために, ソクラテスやプラトンといった, 種ではなく数で互いに異なる人を述定し, その一方で, 動物は, 類であるために, 人間や牛や馬といった, 数だけではなく互いに種が異なるものを述定します. また, 類は特有性と異なります. なぜなら特有性はただ一つの種 - それを特有性とする種 - それとその種の下にある個体を述定するためです (笑うことができるということは人間だけ, ことに人間の述定です) が, その一方で類は一つの種ではなく幾つかの異なる種を述定します. また, 類は差異とも共通の偶有性でも異なりますが, というのも差異と共通の偶有性は, たとえそれらが種が異なる幾つかの事象を述定していても, ‘それが何なのか?’ に対する回答でそれらを述定するものではなく, むしろ, ‘それがどういったたぐいのものなのか?’ に対する回答になります. 人間がどういったたぐいのものなのかと問われると, 我々は理性的であると言います; それからカラスがどういったたぐいのものであるかと問われれば, 我々はそれが黒色のものだと言います - ここで理性的であるということが差異で, 黒色ということが偶有性です. しかし, 我々が人間とは何であるかと問われたときに, 我々は動物と答えます - そして動物は人間の類です.

[*26] アリストテレスは形而上学の Δ 巻「哲学用語辞典」で類の定義として四つの方法を列挙しています:「同じ形相を持つ事物の連続的な生成の存するもの」,「あるものの事物の存在がそれに由来する所の第一の動者」,「平面がさまざまな平面図形の類といわれる意味」と「 その物事の ‘何であるか’ という本質を表すもの」です. ポリピュリオスはアリストテレスの言う「平面がさまざまな平面図形の類と言われる意味」について明瞭に述べていませんが, 最初に述べていることがおおよそ対応するでしょう. それと連続的な生成や第一の動者は二番目に述べていることが対応するでしょう. そして, ポルフュリオスがここで述べている ‘それが何であるか’ に対する回答で定まる類はアリストテレスの言う第四のものです. なお, ポルフュリオスは「表現する」と述べていますが, このことは類や種といったものを, その存在はさておいて「言語的」なものとして主張し, プラトンとアリストテレスの差異が目立たなくなっています ([2] の註を参照).

[*27] ここで「語る」ということは,「X は Y である」と述語付けること, すなわち主語 X に対して「説明規定を与える」ことです

[*28] ポルフュリオスは述定を類, 種, 差異, 特有性と偶有性の 5 項目に分けて述べています. また, ポルフュリオスはアリストテレスの範疇 (カテゴリー, 最高位の類) が 10 種類あることをのちに述べています.

[*29] これはアリストテレスの用語で, 個体については「数において異なる」と述べています.

だから, それら (類) で幾つかの事象が語られるという事実が, ただ一つだけの個体を述定するものから類を区分します; それら (類) が種で異なる事象を語るものであるという事実が, 種や特有性として述定するものをそれら (類) から区分します; そうして, それら (類) が 'それが何なのか?' に対する回答で述定するという事象が, 'それが何なのか?' ではなくむしろ 'それがどういったたぐいのものなのか?', あるいは 'それが何に似ているか?' ということに対する回答に対して述定する差異や共通の偶有性からそれらを区分します. と, ここで述べた類の素描については, 何らの過剰も何らの不足もありません.

A.2.3　種について

我々が種 (εἶδοσ, species) と呼ぶものは, 第一に, あらゆるもののものの形 (εἶδοσ, form)[*30]です - それはこう言われています:

ことのはじめにかの姿 (εἶδοσ) は王国統治に相応わしきものであれ... [*31]

我々はまた前述のもののあつまりで, とある類の配下にあるものを種と呼びます[*32] - 我々が, 動物を類とするときに, 人間を動物の種として, また白色を色の種として, 三角形を図形の種として呼び慣わしているようにです[*33].

もし, 類というものを表現するときに我々が種に言及して (我々は類を 'それは何なのか？' に対する回答で, 種が異なる幾つかの事象を述定するものだと述べました.), また種とは類の配下にあるものであるとここで言うのであれば, 類はある何かの類であって, 種はある何かの種であるから, 両者の説明規定で双方を利用することが必要であるということを実現するものでなければなりません.

そこで, 彼ら (逍遥学派) は種を次のように表現します: 種は類の下で整理されたものです; それから: 'それは何なのか?' に対する回答で類が述定するものです. それからま

[*30] 前述のように種は原文では εἶδος で, この εἶδος の意味は「ものの形」です. ちなみにプラトンはエイドス εἶδος とイデアἰδέα を区別せずに用いていますが, アリストテレスはプラトンのイデアの意味でもっぱら εἶδος を用いています. このように混同して使われるのも本来の意味に「**ものの形**」の意味があるためです. なおポルフュリオスは外観を μορφή(shape), 姿を σχῆμα(figure) と区別して記述していますが, 種 εἶδος はその双方の意味を含み, さらにはものの本質を含む意味でもあります. たとえば εἶδος が白いミルクならば, μορφή は白く塗ったものが対応します. つまり, μορφή が表面的なものであるのに対して εἶδος は奥行があって, さらには本質的なものを意味します.

[*31] 悲劇作家エウリピデスの失われた悲劇「アエオルス (Aeolus)」の一節とのことです. アエオルスはティレニア海の王で風の主です. そして 6 人の息子と 6 人の娘の父親でした. ところで息子の Macareus が娘の Canace に恋してしまい, そうして... といかにもエウリピデスらしい悲劇らしいのですが, 残念ながら断片のみが残っているだけです.

[*32] 第二の意味として,「種」には「類」の配下にあるもの, つまり, 類 - 種という階層があることを述べています.

[*33] ここでの例で人間は本質的存在ですが白色と三角形は本質的存在ではなく質 ('どのようなものか' に対する回答) です. つまり種になるものは本質的存在ばかりではないということを述べています. また, 類は種により, 種は類によるという循環的な定義になっています. なお, ストア派では「種は類に含まれる」,「種は概念」であって, 類と種が互いに言及し合うという循環性がありません.

たこのように: 種は, ‘それは何であるか’ に対する回答で, 数で異なる幾つかの事象を述定します - ところが, これでは最も特定的 (εἰδικόσ, special) なものとただ一つの種の表現になり, そうでなければ他のものが最も特定的でないものにも対応することになるでしょう[*34].

私が言っていることは次で明瞭になるでしょう. それぞれの述定の型 (範疇, カテゴリー)[*35]には, 最も綜合的 (γενικόσ, general) な事象があれば, 他に最も特定的 な事象もあります; そして最も綜合的なものと最も特定的なものの間には他の事象もあります. 最も綜合的なものにはそれよりも上位の事象がありません; 最も特定的なもの, このうしろにはそれよりも下位の種がありません; それから最も綜合的なものと最も特定的なものの間には同時に複数の類や種があります (とはいえ, あるものものと他のものとの間に関係を持たされています)[*36].

私が言っていることは単一の述定の型 (範疇, カテゴリー) であれば明瞭になるでしょう. 本質的存在 (οὐσία, substance) はそれ自体が類になります[*37]. その下に物体 (σῶμα, body) があり, 物体の下で魂のある物体があり, その下に動物が; 動物の下に理性的動物があり, その下に人間があり; それから人間の下にソクラテスやプラトンや特定の人々がいます[*38].

[*34] ‘それが何であるか’ という問いに対し, それはソクラテスという個人, あるいは人間という種の二通りの回答が考えらます.

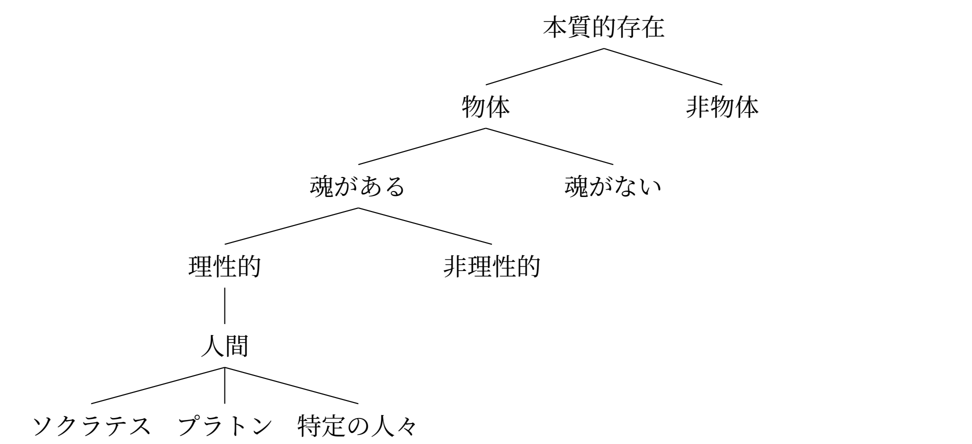
[*35] バーンズは ‘type of predication’, Owen は ‘Category’, そして Boethius は ‘praedicamento’ と「範疇」と同じ意味で訳しています.

[*36] このように綜合的と特定的はちょうど対称的な関係で,「X が Y よりも総合的である」であれば「Y は X よりも特定的である」と言えるものです. そして中間的なものとは「X は Z よりも総合的である」かつ「Z は Y よりも総合的である」ときに Z が中間的なものになります. そして, この中間的な事象は「下位の類」とも呼ばれます. このように述定には階層があり, この点はラッセル (Russell) の導入した型理論から見ても面白いものです. まず, フレーゲの函数概念では主語と述語の関係はもはやなく, 項の位置だけが問題になり, 函数それ自体よりも, 函数に項を入れることでできた命題に視点があります. つまり函数は変数項に値を入れることで一つの命題を生成することのできるというフレーゲの言うところの「飽和されるべきもの」です. それに対してラッセルの型理論でふたたび主語と述語の関係を項と函数それ自身に切り分けた階層として復活させています. また, フレーゲは真や偽を概念の外延として定義していますが, アリストテレスは真や偽は命題の状態を示すもので,「存在するものを存在しないと言い, あるいは存在しないものを存在すると言うこと」が偽で,「存在するものを存在すると言い, あるいは存在しないものを存在しないと言うこと」が真 ([3]11b27) とし, 具体的な対象ではなく状態を示す値として導入しており, フレーゲの論理主義が陥いる大きな陥穽 (命題に外延が存在するとは限らないこと) を結果として上手く避けています!

[*37] 本質的存在 (実体) の内の一つです.

[*38] これを図示したものがいわゆる「**ポルフュリオスの樹**」です. なお,「ポルフュリオスの樹」という言葉自体は 6 世紀の Sergius of Reshana の著作まで遡ることができるようですが, その実体は現在描かれているものとは違うものであるとのことです. また, その他の古代の註釈者は「鎖」,「線分」等と呼んでいるとのことです [45].

ポルフュリオスの樹



これらの事象について, 本質的存在が最も綜合的で, ただ一つだけの類であり, 人間は最も特定的で, 一つの種に過ぎません. 物体は本質的存在の一つの種で, 魂のある物体の類です. 魂のある物体は物体の一つの種で, 動物をその種とする類です. また, 動物は魂のある物体の一つの種で, 理性的動物の類です. 理性的動物は動物の一つの種で, 人間の類です. 人間は理性的動物の一つの種ですが, 特定の人々の類ではありません - 単なる種です.

個体 (ἄτομοσ, individual) [*39]に先行して存在する全ての事象は単なる一つの種であって一つの類ではありません. だから, ちょうど本質的存在が, いかなる類もその前に存在しないような上位のものであり, 最も綜合的な事象であるかのようにです. そして人間は, そのうしろに他のどのような種もなく, ただ個体だけ (ソクラテスやプラトンは個体のために) を除いて本当に何物にも分割され得ない, ただの一つの種であり, そして (個体に) 近接 (προδεχέσ, proximate) [*40]する種でしかなく, そして我々が語ったように, 最も特定な事象です. 中間的な事象というものはそれらに先行して存在するある事象の種になり, そして, それらのうしろにある事象の類になります. だから, 二つの関係が並立し, その一つはそれらに先行して存在する事象に対する関係 (それらがそれらの種であると語られます), もう一つはそれらのうしろの事象に対する関係 (それらがそれらの類であると語られます) です. その両端は単一の関係を持ちます. というのも綜合的な事象はそれの下にある事象との関係を持つために, それら全てのものの類であれば, それの前にある事象と関係を持つことがなく, 最も最上位で第一の起源であれば, 我々が語ったように, それよりも上位の類はありません. そして, 最も特定な事象は単一の関係を持ち, それに先行して存在する事象に対するものがその一つで, それは一つの種であって, そのうしろにある事象との関係を持ちません. 実際, それもまた個体の種と呼ばれます- しかし, 個体をそれが包含

[*39] 不可分のものという意味. 原子 (atom) の語源です.
[*40] προδεχέσ(proximate) は間に中間的な種等が入らないという意味です.

する限り, それは個体の種であり, それに先行して存在する事象でそれが包含される限り, それらの種です.

だから彼 (逍遥学派) らは最も綜合的なものをこのように区別します: それは類であっても種ではなく; それから再び; それの上に他の上位の類はありません. 最も特殊なものは: それが種であっても類ではなく; そして: 種であるとすれば, 我々は再び種に分割できず; それから: ‘それは何なのか？’ に対する回答で, 数で異なる幾つかの事象を述定します. その両端の間の中間のものを彼等は下位の類や種と呼び, そして彼等はそれらの内の個々が種や類であると断定します (ただし, ある一つのものや相互に関係を持たされたものとしてです). 最も特殊なものの前にあって, 最も綜合的なものへと遡る途上にある事象は, 類や種や下位の類であると語られます.

*** アガメムノーン (Ἀγαμέμνων, Agamemnon) [*41]はアトレウス (Ἀτρεύσ, Atreus) の子供という類, それからペロプス (Πέλοφ, Pelops) の子孫という類, そしてタンタロス一族であり, 最後にはゼウスのそれとなるようにです. しかし, 系図であれば, その大半は起源を遡ると単一の個人 - 要するにゼウス - になりますが, 類や種の場合はそうではありません. というのも, そういった存在するものが全ての共通の類でないどころか, ある単一の最上位の類によって全てのものが同じ類に属するということではありません - ちょうどアリストテレスが主張するように[*42]. (アリストテレスの) 範疇論で提起されているように, 第一の類は 10 種類 - 10 個の第一の根源 (ἀρχέ, origin) になります[*43]. だから, たとえ, あなたが全てを実在と呼んだとしても, おそらく, あなたはそうするでしょうが, 彼 (アリストレス) はこう言っています, 同名異義的であっても同名同義的ではないと. というのも, その存在が全てに共通する単一の類であったとすると, 全てのものは同名同義的な存在であると語られます. しかし, 第一の事象が 10 個あり, それらはただ名前を共通に持ち, さらに名前に対応する説明規定がありません[*44].

最も綜合的な事象は, つまり, 10 個です; 最も特有的なものはある一定の数になりますが, 無限個のものでありません; 個体 - その事象は最も特有的な事象のうしろにあると言うべきですが - は無数にあります. それが, 最も綜合的な事象から最も特定的な事象へ降

[*41] ミノス王アガメムノーンは前述のようにタンタロスの子孫ですが, その父はアトレウス, その祖父はペロプス, そして曾祖父がタンタロス, タンタロスはというとゼウスの息子という説があります.

[*42] プラトンやストア派の哲学者は万物に共通する類を想定していたようですが, アリストテレスはそれに反論しています. 彼の範疇論は後述のように 10 種類に最高類 (カテゴリー) を分類しています

[*43] アリストテレスによる最上位の類, つまり範疇 (カテゴリー) の分類のことです. アリストテレスは最上位の類を範疇 (カテゴリー) と呼び, それらを本質的存在, 量, 性質, 関係, 場所, 時間, 態勢, 所有, 能動, 受動の 10 種類に分類しています.

[*44] 範疇論でアリストテレスは「同名異義的」と呼ばれるのは名称だけが共通であり, その名称に対応した事象の本質を示す説明規定が異なるもの,「同名同義的」とは, その名称が共通であり, その名称に対応した事象の本質を示す説明規定も同一であるものと述べています. この場合は一つの類があって名前が一つだとしても, 範疇が 10 種類あり, それぞれの説明規定が異なるために「同名異義的」になると主張しています.

下しようとする人にそこで止めて, それから特徴的な差異で中間物を分類しながら, それらを経由して降下すべきだとプラトンが忠告した理由です; それから, 彼 (プラトン) は我々に無限はそのままにしておけと言っていますが, というのもそれらの知見が有り得ないからです. だから, 我々が最も特有的な事象に向かって降下するときに, 分割したり複数のものをかき分けて行くことが必要であり, そして, 我々が最も綜合的な事象へと上昇するときにはその複数のものをまとめて行くことが必要です. というのも種 - そして, よりいっそうに類 - は多くの事象を一つの本性へとまとめるからです; 特有性, あるいは特異性は, 反対にその一つのものを常に複数のものに分割します. だから種において分有することで, 多くの人々は一つの人間 (という種) になり, そして, 特有性で, その一つで共通の人間 (という種) は個体になります - というのも個別性は常に区分するものであり, 一方で共通性は集約的で, 統合的だからです.

類と種 - それらの各々が何であるか - が表現されており, そして類は一つであったとしても種は幾つかになり (というのも類を切り分けることは常にいくつかの種を生成することになるためです), 類は常に種 (そして全ての上位の事象は下位の事象) を述定しますが, 種は近接する類や, より上位の事象を述定するものではありません - というのも, それは入れ替えが効かないためです[*45]. なぜなら, 等しいもので等しいものを述定 (嘶くことで馬をするように) したり, あるいは (外延が) より広範なもので (外延が) より狭いものを述定 (動物で人間をするように) するといった, どちらかの事態でなければならないのです; ところで (外延が) 狭いもので (外延が) 広範なものを述定するのではありません-あなたは人間は動物であると言うことがあっても, 動物は人間であるとは言いはしないでしょう[*46].

種であると述定されるものは何でも, それらの事象を, 必然性により, その種の類がまた述定するでしょう - それから最も綜合的な事象である限り類を類が述定するでしょう. だから, ソクラテスが人間であり, 人間が動物であり, 動物が本質的存在であるということが真であれば, ソクラテスが動物でも本質的存在でもあるということも真になります. そして, より上位の事象は下位の事象を, 種は個体を, 類は種と個体の双方を, そして, 綜合的な事象は類 (あるいは複数の類, 幾つかの中間的なものと下位の事象があれば) と種と個体を常に述定するためです. 最も綜合的な事象はそれの下の全て - 類や種や個体 - を語ります; もっとも特有的な事象に先行してある類はすべての最も特有的な事象や個体を語るものだからです; そして単一の種である事象は全ての個体を語るものであり; さらに個体はただ一つの特有なものと語られます.

[*45] 「A は B である」から「B は A である」は主張できません. そして, 類は種の上位概念であるために下位概念のある種で上位概念の類を説明できないと述べています. また後述の特有性はものの本質を説明するものではありませんが, 主語と述語の交換が効くものです.

[*46] 主語と述語の入替ができない例です. より上位の概念の「動物」の外延が下位の概念である「人間」の外延よりも広範であるために, 人間で動物を述定できません. ここでの「広範」,「狭い」は外延の大きさです. このようにポルフュリオスはクラスの分析を行っています.

ソクラテスは一つの個体であると語られますが，このことは**この**色白の人も，さらには**この**近寄ってくる人も，ソフロニスクス (Σωφρονίσκου,Sophroniscus) の息子 (ソクラテスが彼 (ソフロニスクス) の一人っ子であったなら) もそうです．このような事象を個体と呼びます．なぜなら各自は固有の特徴で構成されて，他のいかなるものでも同じものが決して見られないものの集積物のためです - ソクラテス固有の特徴は決して他の個体で見付けられないでしょう．その一方で，人間 (ここでは共通の人間という特有性を意味しています) 固有の特徴は幾つかの事象で - あるいは，むしろ，すべての特有の人間で，彼等が人間である限り - 同じであるということが判るでしょう．

このように個体はその種に，そして種はその類に包含されます．というのも類は全体を整理したあつまりで，個体は一部，それから種は全体であったり一部だったりします- つまり，(種は) 一つのものの一部であり，それから他の諸事象の間では (他の事象ではなく) 全体です (というのも全体は諸部分の中にあるからです)．

我々は類と種について，それから最も綜合的な事象が何であり，それから最も特有的なものが何か，そして同時に類と種になるのはどのような事象であるか，そして，何が個体になり，類と種がそう呼ばれる幾つかの方法について議論しました．

A.2.4 差異について

差異 (διαφορά, difference) は共通的，固有的，そして，最も固有的と呼ばれるべきです．というのも，一つの事象がある相違点を持った事象から異なると語られるのは一般的に，それがいろいろな状況で，それ自身や他の事象との関係の双方で，ある相違点によって区別されるときです - ソクラテスはプラトンと相違点で異なり，さらには実のところ彼自身も少年のときや成人のとき，そしてあることで動いているとき，あるいは止まっているとき，　おまけに彼がどのようなものであるかについて，諸々の相違点に関して異なります．一つの事象が特有的に相違点のある事象と異なっていると語られるのは，それがそれと離在不可能な偶有性で異なるときです - 離在不可能な偶有性は，たとえば，青い目であること，鉤鼻であること，傷の堅くなった痕といったものさえもです．一つの事象が最も特有的に異なると語られるのは，種を成す差異 (種差) によって区分されるときです - 人間は差異，すなわち理性的であることによって，馬と異なるようにです．

一般的に，全ての差異は，それが何かに付与されるときに，その事象を多種多様なものにします; つまり，共通で，特有な差異はそれを他の別様 (=別物のよう) にしますが，最も特有な差異はそれを全くの別物にしてしまいます．というのも差異のあるものは物事を別様にし，さらにあるものはそれらを別物にするためです．ここで，それらを別物にするものが種的なものと呼ばれ，それらを別様にするものは単に差異と呼ばれます．そんな訳で理性的であるという差異が動物に付け加えられると，それを別物にして動物の一つの種を作ることになります; ところが運動しているという差異は，ただ止まっていることと比べて

単に別様にする程度です; だからあるものはそれを別物に, あるものは単に別様にします. ここで, そういった事物を別物にしてしまう差異のために, 差異による類の種への分類が行われ, それと定義が, 類やこのたぐいの差異から構成されるもので表現されますが, ところで, あるものを単に別様にしてしまう差異からは, 多様性と, さらにそれが, どのようなものであるかということに関する変化が構成されるだけです.

最初から再び始めましょう, あるものの差異は離在可能で, そしてあるものは離在不可能であると我々は言うべきです - 動いていることや止まっていること, 健康であることや病んでいること, さらにはそれらに似た事象, こういったことが離在可能なことです; 鉤鼻に獅子鼻, あるいは理性的であるとか非理性的であるといったことが離在不可能なことです. 離在不可能な差異であるものは本質的であり, またあるものは偶有的です - 理性的であるということは人間それ自体の本質で, 死すべきことも, そして, 知識を持ち得ることもそうです; とはいえ鉤鼻であることや獅子鼻といったことは偶有的で, それら自体の本質ではありません. 本質的な差異が前もってあれば, それらはその本質的存在の説明規定に採用され, そして, それらはその事象を別物にします; 偶有的な差異はその本質的存在の説明規定にて語られることも, その事象を別物にすることもありません - しかし, 別様にします. さて, 本質的な差異はそれ以上でもそれ以下であることも許容しませんが, 一方で偶有的な差異は, たとえ, それらが離在不可能であっても, 増大や減少を受け入れます[*47]; なぜなら, 類や結果として類を分割する差異の双方がより多く, あるいはより少なく述定されることはありません. というのも各々の事象の説明規定を全うする差異があるためです; そして, 任意の事象について, それであるということは, それがひとつであり, それと同様のものであるために増加も減少も許容されませんが, その一方で, 鉤鼻であることや獅子鼻であること, あるいはとある色であるといったことの双方は (その程度が) 増えたり減ったりします.

三つの差異の種[*48]が見受けられますが, ここであるものは離在可能で, またあるものは離在不可能であり, それから離在不可能のもののあるものは本質的, またあるものは偶有的であり, さらに本質的な差異のあるものは, 結果として我々が類を種に分割することになり, それからまたあるものは分割された事項が結果として種になります. たとえば, 以下で述べるものの全てはそのものの動物の本質的な差異になります - 魂があって知覚がある, 理性的であることと非理性的であること, 死すべきことと不死であること - 魂があって, 知覚がある実体だからです), ところで死すべきことと不死であること, 理性的であるということと非理性的であるという差異は動物を分割する差異です (というのもそれらに従って我々は類を種へと分割するからです). しかし, まさにこれらの類を分割する差異が種を構成する上で十全で, 構成的なものであることが判ります. というのも動物は理性的

---

*47 ものの程度を差異は受け入れませんが, 偶有的な差異は受け入れることを述べています.

*48 離在可能なもの, 離在不可能なもの, それと偶有的なものの三種類です.

であることと非理性的であるという差異で切り分けられ, それから再び死すべきことと不死であるという差異で切り分けられます; それから理性的であって死すべきものであるという差異で人間が構成されることが判り, 理性的であって不死であることから神が, それから非理性的であって死すべきものが非理性的な動物になります. この方法で, 生命があるということと生命がないということの差異と知覚があるということと知覚がないということの差異は最高位の事象の本質的存在を分割し, 生命があるということと知覚があるということの差異から互いに集められた実体で動物が生成され, その一方で生命があるということと知覚がないという差異が植物を生成します. このように同一の差異でも, 一方の取り上げ方では種を構成するような構成的なものであり, もう一方では類を種へと分割するような分割的なものになり, それら全てが種を構成するようなもの (種成的差異) と呼ばれます; それらは類の分割や定義の双方で殊更に有用です - しかし, 離在不可能な偶有的な差異と幾分か離在可能なものはそうではありません.

それらを定義した上で, 彼等 (逍遥学派の哲学者達) はこう述べています: 差異とは, 種が類をそれによってはみ出すものであると. というのも, 人間は理性的であることと死すべき存在であることで動物から飛び出しているように - 動物はこれら双方の何れでもなければ (するとどこから種は差異を得るのでしょうか?), 背反する差異全てをそれ (動物) が保有している訳でもありません (そうなら同じものが同時に背反するものを持つことになってしまうからです); むしろ, 哲学者達が主張するように, それは可能態としてその下の事象の差異全てを保有しているものの, 現実態としては何等も保有しません[*49]. そして, この方法で同じ事象について何でもないことからは何も出て来ることがなければ, 同じものから背反する事象が見出されることも決してありません.

彼等 (逍遥学派の哲学者達) はまたそれ (種成的差異) をこのように定義しています; 差異は 'それがどういった類のもので, どのようなものであるか' に対する回答で, 種で異なる幾つかの事象について述定するものです. だから理性的で死すべきものであるということは, 人間を述定したときに,'人間はどのような類で, どのようなものであるか?' に対する回答で語られることであって, '人間は何であるか?' に対するものではありません. 人間とは何であるかと尋ねられたときに, こう言うのが適切です: 動物であると; ところで, 彼等が 'どのような類の動物であるか?' という質問を付け足すのであれば, 我々は理性的であって死すべきものであると適切に表現することになるでしょう. というのも質料 (ὕλη, matter) と形相 (εἶδοσ, form) から構成されている対象に関しては, 少なくとも質料と形相との類比として, 丁度, 銅像が青銅を質料, その姿を形相として構成されているように,

[*49] アリストテレスは運動の現象を説明するために「可動態 (δύναμισ)」と「現実態 (ἐνέργεια)」という概念を導入しています. まず, アリストテレスは後述の銅像の例のように事物が形相 (εἶδοσ) と質料 (ὕλη) の結合体であると考えています. このときに事物の可能態は形相が潜在的な状況にあって自己を発揮していない状態であり, そして, 現実態は形相がその自己を発揮した状況を言います. つまり, 可能態は実体が別の態へと遷移するための要因です. だから x という可能態を壊したからといって, 現実態の x が壊される訳ではありません.

共通で種としての人間は, 質料に対して類が, 形相に対しては差異が対応して構成されており, そして, これら - 理性的であり死すべき存在である動物 - ということが全体として人間という種になり, ちょうど, ここでの銅像の場合と同様です[*50].

彼等 (逍遥学派の哲学者達) はまたこういった差異というものの大枠を次のように説明しています: (種成的) 差異は同じ類の下にある事象を区分するようなものの本質です - 理性的であることと非理性的であることは人間と馬を区分しますが, これらは同じ類, 動物の下にあります. 彼等はまたそれらをこう表現します: 差異は, それによってそれぞれのものが異なるものです. たとえば, 人間と馬は実際に類は異なりません - 我々 (人間) と非理性的であるという事象の双方は死すべき動物です. しかし, 理性的であることが付け加えられると双方が区分されてしまいます. さらには我々と神々の双方は理性的です. しかし, 死すべきものということが追加されると, それらから区分されます.

差異という話題について詳しく述べると, 同じ類の下で事象を区分する羽目になったものだけが差異ではなく, むしろ, それらの本質に与り, そして, 何であるかということの一部を成すものだと彼等 (逍遥学派の哲学者達) は語っています. ここで, 船に乗って航海するといった性分が, たとえ人間の特有性であったとしても, 人間の差異ではないからです; ある動物に航海する性質があり, 他ではそうでないと我々が言うかもしれませんが, たとえ他からそれらが区分されたとしても, それでも航海するという性質はそれらの本質を全うにするものでもそれの一部でさえもありません - むしろ, 固有的に種成的であると言われるような, これらの差異と同類のものでないために, 実体の素質の一つでしかありません. 差異が多様な種に分割するときに限って, さらには, それが何であったかに差異が含まれるときに限って、差異は種成的となります.

これで差異については十分です.

A.2.5 特有性について

彼等 (逍遥学派の哲学者達) は特有性 (ἴδιον, property) を四つに分類しています: とある種だけの偶有性であるもの, たとえそれの全てでなかったとしても (医者にかかっている, あるいは幾何学を勉強中の人のように); その種全ての偶有性であるもの, たとえそれだけでなかったとしても (人間が二本足であるように); 種だけ, そして種の全て, そして, あるときに保持するもの (人間が老年では灰色になるように); それから四番目に, ‘単体, 全て, そして常に’ で一致するところのもの (人の笑うことができるということのように). というのも人は常に笑っている訳ではないので, 人が笑っていると語られるのは彼がいつ

[*50] 実体は形相と質料の「結合体」として現われるというのがアリストテレスの主張です. プラトンのイデアのように超越的なイデアが個体と別個に存在し, 個体がイデアを真似る, つまり, 分有するという考えと対立しますが,「手引」ではその形相がものの内なのか, 外にあるのかについては述べていません. だからプラトンとアリストテレスの違いが表沙汰にはならず, そのお陰で双方の考えが調和していると言えます.

も笑っているということではなく, 彼が笑うことができるという天性であって - そしてこれは彼が常に持ち, 馬が嘶くのと同様の, 通常の天性なのです. そして, 彼等が厳密な意味でそれらが特有性であると言いますが, なぜならそれらが入替が効くからです[*51]; もし馬であれば, 嘶くでしょうし, もし, 嘶くのであれば, 馬なのです.

A.2.6 偶有性について

偶有性 (συμβεβηκόσ, accident) [*52]はそれら (偶有性) の基体 (ὑποκείμενον, subject) [*53]を壊すことなしに, 行き来する事象です. それらは二つに分類されます: あるものは離在可能なもので, またあるものは離在不可能なものです[*54]. 眠ることは離在可能な偶有性ですが, 一方でカラスとエチオピア人が黒色であることは離在不可能なことです- それらの基体を破壊することなしに白いカラスや自分の肌の色を失っているエチオピア人を想像できます[*55]. 彼等 (逍遥学派の学者達) はそれら (偶有性) をこう定義します: 偶有性は同一物で保有したり保有しなかったりすることが可能なものです; あるいは: 類でも差異でも種でも特有性でもないものですが, 基体の中に常 に内在するものです.

A.2.7 共通の特徴

我々が提案した全ての事象 - 類, 種, 差異, 特有性, 偶有性のことを私は言っています - が述定されましたが, 我々は, それらに対して共通な特徴と固有の特徴として何が述べられるのか語らなければなりません.

それら全てに共通する点は複数の事象を述定していることです. ところで, 類は種のことと個体のことを述定します. これは差異もまたそうですが, ここで種はそれらの下にある個体を述定するものであり, 種の特有性は, それらの特徴になる種や, その種の下にある個体について述定し, 偶有性は種と個体の双方を述定します. たとえば動物は馬や牛といった種について, さらに馬や牛は複数の個体について述定し, それから非理性的であるということは馬や牛やその複数の個体を述定し, 人間のような種はその複数の個体だけを

[*51] 「A ならば B」の A, B の入替ができるという意味です. つまり, 特有性とは「ものの本質を説明するものではないが, そのものを特定することができるもの」なのです. だから「A ならば B」であり逆の「B ならば A」も成立するものだと言っているのです.

[*52] συμβεβηκός は動詞の συμβαίνειν に由来し, 本来の意味は予期せずに ‘生じる’ ことや ‘発生する’ ことですが, アリストテレスは保持しているという意味でも用いています. このようにたまたま生じさせられたり生じたり, あるいは何かを保持していることで用いられています.

[*53] 基体とは「他の事物は ‘それ’ の述定とされるが ‘それ’ 自らは決して他の何者で述定とされない ‘それ’」([3]1028b36) のことです. この説明から基体は構文的に主語以外になり得ないものであることが判ります.

[*54] 最初からある偶有性は離在不可能なもの, そうでないものが離在可能なものです. また, ‘... ということが可能である’ という一節を追加するとができるものが偶有性です.

[*55] 偶有性とはそれ無しの状態が想像ができるものです. たとえば X が Y の偶有性であれば, Y が X でない状態を想像することができるようなものです.

述定しますが, 特有性については, 笑うことができるといったことが人間やその複数の個体の双方を述定します. そして, 黒色ということはカラスの種やそれらの複数の個体の双方が, 離在不可能の偶有性, 人間や馬が動いているということは, 離在可能な偶有性です - しかし, 第一義的には複数の個体を, また, 第二義的には, それら複数の個体を包含する事象の説明規定になります.

A.2.8 類と差異

類と差異の共通点はそれらが種を含むことができるという事実です; 差異もまた種を含みますが, 類が包含するものの全てという訳ではありません - 理性的であること (という差異) は- 動物という類がするように非理性的であるという事象を包含しませんが, それにもかかわらず, 人間と神という種を包含します.

類としてある類を述定するものはさらに, その (類の) 下の種もまた述定し, そして, 差異としてある差異を述定するものはまた, それ (差異) から構築される種を述定します. たとえば類としての動物については, 本質的存在と魂のあるものがそれ (動物) を類として述定します - それからこれらの事象はまた動物の下の全ての種, 個体に至るまでを述定します; だから, 理性的であるという差異については, 理性を用いるということが, それ (理性的であるということ) を差異として述定します - そして理性を用いるということは単に理性的であるということだけではなく, 理性的であるということの下の種をもまた述定します[*56].

共通点はまた, もし類か差異のどちらか一方が除去されたのであれば, その下にある事象も一緒に除去されるという事実です. だから, もしも動物がいなければ馬や人もおらず, さらに理性的であるということがなければ, 理性を使う動物はありえないでしょう.

類に特有なことは, 差異や種や特有性や偶有性以上により多くの事象をそれら (類) が述定するという事実です[*57]. というのも動物は人間や馬や鳥や蛇に対応しますが, 四本足は四本の足を持つものだけ, 人間は個人だけ, 嘶くのは馬と個々の馬, そして偶有性は同様のより僅かな事象だけです.(我々は類を分類するもので差異を採らなければなりません, 類に含まれる本質的存在を全うするものではないからです.)

また, 類は差異を可能態にて含みます; というのも, 動物のあるものは理性的で, あるものは非理性的だからです[*58].

そして類は類の下にある差異に先立つものであり, このことが, 類の差異を除去しても,

[*56] ‘人間は動物である’ と ‘ソクラテスは人間である’ から ‘ソクラテスは人間である’ が成立し, 同様に ‘人間は理性的である ‘と ‘理性的であれば動物’ からは ‘人間は動物である’ と**推移律**が成立すると述べています.

[*57] このことについてはアリストテレスもトピカにて同様のことを述べています.

[*58] アリストテレスは形而上学の Δ 巻にて「類の諸性質が差異と言われる」と述べており, ここでの「可能態」は類の諸性質としての差異のあり方を指しています.

その類が除去されない理由です．というのも，もし動物を除去すれば理性的や非理性的といったことも一緒に除去されます．ところが，差異は類を一緒に除去しません; というのも，たとえ，それら全てを除去したとしても，知覚があり魂のある本質的存在が考えられるでしょう - そして，それが動物というものです．

また，いままで語られてきたように，類は 'それは何であるか?' に対する回答で，差異は 'それがどういったたぐいのものか？' に対する回答として述定されるものです．

また，それぞれの種には一つの類がありますが (たとえば，人間に動物があるように)，差異になると複数になります (たとえば，理性的であるということ，死すべき存在であること，知恵や知識を受容することができる，こういったことで人間は他の動物と異なります)*59．

類は物質に似ていて，差異は形相に似ています．

他の共通性や特有な事象は類や差異に先立っています - が，これらは十分ということにしておきましょう．

A.2.9　類と種

類と種は共に，今まで述べてきたように，複数の事象を述定します (もしも同じ事象が種と類の双方であるなら，その種を類としてではなく種として採用しています.)

(類と種の) それらの共通性はそれらが述定する事象の前にあるという事実，それと各々が一つの全体のものであるという事実です．

(類と種の) それらは類が種を包含するということで異なり，種は類に包含されても類を包含しません．というのも類は種よりもより広範囲のものだからです*60．

また，類は (種よりも) 先立って存在していなければなりません，そして，種成的差異によって形付けられることで，種を生成します．だから類はまた天性によって先立って存在するものです; そして，それらは一緒に削除しますが，一緒には削除されず，さらにもし種が存在するなら類もまた確かに存在しますが，類が存在するからといって種もまた存在するという訳ではありません．

類は同名同義的に類の下にある種を述定しますが，種は類を述定しません*61．

さらに，類は類の下にある種を含むことから種よりも一層広範で，種は種自体の差異によって類よりも一層広範です*62．

*59 ここでの主張は，まず種に近接する類が一つ存在するということ，それに対して差異は類の諸性質であるために複数存在することを意味しています．

*60 類は常に種よりも広範囲である.(トピカ [4] 121b3-4.

*61 ここで「同名同義的に」とは類の下にある種が類の説明規定から述定されるだけでなく，類からも述定できることを意味します．

*62 最初の類が種に対して「広範」であることはその包含関係によるものですが，次の「広範」であることは種が類よりもより多くの差異で語られるということです．要するに「内包外延反比例増減の法則」について言及していると言えます．

そして, 種が最も綜合的なものになる訳でもなく, 類が最も種成的なものになることでもないでしょう*63.

A.2.10　類と特有性

類と特有性は共にそれぞれの種に続くという事実があります; もしも人間であれば, 動物であるということが; そして人間であれば, 笑うことができるということです*64.

類は等しくそれの種を述定し, それから特有性もまたその中で分有する (μετέχειν, participate) *65ものを述定します - 人と牛は共に動物で, アニュトスとメレトス*66は共に笑うことができるといったあんばいです.

類は同名同義的にそれ自身の (下にある) 種を, 特有性はそれが特徴となるものを述定するという共通性もあります. それら (の共通性) は類が先行してあるということと特有性があとにあるということで異なります - たとえば, 動物がまず先行して存在しなければならず, それからあとに差異と特有性で分類されなければならないのです.

類は (その下にある) 幾つかの種を述定し, 特有性はそれを特有性とする一つの種を述定します.

特有性はそれを特有性とするものを交互に述定しますが, 類は何物をも交互に述定することはありません- もしも動物で人間でない場合, どのような動物も笑いません; もし, 人間であれば笑うことができる, 等々です*67.

また, 特有性はそれを特有性とする全ての種を保持し, その (種) 一つだけで, また常にそうです; 類はそれを類とする全ての種を保持し, そして常にそうですが - その (種) 一つ

*63 ここでの「綜合的」と「種成的」は類と種の階層的な関係を示すもので, それぞれ「上位」と「下位」で読み替えることができます. なお, 類と種の関係は分類学で用いられる学名の付け方に反映されています. 学名はリンネによる二名法で記述されますが, この二名法ではその動物/植物が属する「種 (species)」と種が属する「属 (genus)」を用いており, ます, 最初にラテン語の属名, それからラテン語の種名を記述することになっています. この表記方法はオブジェクト指向プログラミングでも属性やメソッドの表記でも見られるものです.

*64 要するに, 類を動物とするときに, その下にある種の人間に対して ‘X が人間’ であれば ‘X は動物’ であり, 同様に種を人間とするときにその特有性の ‘笑うことができる’ に対して ‘X が人間’ であれば, ‘X は笑うことができる’ と X の述語として類や特有性が続くということを意味しています.

*65 プラトンの言う「分有」は原型としてのイデアと模像としての個体の関係を述べたものです:「美そのもの以外に何か美しいものがあるなら, それは他ならぬそのものを分有することによって美しい.」(パイドンより). つまり, 参与, 与ることや共有することとおおまかに言えるでしょう. アリストテレスが使う場合は「与る」と訳されることが多く,「与られるものの説明規定を受け入れることができるということ」(トピカ [4] 121a11) を「与ること」の定義としています. ここではポルフュリオスの師プロティノスが新プラトン主義の創始者であることから「分有」と訳しました.

*66 アニュトス Ἄνυτος は政治家, メレトス Μέλετος は詩人で共にソクラテスの告発者です.

*67 主語と述語の関係で言えば, 類が述語になるのはその下にある複数の種に対してで, 一つの種だけの述語になりませんが, 特有性についてはそれを特有性として持つ種に対して主語と述語の関係で主語と述語の入換が効く, つまり, 述定することに関し, 類は 1 対多であるのに対し, 特有性は 1 対 1 になるということです.

だけではありません[68].

さらに, もし特有性が除去されたとしてもそれら (特有性) は類を一緒に除去しません; ところが, もし類が除去されてしまえば, それら (類) は特有性が所属する種を除去し, それから, 特有性であるものが除去されてしまうと, 特有性それ自体もまた一緒に除去されてしまうからです.

A.2.11 類と偶有性

類と偶有性の共通点は, すでに言われているように, それらが複数の事象を述定するという事実です - ここで偶有性は離在可能であったり, 離在不可能であったりします. というのも動いているということは幾つかの事象を述定し, カラスやエチオピア人やある非生物の事象でも同様だからです[69].

類は偶有性と異なり, 類はその (下にある) 種に先行していますが, その一方で偶有性は種に後行しています- というのも, たとえ離在不可能な偶有性が採られたとしても, 偶有性とするもの (種) がその偶有性よりも先立って存在しているからです.

類の中で与るものは等しく与り, 偶有性の中で与るものはそうではありません - というのも偶有性が与るということは増加や減少を許容しますが, 類の中で与るということはそうでないからです[70].

偶有性は個体にて先行して存在しますが, しかし, 類や種は性質上, 個別の本質的存在に先行しています.

類は ‘それが何であるか’ に対する回答でそれらの下の事象を述定し, 偶有性は ‘それがどういったたぐいのものか？’ や ‘それが何に似ているのか？’ に対する回答です. だからエチオピア人がどういったたぐいのものなのかと問われるなら, あなたは黒色だと言うでしょう; ソクラテスがどうなのかと問われるなら, あなたは彼が腰を下して座っているとかその辺を散歩していると言うでしょう.

我々は類が他の 4 つ (種, 差異, 特有性, 偶有性) とどのように異っているかを述べてきました; そして, それらの各々がまた他の 4 つの事象と異なり, だから, 5 つの事象があれば, 各個が他の 4 つと異なっているために, その違いは 4 かけ 5 で全部で 20 になります. ところで, それらを続々と数えてみると, 第二の場合は 1 つの違いで短くなり, すでに判っているように, 第三の場合は 2 つ, 第四の場合は 3 つ, そして第五の場合は 4 つになります; それゆえに, 相違は 4, 3, 2, 1 - だから 10 になります. 類は差異, 種, 特有性と偶有性と異なります - だから 4 つの違いがあります. 差異に関して, それらが種とどのように違

[68] 類は複数の種を包含し, 特有性は一つの種だけと結びつくことを述べています.

[69] 主語と述語の関係において, 類と偶有性はともに複数のものの述語となり得るということです.

[70] 類は説明規定で語られるもので, 程度が語られるものでありませんが, 偶有性はその程度を語ることができるということです. つまり, 人間に対してはどの程度, 人間であるかを語れませんが, カラスの色の黒さに対しては, その黒色の程度を語ることは可能であるということです.

うかということは類がそれらとどのように違うかを語られたときに語られており, それから種がどのように類と異なるかは, 類が種とどのように異なるかが話されたときに語られています. だから, どのように種が特有性や偶有性と異なっているかを語ることが残っています. そしてこれらの差異が 2 つなのです. 特有性が偶有性とどのように異っているかを語ることが残っているでしょう; それらは種, 差異や類とどのように異っているかについてはすでにそれらとの関係でこれらの差異について語られています. だから, 我々はその他の事象との関係で類については 4 つの差異, 差異では 3 つ, 種では 2 つ, そして特有性 (偶有性との関係で)1 つになります: それらは全てで 10 になり, それらの内の四つ - それらは類とその他の関係のもの - を我々はすでに説明をし終えているのです.

A.2.12 差異と種

差異と種の共通点は, それらが等しく分有するという事実です: 個々の人間は等しく人間性を分有してまた理性的であるという差異を共有します.

またそれらの共通点は, それらが常にそれらの中で分有するもので先立って存在するものであるという事実です; というのもソクラテスは常に理性的であり, そしてソクラテスは常に一人の人間なのです.

差異にとって特有なことは, それらが 'それがどのようなたぐいのものであるか？' に対する回答で, それから種は 'それが何であるか?' に対する回答で述定するという事実です. たとえ人間がものの集まりとして取られたとしても, 彼は単なるものの集まりではなく, むしろ差異がその種にして本質的存在を与えることになります.

再び, 差異はいくつかの種でしばしば観察されるものです - たとえば, 非常に多くの動物が四本足で, 種が異なりますが, その下にある個体に対してのみ種が適用されます.

また, 差異はそれらの種に先立って存在します. というのも, もしも理性的であるということを除去してしまえば, それで人間も除去されますが, だからといって人間が除去されても理性的であるということは削除されません, 神が存在するからです.

そして, 差異には別の差異が混入しています: 理性的であることと死すべき存在であるということは人間という本質的存在に混入しています. しかし, 種は種で混入されることがなく, むしろ他の別の種を生成します. ある一頭の馬とある一頭の驢馬をかけ合せると騾馬が生れます; ところが馬はというと, 単に, 騾馬に生ませるために驢馬と交配させられるという訳ではあり ません[*71].

[*71] 複数の差異を混合させることはできても, 複数の種を混合させることができないということです.

A.2.13 差異と特有性

差異と特有性は共にそれらで共通するもので等しく共有されるという事実があります: 理性的であるという事象が等しく理性的であるということと笑うことができるという事象は等しく笑うことができるということです.

常に, そして任意の場合で先行して存在することは双方に共通ることです. というのも, たとえ二本足のものがバラバラにされたとしても, ‘常に’ ということが, その本質に対する関係で語られるのです. というのも笑うことができるということも ‘常に’ そのような天性があっても常に笑っている訳ではありません.

差異に対する特有性はそれらがしばしば幾つかの種で語られるという事実です - たとえば, 理性的であるということは人間と神の双方にあてはまります - ところで特有性は一つの種 (それが特有性となるものの種) にあてはまります[*72].

差異は, それらが差異となるものの事象に続きますが入換えが効きません[*73]. ところで特有性はそれらが特有性となる事象を交互に述定するため, だからそれらは入替が効きます.

A.2.14 差異と偶有性

差異と偶有性の共通点はそれらが幾つかの事象で語られるという事実です.

離在不可能な偶有性に関して, 共通点はそれらが常に, そして全ての場合に対して先行してあるという事実です: 二本足であるということは常に全てのカラスに対して先立って存在することで, そして同様に黒色であるということもそうです.

それら (差異と偶有性) は異なります. なぜなら差異は包含するものの包含されません (理性的であるという差異は人間を含みます) が, ところで偶有性はある点でそれらが複数の事象にある限り包含し, そして, ある点でそれらの基体が一つでなく複数の偶有性を受容するものに包含されます.

差異は増加可能でも減少可能でもありませんが, それに対して偶有性はそれ以上やそれ以下になることを許容します[*74].

逆に差異は混合しませんが, その反対に偶有性は混合することがあります[*75].

そのような共通点であり, そしてそのような差異と他のものの固有の特徴があります.

[*72] 理性的であるという差異は人間と神の双方で適用できることから判るように差異は複数の種に適用ができます. しかし, 特有性はただ一つの種のみに適用することができるだけで, この点で差異と異なっています.

[*73] 主語と述語の入れ替えができないということです.

[*74] 差異は類の諸特徴として現れるもので, その特徴は度合を持つものではありません. だから, その度合いの増減といったことが生じません. しかし, 偶有性は度合を持つので, 増減を語れます.

[*75] 類が互いに混合することがないのと同様に差異も互いに混合することがないと述べています.

種がどのように類と差異で異なるかということは, 我々が類がどのようにその他のものと異なり, 差異とどのようにその他のものと異なるかを語ったときにすでに語られています.

A.2.15　種と特有性

種と特有性は共にそれらを交互に述定するという事実があります: もしも人間であれば, 笑うことができます; もしも笑うことができるのであれば, 人間です (笑うことができるということはものの天性として採られるべきことだとしばしばそのように語られています.)*76

種はそれらの中で分有するものの中で等しく先立って存在するもので, さらに特有性はそれらが特有性となるものの中にあります.

種は他の事象でもまた類になるような種の中で特有性と異なりますが, だからといって特有性はその他の事象の特有性になることができません.

種は先行して存在する属性で, さらに特有性は種にて偶発的に生じることです.

また, 種は常にそれらの基体で実際に先行して存在するものですが, その一方で特有性は可能態のなかで時折のものです. というのもソクラテスは常に実際にひとりの人間ですが, だからといって彼が何時も笑っている訳ではありません (たとえ彼が何時でも笑うことができるような天性であったとしても).

そして, もしその定義が異なっているのであれば, 定義された事象もまた異なります. 種の定義は類の下にあること, それから ‘それは何であるか？’ に対する回答で, 数で異なる幾つかの事象を述定すること, 等々です; 特有性はそれに対して単体, そして全ての場合について先立って存在していることです.

A.2.16　種と偶有性

種と偶有性の共通点はそれらが多くの事象を述定するという事実です. その他の共通の特徴はほとんどありません. なぜなら偶有性とそれらが偶有性になるものが互いにかけ離れているからです. その二つの各々で特有なことは種が ‘それが何であるか?’ に対する回答でそれら (種と偶有性) が種であるものを述定するという事実ですが, その一方で偶有性は ‘それがどのようなたたぐいのものであるか?’ あるいは ‘それが何に似ているのか?’ に対する回答で述定するものです.

また, 各本質的存在は一つの種と幾つかの偶有性で, 離在可能, 離在不可能な偶有性の双方を分有するという事実があります.

種は偶有性に先立って考えられるものですが, たとえ, それらが離在不可能 (その偶有性

*76 種と特有性は主語と述語の関係で入換が可能だということです. そして今までの議論から種と特有性は 1 対 1 の関係にあります.

となる何かのために基体が存在しなければなりません) であったとしても, 偶有性は後天的であるという天性で, さらにそれらは偶発的な天性を持つのです.

種で分有するということは偶有性では等しく生じます - 離在不可能のそれなら - 等しくはありません. というのも他の人と比較されている一人のエチオピア人は肌の色の黒さが薄くなっても濃くなっても構わないからです.

特有性と偶有性についての議論が残っています; というのも, どのように特有性が種, 差異や類と異なっているかが語られているからです.

A.2.17 特有性と偶有性

特有性と離在不可能な偶有性との共通点はそれらを除外してしまうと, それらが観察されたものについての事象が存在しなくなるという事実です. というのも笑うことのできるということがなければ人間は存在せず, そして黒色がないならばエチオピア人というものは存在しないのです.

ちょうど特有性が全ての場合と常に存在しているのと同様に, 離在不可能な偶有性もまたそうです.

それら (特有性と偶有性) は (笑うことができるということが人間の中にあるように) 一つの種だけの中で存在しているということで異っており, 一方で離在不可能な偶有性, たとえば, 黒色は, エチオピア人単体だけ存在するのではなく, カラスや乳牛や黒檀やそういった他のものに対しても存在しているのです*77.

また, 特有性はそれらが特有性となるものを交互に述定するものですが, その一方で, 離在不可能な偶有性は相互に述定されるものではありません. 特有性の関与は等しく生じますが, 偶有性ではより多かったりより少なかったりするのです*78.

ここで述べた以上の他にも共通で固有な特徴があります. しかし, それらの事象を差別化したり, それらが共通に有するもの指定することでは双方で十分です.

*77 特有性は一つの種の中に存在するものの, 偶有性は他の色々な種の中に存在するということです.

*78 特有性と種は 1 対 1 に対応し, だからこそ主語と述語の関係で, 主語と述語としての入換が効くのです. ところが, 離在不可能な偶有性と種については 1 対 1 の対応とならないために主語と述語の関係で入換が効かず, さらに偶有性の性格上, その程度を表現することが可能だということです.

あとがき

まだ本文も出来上がってはいませんが, 最近思うことを幾つか.

まず, ここ最近は個々で生み出されたものが何かと思わぬ形で統合された形で現われるということでしょうか.

Maxima 本を 2004-2006 にかけて執筆していたころが牧画的にさえ思えます. 当時は Maxima と他の関連するアプリケーションについて述べればよかったためです. ところが現在は, ビックデータやクラウド等, より大規模で統合的なものへと移り, 解説も一つの言語, 話題では済まなくなっています. 実際, SageMath のことを語るだけでも Python と NumPy の話は最低しなければなりませんが, それだけで数式の表現ができないために, 数式自体の話に加え, その背後にある思想についても語らなければならないためです.

これは浮世でも同様で, たとえば, CAE 分野で制御系なら制御系だけ, 機構なら機構だけ, 構造だけなら構造だけ, 流体なら流体だけの解析といった時代は遥か昔に過ぎ去り, 流体と構造の連成解析は当然のこと, さらにそれ以上の規模の解析へと移っています. 実際, 自動車の解析もハイブリッド車になると, 従来の機構や構造解析も大規模で高精度の解析になり, その挙動に関係する些細な事象も含め, 電池の化学変化 (温度依存!) も考慮した複合領域の解析になります. ここまで来ると単一のアプリケーションは当然のこと, 多少, ファミリー展開したパッケージでも不足な状況で, FMI といった規格でアプリケーションを結合した解析へと移行しつつあります. こうなると, 開発元やベンダーがライブラリを開発するだけでは足らなくて, コミュニティベースで物理現象やノウハウのライブラリ化が進行している状況です.「**Modelica 言語**」や「**Industry 4.0**」を調べるにつけ,「**ものづくりの DNA**」と切株を守っている間に世界は大きく派手に動いていたのだと実感する今日この頃です. この DNA にしても, 実は怪しいものです. 日本の戦後の発展が何が原因であったのかという議論で,「**勤勉な国民性**」といった神話がありますが, 実際は, 第二次世界大戦中に急拵えの理科教育や学生の工場への動員といったことが下地になっており, 戦後の経済拡大で工場に資金が流れ込んだことが大きいでしょう. そういった工場を「**ファブレス**」といった流行語でそれにとって代わるものがないにも拘わらず手放したこと, すなわち, 社会的に雇用の幅を狭めるという決断を安易に行ったことは, 1980 年代以

降の教育の軽視, 教養への蔑視が根底にあり, それがこのような分析力の欠如に繋ったのではないかと思います.

現時点, AI がなにかと騒がれていますが, 結局のところ, 計算機は計算機でしかなく, 無様に動物の脳をモデル化した程度でしかありません. これが格段に発達したにせよ, 現在のスマートフォンがあらゆる家電製品を駆逐した程度, つまり, 携帯オーディオやカメラが不要になったところで, それよりも便利なものを利用しているだけでしかなく, やっていること自体は現在の延長線上にあることです. 重要なことは全てがより抽象化され普遍を目指していることで, そのためには「**それが何であるか**」,「**それがどのようなものであるか**」を「**正しく語り**」,「**正しく考察する**」ことを促す教育でなければ対処できないでしょう. 安易に「**Excel や Word が使えれば十分**」といった普遍性に背を向けた「**丁稚教育**」は, そのような現在の風潮から逆行し, 能力を無駄に費やす程度で間違いなく終ることでしょう. それに加えて安易な「**リベラル・アーツ**」の提唱も単なる気休めにしかならないでしょう. 本質の改善とは異なるためです.

これを書いた平成の末期は何かと憤激に駆られた時期でした. それから暫く何も書けなくなっていましたが, 漸く, 書けるだけの

参考文献

[1] 阿部謹, 刑吏の社会史, 中公新書, 中央公論社, 1978.

[2] 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.

[3] アリストテレス, 形而上学 (上下), 岩波文庫.

[4] 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.

[5] アリストテレス, (新) アリストテレス全集 2, 分析論前書・分析論後書, 岩波書店, 2014.

[6] 飯田隆, 言語哲学大全 I 論理と言語, 勁草書房, 1987.

[7] 井筒俊彦, イスラーム思想史, 中公文庫, 中央公論社,1991.

[8] 今道友信, アリストテレス, 講談社学術文庫, 2004.

[9] 上坂吉則, ニューロコンピューティングの数学的基礎, 近代科学社, 1993.

[10] 大石進一, 精度保証付き数値計算, コロナ社, 2000.

[11] 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.

[12] ガザーリ, 哲学者の意図, 岩波書店, 1985.

[13] 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/labo/text/ieee_format/`

[14] 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.

[15] 後藤和茂,BLAS の概要 (http://jasp.ism.ac.jp/kinou2sg/contents/RTutorial_Goto1211.pdf), 2006.

[16] 柴田有, グノーシスと古代宇宙論, 勁草書房,1982.

[17] 清水哲郎, オッカムの言語哲学, 勁草書房, 1990.

[18] 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.

[19] 新開謙三, 疑微分作用素 - 偏微分方程式解法への応用-, 裳華房, 1994.

[20] 杉晴夫, 神経とシナプスの科学 現代脳科学の源流, ブルーバックス, 講談社, 2015

[21] 鈴木航介, 合田隆, 重点解説 モンテカルロ法と準モンテカルロ法, コロナ社, 2025

[22] 藤野登, 論理学 -伝統的形式論理学-, 内田老鶴圃, 2003.

[23] 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

[24] 田中千里, イスラム文化と西洋, 講談社,1991.

[25] 田中 尚夫, 選択公理と数学, 星雲社, 1987.

[26] 聖トマス, 形而上学叙説, 岩波書店, 1935.

[27] 永嶋 哲也, 普遍と心理 アベラルドゥス言語哲学の研究, https://ir.lib.hiroshima-u.ac.jp/files/public/3/30917/20141016180057479520/diss_otsu4060.pdf

[28] 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.

[29] 平井有三, はじめてのパターン認識, 森北出版, 2012.

[30] プラトン (著), 藤沢 令夫 (訳), 国家, 岩波文庫, 岩波書店, 1976.

[31] フレーゲ, フレーゲ著作集 1 概念記法, 勁草書房,1999.

[32] フレーゲ, フレーゲ著作集 3 算術の基本法則, 勁草書房,2000.

[33] 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.

[34] ボエティウス (著), 石井雅之 (訳), ポリフュリウス・イサゴーゲー註解, 中世思想原典集 5 後期ラテン教父, 平凡社, 1993.

[35] 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

[36] ポルピュリオス (著), 水地宗明 (訳), イサゴーゲー, 世界の名著 (続 2), 中央公論社, 1976.

[37] M.Lynne Murphy, Ane. Koskela, 意味論キーターム辞典, 開拓社, 2015

[38] 皆本晃弥, IEEE754 と数値計算,
http://www.ma.is.saga-u.ac.jp/minamoto/doc/kyudai.pdf

[39] 山内志朗, 普遍論争, 平凡社ライブラリー, 2008

[40] 横田博史, はじめての Maxima, I/O Books, 工学社, 2006.

[41] 横田博史, はじめての Maxima 改訂 α 版 (MathLibre に収録)

[42] 横田博史, 数値計算・可視化ツール Yorick, I/O Books, 工学社, 2010.

[43] 吉田光邦, 錬金術 - 仙術と科学の間 -, 中央公論新社, 2014.

[44] ルーベンスタイン (著), 小沢千重子 (訳), 中世の覚醒 - アリストテレス再発見から知の革命へ, 筑摩書房, 2018.

[45] J.Barnes, PORPHYRY INTRODUCTION,Oxford University Press, 2006.

[46] Bell, Toposes and the Local Set Theories, Dover, .

[47] Birman, Braid groups and mapping class groups, Ann. Math, Princeton University Press, 1963.

[48] Boethius, Isagoge, http://www.forumromanum.org/literature/boethius/isag.html

[49] G. J. Brose, MATLAB 数値解析,Ohmsha, 1998.

[50] Haigh, An interview with Jack J. Dongarra, http://history.siam.org/pdfs2/Dongarra_%20returned_SIAM_copy.pdf, 2004.

[51] Jean van Heijenoort, FROM FREGE TO GÖDEL A Source Book in Mathematical Logic, 1879-1931, Harvard University press, 1967.

[52] Goodger, reStructuredText ディレクティブ, http://docutils.sphinx-users.jp/docutils/docs/ref/rst/directives.html

[53] An interview with Charles L. Lawson, http://history.siam.org/pdfs2/Lawson_final.pdf, 2004

[54] Mac Lane, The Category theory for working Mathematician, Springer

[55] Mac Lane, Moerdijk, Sheaves in Geometry and Logic, A First Introduction to Topos Theory, Springer Verlag, 1992.

[56] Porphyry, Introduction(Iagoge) to the logical Categories of Aristotle, http://www.ccel.org/ccel/pearse/morefathers/files/porphyry_isagogue_01_ intro.htm

[57] Porphyry, Letter to Marcella, http://www.tertullian.org/fathers/porphyry_marcella_02_text.htm

[58] B.Russell, The Principles of Mathematics,W.W.Norton & Company,Inc.,1996.

[59] B.Russell & A.N.Whitehead,Principia Mathematica to *56, Cambridge Mathematical Library,Cambridge University Press,1997.

[60] 人工知能学会誌, 特集「自然界に見出す数物構造を利用した知的情報処理」, 2018.9 No.5

[61] Diving into Python: http://www.diveintopython.net/toc/index.html

[62] MathWorks 日本: http://www.mathworks.co.jp/

[63] JeanFrancoisPuget: A Speed Comparison Of C, Julia, Python, Numba, and Cython on LU Factorization https://www.ibm.com/developerworks/community/blogs/jfp/entry/ A_Comparison_Of_C_Julia_Python_Numba_Cython_Scipy_and_BLAS_on_LU_Factorization , 2016.

[64] アテナイの学堂 http://ja.wikipedia.org/wiki/アテナイの学堂