

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 -

USO824 版

横田博史

平成 28 年 10 月 02 日 (日)

数式処理システム SageMath への招待 ©(2016) 横田 博史著
この著作の誤り、誤植等で生じた損害に対して MathLibre
のメンバー、著者は一切の責任を負いません。

まえがき

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

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

なお、この文書はまだ下書き以前の段階で、多少の間違いどころか致命的な間違いや嘘も大量に含んでいます！だから USO800 版なのです。だから現時点ではこの文書の内容の保証は十分できないこと、そのためこの文書の二次配布はご遠慮願います。とはいえたる秘密の文書ではありません。GitHub で公開しているので、リンク先の紹介等は構いませんし、間違いや問題点の指摘は歓迎します。どこに転がるか判らない代物ですが、どうぞ（生？）暖く見守ってやって下さい。

平成 28 年 07 月 29 日 (金)

狸穴主人 横田博史

目次

第 1 章 SageMath について	1
1.1 背景	2
1.2 SageMath の使い方	13
1.3 SageMath が目指すものは?	23
1.4 SageMath が包含するアプリケーションとライブラリ	24
1.5 一般の Python パッケージ	32
1.6 この本の方針	32
第 2 章 オブジェクト指向について	35
2.1 SageMath と Python の関係	36
2.2 Python はどのような言語か	36
2.3 オブジェクト指向プログラミングの哲学的側面	37
2.4 集合論について	54
2.5 圏 (Category)	63
2.6 トポス (Topos)	89
2.7 トポスの基本定理	95
2.8 高階論理 λ -h.o.l. とトポス	95
第 3 章 Python について	105
3.1 Python とは?	106
3.2 Python の特徴	107
3.3 有理数を構築してみよう	119
3.4 Backus-Naur 記法 (BNF)	126
3.5 字句解析について	130
3.6 オブジェクトについて	141
3.7 特殊メソッド	167
3.8 記述子 (descriptor)	174
3.9 クラス属性の参照について	176

3.10	名前空間とスコープ	180
3.11	名前付けと束縛	181
3.12	例外	183
3.13	Python の式	183
3.14	単純文	191
3.15	複合文	195
第 4 章	プログラム開発の指針	199
4.1	はじめに	200
4.2	PEP(Python Enhancement Proposal) について	200
4.3	ファイル名やディレクトリ名に関する指針	206
4.4	ライブラリに関する指針	207
4.5	文書文字列の利用について	207
4.6	reStructuredText について	208
第 5 章	数学的对象の表現	237
5.1	はじめに	238
5.2	Python の数の構成	239
5.3	SageMath の数の構成	244
5.4	SageMath のオブジェクト	247
5.5	SageObject の各階層について	255
第 6 章	SQLite について	259
6.1	SQLite 速習	260
6.2	SQLite のキーワード	264
6.3	SageMath から SQLite を使う	264
第 7 章	結び目理論への適用	269
7.1	概要	270
7.2	結び目/絡み目とは	270
7.3	正則射影図	272
7.4	結び目/絡み目の同値性	273
7.5	群について	275
7.6	結び目/絡み目を表現する群	280
7.7	組紐群と置換群	287
7.8	ガウス・コード	294
7.9	LinkDiagram クラス	301

7.10	多項式不变量	319
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	323
第 8 章	SageMath で画像処理	335
8.1	画像の読み込み	336
8.2	OpenCV の利用	340
第 9 章	SageMath の拡張	341
9.1	sagemath からのパッケージ入手	342
9.2	一般の Python パッケージのインストール	342
9.3	GNU R のパッケージのインストール	342
第 10 章	SageMathCloud	343
10.1	SageMathCloud(SMC) とは	344
10.2	利用者登録について	345
10.3	基本設定	346
10.4	プロジェクト	348
10.5	ファイルのアップロードとダウンロード	349
10.6	端末, Jupiter, L ^A T _E X の利用について	352
第 11 章	手引 (ポルピュリオス)	357
11.1	概要	358
11.2	はじめに	362
11.3	類について	363
11.4	種について	365
11.5	種差について	370
11.6	特有性について	374
11.7	偶有性について	374
11.8	共通の特徴	375
11.9	類と種差	375
11.10	類と種	377
11.11	類と特有性	378
11.12	類と偶有性	379
11.13	種差と種	380
11.14	種差と特有性	381
11.15	種差と偶有性	381
11.16	種と特有性	382

11.17 種と偶有性	383
11.18 特有性と偶有性	383
第 12 章 Python のライブラリ	385
12.1 PIL	386
12.2 Pillow	387
12.3 Matplotlib	389
12.4 SciPy	391
参考文献	393

第1章

SageMathについて

πάντες ἔνθρωποι τοῦ εἰδέναι ὀρέγονται φύσει.

全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 車輪の再発明はしない

SageMath は非常にユニークな **オープンソース ソフトウェア** (Open Source Software, OSS と略記) のシステムです。この SageMath は数式処理システムの *Mathematica* や数値行列処理システムの MATLAB の代替となる OSS の**数学環境**を実現することを目的にしていますが、そのために SageMath の開発で採用した手法・手段が非常にユニークであることに尽きます。実際、SageMath の開発者 (William Stein) の「**車輪の再発明はしない**」との言葉からも判るように最初から独自のソフトウェアを構築するのではなく、それどころか既存の優れたソフトウェアを取込むことで必要な機能を実現するという手法です。このような開発手法が可能となった背景に研究目的のために開発された高機能の OSS のアプリケーションが存在していることに加え、それらを動作させるための環境にも余裕があるという二つの前提があります。これらの事実の背景について簡単に説明しておきましょう。

まず、高機能の OSS のアプリケーションの多くは研究の成果として一般に公開され、研究者が関心を持っている分野では非常に優れたものです。しかし、限られた用途や利用者を対象とするために処理言語やデータ構造が独特なものになりやすく、その結果、使い難いもの、逆にユーザ・インターフェイスを含めた使い勝手が良かったとしても専門分野に限定されているのために利用する上で専門的な知識が必要とされたりと誰にでも使えるというものでないことがあります。とは言え他の分野に全く使えないという本質的に特殊なものはなく、他の処理に転用の効くものであったりするのです。

そこで個々のアプリケーションの土台を Python で構築することで、その処理言語を Python で統一し、さらにデータ構造を Pyrhon 上で定義して、それからデータを各アプリケーションとの間でやりとりするインターフェイスを作ってしまうとどうなるでしょうか？このときに利用者に見えるのは処理言語の Python と、その Python を使って定義したデータでしかありません。こうすることで利用者に要求されるのは共通の基盤の Python 言語と対象が Python でどのように表現されているかという水準にまでに落し込むことができるのです。すると Python 上で対象がどのように表現され、どのように処理すればよいかさえ理解できていれば、門外漢でもそれなりに高度な計算が行えることだけではなく、同時に専門家にとっても統一的な操作が可能な環境が得られることで、それまで利用することができなかつた別分野の専門のアプリケーションを利用するための基盤が整備されることになるのです。

このように既存のアプリケーションを繋げて使うという SageMath のやり方が十分に実用的になった背景に、現在の計算機環境が従来と比べると格段に贅沢な環境になっているという現実があります。実際、2016 年現在の携帯電話でさえも 1GHz 以上の動作周波数で複数のコアを持つ CPU、それに加えて 1GB から 3GB 程度の記憶容量、そして、最低でも 8GB 程度の記憶媒体を持ち、さらに高速ネットワークに当たり前のように接続可能な環境になっています。このような「**贅沢な環境**」になる以前はプログラムサイズを可能な限り小さくし、さらに実用的な処理速度を得るためにアセンブリやコンパイラ言語を駆使するといった工夫や最適化を行う必要がありました。しかし、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 のような比較的低速な対話処理言語で繋ぎ合せたシステムでも余程下手なプログラムを組まなければ十分に実用的な処理速度で動作してしまうのです。そして、こちらの方が職人技で最適化したシステムよりも全体的なコストが安く上がるというおまけまであるのです^{*1}。

1.1.2 CAE アプリケーションの場合

この SageMath に見られるように多様なアプリケーションを Python で結合するという手法は近年の商用の CAE(Computer Aided Engineering) ソフトで数多く見られる手法です。この経緯や理由も簡単に説明しておきましょう。

1980 年代から 1990 年代前半にかけて、その当時のいわゆる西側諸国でサッチャリズムに代表される新自由主義の影響を受けた政策によって、国公立大学や研究機関の研究成果を利用したビジネスの展開が強く要求されるようになりました。その結果、それらの研究機関で開発したソフトウェアを商用化し、販売・サポートを目的とした研究機関を母体とするベンチャー企業が数多く創業されました。こうして研究機関から商用化されたソフトウェアは企業の製品開発等でも幅広く利用されるようになり市場も成熟してゆきます。その結果、研究機関を母体とするベンチャー企業が当初想定していた大学や研究機関に近い研究者や専門家から、手続に沿った作業を行うオペレーター的な利用者へと比重が変化することになります。すると非専門家でも容易に操作が可能なシステムが市場から強く求められるようになります。そうなると多少性能が良いだけでは市場で生き残ることが困難になり始め、本筋の機能に加えて操作性の良さや他のアプリケーションとの連携といったことを特徴に挙げるものが増えてゆきます。1990 年代の後半になると市場の支持を得ることに成功した企業によって、その他のベンチャー企業の多くが買収されるようになり、アプリケーションのファミリー展開が行われるようになります。

^{*1} だからといって高速処理への要求がなくなるという訳ではありません。あくまでもコストと所要時間のどちらかを取るかという現実的な問題の一つの選択なのです。

これと平行して製品開発から生産までの工程を一貫して扱おうとする動きが企業から出てきます。たとえば、設計では CAD(Computer Aided Design) を使ったシステムが現在では主流になっていますが、80年代末から 90年代初頭の手書きから CAD の導入段階ではまだ 2次元 CAD が主流、それも製図手段が計算機で置き換えられた程度で、紙の図面を他の工程に回すものでした。そのこともあって計算機を使った解析もあらかじめ CAD から出力した紙の図面から形状等の情報を読み出して解析モデルの構築を行い、それからさまざまな解析を行うという手順でした。しかし、この中間段階の紙は必要なものではなく、設計図データをそのまま解析モデルに変換できてしまえば解析モデル構築のための座標の読取といった作業が不要になります。とはいっても、2次元モデルの CAD では解析モデルの構築も 2次元のままで解析モデルを構築するのか、あるいは 2次元の図面から 3次元化するのであれば、そのための手間やノウハウも必要だったのです。これが 90年代の後半になると 2次元を中心だった CAD も計算機の処理能力の向上に伴なって 3次元 CAD が主流になります。この時点では紙の図面に落す方が面倒で、解析モデルも最初から 3次元モデルになっています。そんな状態で物差し、分度器やコンパスを使った座標の読取を行って解析モデルを構築するということは、その処理に費す時間がコスト的に割に合わなくなる程の複雑なものになります。さらに計算機の処理能力も複雑な形状であっても処理ができる程の能力を持ち、2次元モデルに限定する理由が希薄になったのです。この状況下では CAD の図面データをそのまま解析ツールに流し込んでモデル化した方が遥かに効率的で正確なモデルが構築できるのです。

さらに実際の製品開発では形状の設計や部品の幾何学的な動きだけを設計しているのではなく、その製品の開発と平行して機能的な設計、たとえば、あるスイッチを押すとどういった動きをするかといったこと、全体の制御をどのように行うかといった制御系の設計、色々な電子部品が組込まれるために効率的な放熱やノイズ対策を行うかといったこと、さらには携帯電話のように落し易い製品で、もし、製品を落したときでも部品が衝撃で脱落し難いように内部部品の配置をどのように行うかといったさまざまな観点からの設計が必要になります。これらの作業を実際に装置に組込む前に計算機モデルで動作の検証や調整が行えれば、それまでの中間的な作業、たとえば、図面からの座標の読取を行って幾何的情報を取出すといった作業が不要になり、さらに各工程でモデルを共有することで全体的な作業の流れがより簡素化されて明瞭なものとなります。たとえば、放熱効果が悪いという解析や実験の結果が出て、それに対応して製品の形状を変更したとき、紙ベースであれば、関連部署にその紙を郵送し、関連部署では図面から座標の読取といった処理を行ってモデルの修正を毎度行わなければなりません。それを関係する部署で独立して行うとなれば、最新の図面の版が何なのかを確認する手間や図面の管理も含めて大変なことですが、CAD データとして一元的に管理していれば、その根本のデータに変更があれば、それに共通す

るものが自動的に更新され、そうでなくても何を参照すべきか混乱することも減るでしょう。このように各工程の結果だけではなく、モデルも含めて一貫して扱った方が版の管理の問題やより卑近なところではデータの変換の手間も省けてより効率的なことが理解されるでしょう。これが MATLAB の開発元の MathWork Inc. が提唱している「**モデルベース・デザイン (モデルベース開発)**」に繋がります。

このモデルベース・デザインは製品の設計・開発から製造に至るまでの各段階を一貫して扱おうとする手法で、この手法では製品開発の概念設計の段階から計算機を用いることになります^{*2}。そして、製品の開発にしても従来の単発的な処理ではなく、モデルを共有することで総合的な開発を行うようになっています。このことは「モデルベース開発のための複合物理領域モデリング」[7] に自動車業界の話が出ていますが、自動車もエンジンと車体だけの話ではなく、内部、外部の騒音の問題、ハイブリッド車であれば電池の化学反応(温度に依存)といった化学的な解析も必要で、このように物理的な話以外のさまざまなことを考慮した開発になっていることが判るでしょう。この状況では最早、モデルもあるアプリケーション特有の書式でよいのかという問題も表面化し、そこで MapleSoft の Maple で動作する解析シミュレータ MapleSim ^{*3}では Modelica 言語でモデルを記述し、それから代数方程式を生成する手法になっています。また、製品がどのようなものであるべきかを設計する人は解析の専門家であるとは限りません。あくまでも設計を行う上での参考として色々と計算する程度で、解析に重点を置くよりはプロジェクトや文書の作成や管理の方が重要であったりもします。そのために CAE の裾野が広がるにつれて当初の利用者であった解析の専門家向けから、文書作成やプロジェクト管理が行えるといった解析以外の機能が拡張された「**より敷居の低い**」アプリケーションが歓迎されるようになります。それはじめの段階では MS-Office(特に Excel)への対応と GUI による操作性を売りにする製品が出始めますが、この時点では Tcl/Tk で構築した GUI を従来のアプリケーションに被せる方式がよく用いられてました。

ここで解析ソフトウェアを機能や目的で大きく分けると、モデルの構築を行う「**プリ**」、解析を行う「**ソルバ**」、結果の可視化等の後処理を行う「**ポスト**」の三種類に分類できます。これらは機能や性格が大きく異なるために操作体系も大きく異なった別個のアプリケーションとなっていることが多い、そのこともあって最初に GUI を導入した時点で各機能の GUI に統一感がないどころか操作体系自体が大きく異っていることさえも普通でした。しかし、それでは流石に不便なのでアプリケーションのファミリー展開が行われる

^{*2} ここで解説している製品開発分野でのモデルベース・デザインについては MathWorks Inc. やサイバネットシステム株式会社のウェブページに解説文があるので、そちらも参考にされると良いでしょう。

^{*3} MATLAB の Simulink に相当する数式処理システム Maple の独立したパッケージで、Modelica 言語にまともに対応している数少ないパッケージです。

ようになると操作性に統一感を持たせるようになります。このときの基準となつたのが Microsoft の Office もので、MS-Word や Excel との連携が中心です。このような操作性の向上が図られる一方で、アプリケーションの中身はさほどの変更はありません。実際、アプリケーション内部も含めて修正するということは開発者にとって簡単なことではありません。そもそも、まともに動作しているアプリケーションの中核を担う箇所を無理に弄つて、その結果、従来と異なる結果を出して信頼性を損ねることだけは避けなければなりません。そのような冒険をするよりも可能な限り本体はそのままにして入力出力データを生成する GUI を工夫して作る方が安全であり、皮肉な話ですが、その操作性の向上の方が一般受けは良いのです。そうなると自動処理、カスタマイズやネットワーク越しでの利用といったことも考慮しなければなりませんが、GUI ビルダーとしての色彩の強い Tcl/Tk を用いるよりも Java や Python のように必要とされる機能を持ち、拡張が容易で、ネットワークにも対応した一般的な言語を用いる方が間違ひがありません。

1.1.3 Java と Python

そこで最初に注目された言語が Java です。Java は「一度書けばどこでも動かせる」というキャッチフレーズに加え、ネットワークにもとより対応し、言語的に C に似ている点、オブジェクト指向の言語でプログラムの再利用が容易であるといったことから急速に普及し、現在は幅広い分野で使われ、最適化も進んだパフォーマンスが高い言語の一つになっています。この Java はネットワーク越しに Oracle 等の RDB の制御を行うアプリケーション、さまざまな環境で均質な動作環境が実現できることから GUI を駆使した Cinderella や GeoGebra のようなアプリケーション、Eclipse のような統合開発環境やアップレット等で用いられています。

また Python は言語仕様が比較的単純なために学習し易く、その一方で拡張性が高い言語で、既存のアプリケーションを繋ぎ合せるための「**接着剤**」としての働きを Python にさせることができます。ただし、処理速度は Java と比較してさほど高速ではありません。しかし、Python が苦手とする部分は既存のより高速なアプリケーションやライブラリで処理させ、その結果だけ横取りするようにすれば処理速度の問題は大きく改善されます。また、このときに入出力を抽象化しておきさえすればより高速なエンジンになるアプリケーションやライブラリが現れれば今度はそれを用いれば良いのです。つまりところ個々のアプリケーションやライブラリは今後もより高度化してゆくでしょう。しかし、利用者が得たい結果は数値や数式であったり、グラフ等の画像であったりと、内容はさておいて外見に極端に大きな違いがある訳ではありません。むしろ、システム全体としてどうあるべきかが問われているのです。また Python 言語のオーバーヘッドの問題も計算機環境の改善によって現在は昔程は問題になってはいないのです（とはいって、それを気にする案件

もちろんありますが、それ以上に総合的な経済性の問題があるのです）。また、前述のように Python の言語仕様は他の言語と比べて簡素するために学習に費す手間も少なく済み、過剰に技巧的なプログラムに凝る必要もありません。それに加えてシステムを Python で記述しておけば独自の処理言語を作成しなくても Python をシステムの処理言語にすることができることに加え、オブジェクト指向言語なのでプログラム資産の継承や拡張が容易である点も挙げられます^{*4}。このような利点もあって、2000 年以降、Python を使って既存のアプリケーションを繋いだシステムを構築したものが増えているのです。

1.1.4 多重模範言語

Python にはオブジェクト指向の考えが取り入れられていますが原理主義的でガチガチなオブジェクト指向の言語ではない「**多重模範言語 (multi-paradigm language)**」と呼ばれる多様なプログラム様式を許容する言語になっています。たとえば、伝統的なオブジェクト指向の考えを取り入れた数式処理システムの多くではオブジェクトの実体化としてインスタンスを生成しない限り、そのオブジェクトに付随するメソッドが使えません。この理由ですが、クラスはこれから扱おうとするデータの属性を表現するもので、このクラスによって「**オブジェクトのあつまり (=オブジェクトの類)**」が定められ、このときにオブジェクトは「**類の元**」としてクラスで記述された属性を持ち、そのメソッドは集合内の元に対して許容された「**演算/処理**」、あるいはオブジェクトが有する「**能力/機能**」に相当します。

このことを多項式の因子分解という処理を通して考えてみましょう。オブジェクト指向のプログラム技法で多項式の因子分解を計算したければ、最初にすべきことは、その多項式が所属するクラスを使ってインスタンスとして多項式を表現することで因子分解がそのクラスのメソッドとして用意されます。また、多項式のクラスが存在していないければ、既存のクラスを使って多項式のクラスを創らなければなりません。この場合は、多項式がどのような性質を持ち、どのような演算や処理が必要であるかを属性やメソッドで表現することになります。そのため多項式というクラスやこの多項式の類から派生した類に属しないデータは当然、多項式のクラスのメソッドが使えません。と、メソッドは領域が限定された函数になっているのでメソッドも式、つまり、オブジェクトがどのクラスに属するかで動作が異なることもあります^{*5}。実際、因子分解一つでも $x^2 + 1$ という式は整数、有理数、実数係数の多項式環で分解ができませんが、複素係数環であれば $(x - i)(x + i)$ と分解することができます。つまり、多項式の因子分解を行うだけでも、まず、その式の係数が整数なのか、有理数なのか、それとも実数なのか、あるいは複素数なのかを考えて、それらに対

^{*4} スクリプト言語であっても過去の遺産が生かせるかどうかはシステムとして見れば重要なことです。

^{*5} メソッドの上書き (オーバーライド, over ride) によるものです。

応するクラスのインスタンスとして多項式を生成すればメソッドを使って因子分解が行えることになります。このように多項式の展開をするだけでもそういった多項式という対象が所属する世界を考えなければまともに処理が行えないことを意味します。

このような処理に慣れてしまえば「あたりまえ」のことでしょうが、単に「**整数係数の多項式の因子分解を素早く行いたい**」利用者にとっては余計な作業をすることになります。その定義が覚え難いものであったり、紛らわしいものであれば、この問題点はより一層、大きなものになるでしょう。こういったことを真面目にやっている数式処理ソフトに Singular があります。このアプリケーションでは「環」と呼ばれる数学上の対象を「**世界**」として定めることでやっと多項式の処理が行えますが、何も指定しなければ整数の四則演算しかできません。そのために計算を行うために何かと必要なクラスのインスタンスを生成する必要が生じます。このときに前提となる概念の知識があれば利用する上での手助けになりますが、そうでなければ何かと「**堅苦しい言語**」になってしまう傾向があります。このことは多項式の展開を覚えたての中学生が計算機で多項式の展開をさせるときに「環」の概念を要求することになり、ただでさえ堅苦しい言語に数学的な概念というさらに厄介な障壁(?)が加わることになります。もちろん、こうした処理を「**魔法の呪文**」だと思ってしまう手もありますが、何れにせよ道具を使う都度、「**呪文**」で呼び出すという煩雑な手間が増えることになります。

ところが Python は前述のように多重模範の言語であるお陰でメソッドを通常の函数のように扱うことができたりと、オブジェクト指向の言語としての側面を全面に出さずに利用することができます。だから、多項式環を定義しなくても、多項式を入力して因子分解ができます。また多項式の計算で、「**最初に整数係数の多項式環を生成して、それから因子分解メソッド factor を使って式の因子分解を行う**」と覚えなくても「**多項式を入力して、その式に函数 factor() を作用させる**」だけで済ますことも可能なのです。この点は一般的の利用者にとっては非常に有り難い点ではないでしょうか？

この実例を示しておきましょう：

```
sage: a = x^4 + 4*x^3 + 6*x^2 + 4*x + 1
sage: a.factor()
(x + 1)^4
sage: factor(a)
(x + 1)^4
```

ここで式の因子分解では式の因子分解を行うメソッド factor() を利用して ‘a.factor()’ で分解したり、函数として ‘factor(a)’ で分解させています。どちらにしても多項式環は表に出てはいません。とは言え、多項式環が表から見えないだけで実際は厳として存在してい

ます。実際、SageMath が立ち上がった時点では変数 `x` の多項式環が定義されており、このことは ‘`type(x)`’ の結果が ‘`<type 'sage.symbolic.expression.Expression'>`’ と返却されることから容易に判ります。この返却値の意味することは SageMath ではあらかじめ変数 `x` の多項式環が定義されているということです。ただ変数を増やすこともそういった多項式環を意識せずに、変数として宣言するだけで良いようになっています。

では、面倒なことが多そうなオブジェクト指向の言語には一体どのような有り難さがあるのでしょうか？まず、プログラムを作成する上で、クラスの定義やメソッドの作成で、概念を実装し易いようによく考える必要がある点もあるでしょうが、より大規模なプログラム開発では顕著になります。その大きなものの一つが「**継承**」と呼ばれる機能です。つまり、既存のクラスを雛形として、新しいクラスを構築すると雛形のクラスに付随するメソッドがそのまま新しいクラスのメソッドとして使えるのです。たとえば貴方が開発した言語には実数があっても複素数がない言語だとします。その言語を使って複素数を扱う必要が生じたとき、実数を使って複素数を定義することになりますが、オブジェクト指向言語であれば実数というクラスに付随するメソッドの四則演算が貴方の定義する複素数クラスにそのまま継承されます。その結果、複素数上の四則演算を頭から構築する必要はなく、実数の四則演算を基に足りない処理を追加することで複素数の四則演算を定めることができます。

このようにオブジェクト指向の考えを取り入れている Python を基底に用いている SageMath で数学上の概念を表現、演算や処理も既存のものを活用できるという非常に大きな利点も持つことを意味します。もちろん、オブジェクト指向の有難味はこれだけではありません。プログラムを作成する前に対象の抽象化が必要とされるためにこの抽象化を行うことにより、より普遍性を持ったクラスを構築し、その実体として現実の事象にあてはめることでプログラムに普遍性を持たせることができることが挙げられるでしょう。これらのこととはプログラムを「**資産**」として考えると、それがその場限りや、精々、一世代のみの有効なものに限定されず、以降、類似の問題への適用が可能となることが最大の長所だと言えます。

1.1.5 電池込みだよ

SageMath は Python で記述された数式処理のためのパッケージ SymPy をその基底とし、数式処理だけでも汎用の **Maxima**、数論向けの **PARI/GP**、群論向けの **GAP**、可換環向けの **Singular** といった専門家向けのツールを組込んでいます。しかし、通常の利用ではそのことを意識する必要はありません。あくまでも利用者にとっては Python を使って処理を行うことしか見えません。また、必要に応じてこれらの専門ツールだけを表に出

して使えるようにもなっています、その意味でも、SageMathは独自のシステムというよりは「**数学上の問題を解くための計算機環境**」としての側面を強く持ちます。

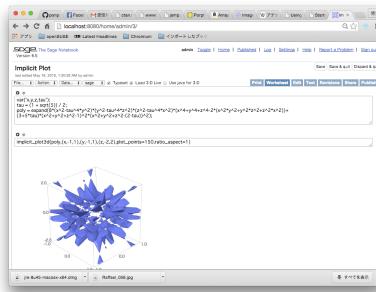


図 1.1 ウエブ・ブラウザを用いた SageMath のノートブック

そしてフロントエンドも仮想端末上でのテキストによるプリティプリントから、ウェブ・ブラウザ上で jsMath を利用したノートブック形式の二つが標準で選べます。後者のウェブ・ブラウザを使うノートブック形式は IPython notebook を利用したもので、ワークシートを公開することでアニメーション表示だけではなくネットワーク上で協調作業を也可能にし、既存の数式処理システムと比較しても見劣りしないどころか、その柔軟性の高さでは逆に勝るシステムです。この非常に柔軟な発想法は Python のいわゆる「**電池込みだよ (Battery Included)**」^{*6}を彷彿させるものです。

1.1.6 SageMath が使える環境は？

SageMath は UNIX 環境で動作します。これは既存の使えるシステムを活用しようとする方針から生じる制約で、この点は仕方がないことです。さて、UNIX 環境には Solaris, FreeBSD や Linux 等といったものがあります。さらに Apple の OSX 上で動作する SageMath もあり、こちらは Linux 版のように仮想端末上で動作するものと OSX のアプリケーションとして動作するものの二種類があります。さて、ここで SageMath が Linux 上で動作すると主張したところで、この Linux にはディストリビューションの違いがあり、さらには同じディストリビューションでもバージョンの違い、同じバージョンでも個々のライブラリ等のアップデートの違いといった些細な違いがあります。こういった要因が組み合わさると SageMath のような非常に複合的なシステムではインストールが大変な作業になり、さらに苦労して構築した環境が実際に正常に動作するかどうかも明瞭と言え難くなります。そこで SageMath の開発者が採用した手法の面白い点

^{*6} 子供の玩具を買って店から出て、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。

は、SageMath が必要な必要とするアプリケーションやライブラリといったものを一切合財を収録した巨大なパッケージとして配布することです。こうすることで微妙なバージョンの違いで悩まされる可能性を大きく下げることができます。さらに MS-Windows 環境のように UNIX 環境と比べてあまりにも異質な環境に無理に移植せずに仮想化環境 (VMware や Oracle VirtualBox) を活用しています。この場合はウェブ・ブラウザを使うので、MS-Windows 上の仮想計算機環境で動作している事情は末端の利用者に判りません。ちなみに SageMath はソースファイルで 280MB 程度、MS-Windows 版になると仮想計算機込みで 1GB 程度と大きなシステムですが、近年の計算機の能力の向上、記憶容量の増大、高速ネットワーク環境といった御利益があるためにさほどの負担にはなりません^{*7}。

この SageMath の導入については UNIX 系の OS であればソースファイルからの構築やバイナリ版の入手による導入の二つの方法が選べます。なお、SageMath のコンパイルは比較的重い処理で、十分な時間がなければバイナリ版の入手を勧めます。ただしバイナリ版でも圧縮した状態で 700MB 程、展開するだけで 2.5GB とインストールするだけでも 3GB は軽く必要になるので計算機に USB メモリスティック分の空容量が必要になることは覚悟して下さい。ただし、SageMath に必要なものが全て含まれるので数式処理のための統合的な環境を導入するのだと思えばそれ程、悪いものではないでしょう。

なお、OSX 版には前述のように UNIX 環境に対する仮想端末を使うバイナリ版と OSX の通常のアプリケーションとして利用できるものの二種類があります。何が何でも OSX を UNIX として利用する意図でもない限り OSX 版を素直に利用と良いとでしょう。その理由は Sage.app を起動すると Finder 上に SageMath のアイコンが現れて図 1.2 に示すように、そこから選べるメニューに SageMath のノートブック形式や仮想端末上の CUI 形式の SageMath、さらには Maxima、R や Singular 等のアプリケーションを個別に仮想端末上で動かすことができるからです。と、このように SageApp を導入するだけで色々なオマケが効せずに得られるのです。だから OSX で科学技術計算を Python で行なっている方は是非とも SageMath を導入すべきです。ちなみに主要な Linux のディストリビューションであれば科学技術計算を行うためのアプリケーションは難無くインストールできる筈ですが、OSX の場合は「FINK」、「MacPort」や「Homebrew」といったパッケージ

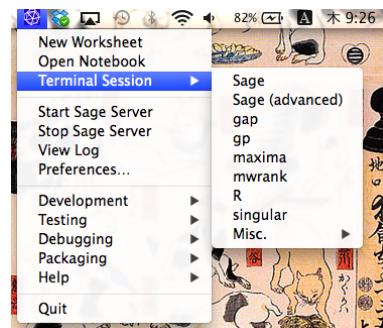


図 1.2 Sage.app のメニュー

^{*7} HD 画質の動画配信で 2 時間程度分でしかないのです。

管理システムが複数存在しているが残念なことに主要な Linux ディストリビューションのパッケージ管理と比べて優れたものではありません。だから、あれやこれやと導入していると微妙なライブラリ等の整合性の問題が生じ易く、Linux のようにシステムに任せて自動的にアップデートさせることもできないからです。しかし、数式処理、統計計算や数値計算にグラフ処理を行うためのアプリケーションが一通り揃った SageMath を導入することで、そのようなリスクを抑えて必要とする一通りの環境を容易に整えることができるというのが長所です^{*8}。

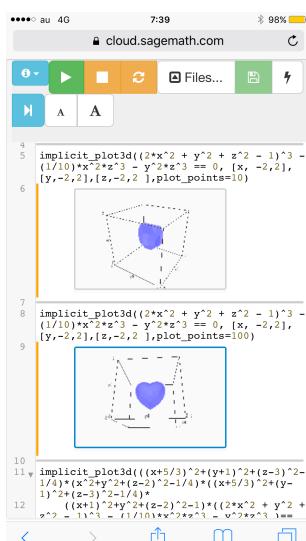


図 1.3 スマートフォンで利用

また、計算機のディスクや処理能力に余裕があるのであれば、いっそのこと MathLibre^{*9}を導入するのは如何でしょうか？ MathLibre は ISO イメージで 3G 程度になりますが、SageMath だけではなくさまざまな数学アプリケーションや TeX 環境、さらには数学アプリケーションに関する日本語文書も含まれています。SageMath で遊ぶも良いし、他のアプリケーションで遊ぶのも楽しいでしょう。そして、これらのアプリケーションの中から自分に本当に必要なものを見付けることもできるでしょう！ MathLibre はそのようなソフトウェアのカタログとしても使えるのです^{*10}。また、SageMath のためにディスクを割けない場合も USB メモリにインストールして、そこから起動といった手段もあります。

もし、SageMath をインストールすることや仮想計算機が利用できなければ、最後の手段として SageMathCloud(SMC)^{*11}を使うという手段があります。この SMC については §10 でも説明していますが、クラウドベースの SageMath でちょっとした計算であれば無課金で利用することができます。こちらは LATEX 等の揃った Ubuntu のシステムとしても使えるので非常に便利です。図 1.3 には iPhone6 Plus から SageMathCloud に接続し

^{*8} とは言え特定のアプリケーションを徹底的に利用しているときに SageMath に含まれるアプリケーションについても同様の環境が構築できるとは限りません。それと SageMath がバージョンアップしたときに折角、構築した環境を再度構築しなおす必要もあつたりとそれなりの手間が必要になります。この点は統合的な環境としての SageMath と専門的な自分の環境の双方を持つことで両者の長所を生かすことが妥当なところでしょう。

^{*9} 以前は KNOPPIX ベースの「KNOPPIX/Math」として開発・配布されていましたが、現在は Debian Live ベースになっています。

^{*10} とは言え、SageMath の容量が TeXLive 等の他のアプリケーションをこのところ顕著に圧迫しているのが現状です。

^{*11} <https://cloud.sagemath.com>

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。このように最近の大画面化したスマートフォンでもちょっととした計算や可視化さえもが可能で、Bluetooth に対応したキーボードとマウスがあれば快適な操作環境も得られるのです。

1.1.7 SageMath の情報

SageMath の情報は本家サイト: <http://www.sagemath.org> から得られます。また、日本語の情報源として Google Group に「**Sage Japan**」もありますが、こちらはあまり活発な活動をしておらず、今後に期待といった状態なので、この本を読まれた方は是非参加して盛り上げて頂ければと思います。さらに日本でも **Sage Days in Japan** というワークショップが開かれることがあります。売り物の書籍としては「群論の味わい-置換群で解き明かすルービックキューブと15パズル」が現在販売されている程度です。その他には「Sage for Newbies」の抄訳の「はじめての Sage」がウェブで公開されています^{*12}。この文書は MathLibre(KNOPPIX/Math) の DVD にも収録されていますが、SageMath に限らず計算機一般の話もあって非常に面白い文書ですが、この文書は Python について幾らかの知識を要求します。実際、SageMath は Python で新たに構成された処理言語を使うシステムではなく、むしろ、さまざまな数学上の問題に対処できる Python 環境でしかないので、逆に言えば、その場限りの処理言語を覚えるのではなくて、より汎用性のある Python を学習するだけでよいのです。実際、この Python を使う上で要求される水準はそれほど高いものではなく、それに加えて Python 自体が学習し易いという大きな長所があるのです。そして Python を習得するということは汎用性の高い言語環境を習得することになるという大きな御利益もあるのです。

1.2 SageMath の使い方

1.2.1 IPython を用いたユーザインターフェイス

さて、ここでは SageMath があらかじめ貴方の計算機に導入されていると仮定して解説します。もしもそうでなければ SageMathCloud で試してみるのも良いでしょう。最初に仮想端末上で `sage` と入力してみてください。すると Python の Shell である IPython が立ち上ります。この IPython は標準の Python 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:

```
| Sage Version 6.5, Release Date: 2015-02-17
| Type "notebook()" for the browser-based notebook interface.
```

^{*12} 筆者のサイトから辿って入手することができます。

| Type "help()" for help.

```
sage: 1 + 1
2
sage: 1 + 1;
sage: p1 = (x + 1)^3
sage: p1
(x + 1)^3
sage:
```

ここでは変数 p1 に ‘ $(x + 1)^3$ ’ を割当てていますが、この式は数式 $(x + 1)^3$ に対応します。ちなみに、Python の幕乗の演算子は “**” で、演算子 “^” はビット単位の演算子の排他的論理和 XOR に相当します。しかし、SageMath で演算子 “^” は幕乗の演算子として扱われていることに注意して下さい。ここで注意すべきことは、全般的に演算子 “^” が幕乗の演算子として置換えられたのではなく、単に SageMath で入力した式が多項式環の元として処理されているためです。この点を忘れて Python の演算子 “^” を使うプログラムを記述すると被演算子が全て Python のオブジェクトであれば本来の XOR の意味で処理されます。そして “sage:” が SageMath の入力行であることを示すプロンプトで、それに続いて式の入力を行います。なお、プロンプトが “sage:” になっていますが実質は Python のシェルである IPython がフロントエンドになっています。そして、入力式の評価は仮想端末では [Enter] キーで、ウェブ・ブラウザ上のノートブックであれば [Shift+Enter] キーで行います。また、SageMath には Maxima のような入力行の末尾であることを示す記号がありません。Maxima の行末尾の記号 “;” は、SageMath では入力行末尾に記号 “;” を置くと結果のエコーバックを行わないという作用があつて全く別の意味になります。また値の割当は記号 “=” で行いますが、この割当でエコーバックは行われません。そこで割当された値の確認は上の例のように変数名を入力するか、print 文^{*13}を用いるのも一手でしょう。

さて、SageMath はオブジェクト指向のシステムです。では p1 にはどのようなオブジェクトが束縛されているのでしょうか？ このオブジェクトの型を調べるときは函数 type() を用います：

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

この函数 type() の結果から変数 p1 には ‘symbolic.expression.Expression’ というクラスのオブジェクトが束縛されていることが判ります。では、このクラスのオブジェクトは

^{*13} SageMath を記述する Python は 2.x のために print は函数ではありません。

どのように処理できるのでしょうか？それを簡単に知る方法があります。SageMath に `p1.` と入力して `TAB` キーを押して下さい。すると下記のように一覧が表示されます：

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

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

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

すると ‘`p1.expand`’ と補完されて候補が幾つか表示されます。このように SageMath のユーザ・インターフェイスには `TAB` を用いた入力の補完機能があります。この補完機能は名前を補完したり、候補が複数存在するときはその候補を列記します。ここでやろうとしていることは名前 `p1` に束縛された多項式の展開で、この一覧に含まれている ‘`expand_log`’ は指数函数を含む式の展開、‘`expand_trig`’ は三角函数を含む式の展開に適した処理を行うメソッドで、処理をしようとしている式は整数係数の多項式のために ‘`expand`’ か ‘`expand_rational`’ で十分です。そこで ‘`expand()`’ で処理することにしましょう：

```
sage: p2 = p1.expand()
sage: p2
x^3 + 3*x^2 + 3*x + 1
sage: expand(p1)
x^3 + 3*x^2 + 3*x + 1
```

ここでの式の展開はオブジェクトの情報だけで式の展開が行えるために ‘`p1.expand()`’ で展開することができます。ところで、Python はマルチパラダイム言語と呼ばれる言語で、オブジェクト指向言語の特徴を全面に出すことなしに使うことができます。だから、命

令として ‘expand(p1)’ と記述することもできます。このように処理するときでも TAB キーを使った補完機能が有効です。このように他の Maxima や Mathematica といった数式処理システムと大差のない使い方もできることが分かるでしょう。

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

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

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

今度はエラーが出ました！このエラーの内容は「**名前 y が何なのか判らない**」ということです。ところで名前 x については文句を言っていないですね。これはなぜでしょうか？そこで函数 type() で名前 x の型を確認しておきましょう：

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

```
NameError Traceback (most recent call last)
<ipython-input-6-6848eca1ead4> in <module>()
      1 type(y)

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

この結果から名前 x には ‘sage.symbolic.expression.Expression’ という型のオブジェクトが束縛されていることが判りますが、名前 y には何も束縛されていないために未定義であることが ‘NameError:’ に続く ‘name 'y' is not defined」 という文言からも判ります。ここで「**NameError**」や「**name 'y'**」という表記には Python の「**名前空間 (name space)**」が関係しています。Python では扱うデータは全てオブジェクトとして捉えます。そしてオブジェクトの名札に対応するものが名前です。この名前とオブジェクトの対応関係のことを「**名前空間**」と呼びます。ここで名前とオブジェクトとの対応付けは「**名前への束縛**」と呼ばれる操作で行われます。そして、この名前空間に含まれている名前の一覧は函数 dir() で確認することができます。

さて、ここでのエラーは名前 y に対応するオブジェクトがないことに起因します。また、x でエラーが出なかった理由は名前 x に対応するオブジェクトが存在しているからです。そして、名前 x に対応するオブジェクトが存在している理由はあらかじめ変数として名前 x が定義されているからです。このように SageMath はマルチパラダイム言語だからと

いっても、全く無条件に Maxima のように利用できる訳ではなく、式を利用する上で必要な名前やオブジェクトをあらかじめ用意しておく必要があります。この場合は函数 var() を使って名前 y が式の変数であることを SageMath に教えてやれば良いのです：

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

ここでは ‘var(‘y’’’ で名前 y が名前空間に変数として加えられたために以降の処理で名前 y を含む式を入力しても、前のようなエラーが出なくなります。なお、複数の名前を変数として登録するときは単純に函数 var() の引数として文字列のリストを与えることになります：

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

ここで SageMath のリストは Python のリストと同一で、演算子 “[]” を使ってオブジェクトや名前の列を ‘[‘x’, ‘y’]’ のように括ったものです。なお、SageMath のリストの生成では使い易いように拡張されたものになっており、たとえば 1 から 10 までの自然数のリストの生成は ‘[1..10]’ で行うことができます：

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

この例では自然数のリストを生成し、それからリストの成分の取り出しを行っています。なお、Python では配列、リスト等の添字は Maxima や MATLAB のように 1 からではなく、C と同様に 0 から開始することに注意して下さい。また、Python では MATLAB 風のリスト処理として**スライス処理**と呼ばれる処理が行えます。ただし、添字が 0 から開始するために微妙な違いが生じるので注意が必要です。たとえば ‘L[0:5]’ で添字が 0 から 4

までの名前 L に束縛されたリストを返却します。また、添字を負の整数とすることでリストの末尾、つまり、右端からの成分を返却できます。たとえば ‘L[-4:-1]’ で添字が -4, -3, -2 の L の成分のリストを返却します。また、‘L[-1:-4:-1]’ で初期値が -1、増分 -1 で -4 までの添字、つまり、-1, -2, -3 の添字の L の成分リストを返却します。

次に代数方程式を解いてみましょう。SageMath では代数方程式を函数 solve() で解くことができます：

```
sage: solve(x-123,x)
[x == 123]
sage: solve(x^2-3*x+1 == 0,x)
[x == -1/2*sqrt(5) + 3/2, x == 1/2*sqrt(5) + 3/2]
sage: var(['y','z'])
(y, z)
sage: solve([2*x -y + z == 0, x^2-y^2+z^2-1 == 0,x^3-z^2+2*y-1 == 0],[x,y,z])
[[x == (0.0255946656987 - 1.63139339042*I), y == (0.0295897155531 - 2.19244726264*I), z == (-0.0215
```

この函数 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。ここで方程式は演算子 “==” を持つ式ですが、0 に等しくなる場合は演算子 “==” と 0 を削除して、式のみでも構いません。たとえば、方程式 $x - 123 = 0$ を解く場合、‘solve(x - 123 == 0,x)’ でも ‘solve(x-123,x)’ でも構いません。そして、函数 solve() は常にリストの書式で解を返却します。また、函数 solve() を使って連立方程式を解くこともできます。この場合、連立方程式は式のリストとして表現し、求めるべき変数も変数リストとして表現します。函数 solve は可能であれば代数的数^{*14}を用いた厳密解を返却しますが、代数的に解けないときは浮動小数点数を用いた近似解を返却します。

では最後に ‘help(expand)’ と入力してみましょう。こうすることで函数やモジュールのヘルプを読むことができます。このヘルプの内容は文書文字列 (docstring) と呼ばれるプログラム内に記述された文字列が対応します。Python では文書文字列に関しても PEP と呼ばれる規約があります (PEP-257 等)。なお、函数 help() は文書文字列の表示を行う Python 組込の函数ですが、前述の IPython^{*15}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` や `? expand` のように記号 “?” のうしろに調べる事項を記述します。この記号 “?” を使ったオンラインヘルプは IPython の機能のために CPython のシェルでは使えません。

^{*14} 整数係数の多項式の零点になる数です。代表的な数として、整数、有理数、純虚数や n 乗根が挙げられます。

^{*15} IPython と matplotlib の組合せは MATLAB 利用者にとっても馴染易い環境になります。

1.2.2 ノートブック形式のユーザインターフェイス

SageMath にはよりモダンな環境があります。この環境はノートブックを模したもので、出力式を美しくレンダリングしたり、グラフやアニメーションのノートブックへの表示といったことが行えます。このノートブックを実現するために IPython Notebook が用いられており、おおよその操作は IPython Notebook に準じることになります。なお、§10 で解説する SageMathCloud では IPython Notebook のフロントエンドを GNU R や Julia といった Python 以外の言語にも対応できるようにした IPython 後継の Jupyter が用いられています^{*16}。

さて、ノートブック形式で利用するときは Unix 環境であれば仮想端末上で `sage -notebook` と入力するか、仮想端末上で SageMath を起動させていれば `notebook()` と入力することでウェブ ブラウザが立ち上がり、そのウェブ ブラウザに SageMath の Notebook が表示されます。MathLibre 上で SageMath を使用するのであれば \sqrt{Math} メニューや Launcher から SageMath を選択すればノートブック形式の SageMath が立ち上がります。また OSX 上の Sage.app を利用しているのであれば、普通の OSX アプリケーションと同様に Launchpad から呼び出せばウェブ・ブラウザを起動してノートブック形式の SageMath が立ち上がります。また、Finder 上には SageMath のアイコンメニューが現われる所以、そこからノートブック形式で SageMath を開いたり、仮想端末から SageMath を起動することもできます。ただし、ノートブック形式で SageMath をはじめて立上げるときにパスワードの設定を最初に行う必要があります。このパスワードの設定後にノートブック形式でウェブ ブラウザに表示され、`New Worksheet` ボタンを押すとノートブック名を設定するダイアログが表示され、それからワークシートへの入力ができるようになります。ここで `jsMath` がインストールされた環境であれば、`Typeset` のチェックボックスにチェックを入れておけば数式が綺麗にレンダリングされます。

また、あなたの計算機（あるいは携帯電話!）がインターネットに接続しているのであれば <https://cloud.sagemath.com/> に接続してみましょう。このサイトは JavaScript に対応したウェブ・ブラウザであれば式の表示や 2 次元グラフとそのアニメーションが利用可能で、PC だけではなく Android 携帯、iPhone や iPad でも利用できます。この SageMathCloud を利用するためには利用者登録が必要ですが、無課金で利用することができます。この SageMathCloud の詳細は §10 を参照して下さい。

^{*16} IPython Notebook もその後継の Jupyter も、Mathematica 流儀のセル単位のノートブックで、セルの評価の方法や命令等の補完といったことは Mathematica と似た操作です。

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

ここでは例として画像の読み込みを行ってみましょう. SageMath には画像処理用のライブラリが色々ありますが, ここでは Matplotlib を用いることにします. この Matplotlib ライブラリについては §8 でも解説しますが, このライブラリのモジュールを用いることで画像を多次元の数値配列に変換することができるのです. さて, Matplotlib を利用することにしましたが, ここで SageMath 上でライブラリの読み込みを行う必要があります. SageMath は Python を骨格とするので, 結局, ここでしなければならないことは Python でライブラリを読み込むことと同じです. このライブラリの読み込みは Python では import 文を用います:

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

この例では画像の描画用に Matplotlib ライブラリの pyplot モジュール, 画像読み込み用の image モジュールをそれぞれ import で SageMath に読み込みます. このときに描画用ライブラリは ‘plt’, 画像読み込みライブラリは ‘mi’ と読み替えておきます. ここで一方が ‘import ...’ でもう一方が ‘from ...’ になっていますが, これらはライブラリを構成するモジュールの呼び出し方の例です. つまり, Python ではモジュールは階層構造を持つており, Matplotlib の直下に pyplot と image というモジュールがあります^{*17}. 最初の ‘import ...’ では ‘matplotlib.pyplot’ とありますが, このような形で直接 pyplot の読み込みを行っています. 一方の ‘from ...’ の例では, ‘from matplotlib’ から親ライブラリ名を指定し, その下にある ‘image’ モジュールを読み込めという命令になっています. さて, ここで ‘as’ を用いてモジュールの読み替えを行っていますが, その理由は次の行で判ります. まず画像の読み込みで ‘mi.imread()’ とされていますね. これはモジュール image の下にある函数 imread() を用いるという意味で, 本来ならば ‘matplotlib.image.imread()’ としなけれ

^{*17} この階層はクラスの継承関係に基づくものです.

ばならないところを ‘matplotlib.image’ の箇所を ‘mi’ で読み替を行うことで ‘mi.imread()’ とできます。つまり、このような呼出ができるようにすることが読み替です。さて、画像はカレントディレクトリからみて ‘Documents’ ディレクトリ内にある ‘ScuolaDiAtene.png’ ファイルです。この画像をここでは函数 imread() で読み込みますが、この函数で読み込まれた画像は多次元の数値配列に変換されます。この数値配列は NumPy と呼ばれるライブラリで定義された数値配列になります。実際、この画像オブジェクトが Python でどのような代物になっているかは函数 type() を使って調べることができます。この例では ‘numpy.ndarray’ という文字列が結果に現われていますね。これは読み込んだ画像のインスタンス i1 が NumPy ライブラリで定義された多次元数値配列であることを示しています。そして、この数値配列の具体的の形（大きさ）はメソッド shape() で調べることができます。ここでは ‘(509, 800, 3)’ と返却されていますね。このことから本来の画像は 509×800 の大きさの RGB 画像であることが判ります。というのも最初の二つの整数値が画像の縦と横の画素数で、最後の ‘3’ が Red, Green, Blue の RGB に対応する数値配列であることを示しています。なお、函数 imread() で読み込んだ画像は 0 から 1 までの浮動小数点数で輝度が表現された数値配列として生成されています。それから数値配列を matrix_plot を使って表示することができるのです。ここで表示した画像の様子を図 1.4 に示しておきましょう：



図 1.4 読込画像の表示例

ここまで処理は SageMath でなくても Python で PIL/Pillow, NumPy や PyLab があればできることです。そこで折角ノートブック形式の UI を使っているので、今度は絵をノートに貼ってしまいましょう。これには特殊な処理は不要です。単純に Image モジュール

ルの関数 `save()` を使って画像を保存してしまえば、あとは SageMath のノートブック側でその画像を貼ってくれます。ここでは ‘`im.save('test.png')`’ で画像の保存を行うと、この式のセルのすぐ下に画像を貼ってくれます。とは言え、表示領域を自動調整してくれる訳ではありません：

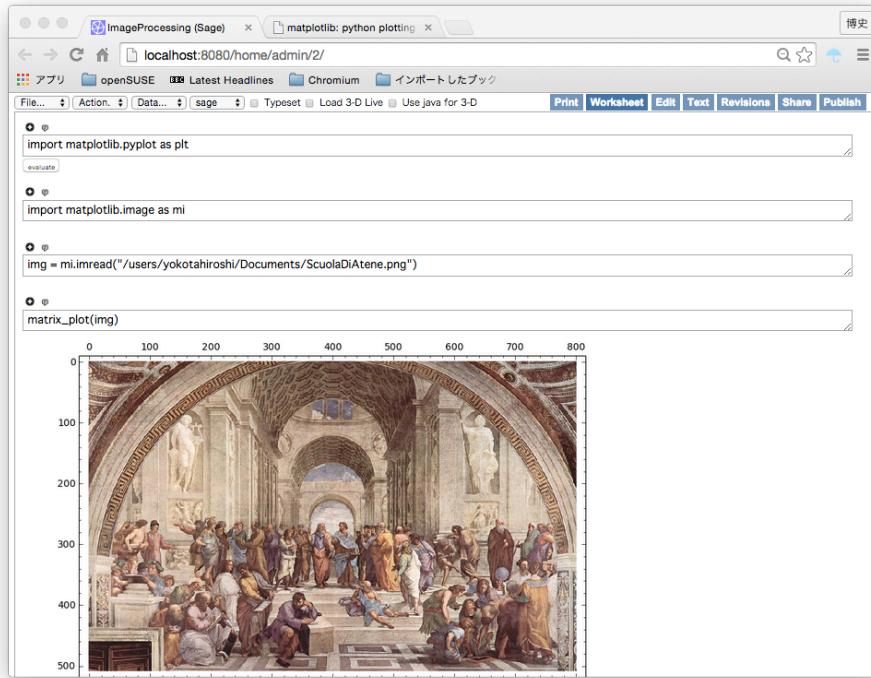


図 1.5 ノートブックへの画像の表示例

と、表示されます。さらに、SageMath のノートブック形式のユーザ・インターフェイスで処理した結果はワークシートとして保存可能で、そのワークシートを一般に公開したり、共有することも可能なのです！

いかがでしょうか？ このように SageMath は商用の数式処理システムと大差がない程の使い勝手と機能を実現させているのです。そして、数式処理システムと身構える必要もなしに、ノートブック環境を持った Python 環境としても使えばよいのです。だから、Python を使って科学技術計算をしたいのであれば、なおさら SageMath を導入しないという理由は皆無なのです。

1.3 SageMath が目指すものは？

まず SageMath の大きな特徴は、それ自体が一つの大きな Python 環境であるということです。ここで「**大きな Python 環境である**」と述べましたが、SageMath のように数多くの OSS を利用することを考えれば利用するライブラリやアプリケーションの整合性という非常に厄介な問題が前面に出てきます。この SageMath の動作環境に Linux がありますが、この Linux 環境と一言で括ってみても、実際は Debian, RedHat, openSUSE, Fedra, Ubuntu, CentOS や Gentoo 等々といったさまざまなディストリビューションと呼ばれるパッケージ管理の手法や/etc 以下のディレクトリ構成の違いを持った Linux があり、その同じ Linux のディストリビューションでも、公開された時期やライブラリの更新から生じる微妙な差異も、このように大きなアプリケーションになると無視できなくなります。では、SymPy の能力を越える処理はどうしているのでしょうか？そのためには SageMath は数式処理エンジンとして GAP, Maxima, PARI/GP, Singular 等の専門の数式処理システムを取り込んでいます。まず、GAP は群論、PARI/GP は数論、Singular は可換環向け、そして Maxima が汎用の数式処理です。ここで Maxima は Common Lisp 上で動作するために組込に適した Common Lisp の実装である ECL が用いられています。さて、このように数式処理だけでも 4 種類あり、専門分野だけでなく基盤となる処理言語もそれぞれが異なります。これらの数式処理のアプリケーションを繋げているものが Python という処理言語です。この Python の拡張は各種のパッケージを追加で読み込むことで行いますが、SymPy という数式処理を Python 上で行うためのパッケージがあるので基本的な数式の定義や処理は SymPy に任せています。

それから SageMath は統計処理アプリケーションの GNU R を包含しています。この GNU R は関連書籍も多く出版されており、統計処理で標準的なアプリケーションと言えます。SageMath に GNU R が含まれているということは数式処理に加えて高度な統計処理も可能であることを意味します。ただし、SageMath に含まれた GNU R は公開された R 向けに公開されたパッケージを全て含むものではなく基本的なパッケージを含むものです。この GNU R で公開されたパッケージの多くを追加することも可能ですが、全てができる訳はないようです。とは言え、GNU R を単体で用いることと比べ、さまざまなツールを駆使して作業を進めることができます。

では数値計算はどうでしょうか？数値計算向けのライブラリで BLAS や LAPACK が有名です。これらは線形代数の諸問題を効率的に解くために数値行列の高速処理を目的とした C や FORTRAN 向けのライブラリで netlib で公開されています。ただし、これらのライブラリは特定の CPU に最適化が行われたものではありません。一般的な数式処理で

は多倍長精度の浮動小数点数計算が可能な反面、(倍精度の) 浮動小数点数に限定した数値計算が不得手な傾向があります。そのために SageMath に含まれている Maxima のように LAPACK ライブラリを持つものもありますが、ここで Maxima の LAPACK は含まれる函数を Common Lisp のコードに単純に変換したもので計算機の構造にまで踏込んで最適化したライブラリではありません。それに対して SageMath の数値計算は遙かに本格的です。SageMath に付属の線形代数ライブラリには netlib 版の BLAS, LAPACK に加えて ATLAS があります。この ATLAS は CPU への最適化が行われた LAPACK ライブラリで、当然、netlib で公開されている素の BLAS や LAPACK 以上の高速処理が望めます^{*18}。そして、これらのライブラリを Python で利用するための NumPy があります。そのために一般的な数式処理で苦手とされる数値計算さえも、数値行列処理に適した MATLAB と大差ない程度の処理が可能になります。

さらに SageMath には上述の数学関連のアプリケーションだけではなく、SQLite や ZODB3 といったデータベース関連のパッケージさえも含まれています。このことから分るように SageMath は単に豪華な Maxima のような単体の数式処理システムを目指しているのだけではなく、実用に耐え得る本格的な数値計算や数式処理機能を持ち、数学の諸問題に対処可能な環境を構築するという非常に野心的なプロジェクトであるという側面が如実に現われているのです。

1.4 SageMath が包含するアプリケーションとライブラリ

SageMath はさまざまなアプリケーションやライブラリで構成された巨大なシステムですが、それらを Python を使って繋ぎ合せたもので、その繋ぎ合せ方も一般の利用者が Python のことさえ理解さえしていれば容易に使えるようにできています。とは言え、どのようなアプリケーションやライブラリで SageMath が成り立っているかを知っていれば、SageMath を利用するにあたって「車輪の再発明」をわざわざするような二度手間を避けることができるでしょう。そのために、ここでは SageMath を構成するアプリケーションやライブラリ等の概要をまとめておきます。

1.4.1 SageMath の中核としての Python

SageMath は Python を使って骨格が構成されています。現在、Python には 2.x 系と 3.x 系があり、SageMath は 2.x 系で構築されています。ちなみに 3.x 系は 2.x にある古典的なクラス型が廃止され、print 文が函数になる等の細かな違いがあって上位互換性があ

^{*18} とは言え、商用の LAPACK(Intel の MKL や AMD の ACML) や GotoBLAS と最適化の度合いで比べるとやや劣ります。

ると言えません^{*19}. そして, この Python という言語は SageMath というシステムを構築するだけではなく, 全体を統括する処理言語としても用いられています. それから利用者と SageMath との対話処理を行うためのシェル^{*20}として **IPython** を用いています. この IPython を用いることで入力の履歴の保存と再利用, 入力行の編集処理といった利用者に役立つ機能を提供できます. さらにウェブ・ブラウザ上の GUI として IPython Notebook が用いられています. この IPython Notebook はノートブック形式の UI をウェブ・ブラウザ上で実現するもので, ノートブックの共用といったことも可能にします. また, IPython Notebook から Python だけではなく他の言語 (GNU R, Julia, Bash 等) も利用できるように言語に依存しない部分の切り分けを行った Jupyter があり, こちらは SageMathCloud にて GNU R, Julia や Bash を利用するために用意されています. この SageMathCloud については §10 を参照して下さい.

つぎに SageMath の骨格を形成するのが SymPy です. この SymPy は Python 上で多項式の四則演算, 展開や因子分解, 初等函数の初步的な微分積分といった比較的簡易な数式処理を可能とするライブラリで, オプションの描画機能以外では Python の他のライブラリに依存しません. この SymPy によって多項式や初等函数等の数式の記号処理に必要な基本的なオブジェクトが Python で定義され, あとはそれらを拡張することで Maxima や Singular といったより本格的な数式処理にオブジェクトを引き渡し, 処理した結果を受け取るという手法を採用しています. それから大規模な数値多次元配列を扱うための基本的な数値計算ライブラリ **NumPy** があります. この NumPy を基に科学技術計算を強化した **SciPy** は信号処理, フーリエ変換, 数値積分, 線形代数, 行列処理, 最適化や統計処理ライブラリ等を含みます. そして, NumPy で定義された数値配列の可視化を行うライブラリとして **Matplotlib** があり, NumPy を中核にした SciPy や Matplotlib の組合せによって Python で MATLAB 並の数値配列の処理能力とその可視化を可能にしています. 一般的に数式処理は数値配列の計算で MATLAB のような数値計算システムに対して精度では勝ることがあっても処理速度で劣ったり, そもそも中規模な数値配列ですら能力的に扱えない面がありますが^{*21}, SageMath は NumPy を中核とする SciPy 等のライブラリを組み合わせて用いることで大規模な倍精度の浮動小数点数の数値配列の処理でさえも MATLAB と同水準の処理を可能にしています.

さらに Python で画像を処理するための基本的なライブラリとして **PIL** があります.

*19 詳細は「What's New In Python 3.0」(<https://docs.python.org/3/whatsnew/3.0.html>) を参照して下さい.

*20 利用者とアプリケーション (ここでは Python) との間を取り持つ役割をするアプリケーションです.

*21 そもそも LAPACK 等の数値行列計算ライブラリを実装していなかったり, 外部ライブラリとして実装したとしても最適化が行われていなかったりします.

この PIL を用いることで画像データの取込、画像データへの出力とさまざまな画像処理が可能になります。なお、PIL 自体は setuptools に対応しておらず、Python 2.X 環境のみの対応で開発が止っているために、この PIL から分岐した Pillow が Python で広く用いられています。

また、Python で記述されたライブラリの導入のために **setuptools** もありますが、**setuptools** を用いなくとも SageMath に用意されたライブラリを別途入手して拡張することも可能です。このことはのちの章で述べることにします。また技術計算分野では沢山の C で記述されたライブラリが存在します。これらのライブラリを利用すれば同機能のライブラリを Python で構築する手間が省けるだけではなく、C を併用することでより高速な処理が可能となるでしょう。その目的のために **Cython** があります。

他の Python パッケージを列記しておきましょう。**Pygments** は Python の構文に従って命令等に着色したテキストを出力するためのライブラリです。**Py-CRYPTO** は Python の暗号化ツールキットで、たとえば SSH 接続でネットワークに繋った計算機（実物、仮想も含めて）間でのデータ転送を行うといったときに必要となります。**python_gnults** は libgnutls のラッパーです。**SQLAlchemy** は Python のための ORM(Object Relational Mapping) ライブラリです。これはデータベースとオブジェクト指向言語との間のデータの変換で用いられます。それから **mpmath** は多倍長浮動小数点数演算ライブラリ、**Pynac** は C++ の数式処理ライブラリ GiNaC(GiNaC is Not a Computer algebra System) の Python への実装です。**CVXOPT** は Python の凸最適化(convex optimization) のためのパッケージです。**NetworkX** は Python で記述されたネットワーク分析用のライブラリですがグラフの描画も可能なので何かと重宝します。**Sphinx** は Python の文書文字列の整形で用いられる標準的なツールの一つで、SageMath では文書生成ツールとして用いられます。ちなみに SageMath のマニュアルは **reST(reStructuredText)** と呼ばれる組版指示(markup)言語のテキストファイルとして記述されており、Python で記述された **Docutils** の命令を用いることで reST から HTML, XML, LaTeX, ODF 等の各書式に変換することができます。**Pexpect** は Python 向けの疑似端末モジュールで、他のプログラムの制御や自動化のためのツールです。名前から予想されるように (Pexpect=Python+expect)，Unix の expect 命令のような処理ができます。**Mercurial** はライブラリ等のバージョン管理のために Python で記述された分散型のバージョン管理システムです。**Twisted** はイベント駆動型のネットワークプログラミングフレームワークです。**MoinMoin** はウェブ・プラウザを利用したユーザ・インターフェイスの構築で用いられています。この MoinMoin は Python で記述された Wiki のクローンで **PikiPiki** を基としたものです。ちなみに「**Moin moin**」の意味ははドイツ語の挨拶の俗語で「おはよう」、「こんにちは」、「こ

んばんは」等の意味に対応することです。また、SageMath には Jinja2 と呼ばれるテンプレートエンジンがあります。これは計算結果を定型的な HTML 等の形式のファイルに落とし込むときに有用です。使い方は、あらかじめ出力すべきファイルに対応するテンプレートを用意し、あとは Python 風の言語を利用して目的の文書を機械的に生成することができます。たとえば、計算結果を HTML で表示したいときに Jinja を使って雛形ファイルを準備しておいて、あとは結果を流し込んで結果の可視化を行うこともできます。ここで名前の「Jinja」は「神社」のことで Jinja の公式サイト^{*22} には宮島の大鳥居がモデルと思われるバナーがあります。

1.4.2 Python 以外の処理言語

SageMath は Python でさまざまなアプリケーションやライブラリを繋ぎ合せたシステムです。逆に言えばそいつたアプリケーションやライブラリ一式が一纏めになっており、Python を介さずに単体で使うこともできますが、このために UNIX 環境ではパスの設定を適切に行う必要があります。また、OSX 版の Sage.App の場合は Finder に表示される Sage メニューから「Terminal Session」→「Misc.」で ecl, Python, iPython といった主要なアプリケーション単体の起動が行えるだけではなく、Bash を起動させる「sh」を選択することで、SageMath から利用できるアプリケーションやライブラリを利用するための環境設定が行われた Bash が起動されるので、その環境で SageMath に付属の各種アプリケーションに利用が可能です。

1.4.3 数式処理と統計に関連するアプリケーション

SageMath の基となっている Python パッケージは SymPy です。この SymPy 単体だけでも多項式の展開や因子分解といった基本的な処理、初等函数の微分や積分といった計算が可能ですが、この汎用の数式処理システム Maxima を併用することで数式処理システムとしての SageMath の骨格を構成します。ただし、Maxima は Common LISP 上で動作する数式処理であるために数値計算はそれ程得意ではありません。また、汎用の数式処理するために各分野の専門アプリケーションと比べると機能的に劣ります。そのためには SageMath では数学の専門分野で主要なアプリケーションが利用できます。たとえば、数論に関しては数論専門の数式処理の PARI/GP や GAP、可換環論向けには Singular を、有限群向けには GAP を利用します。まず、PARI/GP は数論向けに優れたライブラリを持っています。ちなみに数式処理システム RISA/Asir も SageMath と同様に数論函数などで PARI ライブラリを利用しています。それから GAP は莫大な有限群のデータを持っています。それから可換環論向けの Singular ですが、数式処理システム Maxima は 1960

^{*22} <http://jinja.pocoo.org>

年代に開発が進められた古参の数式処理システムであるため、近年、応用が進んでいる Gröbner(標準) 基底を扱うパッケージの機能はそれ程高いものではありません。この弱点を補うために可換環専門の数式処理の Singular を用いています。それから統計処理については GNU R を包含しているのでこの GNU R を用います。ちなみに GNU R には莫大なパッケージが CRAN^{*23}で公開されていますが、SageMath 付属の GNU R にはこれらのパッケージが全て含まれていません。しかし、岡田氏の RjpWiki^{*24}の「追加パッケージをなんでもかんでも追加する」でパッケージを追加する方法が紹介されており、この方法で Sage の GNU R に全部とは言えないものの相当数のパッケージを入れることができます。

1.4.4 光線追跡

3D グラフィックス処理では光線追跡ソフトウェアの Tachyon があります。Tachyon には3次元分子可視化プログラムの VMD(Visual Molecular Dynamics) が組込まれています。ところのこの VMD は本来は分子動力学計算アプリケーションのプリ・ポストアプリケーションとして用いられていたようですが、さまざまな方面で利用されています。このように SageMath では本来の目的とは別の目的で機能が用いられているアプリケーションが多くあります。ここで SageMath で Tachyon を利用した例を示しておきます：

```
t6 = Tachyon(camera_center=(0,-4,1), xres = 800, yres = 600,
               raydepth = 12, aspectratio=.75, antialiasing = True)
t6.light((0.02,0.012,0.001), 0.01, (1,0,0))
t6.light((0,0,10), 0.01, (0,0,1))
t6.texture('s', color = (.8,1,1), opacity = .9, specular = .95,
           diffuse = .3, ambient = 0.05)
t6.texture('p', color = (0,0,1), opacity = 1, specular = .2)
t6.sphere((-1,-.57735,-0.7071),1,'s')
t6.sphere((1,-.57735,-0.7071),1,'s')
t6.sphere((0,1.15465,-0.7071),1,'s')
t6.sphere((0,0,0.9259),1,'s')
t6.plane((0,0,-1.9259),(0,0,1),'p')
t6.show()
```

この例では Tachyon のオブジェクトをあらかじめ生成し、そのオブジェクトに光線追跡すべきオブジェクトや光源の情報を追加してメソッド show() ではじめて光線追跡の計算実行と結果表示を行います。ここで生成した画像を図 1.6 に示しておきますが、メソッド show() によって SageMath を仮想端末上で動かしている場合は外部アプリケーション

^{*23} <http://cran.r-project.org/>

^{*24} <http://www.okada.jp.org/RWiki/>

で、ノートブックであればそのノートブックに画像が表示されます：

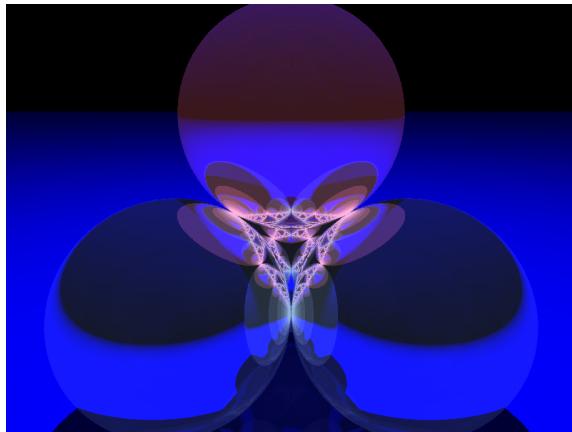


図 1.6 Sage から Tachyon を利用

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

1.4.5 数値計算ライブラリ

数値行列計算では **BLAS** や **LAPACK** といったライブラリが有名です。ここで「**LAPACK(Linear Algebra PACKage)**」は数値行列処理を目的とするライブラリで、線形方程式の求解や固有値問題が扱えます。この LAPACK は「**BLAS(Basic Linear Algebra Subprograms)**」と呼ばれるベクトルと行列の基本的な計算を効率良く処理することを目的としたルーチン群上で構築されています。BLAS の公式標準実装は **netlib** *²⁵で公開されていますが、BLAS の性能が LAPACK の性能に直接関係するために最適化を行った BLAS が幾つか存在します。まず、フリーのものでは「**ATLAS(Automatically Tuned ALgebra Software)**」と「**GotoBLAS2(後藤 BLAS)**」*²⁶で、商用のものには Intel の「**MKL(Math Kernel Library)**」や AMD の「**ACML(AMD Core Math Library)**」です。これらの最適化 BLAS については一般的な利用者は MKL, ATLAS や GotoBLAS で十分、ヘビーユーザなら MKL か

*²⁵ <http://www.netlib.org/blas/>

*²⁶ ATLAS と GotoBLAS2 はに BSDL で配布されています。

GotoBLAS, 新規アルゴリズム開発等の学術向け用途には GotoBLAS が良いようです [9]. なお, BLAS は数値行列処理を高速に行うことを目的にしたライブラリですが, 実際に効果が出るのはサイズの大きな計算量を必要とする行列で, 逆に計算量が極端に小さな行列では遅くなる傾向があります. そのために BLAS を単純にブラックボックスとみなして使うべきではなく, どのような原理で動作しているかを理解しておく必要があります [9]. なお, SageMath にはこの中で ATLAS が組込まれており, SageMath をソースファイルからコンパイルするときは, この ATLAS 関連のコンパイルで相当な時間を費します.

GSL(GNU Scientific Library) は C と C++ 向けの科学技術計算用のライブラリで, 亂数生成, 特殊函数, 最小二乗法を用いたデータ補間等の 1000 を越える数学ルーチンを含んでいます.

1.4.6 テキスト処理

iconv(Internet Codeset Conversion Library) は Shift-JIS, EUC, UTF-8 等の文字コード変換で用いられる標準 API です.

1.4.7 端末関連

freetype はフォントエンジンを実装したライブラリで, フォント関連の処理ができます. **readline** は仮想端末上で Emacs 風の編集や履歴操作を可能とします. **termcap** は端末機能が記載されたデータベースです.

1.4.8 その他のライブラリ:

boehm_gc はガーベジコレクションを行うためのライブラリです. **FLINT(Fast Library for Number Theory)** は数論の計算を行うために最適化された C のライブラリであり, GMP との併用を前提としています. **MPIR** は多倍長整数, 有理数ライブラリ, **MPFI** は区間数演算ライブラリ (INRIA) です. 区間数は数を区間として表現したもので精度保証 (物理定数や測定誤差のプレを包み込むことで精度の保証を行うこと) で用いられたりします. 区間数演算ライブラリは, その区間数に対して四則演算を導入するものです. **MPFR** は多倍長浮動小数点演算ライブラリ (INRIA), **NTL(Number Theory Library)** は数論向けの C++ ライブラリで GMP と併用することを前提にしています. **IML** は整数行列ライブラリで ATLAS/BLAS+GMP と併用が前提になっています. **cddlib** は多面体生成ライブラリ, **Cephes** は数学ライブラリ, **lcalc** は L-函数の計算, **GD** は画像処理ライブラリ, **PPL** は Parma Polyhedra Library の略記であることから判るように多面体を扱うためのライブラリです. **Givaro** は数論や代数計算のためのライ

ブライアリで $GP(q)$ や \mathbf{Z}/p 上のベクトルや行列が扱えます。 **LibBox** は、密、あるいは疎行列に対する計算線形代数計算向けのライブラリです。 **zn_poly** は $\mathbf{Z}/n[x]$ 上の多項式向けの計算ライブラリ、**libgpg_error** は GnuPG 関連のライブラリです。 それから **Symmetrica** は古典群と対称群の表現、対称函数や対称式、有限群の函手を扱い、**Clique** はグラフのクリーク (clique) 探索が行えます。 また、**Gfan** は Gröbner 基底の計算が行え、**GLPK(GNU Linear Programming Kit)** は線形計画法 (LP) や混合整数計画問題 (MIP) を解くためのソルバーです。 そして **fplll** は LLL-reduces euclidean lattices、**PALP** は多面体の頂点や面の数え上げを行うルーチンを含んでいます。**cvxopt** は凸最適化を行うためのライブラリです。**GMP-ECM** は整数の素因数分解向けの Curve Method を収録したツールです。

1.4.9 データベース関連:

SQLite は比較的軽量な SQL データベースエンジンで、小規模なデータベースの構築に向いています。この SQLite はサーバが不要で、さらに他の RDBM のような事前設定を行う必要がありません。だから必要に応じて思い立ったときに即座に使うことができます。この SQLite を SageMath から利用するために sqlite3 パッケージを利用します。この SQLite の利用については §6 を参照して下さい。それから **SQLAlchemy** は Python の SQL ツールキットです。**ZODB3** は Zope Objective Database で、**OpenCDK** は公開暗号開発キットです。

1.4.10 ユーティリティ:

f2c は FORTRAN のコードを C のコードに変換するためのツールです。**patch** は文字通りソースファイルのパッチ当てに用いられます。**lint** は C コードの検証が行えるツールです。**gnutils**、一般的な画像ライブラリとして **libpng** と **zlib** があります。**libgrypt** は GnuPG で用いられているコードに基いた汎用的な暗号ライブラリです。**SCons** はソフトウェアのビルドツールです。

1.4.11 Sage のパッケージ

extcode,boost-cropped,ratpoints は橙円曲線上の有理点を計算するためのパッケージです。**sympow**、**conway_polynomials** は Conway 多項式のパッケージです。**gdmodule,rubiks** はルービックキューブを Sage で扱うためのパッケージです。**genus2reduction**、**libm4ri,eclib,polybori,elliptic_curves** は橙円曲線パッケージです。**graphs** はグラフ理論向けパッケージです。**polytopes_db** は polytope のデータベースです。

1.5 一般の Python パッケージ

SageMath に標準に含まれていない Python パッケージもインストールすることが可能なことがあります。一つは SageMath の環境で Python パッケージを構築してしまえば良いのです。また、あらかじめ用意されたパッケージもあり、そのパッケージを利用するためには該当するパッケージを入手して `sage -i パッケージ` でインストールすることができます（もちろん、全ての環境で可能とは限りません）。ここで、「`sage -i`」でパッケージをインストールする場合はパッケージの構成は SageMath のディレクトリの直下の `spkg/build` で行われ、正常にインストールされると生成されません。なお、パッケージの構築やインストールの記録は `logs/pkgs` にパッケージ名に修飾子が「`.log`」のテキストファイルとして収められます。

1.6 この本の方針

さて、この本はどのような方針で進めるべきでしょうか？一つの方法は SageMath のマニュアルを片っ端から翻訳して載せることです。そうすると SageMath だけではなく、Maxima, Singular といったアプリケーションの解説も必要になり、幾ら紙面があっても足りないでしょう。もう一つは理論的背景を解説することです。こちらも基本概念から解説することはとても大変なことです。だからと言って高校数学程度の幾つかの例題で SageMath がどのように使えるかを試すことだけでは物足らなく、別に本にしなくてもワークシートをそのままインターネット上に公開する程度で十分でしょう。

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

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

この分類からも判るように、SageMath を使いこなすためには Python の知識が重要であることが判りますね。だからといって **Python というプログラム言語の A から Ω まで** をこの本で開陳する必要はないと判断します。そこで、この本では「**SageMath を使っ**

てみたいという人」を対象にし, SageMath を調べることで Python にも慣れてしまう戦術で進めるつもりです.

そして私は特に「Python を使って数学をどのように表現しようとし, 実際にどう表現しているのか」ということを中心に探って行きたいと思っています. 具体的には SageMath で数学の対象がどのように表現されているかを, PEP と呼ばれる Python の開発で重要な規格書に相当する文書の記載や, 実際の実装方法を SageMath のソースファイルを必要とあらば観察し, これらの表現がどのように Maxima などに引渡されているかを確認するのです. このことは何時か SageMath の拡張や類似の環境を構築する際に大きく役立つ筈です. そして, この方針こそが SageMath を育てて行くという観点からも重要なことではないかと私は思っています.

第2章

オブジェクト指向について

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

2.1 SageMath と Python の関係

SageMath は Python で構築した数式処理アプリケーションというよりも既存の数学アプリケーションを Python で繋ぎ合せて創り上げた数学のための統合環境です。したがって数学に関するアプリケーションを使う必要があるって Python が自在に使える方にとっては是が非でも徹底的に使い倒すべきシステムなのです。そして Python について不案内な人にとっては Python がどのような言語であるかを知つておくに越したことはないという結論になります。

2.2 Python はどのような言語か

Python がどのような言語であるかは Google 等の検索エンジンを使って検索したり、 Wikipedia で調べればおおよそのことが判ります。実際に Wikipedia で調べてみるとオランダ人の **グイード ヴァン ロッスム (Guido van Rossum)** ^{*1} が作った OSS(Open Source Software) のプログラム言語であること、オブジェクト指向プログラミングに対応していること、それに加えて BBC 制作のコメディ番組「**空飛ぶモンティ・パイソン**」への言及もあります。さらに記事を読み進めてゆくとプログラマの生産性とコードの信頼性を重視した設計、核となる構文や文法を必要最小限に抑えていること、そして大規模な標準ライブラリがあることが書いてあります。

しかし、Python の特徴には機能、拡張性や言語的な仕様に留まらない文化的な側面もあります。たとえば Python の開発ではコミュニティの存在が前提にあり、そこで議論を反映した「**PEP**」と呼ばれる文書を基に開発が進められている点が挙げられます。なお SageMath は Python の上に立脚したプロジェクトであるために Python 流儀の影響を大きく受けますが、SageMath にはまた SageMath の流儀があるために PEP に厳密に従うのではなく、Python 側も周囲にその流儀を強いることがありません。

このようなことから SageMath を理解するためには Python に関する知識が必要になりますが、言語としての Python は比較的簡素な文法を持ち、学び易い言語のために根底にある「**オブジェクト指向プログラミング**」について理解を深めている方が実際の運用では重要です。そこでこの章では Python の言語仕様ではなく、その基礎になる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について語ることにします。

^{*1} カレル チャペックの戯曲「ロボット (R.U.R.)」は Rossum's Universal Robot(ロッサム汎用ロボット) だったりします。

2.3 オブジェクト指向プログラミングの哲学的側面

2.3.1 プラトンのイデア論

Python は「オブジェクト指向プログラミング (Object Oriented Programming)」に対応した計算機言語です。これはオブジェクトという概念を導入することでプログラミングの生産性向上を図っていると説明されます。ここでオブジェクト指向プログラミングには大きく分けて「クラスに基くもの (class based)」と「プロトタイプに基くもの (prototype based)」の二種類があります。前者のクラスに基くものでは扱うべき対象をクラスという形で抽象化し、このクラスを雛型として与えられた対象をインスタンスと呼ばれるデータとして表現することで処理を行いますが、後者のプロトタイプに基くものは、前のようにクラスを構築することなしに既存のオブジェクトを雛型として用います。ここで Python は前者のクラスに基くものになります。そのためにプログラムが扱うデータをクラスが実体化したインスタンスとして捉えてその処理を行います。その際にクラスには属性やメソッドを備えているので、インスタンスの処理でそれらが使えること、クラスには親子関係に類似した階層構造があり、下位のクラスでは上位の属性やメソッドが継承と呼ばれる手続で利用可能といった長所があります。ただこの本でこういった話を延々と進めても一向に面白くないので脇道に大きく逸れることにします。

このクラスに基くオブジェクト指向プログラミングを街学的に説明するためにプラトン (Πλάτων, Plato) の「イデア論 (Theory of Forms)」が引っ張り出されることがあります。このイデア論によれば我々が考察の対象とする現実の物、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になります。だから貴方のそばに居る三毛猫の「みけ」はそれに対応する「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」になるという主張です。そしてイデアは思惟によってのみ知覚できることに加え、さらには「永遠不滅」といった超越的な性質を持っています。このことからイデアは現実にある対象を「理想化したもの」で、ちょうど「雛型」のような役割をしています。ここでプラトンはイデア界こそが真実の世界であって現世についてはイデアが投影された「影の世界」、要するに「模倣物 (*εικών*) の世界」と見なしています (c.f. 「洞窟の比喩」 [15])^{*2}。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、まずオブジェクトがイデアに対応し、計算機で扱う個々のデータそのものはオブジェクトが計算機内部で「実体化したもの」と説明することができます。ちなみにオブジェクトが計算機上

*2 「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したものが後述のグノーシス主義になります。

のデータとして「**実体化**」することを「**インスタンス化 (instantiation)**」, そして, 「**実体化したオブジェクト**」のことを「**インスタンス (instance)**」と呼びますが, 「**イデアの現世における実体化**」も英語では同じ「**instantiation**」になります. と, 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ることが判ります.

さて, 誰がイデアを実体化し, それもどういった理由なのかという素朴な疑問になると途端にプラトンは歯切れが悪くなります. プラトンによるとイデアを実体化させたのが「**デーミウールゴス (δημιουργός, demiurge)**」で, イデアを模倣して世界を創世した理由は貪欲な神「**エロース (Ἔρως, Eros)**」がイデアの美に憧れたためと述べていますが納得できるものではありません. また, イデアは美や善に関わるもので醜いものや惡にイデアは存在しないと述べていますが, 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.

このような「**機械仕掛けの神 (Deus ex machina)**」^{*3}を持ち出されても信じるしかないところは哲学というよりも宗教であり, 実際, イデア論はヘレニズム世界のさまざまな宗教に大きな影響を及ぼします. まず, プラトンのイデア論を基に超越的な「**一者 (το ἕν, to hen)**」からの流出による世界の創造(流出説)を取り入れた「**新**



図 2.1 ヘルメス・トリスマギストス
図 2.1 ヘルメス・トリスマギストス

プラトン主義^{*4}, 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星辰の支配を受ける人間, 死後の超越的な神への魂の帰一を柱とする「**グノーシス主義 (Γνωστις)**」に繋がります. ここでヘルメス・トリスマギストス(三重に偉大なヘルメス, Hermes Trismegistus, Ἡρμῆς 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(Θωθ)^{*5}がヘレニズム時代に(おそらくエジプトで)融合し

^{*3} 古代ギリシャ悲劇で收拾がつかなくなった話を解決するために唐突に神が登場すること(たとえばソフォクレース(Σοφοκλής)の「ピロクテーテース(Φιλοκτήτης)」の終盤に現われるヘーラクレース(Ἡρακλῆς)です).

^{*4} これは後世の呼び名で、その信奉者達はプラトンの思想そのものと思っていました.

^{*5} 頭がトキ(ibis)の神様です.

たもので、鍊金術では「賢者の石」⁶を実際に手にした人物⁷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)」[10]によると人間は元来、美しい神の似姿として創られた神の子で、あるとき彼は高次で純粋な天界⁸から下位の地上に向います。その際に通過した恒星、土星、木星等の星辰の支配を受けることになり、地上にてフュシス ($\varphi\sigmaις$, 自然) 内に写った自分の姿に恋してフュシスと愛欲に陥り、**フュシスは愛する者を捕へ、全身で抱きしめて互に交わった** 結果、人間はフュシスに捕えられて肉体を牢獄とする存在になったといいます。この伝説⁹が人間の本質が神の似姿のために不死であるものの消滅する肉体に囚われ、その上、星辰に支配された存在であるという二面性を持つことの説明になっていますが、同時にヘレニズム文化圏でのオリエント諸国からの占星術の影響とイデア論を中心とした哲学が秘儀化して宗教へと変じてゆくありさまが刻印されていると言えるでしょう。

なお、この世はデーミウールゴスが誤って創造したものだという厭世的な観点は新プラトン主義はもちろんのこと、キリスト教徒の主流派からも反駁されます¹⁰。それどころか本質的に默示的な宗教であったキリスト教は徐々に合理的な宗教へと変貌します¹¹。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¹²であった教父アウグスティヌス (Augustinus Hippoensis, Augustine of Hippo) が新プラトン主義をキリスト教神学の理論付けに用いたことが大きく影響しています。このときに新プラトン主義のフィルターを介した形でアリストテレス (Αριστοτέλης , Aristotle) の哲学も部分的に受容されます。キリスト教と古代の間には大きな断絶があるのも事実ですが、ヘレニズム文明の同心円的な宇宙観、星辰信仰やイシス信仰をマリア崇敬として引継ぐ等、ヘレニズム文明の遺産を引き継いでいる一面もあるのです。

ここで本題に話を戻すと、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でのクラスとインスタンスの関係に類似がみられるものの、そのインスタンス化の手順の類似に留まります。実際、真っ新たなシステムで「三毛猫!」と唱えれば完全無欠な三毛猫のクラ

*6 賢者の石は鍊金術師が探し求めた究極の靈薬で、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。

*7 図 2.1 の恰好の人物をどこかで見たことがありますか？MIT の SICP (Structure and Interpretation of Computer Programs) の扉絵の人物に似ています。つまり λ -函数概念は計算機科学の「賢者の石」であって「A ニシテ Ω」なのです！

*8 この宇宙観は同心円状階層構造を有する天動説です。

*9 おおよそ宗教や宗教的な代物はその伝説を統合と生成するものです。現在でもカトリックでは列聖で、共産主義は英雄で聖者と伝説を生産するという有様です。そして新たな伝説や聖人達を量産することを止めて博物館や図書館で安心して閲覧できるようになった時点が宗教の死なのです。

*10 全知全能の神は半端なことをする筈がないという反論です。

*11 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・ハマディ文書のように瓶に入れて洞窟に埋められたりします。

*12 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。

スが我等のプログラム上にいきなり降臨することはありません。そしてクラスは「**神聖ニシテ侵スヘキアラス**」な超越的な代物ではなく、現実の対象から抽出されるべきものなのです。だから我々が扱う対象は「**どのようなものであるかを語れる**」もので、クラスは対象が「**何であるかを語るもの**」でなければなりませんが、このようなものに「**概念**」があります。そこで概念が何であるかを述べることにしましょう。

2.3.2 概念について

プラトンのイデアは「**思惟にのみによって知覚されるもの**」で永遠不滅なものです。こういった代物が個体とは別個に勝手に実在し、その上、「**思惟で知覚できる**」ように努力しなければならないという状況は人間の手に余る状態なのです。このイデアのように思惟によって知覚されるものに「**概念 (concept)**」もあり、こちらはイデアのような超越的で天下り的なものではありません。この概念が実在するかどうかはスコラ哲学で「**普遍論争**」と呼ばれる論争になっていますが、その存在の有無はさておいて概念は我々の対象に対する理解に従うものです。実際、概念がどのようにして得られるかと言えば対象を特徴付ける「**微表**」、つまり、「**属性**」を抽出し、これらの属性を共通性で纏めることから得られます。要するに「**何であるか?**」や「**それがどのようなものであるか?**」という問への回答から、それを特徴付ける形や色や機能を纏めることで得られるものです。すなわち概念は人間が認知し得る具体的なもの、たとえば対象の形や色といった「**形相 (εἶδος)**」から出発し、我々が対象をどのように語るかということ、つまり、「**説明規定 (λόγος, 口ゴス, account)**」なのです。このように概念はイデアとは逆に個体や現象等の具体性なものから抽象性・普遍性を目指すものです。なお、概念は「**名辞 (term)**」としても現れますが、名辞はあくまでも概念が乗る器であって概念そのものではありません。

このように概念は人間が認知し得る具体的なもの、たとえば物の形や色といった現物の「**形相**」から出発するため、人間とは無関係にどこかに実在し、永遠不滅で超越的なイデアと異なります^{*13}。そして概念は我々が対象をどのように語るかということ、つまり、「**説明規定**」であり、だからこそ、その対象への理解が深まることで語られることの内容が深まることも理解できるでしょう。そしてこの「**それが何であるか?**」や「**それがどのようなものであるか?**」といった問への回答についてより深く考察した人物の一人がアリストテレス (*Aριστοτέλης*) です。

このアリストテレスとプラトンの思索の方向性の違いはラファエロ (Raffaello) の有名

^{*13} なお、プラトンはイデアを *ἰδέα* とも *εἶδος* とも区別せずに呼んでいます。このことから判るようにイデアはものの形状に深く関係するものだったのです。これはアリストテレスも同様ですが、プラトンのイデアのことを専ら *εἶδος* と呼んでいます。

な絵画「アテナイの学堂」[39]*14の中央に起立している両者の手の違いで表現されていることはよく知られていることです。図2.2に示すようにプラトンは天上(イデア=抽象)を指し、アリストテレスは地上(形相=具象)を示すというかたちにです。つまり、イデアは地上の個体から超越した、天界に存在する超越的なものであるのに対し、概念は地上の個体の微表から取り出されるということなのです*15。

そうして得られた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。なぜなら概念はあるものを語るもののです。だから対象 A を語るときに「**A は X である**」となれば概念は対象 A の述語 X として現れます。さらに対象 A だけに限定されずに他のある対象 B についても「**B は A である**」と語り得るという意味です。このように概念は複数の対象の述語と成り得る性質があります。この複数の主語に対してその述語になる性質のことを「普遍」と呼びます*16。たとえば「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。だから「猫」という「概念」は普遍になりますが、「みけ」や「たま」は個体に強く結びつけられているために「これがたまです」と個体を特定するだけで、複数の主語を取り得るという意味の普遍ではありません。この「みけ」のように個体に結び付けられた概念を「個体概念」と呼びます。

「猫」という括り(あつまり)に対して「三毛猫」、「黒猫」、「白猫」、「虎猫」等の毛並で分類することもできます。これらは「猫」の毛並について述べたもので、こちらは「猫」という概念よりも個々の猫をより詳細に説明するものになっています。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



図2.2 アテネの学堂より: プラトンとアリストテレス

*14 ルネサンスにヘルメス文書とプラトンの全集がフィチーノ (Fichino) によってラテン語に翻訳され、これらは「古代神学」と呼ばれ、これを契機に新プラトン主義は大きな影響をルネサンスに与えており、この「アテナイの学堂」もその影響を受けた作品の一つなのです。

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

*16 いろいろなものを取り替えて使えるものに「ユニバーサル」の名前を冠していること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。

「個体に近い側の概念」の二種類があることが判ります。そして、「対象を類似する対象とまとめて包括的に語ろうとする概念」は「個体をより詳しく説明する概念」を包含します。このように一つの対象を語る二つの概念があって、一方の概念が他方の概念を包含するときに包含する側の概念を「上位概念」と呼び、逆に個体をより詳しく語ろうとする概念のことを「下位概念」と呼びます。これらの二つの概念をその普遍性で比較すると上位概念がより普遍になります。たとえば「三毛猫は猫である」という命題では「猫」が上位概念で「三毛猫」が下位概念になります。そして、「猫」と「三毛猫」の二つの概念を比べると、より細かく個体の「みけ」を説明している概念が下位概念の「三毛猫」です。実際、「三毛猫」は「猫である」ことに加えて「毛の色が黒・茶・白の三色である」の二つの性質が述べられるからです。

また上位概念を「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」になります。そして種の違いを示す微表(特徴)を「種差」と呼びます。たとえば先程の「三毛猫」、「虎猫」、… の例では「毛並」の違いが種差になっています。それから「上位」と「下位」の意味はどちらがより普遍的であるかということに対応しており、より普遍的な概念である上位概念は下位概念を包含します。そしてには概念には上限と下限があり、上限となる最上位の概念を「範疇 (カテゴリー, Category)」、下限と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概念で、当然、個体に最も近い概念になります。それに対して範疇は個体を含む概念の中で最も普遍的な概念になります^{*18}。

この概念の階層構造についてアリストテレスは「範疇 (カテゴリー) 論」等の著作で述べています。ところで、ともすればプラトンに批判的なアリストテレスをイデア論が秘儀と化はじめた新プラトン主義の哲学者達は「師の思想を秘匿するために批判していた」と捉えるようになります。アリストテレスの論理学は新プラトン主義の哲学への入門書としても重要視されています^{*19}。このアリストテレスの受容に決定的だったのはポルピュリオス (*Πορφύριος*, Porphyry of Tyre) が記述した「手引 (エイサゴーゲー, *Eἰσαγωγή*, Isagoge [26])」^{*20}です。この「手引」でポルピュリオスはプラトン

^{*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) に対応するもので、さらに属の直下に種があつて、その間には何かが入るものではなく、種は類の直下の概念としての性格があります。

^{*18} だからアプリケーションのメニューでは最上段が「カテゴリー」という名称で分類されているのです。

^{*19} 当時のライバルのストア派 (*Στωικοί*, Stoics) の論理学は現代論理学に類似した命題論理学ですが、残念ながらストア派の論理学は断片でしか伝わっていません。

^{*20} 英訳はボエティウス (Boethius) のラテン語訳 (イサゴーゲー, Isagoge) を オーエン (Owen) が翻訳したもの [33] とバーンズ (Barnes) がギリシャ語文献から翻訳したもの [26] があり、前者はエイサゴーゲー

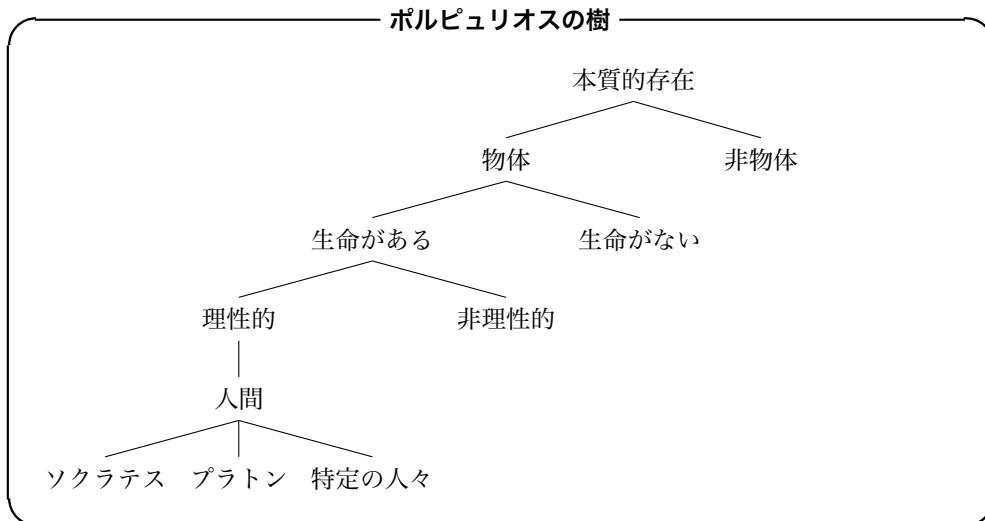
とアリストテレスを無理なく結び付けることに成功し、その結果、「手引」は(新プラトン主義の)哲学を学ぶにあたって最初に読むべき本とされます。さらにはこの「手引」は西ローマ帝国崩壊の混乱のうちに西ヨーロッパに残された数少ない哲学書のひとつになって、西ヨーロッパの哲学に大きな影響を及ぼすことになります。

さて、この「手引」によるとものごとを語るということには「類」、「種」、「種差」に加えて「**特有性**」と「**偶有性**」があります。ここで類 (genus) と種 (species) の語源はギリシャ語の $\gammaένος$ と $\epsilonιδος$ で共に「形」という意味があり、このことから伺えるように概念は形の類似や違いをもとにしていた経緯があります^{*21}。そして類や種は「**それが何であるか?**」という問への回答です。これに対して「**種差**」と「**特有性**」、それから「**偶有性**」については「**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「**種差**」は「**種を特徴付けるもの**」、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指摘できるようなもの**」、つまり固有の特徴です。それに対して「**偶有性**」は「**その程度を語ることができるもの**」で、たとえば日焼けした子供ならば「薄く日焼けしている」、「よく日焼けしている」のように程度が表現可能であり、さらには「**日焼けしていない**」のように日焼けしているという属性を持たないという状況も考えられることがその特徴です。

それからポルピュリオスによる述語の「類」、「種」、「種差」、「特有性」と「偶有性」による分類は大きな影響をさまざまな分野に与えています。その一つに「**ポルピュリオスの樹 (Arbor Porphyrianae)**」と呼ばれるものがあります：

ゲーが範疇論の入門書という古来からの立場、後者がアリストテレスの論理学全体への入門書としての立場で、さらに詳細な解説があります。

*²¹ 分類学でも当初は形態や機能的な側面、ダーウィン以降は進化論、現在はDNA等の分析が入っていることと同様です。



これは類を種で分類するということをのちの「手引」の註釈者達が視覚化したもので、さまざまな分野で階層構造を示す「樹形図」です。またリンネ (Carl von Linné) が始めた学名の命名方法は「**二名法**」と呼ばれる方法で、これは種による類の分類そのものです。つまりこの命名方法は動物/植物が属する種とその種を包含する属に対して最初に属 (genus) のラテン語名、それから種 (species) のラテン語名を列記する方法です。たとえば人類の学名は ‘Homo sapiens’ ですが、ここで属が Homo、種が sapiens です。この二名法はオブジェクト指向プログラミングでも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います。そしてサブクラスの表記では二名法の繰返しで実に長いクラスのメソッドの指定を行いますが、全体を俯瞰するときには樹形図が用いられます。

2.3.3 内包と外延

「概念」を語る場合はまず「**それが何であるか?**」という問に対しても我々はそれがどのようなものであるかを説明するか、あるいは個体を列挙して説明するかどちらの方法になります。このように説明の方法には二通りの方法があり、一つが「**内包**」、もう一つが「**外延**」と呼ばれる方法です。最初の「**内包**」は概念が持つ徴表/属性で構成され、「**外延**」は概念が適用される対象を列記することで構成されます。「**猫**」という概念であれば、その内包は「**動物である**」、「**4本足で歩く**」、「**柔らかい肉球を持つ**」、「**ニヤオと鳴く**」等の属性 (性質) から構成されるでしょう。一方で外延なら「**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。

これら内包と外延には「**内包外延反比例増減の法則**」と呼ばれる関係があります。これは内包が増大するに従って外延が減少し、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のことです。たとえば「猫」という概念に対して「茶、黒、白の三色の毛並である」という内包を追加すると「三毛猫」以外の「白猫」、「黒猫」等の猫が「猫」と「茶、黒、白の三色の毛並」の外延から消えてしまいますが、逆に「三毛猫」という外延に「白猫」という外延を追加すると「茶、黒、白の三色の毛並である」という内包が消えてしまいます。つまり、内包が増えるということは、それだけ述語付けられることで個体に近付く結果、外延を構成する個体が絞られ、逆に外延を構成する個体が増えれば個体から離れて普遍的な事柄を抽出するために内包が減少するということなのです。このことからも判るように上位概念とその下位概念とを外延で比較すれば、より大きな外延を上位概念が持ち、逆に下位概念は外延が小さくなるものの、内包に関しては逆に下位概念が上位概念より大きなも内包を有するというものです。

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

床屋の逆理

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

この逆理は古来より「**クレタ人の逆理**」として知られていました：

クレタ人の逆理

クレタ人はうそつきである。

この命題をエジプト人やギリシャ人が主張したのであれば問題がありませんが、エピメニデス (*Ἐπιμενίδης*, Epimenides) というクレタ人^{*22}が主張したためにややこしくなったものです。これらの逆理の本質は前述の論理式 ' $x \neq x$ ' 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という循環的な定義になっています。このようにラッセルの逆理は非常に単純な式ですがその効果は絶大で、ラッセルが書き上げたばかりの著作「Principles of Mathematics」[35] やドイツの数学者フレーゲ (Frege) が独

*22 紀元前 6 世紀頃のクレタのクノッソスの哲学者だそうです。

自の図式のために嫌がる出版社を説得して二部に分けて出版した「算術の基本法則」[17]といった著作の成果を葬り去るに十分でした^{*23}。

ラッセルやフレーゲの論理主義^{*24}やカントール (Cantor) の(素朴)集合論^{*25}に批判的であったポアンカレ (Poincaré) は彼のエッセイ集「科学と方法」[18]で幾つかの逆理を分析しています。たとえば「偶数の集合」や「身長 170cm 以下の人の集合」といった集合の定義では「自然数の集合」や「人間の集合」といった集合の概念に触れずに集合がきちんと定義ができます。これらの定義方法を「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 ([18], p.204)。この指摘に対してラッセルは「型理論」と「悪循環原理」を導入することで非可述的な命題の排除に成功したもの今度は数学的帰納法が使えないという重大な副作用が生じます。そこで「還元可能性公理」と呼ばれる公理^{*26}を導入すれば今度はその天下り的性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、現在の集合論では後述の集合の公理系で「集合」を定め、それ以外の命題の外延のことを「類」、あるいは「クラス」と呼んで集合と区分して「ラッセルの逆理」を排除しています。

2.3.4 定義すること

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

ところでキュニコス派のアンティステネス (Ἀντισθένες , Antisthenes) は定義 ($\lambda\circ\gamma\circ\varsigma$)

^{*23} 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。

^{*24} 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」[36]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。

^{*25} 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。

^{*26} 「任意の階の命題函数には、それと同値な可述的函数が存在する」, Principia Mathematica の表記を用いると $(\exists\varphi).\psi x. \equiv_x .\varphi!x.$ と記述されます。

について「それが何であるかを説明するもの」と述べています。その一方で「**一つの主語は一つの述語あるのみ**」[2]と主張し、「馬は認めて馬性を認めない」と類や種といった概念を認めないとという立場です。これと彼の弟子のディオデゲネス ($\Deltaιογένης$, 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むしり取った鶏を持って行って「これがプラトンの人間だ!」と言った逸話を思い起こすなら、説明規定を並べていったところで個体そのものになるとは言い難いことを述べていると言えるのです。この立場に立脚するならクラスを定義することは妥当なことではなく、むしろ、クラスを持たずに既存のものを複製するオブジェクト指向プログラミングが類似していると言えるでしょう。

2.3.5 プラトニズム

では概念やイデアは実在するものでしょうか? イデア論を認めるのであればイデアは個体から独立して存在しますが、概念となるとなかなか厄介な問題です。たとえば「三毛猫のみけ」を観察することで「猫」や「三毛猫」といった概念に到達できるとはいえ、だからといって「みけ」が「猫」や「三毛猫」といった概念に先立って存在している訳ではありません。それ以前に存在した猫や三毛猫によって「猫」や「三毛猫」といった類や種が定義されているからです。ここで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、それが実際に存在するものかどうかを明確に述べていません。また入門書として著名な「手引(エイサゴーケ)」の著者であるポルピュリオスは類や種といった概念(普遍)が存在するものであるかどうかを触れないことを手引の最初の章で述べています。この手引をラテン語に翻訳したボエティウス (Boethius) は手引の註釈を記していますが、特にその第二注釈が西ヨーロッパ中世のスコラ哲学にて「普遍論争」を引き起すことになります[21].

この概念/イデアの存在は物理学の原理や数学の定理の方が先に存在して、それらを学者が発見すると考えるか、到達した概念から原理や定理が導出されると考えるのかといった議論にも繋がります。ここで事物の前に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。それに対して事物のあとに概念があると考える立場を「**唯名論 (Nominalism)**」と呼びます。

ここで実在が問題となった背景ですが、アリストテレスが創始し、その後に発展した論理学、いわゆる伝統的論理学で扱う命題には「**存在含意 (external import)**」と呼ばれる条件が付随しています。この存在含意は命題の主語が実際に存在しているという一種の暗黙の条件です。このことはアリストテレスが用いた古代ギリシア語が属する印欧語族では ‘A = B’ という命題にて、その主語 A と述語 B の関係として表現する「**繫辞**

(copula)」として主語 A が存在する意味が付随する「**存在動詞**」と呼ばれる動詞が用いられることに関係しています。たとえば日本語の「A は B である」^{*27}を印欧語族の一つである英語で「A is B」と置換した場合、日本語の「は」は A と B が一致すること意味する以上の意味を持ちませんが、be 動詞は主語の A が存在するという意味が付随する「**存在動詞**」と呼ばれる動詞であるために「A = B」の意味だけではなく、むしろ「A が存在して A = B である」の意味を持つ命題になるのです。このように伝統的論理学の命題には主語の隠れた存在性という条件である存在含意が含まれているのです。ただし、この存在含意は現代の論理学の創始者のフレーゲ (Frege) の概念記法では除外されて現在の論理学はありません。これは現代の論理学が命題を扱う命題論理学であるからです。

また伝統的論理学では主語と述語の関係の考察を中心に行っており、「**名辞論理学**」と呼ばれます。それに対して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^{*28}。さて、伝統的論理学の命題は、その主語に対して存在含意を前提にして論理学が構築されているために「**非存在**」のもの、したがって存在が不確かな「**仮説**」に対して「**三段論法 (Syllogism)**」と呼ばれる推論が伝統的論理学では使えないのです。この「**三段論法**」は「**大前提**」、「**小前提**」と「**結論**」の三つの命題から構成される「**推論の規則**」です。たとえば「**提言三段論法**」の例を次に示しておきましょう：

定言三段論法の例

大前提： 人間は死すべきものである
 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ものである

三段論法の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**」や「**帰納的に求められるもの**」でなければなりません。ここでイデアや概念といった普遍の存在を認めてしまえば存在含意を充すので、この推論を行う際の障害がなくなります。ところが存在が不確かな仮説となると、どのような結論が出ても不思議はないのです。実際、イスラム哲学においてアッバース朝の公認神学であった「ムアタズィラ (Mu'tahzilah)」と呼ばれる超合理主義派は自らを「**正義と神の唯一性の提唱者**」と自称していた程ですが、彼等は三段論法を駆使してともすれば異端的な結論を導出していたとのことです[5]。それに加えて「**神の人格表現の否定**」を主張してクルアーン (コーラン) で述べられた神の人間的

^{*27} 「A は B である」という命題に「ある」が何気に含まれていることに、このような用語を作り定着させた明治の人々の何気ない凄さを私は感じます。

^{*28} 古代ギリシャの哲学学派ストア派にも論理学があり、こちらも命題論理学です。ただし、ストア派の論理学や伝統的論理学に欠けているのが「すべて」や「存在する」に対応する量化詞です。量化詞は19世紀末にフレーゲが函数概念と同時に導入しています。

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^{*29}。つまり「**哲学こそが全て、宗教は一般大衆向けの幼稚な哲学**」という考え方を持っており、のちの「正統派」によってムアタズィラの著作が根絶させられるという憂き目にあっています。この様子は19世紀以降、ヨーロッパ諸国の軍事力に圧倒された結果、世俗的な社会改革を行うものの宗教的保守派や原理主義、そして改革を受けれない大多数の大衆によつて再三、妨げられ、改革の失敗後に極端な復古が生じるというイスラム教諸国でよく見られる動向と類似していなくありません^{*30}。

2.3.6 イデア論の問題点

ところでイデア論にもいろいろと問題があります。イデア論に対する反論で有名なものが「**第三の人間**」と呼ばれるものです。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ます。すると、これらに対して人間として類似していることを示す尺度としての「**人間のイデア**」が必要になり、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「**第三の人間**」とその他のイデアに対しても類似の尺度になるイデアが存在しなければならず、以降、第四、第五、第六…の人間が存在することになります。このように議論が收拾できないこと自体にイデア論に無理があるのではないかという反論です。この有様にアリストテレスも「形而上学」にて「**物を数えようとする場合に、数が少なくては数えられないと思って、その数を増やして数えようとする者のごときである**」とイデア論を批判しています[2]^{*31}。

また理想的な人間として例えられるソクラテスにしても、赤ん坊、子供、若者、壮年、老年といった過程を辿ることになりますが、それぞれの瞬間にイデアがある筈で、その瞬間瞬間のイデア同士の関係はどうなるのかと話が簡単になるどころか逆に複雑になっていきます。また種から芽が出てやがて木になり、それが老木になって倒れて腐るといった個体の生成、変化や運動、最後に消滅する理由がイデア論からは説明できません。結局、機械仕掛けの神を引っ張り出して創世神話や生物の生殖の理由を説明したとしても、何気ない現象の説明にはとても無理があるのです。

*29 νοῦς や λόγος とみなす訳です。その為に同時代の神学者からは「神よ」と呼びかけるのではなく「知恵よ」と呼びかけばいいではないかと皮肉られています。

*30 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。

*31 形而上学 第一巻九章

2.3.7 形相 ($\epsilon\hat{\imath}\delta\omega\varsigma$)

アリストテレスは師匠のプラトンと異なり、観察に立脚したより現実に則した考え方をしています。まず、アルストテレスの「形相 ($\epsilon\hat{\imath}\delta\omega\varsigma$, eidos)」はプラトンの「イデア ($\iota\delta\varepsilon\alpha$)」のような「個体から離れた存在 ($\chi\omega\rho\iota\sigma\tau\alpha$)」ではなく、むしろ、現実にある個体は「形相」とこれといった特性を持たない「質料 ($\beta\lambda\eta$)」との「結合体 ($\sigma\beta\gamma\omega\lambda\omega\nu$)」として捉え、形相こそがその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ものとして捉えています。これを木の種の話に戻すと、まず、木としての形相が種の内部に存在し、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、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ゆくという説明になります。

このアリストテレスの考察を現在の科学と比べてどうかと言えば細かな点では怪しいかもしれません、現代の科学でも結局、対象が何であるか、どのような理由でそれがそれ自身であるかを説明しようとするものであり、この流儀はアリストテレスの考察にその源流があることが判ります。だからこそアリストテレスは「万学の祖」なのです。

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、質料はそれ自体では何らの特性を持たないものですが、これをビットの列に、それから形相をデータ構造等の意味付けに対応付けることができるでしょう。すると計算機内部のデータは形相と質料の結合として表現されることになります。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出さなければ收拾がつかないのですが、現在のようにプログラムといったソフトウェアも含めて考慮するとき、形相因は非常に説得力のあるものなのです。

2.3.8 アリストテレスの範疇 (Category)

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

アリストテレスによる範疇

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

ここでの「**本質的存在 (実体, οὐσία)**」は「**第二の本質的存在 (第二実体)**」と呼ばれるもので、「人間」, 「猫」, 「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り得るもので、ちなみに「**第一の本質的存在**」は「私」, 「みけ」, 「ソクラテス」等の個体により近くて普遍性を持たないもので、これらの本質的存在はギリシア語で「**ウーシア (οὐσία)**」と呼ばれ、「**存在**」を意味する動詞 εἶναι を名詞化したものに由来し、「**実体**」が訛語として当てられています。このアリストテレスの分類に対し、18世紀の哲学者カント (Kant) は量, 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：

カントによる範疇の分類

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

これらの範疇で重要なことは、「**それが何であるか?**」や「**それがどのようなものであるか?**」という問に対する答は、ここで述べた範疇の何れかになるということなのです。ま

た, そのようにして語られるものには, 類, 種, 種差, 特有性と偶有性によって階層構造が入ります. そしてこれらは, これから我々が考察する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とその構造に深く関わるのです.

2.3.9 オブジェクト指向プログラミングにおけるクラスの表現

ここでようやくオブジェクト指向プログラミングの話に戻しましょう. まず, 扱うべき実際のデータが個体であり, このデータをオブジェクトが実体化したものとして捉えられ, それから個体を何であるか, 何であるべきかを定める形相に相当するものがオブジェクトのクラスに対応し, クラスの記述では, そのクラスを具体的に定める属性を記載することになりますが, 属性は「どのようなものなのか」という問に対する答が何らかの値で表現されるのであればその値, 何らかの機能であれば, それをメソッドとして表現すれば良いのです. たとえば, 「猫」であれば「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」等の属性や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, そのクラスで表現されたがオブジェクトが「猫」であり, 「みけ」は「猫」というオブジェクトが実体化したもの, すなわち, インスタンスとして捉えられるという訳です. このときにクラス間の関係はどのようになるでしょうか? 概念では類と種といった階層が入ります. これに似たものとして次に述べる「**継承**」という関係があります.

2.3.10 継承

概念には先程の説明のようにより大きな外延を持つ概念と, より小さな外延を持つ概念があり, より大きな外延を持つ概念を上位概念, 小さな外延を持つ概念のことを下位概念と呼びました. 概念を内包で書換えてしまうと下位概念の内包は上位概念の内包を基に上位概念に含まれない内包を付与したものになります. このことは「上位概念」に含まれる属性をそのまま引き継いで, その概念に新しい「属性」を与えれば新たに「下位概念」が構築できることを意味します. この操作がオブジェクト指向プログラミングでの「**継承**」に相当します.

この継承という考えは非常に自然な考え方です. 実際, ある新しい動物を見たときに, その動物が何に属するといった系譜が創られます. その動物の調査が進むにつれてさらに細かく分類されることがあります. この場合, 新しく分岐する動物は旧来の分類を基にして新しい分類が行われるでしょう. これと同様に扱うべきデータをあるオブジェク

トの実体化として記述したとしても、のちにデータの理解が深まることで、そのデータがより細かく分類されることはそう珍しいことではありません。このことは最初に大きく分類したクラスをより下位のクラス、すなわち、サブクラスへとさらに細かく分割することに相当しますが、この細分化は上位のクラスにない値やメソッドを追加することで行われます。このことは最初のクラス構築が間違っていない限り、システムの大枠を変更することなしに自然に拡張が行えることを意味します。

ただし、この継承を上手く行うためには、系統立った分析が必要になることが言うまでありません。この分析を誤れば、継承が自然に行うことのできないシステムができあがることになります。ここで継承関係が一子相伝的な継承であれば、その属性やメソッドが何処から引き継がれたかを探すことが直線的な関係になるために容易です。しかし、実際の継承は複数のクラスからの継承を含む複雑なものになるでしょう。それに加えて経済的な側面も考えなくてはなりません。実際、あまりにも複雑怪異な継承関係は扱う側にとっても不要な混乱を招く畏れがあるだけではなく、メソッドや属性の検索という観点からも不利になる可能性があります。実際、クラスをあまりにも小分けにすることで分類を細かくしてしまえばどうなるでしょうか？たとえば「猫」から個体の「みけ」に至るまでに「三毛猫」が間に一つ入ると、「アジアの猫」、「東アジアの猫」、「日本猫」、「三毛猫」が入ると素朴に考えても、猫の毛並だけを問題にしているのであれば「アジア」、「東アジア」、「日本」といった地域はさほど問題にはならず、冗長でさえあることは理解できるでしょう。さて、このような直系的な継承関係であったとしても、ここで「みけ」が持つ「猫の属性」や「猫の習性」を知りたくなったときにどのようなことが生じるでしょうか？このときに最初に「みけ」が属するクラスから順に調べてゆくことになりますね。すると、最初の継承関係であれば「三毛猫」を間に一つ挟む程度で済むことが、後者の継承関係になると「アジアの猫」、「東アジアの猫」と「日本猫」の三つのクラスを間に挟むため、これらのクラスで検索を行う必要が出てくるのです。このように検索の手間が増えてします。これが複数のクラスを継承する関係であれば、属性やメソッドの検索により多くの時間を要する可能性が生じることが理解できるでしょう。さらに、この検索の手間の問題だけではなく、この属性やメソッドの検索順位をどのように定めるかで新しいクラスの属性やメソッドが反映されなくなる可能性も出てきます。この問題については「C3 MRO」といった手法で改善が図られていますが、最初のクラスの分析が非常に重要なことは言うまでもないでしょう。

2.4 集合論について

2.4.1 集合論言語について

ここからは唐突ですが集合論について解説を進めて行こうと思います。その理由ですが、「これが何であるか?」と言う問に対して我々はさまざまな説明を行います。この説明では、そのものの特徴やものの程度といったことで、これらのこと整理することで概念に到達すると述べました。そうするとその概念で説明され得るもの集まりが外延になり、それが類や種であったりします。ところでフレーゲが概念記法で開始した論理主義は非常に厳密な数学の基礎を与えるかのように見えましたが、「**それ自身でないもののあつまり**」という命題を考えることで呆気なく破綻してしまいました。つまり、「**任意の命題はその外延を持つ**」という前提で構築していたにもかかわらず、この「**ラッセルの逆理**」のように外延を持たない命題をその体系から排除できなかったために矛盾が生じてしまったからです。集合論の創始者のカントール (Cantor) は、素朴な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様ですが、この論理主義の失敗から単純に「**命題を充すもののあつまり**」と「**集合**」との間に境界線が必要との認識が生じます。そこから発生したのがここで述べる集合論なのです。この集合論にはツエルメロ (Zermelo) の公理系を基にフレンケル (Frankel) の公理等を追加した公理系と、それらの公理とは独立した「**選択公理**」と呼ばれる重要な公理があり、これらの公理の組み合わせで Z, ZF や ZFC 等と略記され、これらの公理系を基に集合論が構築されています。そして、この公理系を語る必要がありますが、この集合論にはその体系で扱う対象を語るために「**言語**」があり、それが「**集合論言語**」と呼ばれる言語です。ここでは集合論の論理式で用いる記号について説明しておきましょう：

— 集合論で用いる記号系 —

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

ここでは元が集合に属するという意味で用いる記号 “ \in ” と対象の同値性を示す記号 “ \equiv ” 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。またこの本では式 a を b で定義することを記号 “ $\stackrel{\text{Def.}}{=}$ ” を導入して ‘ $a \stackrel{\text{Def.}}{=} b$ ’ と表記します。それから記号 “ \equiv ” を同値性を意味する記号として $A \equiv B \stackrel{\text{Def.}}{=} (A \supset B) \wedge (B \supset A)$ で定義し

ます。また、集合の重要な記号に記号“ \cup ”や記号“ \cap ”等がありますが、これらの記号は後述の集合論の公理から順に定めることにします。

これらの集合論の記号系に含まれる記号を用いて、集合論で用いられる論理式を次の形成規則に従って定義することができます：

論理式の形成規則

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

この論理式の形成規則を持つ系を「**集合論言語**」と呼び \mathcal{L} と表記します。この形成規則は帰納的なもので、論理式の具体的な表記は §3 の §3.4 で述べる BNF 記法に基く表記等で表現することができます。なお、以降の説明では集合論言語 \mathcal{L} の記号や形成方法を用いて、記号“ \cup ”や記号“ \cap ”といった記号を必要に応じて定義します。ただし、この論理式の形成規則は、集合論で扱う論理式の形成方法について述べたもので、個々の論理式の意味や意義について述べたものではなく、どのような論理式が集合論の体系に受け入れられるかを述べたものではありません。それを規定するものが次の集合論の公理系になります。

2.4.2 集合論の公理系

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

集合論の公理系

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

では「**集合論の公理系**」で挙げた公理について順番に解説しましょう。

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

■対公理 (Axiom of pairing) 集合 x, y を成分とする「**対集合**」の存在を保証する公理です。ここで集合 x, y の対集合を $\{x, y\}$ と記述します。特に $\{x, x\}$ を $\{x\}$ と表記して「**1-要素集合**」と呼びます。この対集合 $\{x, y\}$ は集合 x と y をその成分として持つということを意味するだけで、集合 x と集合 y の順序等の関係について何も述べていません。そこで $\langle x, y \rangle \stackrel{\text{Def.}}{=} \{\{x\}, \{x, y\}\}$ によって集合 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。この順序対では $x = y$ でない限り $\langle x, y \rangle = \langle y, x \rangle$ にはなりません。そして成分が 3 以上の順序対を構成することができます。この順序対の構成方法を以下に纏めておきましょう：

順序対の構成方法

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

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

■幂集合公理 (Axiom of power set) この公理に現われる記号 “ \subseteq ” は $a \subseteq b \stackrel{\text{Def.}}{=} \forall x(x \in a) \supseteq x \in b \vee x = b$ で定義される記号で、この公理の意味は集合 x の任意の成分を外延として持つ集合の存在を保証します。この公理と外延性公理から唯一存在が保証される集合を「**幂集合**」と呼び、 $\mathfrak{P}(x)$ で集合 x の幂集合を表記します。

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

■無限集合公理 (Axiom of infinity set) 無限集合の一つの作り方を定める公理で、 v が集合のときに $\emptyset \cup \{v\}$ が集合となることを保証します。さて、空集合公理から空集合 \emptyset は集合です。この空集合と無限集合公理から $\emptyset \cup \{\emptyset\}$ も集合になることが保証されます。さら

に $\emptyset \cup \{\emptyset \cup \{\emptyset\}\}$ を構成すると、これも集合になることが無限集合公理から保証されます。このように空集合 \emptyset から開始して、この処理を繰り返すことで $\emptyset, \emptyset \cup \{\emptyset\}, \emptyset \cup \{\emptyset \cup \{\emptyset\}\}, \dots$ という集合の無限列が構成できます。実は、この集合の無限列を自然数の定義とすることができます：

自然数の定義

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

この定義では、空集合 \emptyset が自然数の 0, $\emptyset \cup \{\emptyset\}$ が自然数の 1 に対応し、以降、集合の無限公理で認められた集合の生成規則に従って他の自然数が続々と生成されます。このように空集合公理と無限集合公理を含む公理系では自然数を構成することが可能で、逆に言えば自然数を保持しているとも言えるのです。

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

超限順序数

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

ここで定義した自然数に「大小関係」を導入することができます。つまり、自然数 a, b に対して $a < b \stackrel{\text{Def.}}{=} a \in b$ で記号 “ $<$ ” を導入し、同様に記号 “ \leq ” を $a \leq b \stackrel{\text{Def.}}{=} a \in b \vee a = b$ で定義します。それによって定義した自然数に大小関係が自然に導入できます。さらに自然数 a に対して $a + 1$ を $a + 1 \stackrel{\text{Def.}}{=} \emptyset \cup \{a\}$ で定め、この $a + 1$ を a の「後続」、あるいは「後者」と呼びます^{*32}。この自然数については順序数で再度触れることにしましょう。

■置換公理図式 (Axiom schema of replacement) 図式のはじめの $\phi(x, y) = \phi(x, z) \supset y = z$ を $F(x) = y$ で置換えると集合 x の函数 F による像も集合になる公理で、同時に集合そのものに制約を入れる公理です。

この公理はフレンケルによって導入されたのですが、もともとツェルメロが入れていた公理は「分出公理 (Axiom of displacement)」と呼ばれる次の公理です：

*32 自然数の後者関係についてはフレーゲの「概念記法」[16] ではじめて厳密に述べられています。

—— 分出公理 (Axiom of displacement) ——

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

この分出公理の意味は素朴集合論のように任意の命題が外延を持つというものではなく、既存の集合から指定された命題を充す集合が存在するという公理で置換公理図式から導くことができます。実際、置換公理図式 A8 の $\phi(x, y)$ を $\psi(x) \wedge x = y$ で置換えることで分出公理 A7' が直ちに得られ、この分出公理から得られる集合を $\{u \in x : \phi(u)\}$ と表記します。そして、この分出公理から幾つかの重要な集合の生成方法が定義できます。まず、集合 x, y に対して $\{u \in x : u \in y\}$ で得られる集合を $x \cap y$ と表記し、集合 x と y の「**共通集合**」と呼びます。それから $\{\langle u, v \rangle : u \in x \wedge v \in y\}$ によって得られる集合を $x \times y$ と表記し、集合 x と y の「**直積集合**」と呼びます。

ここで命題 $\phi(x)$ の外延 $\{x : \phi(x)\}$ を考えてみましょう。この外延はその元がある集合の元であると保証されないために分離公理から集合であるとは断言できません。このような命題の外延のことを「類」、あるいは「**クラス (class)**」と呼び、「**集合**」と区別します。オブジェクト指向の「**クラス**」が「**クラス**」と呼ばれるのも複数の「**述語**」に対応する「**属性値**」や「**メソッド**」から構成されるものの、それらが定める外延がとある「**集合**」から切り出したものになるとは限らないからです。では素朴集合論で問題となつた「ラッセルの逆理」をもう一度考えてみましょう。この逆理の本質は外延 $\{x : x \notin x\}$ が素朴集合論の集合から排除できないために生じていると述べました。そこで分出公理を認めるとあらかじめ集合として認められたものから命題 $x \notin x$ を充す x を取り出さなければなりませんが $x \notin x$ より自分自身を包含しない集合が構成できなければ集合として存在することができないためにこの命題の外延は集合にならず、この体系から除外することができるのです。

この分出公理は逆理の排除という目的では有効ですが、この公理がフレンケルによって置換公理図式で置換えられた理由として「大きな集合の生成ができない」ということに尽きます。ここでは「選択公理と数学」[13] で紹介されている例を挙げておきましょう：

最初に函数 f を

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

で定めます。このとき函数 f の値域 $\text{rng}(f)$ は:

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

で与えられることになりますが、この値域 $\text{rng}(f)$ が集合になることが分出公理からは導出できません。ここで置換公理を認めると函数による集合の像も集合となることが保証されるので、その値域 $\text{rng}(f)$ が集合になります。このように新たな集合を作り出せる置換公理の方が、集合から集合を取り出すということで集合であることに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されるでしょう。

■正則性公理 (Axiom of regularity) この公理によって $a \in a$ を充すものが集合から排除され、このことから集合と集合の元を区別する公理であると言えます。また、この公理から $\dots, x_3 \in x_2, x_2 \in x_1, x_1 \in x_0$ を充す**集合の底なしの無限列**: 「 $\dots x_3, x_2, x_1, x_0$ 」も排除されます。このような底なしの無限列があると困る点に軽く触れておきましょう。最初に空集合公理と無限公理の二つを認めると自然数を導入することができます。このときに大小関係も前述の方法で関係 \in から導入することができますが、正則性公理があれば $\dots \in x_2 \in x_1 \in x_0$ となる集合の列 x_i は底なしの無限列にならないので必ず $x_n \in \dots \in x_2 \in x_1 \in x_0$ を充す集合 x_n が存在し、このことから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の導入ができる利点があるからです^{*33}。また関係 \in に対する無限降下列が存在しないことは公理 A8':

—無限降下列の非存在性—

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

とすることができます。この「**無限降下列の非存在性公理 A8'**」と「**正則性公理 A8**」の間には $A8 \supset A8'$ が成立しますが、その逆の $A8' \supset A8$ が成立するためには次の「**選択公理**」が必要になります [13]。

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

■選択公理 (Axiom of choice) 空集合と異なる集合から、その成分を取り出すことができるという公理で、後述の ZFC 公理系の “C” に該当する公理です。この選択公理は他の集合論の各公理から独立した公理であり、この公理なしでも「数学」を構築することができます。この選択公理は何かと便利な公理ですが、この公理から非常に厄介な逆理が幾つか導きだせることができます。その逆理の一つの「**バナッハ-タルスキ (Banach-Tarski) の逆理**」を紹介しておきましょう：

—— バナッハ-タルスキの逆理 ——

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

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

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

この逆理を適用するとゴルフボールの表面を適当に分割して、それらを貼り合せたもので地球が覆えることになります。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になります^{*34}。さすがにこの定理は日常的な常識から大きく外れたもので逆理としか言い様がありませんが、このような逆理が導き出せるにせよ、この公理を認めたときの御利益が大きい公理です。なお、この公理を認めない場合、任意の自然数の部分集合が最小元を持つことが利用されます。

ここで A1 から A9 までの公理系の組み合せ表を以下に示しておきましょう：

—— 集合論の公理系 ——

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

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。

^{*34} 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畳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

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

最初の述語 $\text{Trans}(u)$ は集合 u が推移的であることの定義になります。ここで述語 $\text{Trans}(u)$ の意味するところは x が集合 u の元であり, y が x の元であれば y も集合 u の元となるということです。ここで記号 “ \in ” を記号 “ $<$ ” で置換えると「 $x < u$ かつ $y < x$ ならば $y < u$ 」が得られ、このことから通常の大小関係で見られる推移律に対応すること容易に判るでしょう。次に述語 $\text{Ord}(\alpha)$ を使って集合 α が順序数であることの定義を行っています。この述語 $\text{Ord}(\alpha)$ の意味するところは、まず、集合 α が推移的で、それから集合 α に属する任意の x, y に対して $x \in y$, $y \in x$ か $x = y$ の何れかの関係が成立することです。ここでも記号 “ \in ” を記号 “ $<$ ” で置換えると順序数 α に対して $x < \alpha$, $y < \alpha$ となる $x, y \in \alpha$ に対して $x < y$, $y < x$ か $x = y$ の何れかの関係が成立すること、つまり、集合 α が全順序集合であることを意味しています。たとえば、自然数全体の集合 $\omega = \{0, 1, 2, 3, \dots\}$ の元 u は $\text{Trans}(u)$ を充すために推移的で、さらには $\text{Ord}(u)$ を充すので順序数になります。そして、この順序数の定義からはさまざまな集合の無限列からも順序数が得られることが判ります。たとえば置換公理で紹介した $\{\omega, \mathfrak{P}(\omega), \mathfrak{P}^2(\omega), \dots\}$ も順序集合で、また、 ω は自然数を含む順序数の中で最小の順序数です。

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

では次に順序数全体 OR を定義しましょう:

順序数全体

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

この順序数全体 OR は大き過ぎるために ZFC では集合ではなくクラス(類)になります

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 $OR \in OR$ になって OR の後者 $OR + 1$ を考えることができますが推移率から $OR \in OR + 1$, 一方で OR は順序数の全体なので $OR + 1 \in OR$ になって矛盾が生じます. これが素朴集合論で「**プラリ=フォルティの逆理**」と呼ばれる逆理です. ただし, ZFC では正則性公理から OR は集合ではなくクラスになるため, この素朴集合論上の逆理も ZFC では「OR は集合でない」という定理になります.

2.4.4 モデルと宇宙

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

このモデル M は集合論言語 \mathcal{L} の文 φ の意味を判断する上での文脈に相当します. ちなみに日常の文でも文脈によって, その意味が真であったり偽となったりすることがあります. たとえば, ある人達の会話で「彼はイケメン」という話が出たとき, その会話をしている人達にとっては「彼」が誰なのかは自明なことですが, この人達と無関係な人にとって「彼」が誰を指すのか不明なために真偽の判断ができないものです. これはモデルでも同様で, モデル M で文 φ の意味が真であったとしても別のモデル N では偽となることがあります. ところが, 文 $A \supset A$, 日常語なら「 A は A である」のように文脈と無関係に常に真になる文もあります. このように文脈とは無関係に常に真となる文のことを「**恒真式**」あるいは「**トートロジー (tautology)**」と呼びます.

2.5 圈 (Category)

2.5.1 カテゴリーと圈

さて、ここまで集合論の話をしました。次にここからは数学の「**圈 (Category)**」について解説します。この数学用語の「**圈**」は英語では「**Category**」が対応します。ところで、この Category という言葉は哲学用語ではアリストテレスの「**カテゴリー**」、その日本語の訳語として「**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 (οργανον))**」と呼ばれる著作群の筆頭に置かれ、アリストテレスの哲学を学ぶ上で最初に読まれるべき書物とされていたとのことです[1]*35。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ると言えるでしょう。実際、MacLane[31]によると「**Category**」という言葉はアリストテレス (Aristotle) とカント (Kant) の「**カテゴリー論**」に由来するもので、「**functor**」はカルナップ (Carnap) の著作に由来すると述べています。

ここでは圏を MacLane の本: The Category theory for working Mathematician[31](以後, CMW と略記) の定義に沿って解説しましょう。この CMW[31] ではメタグラフとメタ圏を定義し、それから集合に対してグラフと圏を定義しています。また、集合と述べた場合は ZFC 公理系の集合で考えており、この文書でも、その考えを引き継ぐことにして解説を進めることにします。この CMW では他の圏論の本とはやや異なり、メタグラフやメタカテゴリーから話を始めます。ここで「**メタ**」が頭に付く理由ですが、後述の対象や矢の類が集合になるとは限らないものを考えており、後述のグラフや圏の一種の雛形になっていると言えるからです。そして、ここで述べる事項は数学的対象そのものや性質をより抽象化したものとなっています。このことによって、最上位の概念が範疇 (カテゴリー) になるのと同様に、抽象化を行った本質的なものが圏論 (カテゴリー) で扱うものになります。

*35 その注釈書としてポルピュリオス ($\Pi\sigma\rho\varphi\upsilon\varphi\iota\omega\varsigma$, Porphyry of Tyre) のエイサゴーゲー [33] が非常に有名でした。§11 に訳を載せています。

2.5.2 メタグラフについて

メタグラフ \mathcal{C} は下記の性質を持つ対象と矢(射)で構成されます:

———— メタグラフ (metagraph) ————

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

この定義から, まずメタグラフは対象から構成されます. ここで対象のあつまりが集合になるとは限りません. そして, 二つの対象の間に矢と呼ばれる関係があります. この矢のあつまりも集合になるとは限りませんし, 任意の二つの対象の間に矢があるとも限りません. このメタグラフの対象は集合の元, 矢は写像をそれぞれ抽象化したもので, この矢による関係がちょうどグラフを抽象化したものに相当しているのです.

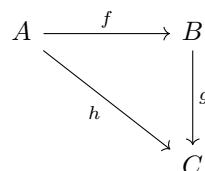
最初に幾つかの記号を導入します. まず, メタグラフ \mathcal{C} の対象のあつまり, すなわち類(クラス)を $\text{Obj}\mathcal{C}$, 同様にメタグラフ \mathcal{C} の矢の類を $\text{Arr } \mathcal{C}$ と表記します. そして, メタグラフ \mathcal{C} の矢の始域になり得る対象から構成される類を \mathcal{C}_0 , 終域になり得る対象で構成される類を \mathcal{C}_1 と表記します. 同様に対象 A を始域, 対象 B を終域とするメタグラフ \mathcal{C} の矢から構成される類を $\text{Hom}_{\mathcal{C}}(A, B), \mathcal{C}(A, B)$, あるいは $\text{Hom}(A, B)$ と表記します. なお, メタグラフ \mathcal{C} の矢 f が対象 A を始域, 対象 B を終域とする矢のときに記号 \in を用いて簡単に $A, B \in \mathcal{C}, f \in \mathcal{C}$ と表記したり, より詳細に $A \in \mathcal{C}_0, B \in \mathcal{C}_1$, および $f \in \text{Hom}_{\mathcal{C}}(A, B)$ と表記することで対象や矢の圏 \mathcal{C} への包含関係を表記します.

ここでメタグラフ \mathcal{C} の矢は二つの対象の間の関係を与えるものと考えた方が自然で, このときに矢の対象の順序が重要になります. さらに矢は伝統的論理学上の「繋辞 (copula)」として考えることもできます. ここで連辞は命題「 A は B である」の中の「... は... である」のように主語と述語の関係を表現する機能を持っており, 伝統的論理学ではこの繋辞こそが命題を構成するものと考えられていたとのことです [4]. さて, ここで矢を \xrightarrow{f} と表記することで矢の式 $f : A \rightarrow B$ から $A \xrightarrow{f} B$ へと図式にすることができます. そして, 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見えてきます.

2.5.3 矢について

さて、「メタグラフ」に含まれる「グラフ」という言葉から「**函数のグラフ**」等の「グラフ」を連想される方も多いかと思います。この函数のグラフはある函数 f の XY-グラフで、これは点 x における函数 f の値 $f(x)$ を XY 平面上の点 $(x, f(x))$ として描いたものの類です。この座標の表記では最初の成分が X 座標、そのうしろの成分が Y 座標になります。この座標を構成する対の順序に重要な意味があります。そこで座標 $(x, f(x))$ を集合論言語 \mathcal{L} の順序対 $\langle x, f(x) \rangle$ として記述してしまいましょう。このときにグラフ全体は $\{\langle x, f(x) \rangle : x \in A\}$ で外延として記述できます。ここで対象 A が ZFC 公理系の集合であれば、このグラフも置換公理図式から集合になります。このように XY-グラフは順序対の集合としての性格を持つことになります。さて、メタグラフの矢 $A \xrightarrow{f} B$ に対しても対象 A, B がともに集合であれば集合 $\{(x, y) : x \in A \wedge y \in B \wedge f x = y\}$ として矢を考えることができます。このときに $x \in A$ に対応する対象 B の元を $f(x)$ あるいは $f x$ と表記し、 $y = f x$ のときに $x \mapsto y$ と表記することで $x \in A$ と $y \in B$ が矢 f を伸立とする関係にあることを示します。ここで空集合 \emptyset を始域とするメタグラフの矢 f を考えると、この矢は空集合 \emptyset でなければならないことが判ります。この実例として後述の Python のオブジェクトの型で None 型が挙げられます。実際、この None 型が空集合 \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**」と呼ばれる矢の操作について解説しましょう。この矢の合成は写像の合成を抽象化したものです。この合成を考える前に「**合成可能対**」というものを考えます。これは矢の順序対 $\langle f, g \rangle$ であり、この順序対を構成できる矢 f, g は $\text{dom } g = \text{cod } f$ を充す矢でなければなりません。この条件を充す順序対から構成される類を $\text{Arr } \mathcal{C} \times_{\text{Obj } \mathcal{C}} \text{Arr } \mathcal{C}$ と表記し、「**合成可能類**」と呼びます。ここで $\text{Arr } \mathcal{C} \times_{\text{Obj } \mathcal{C}} \text{Arr } \mathcal{C}$ に属する順序対 $\langle g, f \rangle$ で $g \circ f$ 、その始域と終域をそれぞれ $\text{dom } f, \text{cod } g$ になる矢に対応させる操作とします。もちろん、このような操作が可能かどうかはメタグラフでは保証されません。この操作が可能な場合に得られる矢のことを矢 f, g の「**合成**」と呼びます。ここで二つの矢 $A \xrightarrow{f} B$ と $B \xrightarrow{g} C$ の合成 $A \xrightarrow{g \circ f} C$ が矢 $A \xrightarrow{h} C$ と一致するときに、これらの矢の関係を次の図式として表現することができます：



この図式では対象 A から矢に沿って対象 C に向う二つの経路があり、この経路に沿う

ことと矢の合成が一意に対応します。この図式には矢 f と g を経由する経路から得られる矢 $g \cdot f$ と矢 h を経由する経路の二つが存在し、これら二つの経路が一致することを意味します。この図式のように図式中のある対象を始域と終域とする複数の経路が存在し、それらの経路の何れを通っても矢の合成が一致する図式を「可換図式」と呼びます。

さて、3個の矢 $f : A \rightarrow B$, $g : B \rightarrow C$, $h : C \rightarrow D$ に対し、矢の合成として $f \circ (g \circ h) \stackrel{\text{Def.}}{=} \langle f, \langle g, h \rangle \rangle$ と $(f \circ g) \circ h \stackrel{\text{Def.}}{=} \langle \langle f, g \rangle, h \rangle$ をそれぞれ構築することができます。これらが一致するかどうかは一般的に正しいとは言えませんが、これらの矢の合成が一致するということ、すなわち $f \circ (g \circ h) = (f \circ g) \circ h$ を「結合律」と呼びます。

また、メタグラフ \mathcal{C} の対象からそれ自身への矢を考えますが、特に任意の $f, g \in \text{Hom}_{\mathcal{C}}(A, B)$ に対して次の図式を可換とする矢を特に「同一矢（恒等射）」と呼び、 1_A あるいは id_A と表記します：

$$\begin{array}{ccc} A & \xrightarrow{f} & B \\ & \searrow f & \downarrow 1_B \\ & & B \xrightarrow{g} C \end{array}$$

この可換図式は $1_B \circ f = f$ と $g \circ 1_B = g$ を充すことを意味し、この可換図式の意味を「同一矢の公理」と呼びます。この公理の重要な点は同一矢が一意に定まるところで、実際、対象 A に対して二つの同一矢 1_A と $1'_A$ が存在するとき、この公理から直ちに $1_A = 1'_A$ が導き出せるからです。このことから同一矢と対象は一一に対応するために対象と同一矢を同一視することができます。このように圏論で中心となるのは対象ではなく、あくまでも矢や函手と呼ばれる圏の間の写像に重点があります。この点は伝統的論理学が線的に主語と述語で構成された命題の分析に費されているのと比較し、現代の論理学がフレーベルの函数概念と量化詞（ \forall や \exists ）の導入によって個体同士の関係へと重点が移っているとも言えるでしょう。

最初の述べたように矢は写像を抽象化したのですが、写像の単射、全射、同相に対応する性質として矢には「mono」、「epi」、そして「iso」があります。これらの性質は他の矢に対する性質として定義することができます：

mono, epi, iso

- **mono** : 任意の矢 $h, g : C \rightarrow A$ に対して $f \circ h = f \circ g$ を充せば $h = g$ になるとき. このときに $f : A \rightarrow B$ と表記します.
- **epi** : 任意の矢 $h, g : B \rightarrow C$ に対して $h \circ f = g \circ f$ を充せば $h = g$ となるとき. このときに $f : A \twoheadrightarrow B$ と表記します.
- **iso** : $g \circ f = 1_A$ かつ $f \circ g = 1_B$ を充す矢 $g : b \rightarrow a$ が存在するとき. このときに $A \equiv B$ と表記します.

矢の mono, epi, iso といった性質は、対象が集合であれば通常の写像の单射、全射、同射に対応します。しかし、圏の対象が集合であるとは限らないために勝手が異なります。また、矢 f が iso であれば矢 f は mono で epi にもなりますが、mono で epi だからといって iso になるとは限りません。ただし、対象が集合であれば矢は通常の写像が対応するために mono で epi であれば iso になります。

2.5.4 メタ圏について

メタグラフ \mathcal{C} が「**メタ圏**」であるとはメタグラフ \mathcal{C} の矢の合成が可能であり、同時に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 \mathcal{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

これらメタグラフとメタ圏は非常に形式的な定義です。そして、メタグラフやメタ圏を構成する対象や矢のあつまりがどのようなものであるかといった言及はありません。ここで対象の類が ZFC 公理系等の公理系で集合を構成するときにメタグラフとメタ圏はそれぞれ次の節で述べる「**グラフ**」や「**圏**」になります。

2.5.5 グラフと圏について

メタグラフやメタ圏には、その対象と矢が何であるかということに制約がありません。実際、 \mathcal{C} をメタグラフ、あるいはメタ圏としたときに \mathcal{C} の対象が構成する類 \mathcal{C}_0 と \mathcal{C}_1 については形式的な定義以上のものはありません。そこでメタグラフやメタ圏に制約を入れます。ここで入れる制約は対象と矢が構成する類が集合となるものに限定するというものです。この制約を入れたメタグラフやメタ圏では対象や矢の性質を考察するときに集合論

の成果が使えるようになります。そこで、この対象と矢が集合を構成するあつまりが ZFC 公理系で集合になるメタグラフ \mathcal{C} のことを「**グラフ**」、同様にそのようなメタ圏のことをと呼び、今後はこれらを中心に考察することにし、以下にグラフと圏の定義を纏めておきましょう：

—— グラフ (graph) の定義 ——

グラフ \mathcal{C} は次の性質を充す：

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

ここでグラフ \mathcal{C} の対象全体の集合 O のことをメタグラフの表記に従って $\text{Obj } \mathcal{C}$ 、同様に矢全体の集合 A の集合のことを $\text{Arr } \mathcal{C}$ と表記することにします。ただし、**O** や **A** も適宜用いることにします。このグラフに対して圏を次で定義します：

—— 圏 (category) ——

グラフ \mathcal{C} が次の性質を充すときに圏と呼ぶ：

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

前述のように圏では対象はさほどの意味を持たず、むしろ、矢や圏の間の写像に対応する函手の方が重要になります。実際、対象はその恒等矢と一对一に対応させられるために対象と矢を同一視できるからです。その結果、対象そのものよりも矢や後述の函手を用いて圏の構造や性質を探ることがより重要になります。

2.5.6 双対圏

圏 \mathcal{C} の対象と矢に対し、対象はそのまま対象に写し、矢に対しては、その矢の始域と終域を入れ替える操作を考えます。つまり、圏 \mathcal{C} の対象 A はそのまま対象 A に対応させる一方で、矢 $f: A \rightarrow B$ を矢 $f^{\text{op}}: B \rightarrow A$ で置換える操作です。この操作によって二つの矢 $f: A \rightarrow B$ と $g: B \rightarrow C$ の合成 $g \circ f$ に対しては矢 $(g \circ f)^{\text{op}}$ が対応しますが、矢の合成の方法から矢 $f^{\text{op}} \circ g^{\text{op}}$ となることがわかります。この操作 ${}^{\text{op}}$ で得られるものを「**双対**」と呼びます。

この操作 \circ^P は対象に対しては恒等矢であり、矢に対しては、その矢の始域と終域を入れ替え、矢の合成に対しては、その矢の双対の順番が逆順になります。この双対によって圏 \mathcal{C} から新しい圏が構築され、この圏のことを圏 \mathcal{C} の「**双対圏**」と呼び、 \mathcal{C}^{op} と表記します。

2.5.7 圈の例

ここでは具体的な圏の例を挙げることにしましょう。最初にちょっと特殊な圏を考えます。この圏は有限個の元を持つ集合 A で矢として恒等矢しか持たないものとします。このように恒等矢しか持たない圏は「**離散圏**」と呼ばれます。この離散圏では各対象 $A \in \mathcal{C}$ と対応する恒等矢 $A \xrightarrow{1_A} A$ の他には矢がない圏なので、この矢と繋ぎの「... は... である」を対応させてみましょう。すると $A \xrightarrow{1_A} A$ の意味を「 A は A である」と解釈することができます。そして離散圏は対象 $A \in \mathcal{C}$ を始域とする矢は恒等矢 1_A しか存在しないので、任意の対象 $A \in \mathcal{C}$ に対して「 A は A である」としか言えない圏であることが判ります。これは犬儒派のアンティステネス (*Αντιστένες*) の主張する「**一つの主語は一つの述語あるのみ**」^{*36} と普遍を認めない立場に対応します。

その他の重要な圏の例を挙げておきましょう：

- **Set:** 対象が集合、矢が通常の写像
- **Set_{*}:** 対象が基点付きの集合、矢が基点を基点に写す通常の写像
- **Cat:** 対象が圏、矢が函手
- **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
- **Vect:** 対象がベクトル空間、矢が準同型写像
- **Top:** 対象が位相空間、矢が連続写像
- **Top_{*}:** 対象が基点付きの位相空間、矢が基点を基点に写す連続写像
- **Toph:** 対象が位相空間、矢が連続写像
- **\mathcal{C}^{op} :** 圏 \mathcal{C} の双対圏

ここで述べている集合、圏と位相空間は、より正確にはそれぞれ「**小集合 (small set)**」、「**小圏 (small category)**」、「**小位相空間 (small topological space)**」と呼ばれます。これは「**グロタンディーク宇宙 (Grothendieck Universe)**」との関係を表現するものです。このグロタンディーク宇宙は圏の対象全てと後述の対象の演算結果を含む大きな集合で U と表記します。このことから対象はグロタンディーク宇宙に元として包含される

^{*36} 形而上学 [2] 5 卷 29 章, 1024b34

ために「**小**」，この宇宙そのものはグロタンディーク宇宙の元として包含されないために「**大**」と呼ばれます。また，圏については「**局所的に小さい**」と「**小さい**」の二種類があり，圏 \mathcal{C} が「**局所的に小さい**」とは任意の対象 $A, B \in \mathcal{C}$ に対してその矢の集合 $\text{Hom}(A, B)$ が小集合になる場合，同様に「**小さい**」とは対象全体の集合 \mathcal{C}_0 と矢全体の集合 \mathcal{C}_1 の双方が小集合になる場合です。

2.5.8 グロタンディーク宇宙について

さきほどの圏の例で，対象が「**小集合**」等と**小**が付くものを示しました。ここでの大小はグロタンディーク宇宙との関係で決まると言いました。ここではもう少し細かく説明することにしましょう。

まず「**グロタンディーク宇宙**」に包含される対象なら小で，宇宙そのものは大となります。このグロタンディーク宇宙は集合の公理系に類似する公理を充す対象の集合です。圏の場合，対象や矢の類は集合を構成し，それらの集合に対してグロタンディーク宇宙にて次の演算が許容されています：

——— グロタンディーク宇宙で許容される演算 ———

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

これらの演算は ZFC 公理系であれば問題なく充される集合の演算処理です。これらの集合演算を前提として，以下に示す性質が成立する集合 U のことを「**グロタンディーク宇宙 (Grothendieck Universe)**」と呼びます：

——— グロタンディーク宇宙 U が充すべき性質 ———

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

グロタンディーク宇宙 U 自身は ZFC 公理系の「**正則公理**」によって U の元として含まれることはありません。さらに，この U が集合になるとも限りません。このように

宇宙 U とその元には区分があり、この区分を対象が集合であれば宇宙 U の元を「**小集合**」、対象が圏であれば U の元を「**小圏**」と宇宙 U の元のことを、その元の型の頭に「**小 (small)**」を付けます。逆に宇宙 U は頭に「**大 (large)**」を付けます。たとえば対象が集合であれば「**大集合**」、圏であれば「**大圏**」といったあんばいです。

2.5.9 始対象と終対象

次に重要な対象として「**始対象 (initial object)**」と「**終対象 (terminal object)**」を定義しておきましょう：

——— 始対象と終対象 ———

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

ここで始対象を 0 と表記する理由ですが、集合を対象とする圏 **Set** の始対象は \emptyset で、さらに自然数の構築で \emptyset が 0 に対応し、同様に圏 **Set** の終対象はシングルトン (singleton) と呼ばれる成分が一つだけの集合で、沢山ありますが、ここで $1 = \{\emptyset\}$ であるために 1 を用いています。なお、これら始対象と終対象は圏によって異なります。たとえば、対象の集合を自然数 **N**, 矢を \leq とする圏 (\mathbf{N}, \leq) のとき、自然数 0 は任意の $n \in \mathbf{N}$ に対して $0 \leq n$ を充すことから始対象であることが判ります。また、群を対象年、準同型を矢とする圏 **Grp** では始対象は成分が一つだけの群 (=単位元のみの群) になります。なお、この本では始対象 0 と終対象 1 を自然数の $0, 1$ と混同しないように必ず **始対象**、**終対象** という枕詞を置き、単に $0, 1$ と表記するときは自然数 $0, 1$ を指示するものとします。

ここで始対象と終対象は反変函手 ${}^{\text{op}}$ によってそれぞれ終対象と始対象に写すことができます。すなわち A を圏 \mathcal{C} の始対象とするとき、条件から任意の $B \in \text{Obj} \mathcal{C}$ に対して矢 $f : A \rightarrow B$ が一意に存在します。ここで双対圏 \mathcal{C}^{op} を考えると対象はそのままで矢の始対象と終対象が入れ替えられるので、任意の対象 $B \in \mathcal{C}^{\text{op}}$ に対して矢 $f^{\text{op}} : B \rightarrow A$ が存在することになり、このことから対象 A が圏 \mathcal{C}^{op} の終対象となることが判ります。同様に終対象もその双対圏では始対象になるので 1 と 0 が互いに双対関係にあることが判ります。

なお、集合を対象とし、矢を集合間の写像とする圏 **Set** で終対象 1 は成分が一つの集

合^{*37} $\{*\}$ であり, 矢 $1 \xrightarrow{1_A} A$ が集合 A の成分を定めることになります^{*38}. それから始集合 0 は空集合 $\emptyset = \{\}$ になります. この圏 **Set** の例のように始集合と終集合の双方を有する圏もあります. また, この集合の圏 **Set** では自然数を \emptyset から構築することができますが, このときの 0 と 1 は圏の始対象と終対象にそれぞれ一致します.

2.5.10 函手

圏 \mathcal{C} と圏 \mathcal{D} が与えられたとき, これらの圏に対して, 圏 \mathcal{C} の対象を圏 \mathcal{D} の対象に対応させ, 同様に圏 \mathcal{C} の矢を圏 \mathcal{D} の矢に対応させる写像^{*39}が考えられます. そして, より扱い易い性質として圏 \mathcal{C} の恒等矢 1_A を圏 \mathcal{D} の恒等矢 1_B に写す性質があると良いでしょう. このような写像としては圏 \mathcal{C} から圏 \mathcal{C} 自身への恒等射と, 先程挙げた双対写像^{op}です. ところで矢には合成という操作があります. たとえば, 二つの矢 $A \xrightarrow{f} B$ と $B \xrightarrow{g} C$ の合成 $A \xrightarrow{g \circ f} C$ は函手 $F : \mathcal{C} \rightarrow \mathcal{D}$ によって Ff, Fg と $F(g \circ f)$ に写されます. では写した先の圏 \mathcal{D} で Ff と Fg を使って $F(g \circ f)$ はどのように表記されるでしょうか? これには二通りが考えられ, 一つは $F(g \circ f) = Fg \circ Ff$ と f, g の順番を保つものと $F(g \circ f) = Ff \circ Fg$ と逆になるものです. たとえば同じ圏への恒等射であれば矢の合成はそのままですが, 双対写像^{op}なら矢の合成が逆になります. このように矢の合成の順番を保つかどうかで函手を区分することができるのです.

まず, 最初の矢の始域と終域をそのまま写し, 矢の合成の順序も保つ圏から圏への写像のことを「**共変函手**」と呼びます:

——共変函手 (covariant functor)——

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

- $C \in \text{Obj}\mathcal{C}$ に対し $FC \in \text{Obj}\mathcal{D}$
- 圏 \mathcal{C} の矢 $A \xrightarrow{f} B$ に対して圏 \mathcal{D} の矢 $FA \xrightarrow{Ff} FB$ が対応
- $F 1_A = 1_{FA}$
- $F(g \circ f) = Fg \circ Ff$

ここで共変函手のことを単に**函手 (functor)**と呼びます. この本でも誤解がない限り, 単に函手と呼ぶときは共変函手のことを指します. この共変函手の例として圏 \mathcal{C} 自身への恒等射が自明なものとして挙げられるでしょう.

^{*37} シングルトン (singleton) と呼びます.

^{*38} 圏 **Set** では新プラトン主義者が聞いたら喜びそうな「一者からの流出」で万物は定義されているということです.

^{*39} 圏では対象や矢のあつまりが集合になるからです

つぎに双対写像のように矢の始域と終域を入れ替え、それから矢の合成も順序が逆になる圏から圏への写像を「**反変函手**」と呼びます:

反変函手 (contravariant functor) —————

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

- $C \in \text{Obj} \mathcal{C}$ に対し $F C \in \text{Obj} \mathcal{D}$
- 圏 \mathcal{C} の矢 $A \xrightarrow{f} B$ に対して圏 \mathcal{D} の矢 $F A \xrightarrow{F f} F B$ が対応
- $F 1_A = 1_{F A}$
- $F (g \circ f) = F f \circ F g$

この反変函手の例としては先程の双対写像があります。また、ここで共変、反変函手の重要な例を挙げておきましょう。最初に圏 \mathcal{C} の対象 C, C' に対し、対象 C' から対象 C への矢の類 $\text{Hom}_{\mathcal{C}}(C', C)$ は圏の性質から集合になります。そして矢の集合間の矢として \mathcal{C} の矢を用いることで写像 $C' \mapsto \text{Hom}_{\mathcal{C}}(C', C)$ と写像 $C \mapsto \text{Hom}_{\mathcal{C}}(C', C)$ は圏 \mathcal{C} から(小)集合の圏 **Set** への写像になります。そして、これらの写像は矢の合成の関係から前者の写像 $C' \mapsto \text{Hom}_{\mathcal{C}}(C', C)$ が反変函手、後者の写像 $C \mapsto \text{Hom}_{\mathcal{C}}(C', C)$ が共変函手になることが判ります。

ここで函手 $F : \mathcal{C} \rightarrow \mathcal{D}$ は集合から集合への写像であるために、通常の写像の議論ができます。そして、函手を矢の間の写像として考えるときに、それが単射、全射、同型となる場合を考えられます。ここで函手が「忠実 (faithfull)」であるとは $\text{Hom}_{\mathcal{C}}(A, B) \xrightarrow{F} \text{Hom}_{\mathcal{D}}(FA, FB)$ 同様に、この写像が通常の写像として全射になるときは「**充満 (full)**」と呼びます。そして、函手 F と逆向きの函手 $G : \mathcal{D} \rightarrow \mathcal{C}$ で $GF = 1_{\mathcal{C}}$ と $FG = 1_{\mathcal{D}}$ を充すものが存在するときに函手 F を「**同型 (isomorphism)**」と呼びます。

この函手に対してはもう一つ別の対応関係を考えることができます。まず圏 \mathcal{C} から圏 \mathcal{D} への二つの函手 F と G が与えられたときに圏 \mathcal{C} の矢 $C' \xrightarrow{f} C$ の始域と終域になる対象 $C, C' \in \mathcal{C}_0$ は函手 F によってそれぞれ $F C, F C' \in \mathcal{D}_0$ に写され、また、矢 f 自体も圏 \mathcal{D} の矢 $F C \xrightarrow{F f} F C'$ に写されます。これは函手 G も同様で、対象は $G C, G C' \in \mathcal{D}_0$ に矢は $G C \xrightarrow{G f} G C'$ へとそれぞれ写されます。ところで同じ対象と矢を函手で別物に写しているので、函手 F と 函手 G で写される対象 FC' と GC' 、それと FC と GC の関係が考えられます。この FX から GX への対応関係を α_X と表記しましょう。この対応関係からは圏 \mathcal{C} の矢 $C \xrightarrow{f} C'$ の始域 C と終域 C' に写像 α による関係がそれぞれあるので、 $G f \circ \alpha_C$ と $\alpha_{C'} \circ F f$ が等しくなるということは自然な要請になります。このように函手 F から G への写像 α で右下の図式を可換にする写像のことを「**自然変換**」と呼び、 $\alpha : F \rightarrow G$ と表記します:

$$\begin{array}{ccccc}
 C' & & FC' & \xrightarrow{\alpha_{C'}} & GC' \\
 \downarrow f & & \downarrow Ff & & \downarrow Gf \\
 C & & FC & \xrightarrow{\alpha_C} & GC
 \end{array}$$

函手と自然変換を使って新に圏を構成することができます。つまり、対象を圏 \mathcal{C} から圏 \mathcal{D} への函手、矢をそれらの間の自然変換として新たに圏 $\mathcal{D}^{\mathcal{C}}$ を構成することができます。

2.5.11 コンマ圏

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

$$\begin{array}{ccccc}
 & & b & & \\
 & \swarrow f & & \searrow g & \\
 c & & & & d
 \end{array}$$

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

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

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

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

さらに二つの圏 \mathcal{C}, \mathcal{D} に対して函手 $S : \mathcal{D} \rightarrow \mathcal{C}$ が与えられたときに対象 $b \in \mathcal{C}$ を固定します。それから $d \in \mathcal{D}$ に対して矢 $f : b \rightarrow Sd$ を $\langle f, d \rangle$ と表記します。それから $\langle f, d \rangle$ と $\langle f', d' \rangle$ を考えます。ここで矢 $d \xrightarrow{h} d'$ で次の可換図式が成立するものを $\langle f, d \rangle$ と $\langle f', d' \rangle$ の矢 h とします：

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

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

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

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

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

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

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

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

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

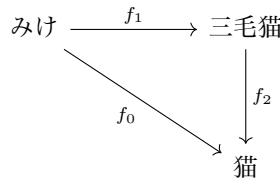
と圏 $(b \downarrow S)$ の可換図式と実質が違わないことがわかります。同様に函手 S を圏 \mathcal{C} の同一矢とすると圏 $(b \downarrow \mathcal{C})$ が得られます。そのためにここで述べた構成方法から得られる圏のことを「コンマ圏」と呼びます。また、スライス圏のことを特にコンマ圏と呼ぶこともあります[11]。

2.5.12 普遍矢

アリストテレスの論理学で、普遍であるとは主語と述語の関係において、述語に成り得るもので、特定の主語をだけを探らないもので、つまり、主語の取り替えが効くということです。たとえば、「みけ」、「三毛猫」、「猫」で、「みけは三毛猫である」が真であれば、「三毛猫は猫である」ことから「みけは猫である」も真になります。さて、「みけ」はとある一匹の猫を指すので、別の虎猫の「とら」に対して「とらはみけである」とはなりませんし、逆も違うのです。個体は主語の取替えということでは「これ」だの「あれ」といった個体を直接指すような主語以外は取れず、その意味では普遍ではありません。ところが「猫」については「みけ」も「とら」も主語になり、「三毛猫」も同様に特定の主語を探りません^{*40}。ここで「猫」と「三毛猫」については「三毛猫は猫」ですが「猫は三毛猫」ではないので、「三毛猫」と「猫」は同値なものではありませんが、共に外延を持つものです。このことから「三毛猫」は「猫」と「みけ」の中間にあるものと言えます。この類と個体の間にあるものを下位の類と呼び、その下位の類のことを直上の類に対して種、直上の類のことを単に類と呼びます。この「三毛猫」と「猫」の例では「三毛猫」が個体の「みけ」に近いことから種、「猫」が類になります。そして、類と種は共に普遍なものです。

次に ‘ A は B である’ ということを ‘ $A \xrightarrow{f} B$ ’ と表記してみましょう。すると ‘みけは三毛猫である’, ‘三毛猫は猫である’ と ‘みけは猫である’ はそれぞれ ‘みけ $\xrightarrow{f_1}$ 三毛猫’, ‘三毛猫 $\xrightarrow{f_2}$ 猫’ と ‘みけ $\xrightarrow{f_0}$ 猫’ で置換えることができます。そして次の可換な図式として表現できることが判ります:

^{*40} 道具で “universal” というと、方向が自由自在であるとか、さまざまな状況に対応できるとか、この取替えが効くという意味で用いられていますね。



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

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

ここで先程の猫の例と同一の図式が得られていますね。ただし、みけ、三毛猫、猫を安易に D, C, B に割り当てる無意味なものになってしまいますが、矢を中心と考えると非常に意味のあるものです。つまり、猫の例の矢 f_0 が普遍被覆 q に対応し、各々が中間的な存在である「三毛猫」や位相空間 C を経由する矢の合成として表現されるという意味があります。そして、この普遍の存在では、「みけ」の場合には類の「猫」が「三毛猫」を種として包含すること、写像 $p : C \rightarrow B$ が被覆空間であるということが前提となっていることです。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 $S : D \rightarrow C$ 、対象 $c \in C$ とするときに c から S 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：

- $r \in D, c \xrightarrow{u} Sr$ とする
- 任意の $d \in D, c \xrightarrow{f} Sd$ から対 $\langle d, f \rangle$ を定める
- $Sf' \circ u = f$ を充すただ一つの矢 $r \xrightarrow{f'} d$ が存在する

これを次の可換図式で表現することができます：

$$\begin{array}{ccc}
 c & \xrightarrow{u} & Sr \\
 \downarrow 1_c & & \downarrow Sf \\
 c & \xrightarrow{f} & Sd
 \end{array}
 \qquad
 \begin{array}{c}
 r \\
 \downarrow f' \\
 d
 \end{array}$$

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

では先程の三毛猫のはなしはどうなるでしょうか? 「みけが三毛猫である」ということと「みけが猫である」ということは対象がある集合に含まれるという包含関係であり、「三毛猫が猫である」ということは種が類に包含されるということ、つまり、集合の包含関係になっていることが判ります。つまり、「みけが三毛猫である」と「三毛猫が猫である」の双方の繋辞はともに包含写像であってもその対象が異なるということになります。ここで猫の圏を次のものとしましょう:



この「猫の圏」は猫の類を幾つかの種(至って適当)に分類したもので、対象は猫の種で矢は猫の種と類の関係です。それから函手 $S : \text{猫の圏} \rightarrow \mathbf{Set}$ を種を集合に、種と類の関係を集合の包含関係に置換えるものとしましょう。すると先程の可換図式は次のものになります:

$$\begin{array}{ccc}
 \text{みけ} & \xrightarrow{u=\epsilon} & \text{三毛猫}_{\text{set}} \\
 & \searrow h=\epsilon & \downarrow Sf=C \\
 & & \text{猫}_{\text{set}}
 \end{array}
 \qquad
 \begin{array}{c}
 \text{三毛猫}_{\text{猫の圏}} \\
 \downarrow f \\
 \text{猫}_{\text{猫の圏}}
 \end{array}$$

このとき、矢 u は矢 h に対して矢 Sf が一意に定まるので普遍矢になりますが、この対応は「みけが猫である」に対してみけが所属する「三毛猫」について「三毛猫が猫である」が一意に対応するというものです。

もうすこし真面目な例を挙げておきましょう。この例は Mac Lane の本 [31] に出ているもので、ベクトル空間とその基底に関するものです。この例ではまず集合の圏 \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 $U(W)$ に対応させることで矢 $X \xrightarrow{f} U(W)$ を構成することができます。それからこの f の対応関係を基に体 K について線形になるように拡張することで新たに \mathbf{Vect}_K の矢 $V_X \xrightarrow{f'} W$ を構成することができます。以上の結果から次の可換図式が得られます：

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

このベクトル空間の例も三毛猫の例と似た状況ですが、 V_X, W は類-種のような階層構造に基づく関係ではなく、変換写像から得られた矢となっている点で異なります。

2.5.13 圈の演算

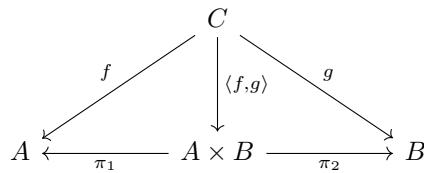
圏 \mathcal{C} の対象について積や幂を定めることができます。最初の対象の積について述べましょう：

対象の積

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

- \mathcal{C} の二つの矢 $X \xrightarrow{\pi_1} A$ と $X \xrightarrow{\pi_2} B$ が存在
- \mathcal{C} の任意の対象 C と C から A, B への二つの矢 $C \xrightarrow{f} A, C \xrightarrow{g} B$ について $f = \pi_1 \circ h, g = \pi_2 \circ h$ を充す矢 $C \xrightarrow{h} X$ が一意に存在する。この矢 h を $\langle f, g \rangle$ と表記する。圏 \mathcal{C} の任意の二つの対象に対して積が存在するときに圏 \mathcal{C} のことを「積を有する圏」と呼ぶ。

圏の積の定義を可換図式として表現することができます：



この定義では対象 C から積 $A \times B$ への矢が一意に存在するという性質から積 $A \times B$ を定めるという定義方法になっています。この性質は $A \rightarrow B$ という命題を考えたときに複数の主語の述語になるという性質に類似したもので、この積 $A \times B$ の性質は「普遍性」に対応するものであることが判ります。つまり、この積の定義は対象 $A \times B$ の具体的な形や構成方法について述べたものではなく、その積の普遍性に基いた定義方法となっていることに注目して下さい。また、 $C \xrightarrow{f} A$ と $C \xrightarrow{g} B$ から A, B は C よりも普遍であり、 $C \xrightarrow{\langle f,g \rangle} A \times B$ から $A \times B$ は C よりも普遍であるものの $A \times B \xrightarrow{\pi_1} A$ と $A \times B \xrightarrow{\pi_2} B$ から $A \times B$ よりも A, B の方が普遍であるということになります。つまり、 $A \times B$ は A, B に近く、 A, B よりも一段落ちる普遍であることが判ります。

なお、圈 \mathcal{C} の矢が通常の写像であれば、対象 A, B の積については $A \times B = \{(x, y) | x \in A \wedge y \in B\}$ と位相空間のデカルト積になります。ところで圈 \mathcal{C} の対象が順序数^{*41}で、矢が \leq のときの $A \times B$ は A と B のどちらかより後者にある対象で与えられるので、対象の積は対象の成分の順序対のようなものになるとは限りません。

この対象の積については次の性質を充します:

対象の積の性質

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

対象の積が結合律を充すことから、2個以上の対象の積は $A_1 \times A_2 \times \dots \times A_n$ と括弧を外した形で表記することが可能であることが判ります。また、対象 A の n 個の積 $A \times \dots \times A$ を A^n と表記することにします。

つぎに「対角矢 Δ_A 」を定義しておきます:

^{*41} 順序数も集合です

対角矢

以下の可換図式を充す矢 $\langle \text{id}_A, \text{id}_A \rangle$ を特に対角矢と呼び記号 Δ_A で表現される。

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

この対角矢 Δ_A は後述の特性写像 δ_A との関係で重要です。

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

対象の直和

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

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

この直和の可換図式は直積の可換図式の双対になります：

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

この直和は $A \xrightarrow{f} C$ と $B \xrightarrow{g} C$ より C は A, B よりも普遍であり， $A \amalg B \xrightarrow{\langle f, g \rangle} C$ から C は $A \amalg B$ よりも普遍であり， $A \xrightarrow{\pi_1} A \amalg B$ と $B \xrightarrow{\pi_2} A \amalg B$ より A, B よりも $A \amalg B$ の方が普遍であるということになります。つまり， $A \amalg B$ は A, B に最も近い普遍であるということが判ります。

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

矢の積

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

この矢の積は，対象の積の可換図式の特殊な例として考えられ，以下の可換図式として

示すことができます:

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

さらに積を有する圏 \mathcal{C} では対象の幂も定義することができます:

——対象の幂——

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

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

ここで矢 ev のことを「評価 (evaluation)」、矢 \hat{g} のことを矢 g の「転置 (transpose)」と呼びます。さらに対象の幂については $\text{Hom}(C \times A, B) \cong \text{Hom}(C, B^A)$ が成立します。

次に「等化 (イコライザー, equalizer)」について述べましょう:

——等化 (equalizer)——

二つの矢 $A \xrightarrow[g]{f} B$ に対して矢 $C \xrightarrow{e} A$ が $f \circ e = g \circ e$ 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 $D \xrightarrow{h} A$ が一つだけ存在するときに矢 e を矢 f と矢 g の等化と呼ぶ。

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

ここで二つの矢 f, g の等化 e は必ずモノになります。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:

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

を充すことになりますが、ここで矢 e が等化なのでこのような矢 h, k がただ一つ存在しなければならず、その結果 $h = k$ 、このことから等化 e がモノであることが判ります。

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

余等化 (coequalizer) —

二つの矢 $A \xrightarrow[g]{f} B$ に対して矢 $B \xrightarrow{e} C$ が $f \circ e = g \circ e$ 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 $B \xrightarrow{h} D$ が一つだけ存在するときに矢 e を矢 f と矢 g の余等化と呼ぶ。

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

余等化については等化の双対であるためにエピになることが判ります。

2.5.14 引き戻しと押し出し

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

引き戻し (pull back)

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

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

引き戻しの性質の性質を幾つか挙げておきます:

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

- 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:

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

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

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

は引き戻しになりますが、ここで

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

をよくよく考えると次の集合の積と等化が現われます：

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

ここで $h \times k$ は $d \in D$ に対して $(h(d), k(d)) \in A \times B$ を対応させる写像です。この例は集合の圈 **Set** で考えたことですが、ここで対象の積と等化が現われていることから引き戻しには対象の積と等化が関係していることが予想できます。実際、対象の積、等化と引き戻しについては以下の関係があります：

積, 等化と引き戻しの関係

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

ここでの証明ですが、まず任意の $A \xrightarrow{f} C \xleftarrow{g} B$ に対して次の等化が考えられます：

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

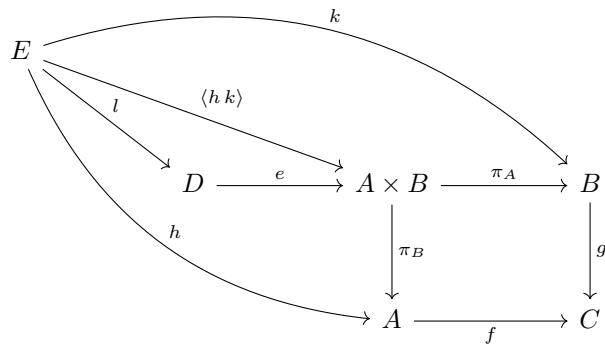
このときに

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

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

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

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

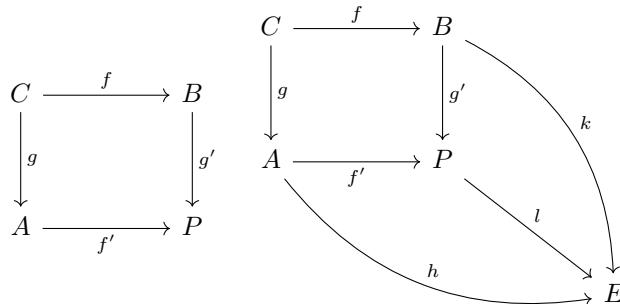


のことから $A \xleftarrow{\pi_A \circ e} D \xrightarrow{\pi_B \circ e} B$ が $A \xrightarrow{f} C \xleftarrow{g} B$ の引き戻しになっていることがわかります。

「引き戻し」の双対に「押し出し (push out)」があります:

——押し出し——

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



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

極限と余極限

引き戻しと押し出し、それと等化と余等化は互いに双対の関係にあります。この直接的なものの見方に加えて別の見方をすることができます。ここでは極限と余極限という概念を導入しましょう。

まず、圏 \mathcal{C} とその部分圏 \mathcal{D} を考えます。それから圏 \mathcal{D} の対象を D_i と添字を付けられたものとし、対象間の矢があれば $D_i \xrightarrow{\delta_{ij}} D_j$ と表記します。そして、この圏 \mathcal{D} を「**図式 (diagram)**」と呼びます。このときに「**錐 (cone)**」と「**余錐 (cocone)**」を次で定めます：

錐の定義

圏 \mathcal{C} の対象 X が錐であるとは、部分圏 \mathcal{D} の対象間の矢 $X \xrightarrow{\nu_i} D_i, X \xrightarrow{\nu_j} D_j$ に対して $\nu_j = \delta_{ji} \circ \nu_i$ を充すときである。

余錐の定義

圏 \mathcal{C} の対象 X が錐であるとは、部分圏 \mathcal{D} の対象からの矢 $D_i \xrightarrow{\mu_i} X, D_j \xrightarrow{\mu_j} X$ に対して $\mu_i = \mu_j \circ \delta_{ji}$ を充すときである。

極限 (limit) の定義

圏 \mathcal{C} の対象 X が部分圏 \mathcal{D} の極限であるとは次の条件を充すときである：

1. 部分圏 \mathcal{D} の任意の対象 D_i に対して $\nu_i = \delta_{ij} \circ \nu_j$ が成立するような矢 $X \xrightarrow{\nu_i} D_i$ が存在する
2. $\mu_i = \delta_{ij} \circ \mu$ を充す対象 $Y \in \mathcal{C}$ と矢 $Y \xrightarrow{\mu_i} D_i$ が存在するときに $Y \xrightarrow{h} X$ が一意に存在する

このとき X を $\lim_{\leftarrow} \mathcal{D}$ と表記する

この極限の双対も考えることができます。極限の双対のことを「**余極限 (colimit)**」と呼びます：

余極限 (colimit) の定義

圏 \mathcal{C} の対象 X が部分圏 \mathcal{D} の余極限であるとは次の条件を充すときである:

1. 部分圏 \mathcal{D} の任意の対象 D_i に対して $\mu_i = \delta_{ij} \circ \mu_j$ が成立するような矢 $D_i \xrightarrow{\mu_i} X$ が存在する
 2. $\mu_i = \mu_j \circ \delta_{ji}$ を充す対象 $Y \in \mathcal{C}$ と矢 $D_i \xrightarrow{\mu_i} Y$ が存在するときに $X \xrightarrow{h} Y$ が一意に存在する
- このとき X を $\lim_{\rightarrow} \mathcal{D}$ と表記する

集合全体の圏 **Set**, 群全体の圏 **Grp** と可換群全体の圏 **Ab**において余極限 $\lim_{\rightarrow} \mathcal{D}$ は \mathcal{D} の元の帰納的極限に対応します.

最後に圏 \mathcal{C} の任意の図式が極限を持つときに圏 \mathcal{C} のことを「**完備 (complete)**」であると言います. 同様に圏 \mathcal{C} の任意の図式が余極限を持つときに圏 \mathcal{C} のことを「**余完備 (co-complete)**」であると言います.

2.6 トポス (Topos)

2.6.1 トポスとは

「トポス (Topos)」はアリストテレスの著作「トピカ (Topica, Τόποι)」に由来するものであり、「Topo」は位相幾何学 (Topology) の「Topo」と同義で「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するものです. そして, トピカで論じられているトポスは偶有性に関するもので 103 個, 類に関するもので 81 個, 特有性に関するものが 69 個, 定義に関するものが 84 個と全部で 337 個のトポスが挙げられています ([3] の註を参照). ここで述べる圏論のトポスはそれに似た働き, つまり, 判断の枠組を与えるものです. 以下, トポスがどのように判断の枠組を与えるかを見てゆきましょう.

2.6.2 部分対象分類子 (subobject classifier)

トポスを導入するためにはまず「部分対象分類子 (subobject classifier)」と呼ばれる圏 \mathcal{C} の対象が必要です. この部分対象分類子は与えられた対象を分類することに対応するもので, 要するに「あれかこれか」の判断に関わるもので. そして, 圏 \mathcal{C} の対象 Ω が「部分対象分類子 (subobject classifier)」であるとは次の性質を充す場合です:

部分対象分類子の定義

- 圈 \mathcal{C} には終対象 1 が存在する
- 圈 \mathcal{C} の対象 Ω に対し、任意の mono: $A \xrightarrow{f} B$ について次の図式が引き戻しになる矢 $A \xrightarrow{\chi_f} \Omega$ が一意に存在する。

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

このとき対象 $\Omega \in \mathcal{C}$ を圈 \mathcal{C} の「部分対象分類子 (object classifier)」と呼び、矢 $b \xrightarrow{\chi_f} \Omega$ のことを「特性矢」と呼びます。

この図式の意味するところですが、ここで圈 \mathcal{C} を集合の圈 Set で説明しておきましょう。このとき、対象 A, B の関係は $A \subset B$ として考えることができます。それから Ω を $\{\text{True}, \text{False}\}$ の二つの真理値の集合だとしましょう。次に \top は $A \xrightarrow{!} 1$ が一意に存在することから mono であり、そのことから 1 を True に写せば、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であることから、矢 χ_f のことを「特性矢」と呼ぶ理由になります*42。

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

対角矢 Δ_A の特性矢 δ_A

圈 \mathbf{E} における以下の可換図式を充す矢 δ_A のことを「対角矢 Δ_A の特性矢」と呼び、 $=_A$ とも表記する：

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

この対角矢 Δ_A の特性矢 δ_A が対象 A の同値性の演算子 $=_A$ を定めています。なお、フレーゲは真理値について $\forall x(x = x)$ を真 (True), $\forall x(x \neq x)$ を偽 (False) として定義

*42 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を与えるものと言えます。

していますが [17], ここで同値性 $=_A$ の定義で真理値集合に該当する部分対象分類子 Ω が与えられていなければならぬことから, フレゲの真理値の定義が妥当では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言えても演算子 $=$ 自体が真理値に依存するために $\forall x(x = x)$ を真理値 True の定義にすることはできないからです. ところで, アリストテレスによると「真や偽は命題の状態を示すもの」であり, 真理値に対応する状態とは, 「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」が真である ([2]11b27) と述べていることは興味深いことです.

同値性 “ $=_A$ ” と同様に連言 “ \wedge ” も定義することができます:

矢 \wedge の定義

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

圏 \mathcal{C} を集合の圏 **Set** とし, 部分集合分類子 Ω を $\{T, F\}$ とするときに矢 “ \wedge ” は (T, T) 以外は全て F 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ます. さらに圏 \mathcal{C} に始対象 0 が存在するときに否定 \neg が定義できます:

矢 \neg の定義

$$\begin{array}{ccccc} 0 & \xrightarrow{!} & 1 & \xrightarrow{!} & 1 \\ \downarrow 0_1 & & \downarrow T & & \downarrow T \\ 1 & \xrightarrow{\perp} & \Omega & \xrightarrow{\neg} & \Omega \end{array}$$

この \neg の定義で $T : 1 \rightarrow \Omega$ に対応する重要な矢 $\perp : 1 \rightarrow \Omega$ を左側の可換図式で定義します. この矢 $1 \xrightarrow{\perp} \Omega$ は矢 $1 \xrightarrow{0_1} 1$ の特性矢で, すこし捻くれた働きをします. 実際, 圏 \mathcal{C} が集合の圏 **Set** のときに矢は通常の写像で, 始対象 0 は空集合 \emptyset になります. さらに $\Omega = \{T, F\}$ のときに矢 \perp を定義する左側の可換図式から, 矢 \perp は空集合 \emptyset を $T \in \Omega$ に写し, それ以外を $F \in \Omega$ にであることがわかります. このことから右側の可換図式から矢 $\Omega \xrightarrow{\neg} \Omega$ は集合 Ω の成分 T と F を相互に入替える写像になることが判ります. このように圏 \mathcal{C} に終対象 1 が存在して部分対象分類子 Ω が存在すれば, 「あれかこれか」という判断に対応する特性写像があり, それによって連言も定義され, さらに圏 \mathcal{C} に始対象

0 も存在すれば否定も定義ができますことになります。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なのです。

2.6.3 基本トポス

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

—— 基本トポスの定義 ———

1. 圈 \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 $A, B \in \mathbf{E}$ に対して幂 $B^A \in \mathbf{E}$ が存在する。
4. \mathbf{E} には部分対象分類子 Ω が存在する。

ここで 1., 2., 3. を充す圏のことを「**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て、その同一性や連言、否定も定義可能なことから一階の論理式が構築可能です。

—— トポスの定義 ———

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

トポスになる圏として代表的なものとして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ることになりますが、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るのです。

2.6.4 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ることにします。そのためには「**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。このとき任意の $1 \xrightarrow{g} A \xrightarrow{h} A$ となる対象と矢に対して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) と呼ぶ。

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

ペアノの公理系との対応では対象 N が自然数 \mathbb{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。

さらにトポス \mathbf{E} が自然数対象を持つときにトポス \mathbf{E} の任意の対象と矢 $A \xrightarrow{g} B \xrightarrow{h} B$ に対して次の可換図式を充たす矢 $A \times N \xrightarrow{f} B$ が一意に存在します:

$$\begin{array}{ccc} & A \times N & \xrightarrow{\text{id}_A \times \sigma} A \times N \\ \langle \text{id}_N \xi_N \rangle \nearrow & \downarrow f & \downarrow f \\ A & \searrow g & \\ & B & \xrightarrow{h} B \end{array}$$

ここで ξ_N は矢 $A \xrightarrow{!_A} 1 \xrightarrow{\xi} N$ です。そして $A \xrightarrow{g} B \xrightarrow{h} B$ が与えられたときに以下の可換図式を充たす矢 $h' : B^A \rightarrow B^A$ が存在します:

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

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

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

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

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

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

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

和 “ $+_{\mathbf{E}}$ ” が次で定義できます:

$$\begin{array}{ccc}
 N \times N & \xrightarrow{\text{id}_N \times \sigma} & N \times N \\
 \swarrow \langle \text{id}_N \xi_N \rangle & \downarrow +_{\mathbf{E}} & \downarrow +_{\mathbf{E}} \\
 N & \xrightarrow{\text{id}_N} & N \\
 & \searrow \sigma & \downarrow \\
 & N &
 \end{array}$$

2.7 トポスの基本定理

トポスの基本定理

- 圈 \mathcal{E} がトポスであり, B が \mathcal{E} の任意の対象であるときにコンマ圏 $(\mathcal{E} \downarrow B)$ もトポスになる.
- A, B をトポス \mathcal{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 $(\mathcal{E} \downarrow A)$ と $(\mathcal{E} \downarrow B)$ の間に函手 $f^* : (\mathcal{E} \downarrow B) \rightarrow (\mathcal{E} \downarrow A)$, $\Sigma_f : (\mathcal{E} \downarrow A) \rightarrow (\mathcal{E} \downarrow B)$ と $\Pi_f : (\mathcal{E} \downarrow A) \rightarrow (\mathcal{E} \downarrow B)$ が存在して $\Sigma_f \dashv f^* \dashv \Pi_f$ を充たす.

2.8 高階論理 λ -h.o.l. とトポス

ここでは「圏論による論理学」[11] に従って函数型高階論理 λ -h.o.l とあるブーリアン・トポス \mathbf{E} との対応関係を紹介します. ここでブーリアン・トポス (Boolean topos) とはトポス \mathbf{E} で部分対象分類子 Ω として真理値 $\{T, F\}$ とするものです.

2.8.1 高階論理 λ -h.o.l. について

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

t は真理値の型を示すものとします。そして、領域 D_α から領域 D_β の写像全体で構成される領域は、その領域の型を $\langle\alpha\beta\rangle$ でその型を定めることができます。ただし、この写像の型については次の規則を入れることにします：

函数の型の表記

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

ここで 1. については単なる略記として認めるることは問題がないでしょう。2. については前半の外側の括弧を外した略記も特に問題はないでしょう。後半の括弧を全て外した表記と前半の表記との関係については、右側の記号との結合が強いとする立場を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の問題にはなりません。ちなみに論理学に函数概念を最初に導入したのはフレーゲですが、彼は函数を二項間の関係とみなしており、その場合、項の場所に注目をしています。それから三変数以上の函数についても二項関係の延長として捉えています。

以上から λ -h.o.l. の型を次で定義します：

λ -h.o.l. の型 (type)

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

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

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

λ -h.o.l. の基本記号

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

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

論理記号 ‘ $=$ ’ の定義

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

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

 λ -h.o.l. の型付き項の定義

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

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

論理記号と式の定義

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

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.	$A_t \supset B_t$	$\stackrel{\text{Def.}}{=} ((\supset_{t \langle tt \rangle} A_t) B_t)$

2.8.2 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	\rightarrow	対象 E
型 t	\rightarrow	対象 Ω
型 $\langle \alpha, \beta \rangle$	\rightarrow	対象 B^A

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

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

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

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

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

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

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

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

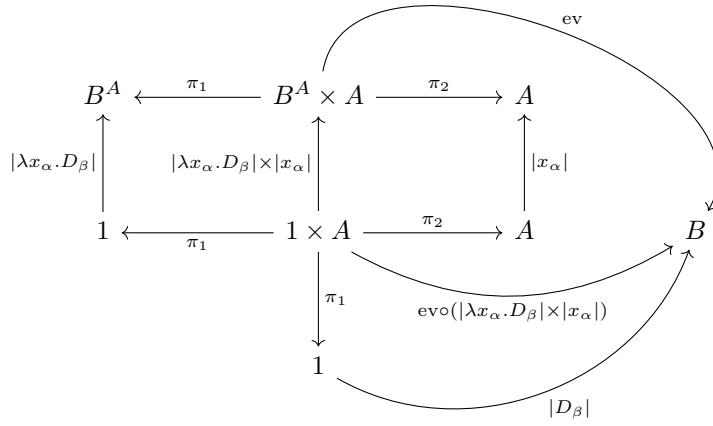
$$\begin{array}{ccccc}
 & 1 & \xrightarrow{|C_{\alpha\beta}|} & B^A & \\
 & \pi_1 \nearrow & & \uparrow \pi_1 & \\
 1 \times A & \xrightarrow{|C_{\alpha\beta}| \times |x_\alpha|} & B^A \times A & \xrightarrow{ev} & B \\
 & \pi_2 \searrow & & \downarrow \pi_2 & \\
 & & A & \xrightarrow{|x_\alpha|} & A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

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

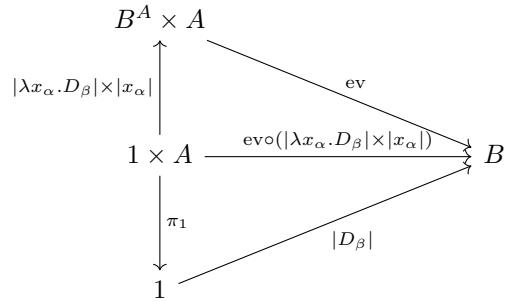
ただし, この図式では矢の積が判り易くなるように $1 \times A$ を射影 π_1, π_2 を使って分解したくどい説明になっています.

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

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



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



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

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

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

この図式の骨子を取り出したものが次の可換図式になります:

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

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

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

$$\begin{array}{ccccc}
 & & A & & \\
 & \nearrow |C_\alpha| & \uparrow \pi_1 & & \\
 1 & \xrightarrow{\langle |C_\alpha|, |D_\alpha| \rangle} & A \times A & \xrightarrow{\delta_A} & \Omega \\
 & \searrow |D_\alpha| & \downarrow \pi_2 & & \\
 & & A & &
 \end{array}$$

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。

2.8.3 λ -h.o.l. の解釈について

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

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

を ‘ $\boxed{\quad} a$ ’ とし、それと「Modus Ponens(MP)」と呼ばれる推論規則^{*43}、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：

$$\boxed{\quad}_{\lambda\text{-h.o.l. の含意 } \supset \text{ の定義}} \supset_{t(t,t)} \stackrel{\text{Def.}}{=} \lambda x_t. \lambda y_t. x_t = (x_t \wedge y_t)$$

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

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

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

で定義されます。この定義もトポス \mathbf{E} の下で

^{*43} ‘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。

トポス E における量化詞

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

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

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

第3章

Pythonについて

3.1 Python とは？

SageMath は「**車輪の再発明**」を行わないという方針が開発の前提にあります。そのために SageMath を記述する言語である Python を使って必要な機能を最初から造り上げるのではなく、必要とされる機能を有する既存のアプリケーションやライブラリを Python を使って繋ぎあわせる戦術を採用しています。だから SageMath を使いこなすためには何よりも Python を使いこなさなければなりません。

この Python は一つの言語仕様であり、その実装には C による CPython, Java による Jython, .Net Framework/Mono による IronPython, さらには Python 自体^{*1}による PyPy があります。ここで念頭に置いている Python は SageMath の記述で用いられている CPython で、その Python のバージョンは 2.7 系で新しい 3.x 系ではありません。これは 2.x 系と 3.x 系に互換性の問題があるためで、SageMath が必要とするパッケージが全て 3.x 系に移植されるまではこの状態が続くでしょう。

ここで処理系とユーザの仲立を行うソフトウェアを卵の内部と外部を隔てる境界の殻にたとえて「**シェル (shell)**」と呼びます。Python の代表的なシェルに、CPython に付属するシェル、IPython や Jupyter^{*2}があります。特に CPython に付属のシェルは「**インタプリタ (interpreter)**」と呼ばれ、そのユーザ・インターフェイスは CLI(Command Line Interface) と呼ばれる昔ながらの行単位で命令文を入力する形態で、簡易な行編集機能を持っています。それに対して IPython や Jupyter はより高度な行編集や履歴機能に加え、ウェブ・ブラウザを使ったノートブック形式のユーザ・インターフェイスも実現しています。

この章では最初に Python 2.x 系の概要を述べ、例として有理数の定義を行います。それから「Python 言語リファレンス」^{*3}にしたがって Python(2.x) の言語的な説明を行います。なお、Python の実例では CPython のインタプリタを主に用いますが、SageMath で IPython が用いられているために必要に応じて IPython 等のシェルについても説明します。

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

^{*2} Jupyter は IPython の後継で、Python に依存する箇所を切り分けて Haskell, Julia, GNU Octave や GNU R 等に対応したノートブック形式のユーザ・インターフェイスを実現しています。

^{*3} Python 言語リファレンス: <http://docs.python.jp/2/reference/index.html>, 2.x 系向けの文書です。

3.2 Python の特徴

3.2.1 多重模範言語

Python はクラスに基づくオブジェクト指向プログラミング言語 (OOPL) と呼ばれる言語の一つで、インタプリタを利用すると命令型プログラミング言語の Basic のように対話形式で、あるいは LISP のように函数型風に、さらには C のように手続型風にプログラミングを行うことが可能です。このように目的に応じてさまざまなプログラミング様式が選べる言語のことを「**多重模範言語 (Multiparadigm programming language)**」と呼びます。

3.2.2 簡素化された構文

Wikipedia に「核となる構文や文法が必要最低限に抑えられている」とあるように Python は構文が簡素です。実際、変数の宣言は不要で、条件分岐は if 文、例外処理は try 文、反復処理は for 文と while 文と必要最低限の構文に抑えられ、その結果、Perl^{*4} や Mathematica で見られる「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仕様です。このことは一つの問題に対するプログラムが多種多様な形態ではなく、均質的なプログラムに収斂する傾向があると言えます。このことに加え、PEP-8^{*5}に代表されるようなプログラム記述のためのガイドラインがあり、それらのガイドラインにしたがってプログラムを記述しさえすればある程度の品質を保持したプログラムが期待できます。このよう Python は覚えることが少なくて過剰な技巧に走る必要性がなく、そのガイドラインに沿っていればある程度の品質が保てるという実用本位の言語です。

3.2.3 構文要素としての字下げ

C や Java 等の多くの言語でプログラムの構造の視覚的把握のために「**インデント (indent, 字下げ)**」が用いられますが、この字下げは任意です。ところが Python の字下げは構文上の必要不可欠な要素で、Python のプログラムは二次元的な構造を有し、その構造を視覚的に把握することができます^{*6}。この字下げは PEP-8 で欧文間隔 (Space) のみの 4 文字で行うことが推奨されています。

^{*4} 駱駝形のプログラム例 (camel code): <https://gist.github.com/cgoldberg/4332167>。4 頭のラクダを ASCII アートで描きますが、プログラム自体が ASCII アートになっています。

^{*5} PEP (=Python Enhancement Proposal): Python 改善提案書

^{*6} 仕様として字下げを持つ言語に FORTRAN77 もありますが、FORTRAN77 はカード読取機の制約に由来するもので、プログラムの構造の可視化を意図したものではありません。

たとえば、C の if 文は直線的に

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

と記述しても、平面的に

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

と記述しても空白文字の Space, TAB や改行による字下げは C の構文上、意味がないためにプログラマーの意図とは別に、プログラムとしての違いもなければ構文上の問題もありません。ところが Python ではクラスやメソッドの宣言、分岐や反復といった構文が複数の行で構成されるときに、その構文を構成する行に対して空白文字の Space や TAB を使って、それらの文字数と並びを含めた水準を揃えて字下げを行う必要があり、

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

のように if 文内部の文（ここでは ‘y = 1’ と ‘y = 0’）と if 文を構成する文節 if と else が同じ字下げの水準になければなりません。より正確には文節の末端に記号 “:” がある場合、次の行から字下げを行うか、その次の行に記号 “:” が含まれないときのみ線的に記述することが許容され、字下げを伴う構文を終えると字下げの水準を一段階戻します。

したがって

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

と記述することは許容されても、記号 “:” を二つ以上含む線的な記述:

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

と字下げの位置がチグハクでプログラムの構造が表現できていない記述:

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

の双方は Python の構文エラーになります。このように Python が構文の一要素として字下げを取り入れたことに類似の事例が歴史的な論理式の表記にもあります。フレーゲの概念記法がそれで、その独特的な論理式の記号に加えて平面的な構造を持っています。ただし、フレーゲが活躍した 19 世紀末から 20 世紀初頭の印刷技術でその図式の印刷は非常に面倒であったために「算術の基本法則」を二巻に分けて出版することになり、さらに論理式の記述で主流になることもなく、ペアノ (Peano) による線的な表記が現在の論理式の表記の源流になっています。この事例を鑑みると、ed や EDLIN といったラインエディタではなくスクリーンエディタが主流でディスプレイも大画面、多画面であることがこの字下げを構文要素の一つとして取り入れる一助になっていること、次に述べる文書文字列というある意味過剰な文書をプログラムの解説文書として包含できること、また、Python の本体を小さくできることも必要なパッケージをインターネット経由で容易に入手可能であることから可能になっていること等と Python はさまざまな技術発展による恩恵を受けている言語です。

3.2.4 文書文字列 (docstring)

Python にはプログラム内部に解説や例題等の文書を包含することでプログラムの文書性を高める工夫があります。このプログラムに埋め込まれた文字列を「文書文字列 (docstring)」と呼びます。このプログラム中に文字列を組込む工夫は LISP や MATLAB 系言語で見られます。ここでは最初に Common LISP^{*7} の例を示しましょう：

```
* (defun add2 (x) を足すよー"2" (+ x 2))  
  
ADD2  
* (documentation #'add2 'function) を足すよー  
  
"2"
```

この例では defun 文で函数 add2 の定義を行い、函数に記載した文字列を函数 documentation で表示させています。なお、この例で左端の文字 “*” が SBCL のプロンプトです。

^{*7} Common LISP の処理系の一つの SBCL です。

数値行列処理に長じている MATLAB に類似した言語をこの本で MATLAB 系言語^{*8}と呼びます。さて、MATLAB 系言語では文書文字列に相当する文字列をファイルから検し出すために函数単位で一つのファイルにあらかじめ記述しておく必要があります^{*9}。ここでは、MATLAB 言語の中では異端の言語である Yorick^{*10}の例を示しておきます：

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

それから add2.i と函数名に対応するファイル名で保存して Yorick 上で函数 include() を使ってファイルを読み込みます。あとは函数 help() でファイル add2.i に記載された文書文字列が表示できます。この仕組は他の MATLAB 系の言語でもほぼ同様です、以下に Yorick の実例を示します。なお、左端の文字“>”は Yorick のプロンプトです：

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

この Yorick の例で示すように MATLAB の系の言語ではプログラム中の文書としての性格をより強く持っています。実際、MATLAB 系の言語ではオンライン・ヘルプの例題を含む文書として記載されています。これらの例と同様のことを Python では

```
>>> def add2(x):
...     """
...     を足すよー    2
```

^{*8} MATLAB クローンの GNU Octave, INRIA で開発され高機能でツールが揃っている Scilab, MATLAB 風の配列処理が可能な C といった雰囲気の Yorick があります。これらの言語は行列や配列の処理方法に類似があり、Python ではスライス処理と呼ばれる手法に対応します。

^{*9} MATLAB ではこのファイルを M-file と呼びます。その理由はファイルの修飾子が“.m”だからです。たとえば、neko() という MATLAB の函数を定義するときは neko.m というファイル名に函数一式とその函数内部でのみ用いる函数を記述します。

^{*10} Yorick は C 風の軽量の言語で多次元配列操作に長けており、Python の配列の拡張スライス操作に類似した操作もあります。

```
...
    """
...
    return x+2
...
>>> help(add2)

Help on function add2 in module __main__:

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

と MATLAB 系言語と同様のことができます。ここで Python で文書文字列を記載する位置は、函数やメソッドに対しては MATLAB 系言語と同様に、それらを定義する def 節の次の行から記載します。この例では直接、インタプリタに定義文を入力しています。最初に文字の列 “>>>” はインタプリタの通常のプロンプトで、次に現れる文字の列 “...” は Python の文が入力途上にあることを示すプロンプトで、函数 add2() の定義が継続中であることをプロンプトを切り替えることを利用者に知らせています。そして、文書文字列のエンコーディングが UNICODE であることを指示するために接頭辞 “u” を配置しています。Python ではオブジェクトの文書文字列を函数 help() で表示することができます。この例では早速定義した函数 add2() の文書文字列を函数 help() で表示していますが、Python ではオブジェクトの属性 __doc__ に記載した文書文字列が割り当てられます。そのことを add2.__doc__ で確認しています。

これらの例では、MATLAB 系の言語と同様のことしか行っていませんが、Python の文書文字列は MATLAB 系の言語が有する文書文字列以上の機能を持っています。まず、文書文字列は 3 個の二重引用符 ("""") で括られた文字列で、3 個の二重引用符で括られることにならない限り、引用符を文書文字列内部に記載することは問題ありません。そのため文書の自由度が高く、後述の reST のような組版指示言語であっても構いません。また、例のようにオブジェクト内の文書文字列の閲覧には函数 help() が使えます。ただし、Python の組込のオブジェクトに関して函数 help() で表示されるものは pydoc モジュールで整形されています。以下にインタプリタ上で函数 help() を使って函数 open() を調べた様子を示します：

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

open(...)
    open(name[, mode[, buffering]]) -> file object
```

```
Open a file using the file() type, returns a file object. This is the
preferred way to open a file. See file.__doc__ for further information.
lines 1-7/7 (END)
```

この例では pydoc で整形されているとはいっても、本文は関数 open() の文書文字列で、内容は ‘__builtins__.open.__doc__’ で確認することができます。なお、インタプリタで関数 help() で引数を指定せずに ‘help()’ と入力するとヘルプシステムが起動してプロンプトが “help>” に切り替わり、この状態で調べたい事項を入力すれば関数 help() と同様の結果が得られます。また、このヘルプシステムから抜けるときは ‘quit’ か ‘q’ の何れかを入力すればインタプリタに戻ります。

なお、IPython や Jupyter を Python のシェルとして用いているときは関数 help() の他に記号 “?” が使えます：

```
In [1]: open?
Type: ?op      builtin_function_or_method
String Form:<built-in function open>
Namespace: Python builtin
Docstring:
open(name[, mode[, buffering]]) -> file object
```

```
Open a file using the file() type, returns a file object. This is the
preferred way to open a file. See file.__doc__ for further information.
```

このように記号 “?” はオブジェクトの名前に空白文字を入れずに続けて入力します^{*11}。

この文書文字列はプログラム内部に組み込むことが容易で、プログラムの解説や例題の記載等にも使えます。そして、文書文字列の表記に関しては PEP-257 にその規約があります。さらに文書文字列を reST(reStructuredText) と呼ばれる「組版指示 (markup) 言語」で記述することが PEP-287 で提唱されています。この reST は表現しようとする書式をアスキーアート風の表記で行うために記述性、可読性が高く、さらに Docutils^{*12}を用いることで HTML や LATEX 等のさまざまな書式への変換が可能で、これらの機能から reST で記述した文書はより高い表現力を持つことができます。ここで図 3.1 に SageMathCloud 上で編集した reST の一例を示しますが、左側が reST の記述、右側がそのレンダリング結果です：

^{*11} IPython では名前の前に記号 “?” の記載が可能ですが、SageMath ではエラーになります。

^{*12} Documentation Utilities <http://docutils.sourceforge.net/>

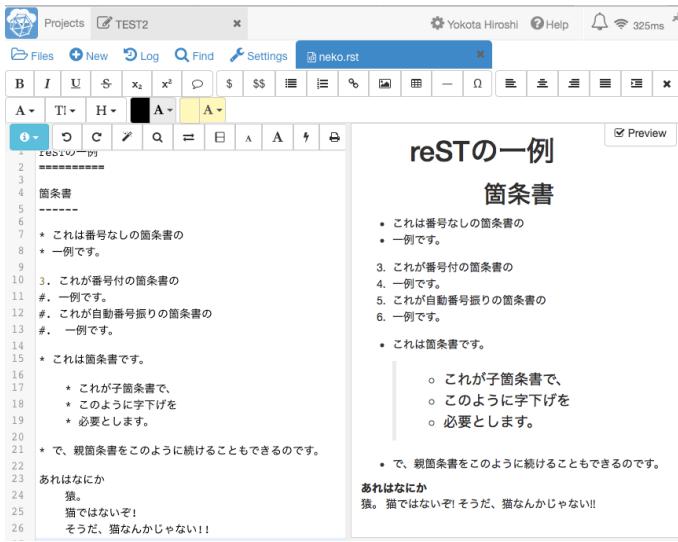


図 3.1 reST の一例

文書文字列はエディタで直接作成することも容易ですが、文書生成ツールの Sphinx^{*13}が標準的なツールです。この Sphinx を使うことで reST 文書を LaTeX, HTML や PDF といった書式の文書に変換することも容易に行えます。このように Python はプログラムの文書化も重視しています。

3.2.5 クラスに基づくオブジェクト指向

Python はクラスに基づくオブジェクト指向プログラミング言語です。ここでオブジェクト指向プログラミング言語は扱う対象全てを「**オブジェクト**」として捉えます。これは数値や文字列といったデータはもちろんのこと、函数やクラスといった諸々のものがオブジェクトになります。さらにクラスに基づくオブジェクト指向プログラミング言語は扱うべき対象（オブジェクト）を抽象化して「**概念**」として捉え、それから「**クラス (class)**」をオブジェクトを説明規定するもの、すなわち**概念の内包**や**概念の外延**として表現します。そして実際に扱うデータは現実のもの（に最も近接するもの）であることから「**概念の外延を構成する個体**」に相当し、クラスが実体化したものとして捉えられます。このクラスの実体化のことを「**インスタンス化 (instantiation)**」、インスタンス化したオブジェクトを「**インスタンス (instance)**」と呼び、インスタンスはそのクラスの成員になります^{*14}。それからクラスもあるクラスのインスタンスになる場合があります。このときに「**クラス**

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

*14 ただし、クラスが集合になるとは限りません！

がインスタンス化したクラス」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)**」と呼びます。そして、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。

Python ではクラスの定義を class 文で行います。以下に最も簡単なクラスの定義を示しておきます*15。

```
class TEST:  
    pass
```

このクラスは名義的な定義を行うクラスで、pass 文は「**何もしない**」ことを意味する文です。このクラスの実体に対応するオブジェクトの生成は ‘a = TEST()’ で名前 a への束縛も行ないます。このクラス TEST では対象の振舞いや初期値が何も決まっていなくて好き勝手ができます。その様子を以下に示しておきましょう：

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

ここでやっていることは最初に TEST クラスを定義したのちに ‘a = TEST()’ で実体化したオブジェクトに名前 a に束縛し、a.pet, a.weight や a.age に値を設定するという C の構造体と同様の処理をしています。ここで名前 a に続く name や age は名前 a で参照されるオブジェクトが属するクラス TEST の「**属性 (attribute)**」であり、クラス TEST が表現する概念の属性に相当します。

次にもう少し複雑なクラスを定めてみましょう。ここで定義するクラスは C の構造体に類似した書式です：

*15 この定義方法で Python 2.x は古典的クラス (旧スタイル)、Python 3.x はクラスタイプ (新スタイル) になるという相違点がありますが、ここでの解説ではその差異が露呈しません。

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

このクラスの定義では二つの属性 `x` と `y` があり、初期値として 1 を設定しています。実際に使ってみましょう：

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

この例ではクラス `TEST` を定義し、「`a1 = TEST()`」でインスタンス化と同時に名前 `a1` にオブジェクトを束縛させて属性を参照しています。ここで属性参照は

〈インスタンス名〉.〈属性名〉、その属性値変更は 〈インスタンス名〉.〈属性名〉 = 〈属性値〉で行います。と、ここまで使い方ではクラスは C の構造体と大差はありません。

Python のクラスには属性だけではなくメソッドという機能があります。たとえば猫に「雨が降る前に顔を洗うような仕草をする」という習性があるために家の猫のミケがそのような仕草をしたときにあらかじめ洗濯物を取り込むことは妥当な行為です。しかし、犬のポチがそのような仕草をしたからといって洗濯物を取り込む理由にはなりません。この場合はミケやポチといったインスタンスが属するクラスが犬と猫で異なっており、おまけに犬にそういう習性がないからです。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です。このメソッドはオブジェクトに結び付けられた函数として表現され、オブジェクトに結び付けられているがために通常の函数とは異なり、そのクラスのインスタンスであるか、そのクラスと継承関係にあるクラスのインスタンスでなければ使えません。

また、クラスに設定された値にせよ、そのクラスであれば一定値になるものや取り得る値が一意でないものや、ある値を持つということ自体がそのオブジェクトを特徴付けるもの、つまり、「**特有性**」である場合、あるいはその値が程度を表現し、その値が設定されない状態も考えられる場合、つまり、「**偶有性**」の場合が考えられます。また属性の役割を考えると、そのクラスを特徴付ける種差や特有性のようにクラス単位で共通になる値を格納する属性、個々のインスタンスごとに異なる値を格納す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を格納する属性を「**クラス変数**」、後者のようにインスタンスごとに異なる値を収納する属性を「**インスタンス変数**」と呼びます。ここでC++ではクラス変数とインスタンス変数で宣言が異なりますが、Pythonにはクラス変数やインスタンス変数といった型の宣言がないために変数の使い方で両者を区別することになります。たとえば、「**クモ**は足の数が8本」の「足の数が8本」はクモというクラスを特徴付ける属性の一つで、「足の数」がクラス変数です。また、「**犬**のニコは年齢が6ヶ月」の「年齢」は犬というクラスに付随する属性の一つですが、個体(インスタンス)で異なるため、こちらはインスタンス変数の例になります。

Pythonのメソッドにはクラス操作に関わる「**クラスメソッド(class method)**」とインスタンス操作に関わる「**インスタンスマソッド(instancemethod)**」の二種類があり、これらのメソッドは変数と異なって全てクラスの属性です。ここでインスタンスマソッドは名前に「**インスタンス**」とあってもインスタンスの属性ではなく、あくまでもクラスの属性です。そのためにインスタンス変数の参照が行えません。そして、クラスメソッドは「**スタティック(静的)メソッド(staticmethod)**」と「**クラスメソッド(classmethod)**」があり、これらのメソッドの定義で「**デコレータ**」が用いられます。つまり、スタティックメソッドの定義で‘@staticmethod’、クラスメソッドの定義で‘@classmethod’が用いられます。さらに、クラスメソッドの定義では第一引数に‘cls’というクラスそれ自体に対応する引数があるために、そのクラスの属性の参照が可能ですが、スタティックメソッドはクラスそれ自体を示す引数を持たないために参照するクラス名を直接指定する必要があります。そのためにクラスメソッドでは指定した属性がそのクラスになければ継承したクラスへと遡った動的(dynamic)な参照が行われるのに対し、スタティックメソッドでは名前で直接指定したクラスの参照に留まります。つまり、「**スタティック(静的)**」という意味は「**属性の参照が静的(スタティック)に行われる**」という意味です。

では先程のTESTクラスのクラス変数に値を束縛し、それらの和を計算するインスタンスマソッドを追加したクラスの定義をしてみましょう：

```
class TEST:
```

```
x = 1
y = 1

def wa(self):
    return self.x + self.y
```

この例ではクラス変数とインスタンスマソッドを属性として持つクラス TEST を定義しています。メソッドの定義自体は Python の函数定義と構文は同様ですが、インスタンスマソッドの定義で操作すべきインスタンスを ‘self’ いう名前で表現して第一引数に与えます。またメソッド内部でインスタンスの属性の参照でも self を用いてインスタンスそれ自身を表現します。なお、ここで定義するインスタンスマソッド wa() はクラス属性 x と y の和を返却します。実際に動かしてみましょう：

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

と、この例のように引数 self はメソッドの引数として現れません。ところで、定義したクラス等のオブジェクトや、オブジェクトのメソッドや属性に何があるかを調べる方法はないでしょうか？この目的に応えられる函数が組込函数 dir() です。以下に起動したてのインタプリタで函数 dir() を使った結果を示しておきましょう：

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

このように組込函数 dir() は参照可能な範囲内、すなわち、スコープ内の名前のリストを返

却する函数で、引数がなければ大域变数と参照可能なオブジェクトの名前のリストを、オブジェクトの名前が引数として与えられるとそのオブジェクトのメソッドや属性の名前のリストを返却します。この例では ‘dir()’ と引数なしではトップレベルのスコープ内の 大域变数のリストを返却しています。なお、大域变数の情報を返す函数には函数 globals() もあり、こちらは指定したスコープ内の 大域变数の辞書を返却し、ここでは大域变数とその 大域变数に束縛された値の辞書が返却されています。

ところで大域变数 `__name__` の先頭の文字 “`_`” には理由があります。つまり、文字 “`_`” や文字列 ‘`__`’ を名前の先頭に持つメソッドや变数は非公開のものであることを意味します。特に名前の先頭に文字列 ‘`__`’ を持つメソッドや属性は隠蔽されるべきオブジェクトの名前で用いられます。ここで文字 “`_`” 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` の属性を `x`, `__y` と `__z`, クラス `test` のインスタンスを `a1` で属性 値の確認を行っています。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 が追加されています。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。このようにと名前の先頭に文字 “_” が二つある属性の隠蔽しているように見えますが、函数 dir() でクラス test の中にある名前を見ると属性 __z だけに文字の列 ‘_test’ が先頭に置かれています。つまり、文字の列 ‘__’ を名前の先頭に持つ属性は文字の列 _⟨ クラス名 ⟩ が先頭に置かれた名前に内部的に置換されます。そのために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 の値の参照ができるという訳です。さらに他の属性と同様に、その属性値の書換も可能です。このように Python の属性の隠蔽の方法は詮索すれば容易にわかるという不徹底な方法です。なお、ここでは函数 dir() を用いましたが似た処理を行う組込の函数 vars() があり、こちらはリストではなく辞書と呼ばれるデータ型で返すという違いがあります。

このように隠蔽の仕組を有する構造体モドキが使えるというだけではクラスの有難味がありません。オブジェクト指向プログラミング言語の大きな有難味の一つは「継承」と呼ばれる機能、つまり、既存のクラスを土台に新しいクラスを効率的に構成できる機能です。この機能を活用することで既存の資産を新たなシステムの開発に生かすことができるのです。次に自然数を拡張して有理数を構築することでその後利益を体験してみましょう。

3.3 有理数を構築してみよう

3.3.1 有理数の表現

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

まず、有理数は二つの整数 n, m を用いて n/m の書式で表現される数です。だから有理数を一つ定めるためには二つの整数が必要です。そこで最初に整数対のクラスを定義することにします。ここで整数対は与えられた有理数に対して一つ定まるもので有理数全体で一定の値にはなりません。したがって、整数対を格納する属性はクラス変数ではなくインスタンス変数であり、インスタンス化の時点で初期化されること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有理数を表示させるようにします。これらインスタンスの初期化や表示、さらには比較や演算を定めるメソッドに「**特殊メソッド**」と呼ばれるメソッドがあり、これらで上記の目的を達成

することができます。これら特殊メソッドの詳細は §3.7 で述べますが、ここでは分母と分子の値の設定を行うためにインスタンス化の時点でインスタンス固有の属性の初期化を行う特殊メソッド `__init__()` と、インスタンスが割当てられた名前が入力されたときにインスタンスの標準的な表示を定める特殊メソッド `__repr__()` の二つを用いて整数対のクラス `PairOfInts` を以下で定義します：

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

このクラスでは、最初に定義したメソッド `__init__()` がインスタンスの初期化を行うための特殊メソッドです。このメソッドの第一引数の `self` は対象そのものを示し、そのうしろの引数 `numer` と `denom` が実際のインスタンスの生成で必要な引数、つまり、インスタンス変数に束縛すべき値です。ここで二つのインスタンス変数の `numer` が分子、`denom` が分母に対応し、「`a = PairOfInts(1,2)`」で対象を生成すると `a.numer` で属性 `numer` の値、`a.denom` で属性 `denom` の値の参照や代入が行えます。それから二番目に定義されているメソッド `__repr__()` がインスタンスの表示に関わるメソッドで、オブジェクトを指示する名前が入力されたときや `print` 文^{*16}でどのような表示を行うべきかを定め、ここでは出力書式を文字列 '`%s/%s`' で指定することで「`a = PairOfInts(1,2)`」でインスタンスを生成したときに名前 `a` をインタプリタに入力すると文字列 '`1/2`' を表示することを意味します。

このことを実際に試してみましょう。ここではカレントディレクトリ^{*17}上に `PairOfInts` クラスの定義をファイル `PairOfInts.py` に記載して `import` 文でインタプリタに読み込みます。ここで Python に読み込んだファイルは「**モジュール**」と呼ばれるオブジェクトになります。また複数のモジュールを一つにまとめたものなどを「**パッケージ**」と呼びます。そして `import` 文によるファイルの読み込みで名前の衝突を避ける方法が後述の「**名前空間**」です。「`import PairOfInts`」で読み込むと読み込みファイル内部で定義したオブジェクトの参照を行うと、この `PairOfInts.py` ファイルで定義したオブジェクトへの参照では先頭にモジュール名（ここでは `PairOfInts`）を付けます。ただ、この程度の内容で煩

^{*16} Python 2.x では `print` 文ですが、Python 3.x では函数 `print()` に変更され、函数の書式、つまり、引数を括弧 “`()`” で括る必要があります。

^{*17} `os` モジュールの函数 `getcwd()` で確認できます。変更は同様に `os` モジュールの函数 `chdir()` の引数として経路を与えればできます。

わしくしても意味がないので, ‘from PairOfInts import PairOfInts’ でファイルの読み込みを行うことでモジュール名(=ファイル名)を外して使えるようにできます:

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

この例ではインタプリタ上で PairOfInts クラスを import 文で読み込み, そのインスタンスを生成して名前 a に束縛させ, 名前 a を入力して ‘1/2’ という表示を得ています. では, メソッド __repr__() を定義していなければどうなるでしょうか:

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

ここでは PairOfInts クラスからメソッド __repr__() を除いた TEST クラスをインタプリタ上で直接定義し, それから生成したオブジェクトを名前 b のインスタンスとして割当てています. ここで名前 b を直接入力するとメソッド __repr__() が定義されていないために ‘<__main__.TEST instance at 0x4f607a0>’ と表示されるだけです. ここで表示されている ‘0x4f607a0’ という値は組込函数 id() が返却するオブジェクトの識別値と一致します. このようにメソッド __repr__() 等で指示されていない限り, 名前をそのまま入力しても名前が参照するオブジェクトの番地が返されます. またメソッド __repr__() が組込函数の函数 repr() に対応し, メソッド __repr__() が定義されていてもインスタンスの属性とそれらの書式を函数 repr() に引き渡すことで同様の表示を得ることができます. このように特殊メソッド __repr__() を定義することでインスタンスの表示を定めることができます. ここで行ったように上位クラスのメソッドを下位のクラスで新たに定義しなおすことを「上書き (override)」と呼びます. またメソッドの引数の型が異なる場合は上書きではなく「**多重定義 (overload)**」と呼びます. なお, ここで注意することは表示される値が同じものだからといってクラスが異なることがあるということです. 実際, Python の整数型の 1 と SageMath の Integer 型の 1 が別物であるた

めに Integer 型のメソッドが使えません。これは SageMath で 1 を入力すると SageMath の Integer 型になりますが、プログラム内の数リテラルの 1 は Python の整数型になります。そして、双方共に print 文では 1 が表示されるためにクラスの違いが表示からは判りません。

3.3.2 有理数のクラスの構築

さて、この自然数の対はあくまでも自然数の対としての性質の他に何もなく、メソッドにも有理数の表記を使って自然数の対を表示する機能しかありません。ここで我々は有理数を Python 上で表現することを目的にしていますが、現時点では有理数を整数の対として表現することだけで有理数として定義ではありません。つまり、「**有理数とは何なのか**」という問への回答には、二つの整数対が与えられたときにどのような条件を充せば有理数として等しいと判断できるのかという判断基準、互いを比較するという手段を与えることがまだ必要なのです。ここで最も原始的な判断基準は分母と分子がそれぞれ等しければ良いというものです。しかし、これだけでは不十分です。たとえば $1/2 = 2/4$ ですが、有理数として自然数の対 $(1, 2)$ と $(2, 4)$ は等しくなければなりませんが単なる自然数の対として前述の安易な手法によると別物です。また $(-a, -b)$ と (a, b) は等しいものであるべきです。この「等しい」という関係は「**与えられた自然数の対 (a, b) と (c, d) が ‘ $ad - bc = 0$ ’ を満すとき**」と定めることができます。これで「等しい」ことの判断基準が一つ定まりました。そこで有理数は一意に整数対で表現されるべきではないでしょうか？たとえば (a, b) と $(-a, -b)$ は等しい自然数で、このことから分母に相当する denom が常に正となるように置き換えたものをその代表に、同様に 0 と異なる整数 c に対して $(a*c, b*c)$ を (a, b) で置き換えるべきです。つまり、RationalNumber クラスは単なる自然数の対のクラス PairOfInts にこの二種類の正規化を加えたクラスであるべきです。では、等しくない場合はどうでしょうか？ここで二つの有理数が与えられたとき何ができるでしょうか？整数が保持する関係には与えられた二つの整数が等しいか、それともどちらかが大きいかということ、すなわち、大小関係という関係があり、この関係は有理数にも入れられます。ここで有理数に対して正規化を行っていれば常に分母は 0 以外の正整数することができるため ‘ $ad > bc$ ’ と ‘ $(a, b) > (c, d)$ ’ が同値になります。これらのこと踏まえて RationalNumber クラスを構築しましょう：

```
from PairOfInts import PairOfInts
class RationalNumber(PairOfInts):
    def __gcd__(self):
        __a = self.numer
        __b = self.denom
```

```
if __a == 0:
    if __b != 0:
        __b = 1
    elif __b == 0:
        __a = 1
    elif __a > __b:
        __d = __a / __b
        __r = __a - __d * __b
    else:
        __c = self.conv()
        return c.__gdc__()

def __cmp__(self, other):
    """有理数の合同性と大小関係を判別するメソッド

    """
    return cmp(self.numer * other.denom,
               self.denom * other.numer)

def conv(self):
    """逆元を返すメソッド

    """
    tmp = self.numer
    if tmp == 0:
        self.numer = 1
        self.denom = 0
    else:
        self.numer = self.denom
        self.denom = tmp

def rexpr(self):
    """有理数の正規化を行うメソッド

    """

```

```

if self.denom < 0:
    self.denom = - self.denom
    self.numer = - self.numer
if self.numer == 0:
    self.denom = 1
elif self.denom == 0:
    self.numer = 1
else:
    tmp = gcd(self.numer, self.denom)
    self.numer = self.numer / tmp
    self.denom = self.denom / tmp

```

class 節で class 新しいクラスの名前(既存のクラスの名前) で既存のクラスの利用による新しいクラスの定義が行えます。このクラス構築の手法のことを「**継承**」と呼びます。この例では最初に import 文で PairOfInts クラスを読み込み、次に RationalNumber クラスの定義を ‘class RationalNumber(PairOfInts):’ で開始し、RationalNumber クラスが PairOfInts クラスを継承することを宣言しています。ここで RatinalNumber クラスに整数対の正規化を行うために二つの整数の最大公約数を求めるメソッド __gcd__(), 逆数を求めるメソッド conv() を定義している他に大小関係を整数から引継ぐメソッドとしてあらかじめ用意された特殊メソッド __cmp__() を先程の ‘a/b > c/d’ を定めるために上書きする形で用いています。このメソッドを定義することで大小関係の二項演算子 “>”, “<” と等価 “==” がこのクラスでも使えるようになります*18.

ここで新しいクラスを定義するときに基になった PairOfInts クラスを「**基底クラス (base class)**」、継承する側の RationalNumber クラスを「**派生クラス (derived class)**」と呼びます。また、この継承関係を親子関係に例えたときに基底クラスを「**親クラス**」、派生クラスを「**子クラス**」、また、この継承関係を上下関係として捉えるときに基底クラスを「**スーパークラス (super class)**」、派生クラスを「**サブクラス (subclass)**」と呼びます。クラスは処理の対象が「**何であるか**」ということと「**どのようなものであるか**」を語るもので、そのまま範疇論の歴史的な議論が活用できます。そして、対象への理解が進めばより細かく分類されるということも理解されるでしょう。その結果、下層のサブクラスは上位のクラスよりもより一層、現実の事物に近いものとなる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になるためにより抽象的(普遍的)にな

*18 ここでの「等価」はオブジェクトの値について「等価」であるかどうかを判断するメソッドで、オブジェクトが同一であるかどうかを判断するメソッドではありません。

ります。

3.3.3 特殊メソッドによる四則演算の導入

さて、この RationalNumber クラスにまだ足りないものがあります。それは四則演算です。その四則演算を次で追加しましょう：

```
def __add__(self, other):
    """有理数の和を定義するメソッド

    """
    numer = self.numer * other.denom + \
            self.denom * other.numer
    denom = self.denom * other.denom
    c = RationalNumber(numer, denom)
    c.reexpr()
    return

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

    """
    numer = self.numer * other.denom - \
            self.denom * other.numer
    denom = self.denom * other.denom
    c = RationalNumber(numer, denom)
    c.reexpr()
    return

def __mul__(self, other):
    """有理数の積を定義するメソッド

    """
    numer = self.numer * other.numer
    denom = self.denom * other.denom
    return RationalNumber(numer, denom)
```

```

def __truediv__(self, other):
    """有理数の商を定義するメソッド

    """
    numer = self.numer * other.denom
    denom = self.denom * other.numer
    return RationalNumber(numer, denom)

```

これらのメソッドも特殊メソッドで、最初のメソッド`__add__()`が和演算子“+”に対応するメソッド、次のメソッド`__sub__()`が差演算子“-”に対応するメソッド、それからメソッド`__mul__()`が積演算子“*”に対応するメソッドで最後のメソッド`__truediv__()`が商演算子“/”に対応するメソッドです。ところで、四則演算は同じクラスの二つのオブジェクトに対する二項演算で、四則演算を表現するメソッドの引数にオブジェクト自体を参照することを意味する変数`self`に加え、同じクラスのインスタンスを参照を参照することを示す変数`other`があることに注意して下さい。

なお、ここで行った有理数の定義は非常に大雑把なものです。実際、有理数を整数対として表現し、それらの四則演算を入れましたが、その代数的構造（分配律、結合率等）や順序関係について何も語っていないからです。比較については四則演算のように比較のための特殊メソッドがありますが代数的構造はそう簡単に導入できません。しかし、SageMathにそのような代数的構造を有するクラスが存在しているため、表現しようとする数学的对象に最も類似するクラスを継承すればより効率的に表現することができます。このことは SageMath を用いる大きな利点になります。

簡単な紹介はここまでにして、次の節から「Python 言語リファレンス」を基にした Python の解説を行います。ただ、その前に Python の構文を表現するために BNF 記法を紹介しておきます。

3.4 Backus-Naur 記法 (BNF)

これから参照する「Python 言語リファレンス」では「**Backus-Naur 記法 (BNF)**」と呼ばれる表記を拡張した「**EBNF(Extended BNF)**」を用いて Python の構文の解説が行われています。この BNF (Backus-Naur Form) は John Backus がプログラム言語 Algol の文法の説明で用いた記法を Peter Naur が改良したものが基になっており、その構文規則は次で与えられます：

Backus-Naur 記法の構文規則

非終端記号 ::= 定義₁ | 定義₂ | ... | 定義_n

この式の意味は演算子 “::=” 左辺の非終端記号が右辺にある 定義_{i ∈ {1, ..., n}} の何れか一つの定義で k 意味します。したがって、生成規則が一通りだけであれば右辺は 定義₁ のみとなるために記号 “|” は不要で、また、定義₁ | 定義₂ であれば 定義₁ でなければ 定義₂ の構成になることを意味します。

ここで構文規則の左辺にある「**非終端記号**」とは何でしょうか？この「**非終端記号 (Nonterminal Symbol)**」は「**形式文法 (Formal Grammar)**」で用いられている言葉で、演算子 “::=” の右辺の各定義に従って変化が生じ得る記号です。この記号は変数と同様の働きをする記号になるので「**構文変数**」とも呼ばれます。非終端記号があれば終端記号もあり、この「**終端記号**」は演算子 “::=” の左辺の定義として現われる生成文法に従ったときに、それ以上の変化が生じない定数のような記号です。この終端記号はその性質から BNF では演算子 “::=” の右辺のみに現われます。なお、本来の BNF では非終端記号を記号 “<... >” を使って **<非終端記号 >** と括りますが、この表記は非終端記号の区別を明瞭にする程度の働きしかしないため、ここでは用いません。

さて BNF の簡単な実例を示しておきましょう。まず「**数字**」は ASCII 文字の “0” から “9” です。これを BNF で表記するなら

BNF による数字の定義

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

になります。ここで ASCII 文字の “0” から “9” が終端記号であることに問題ないでしょう。実際、“0” から “9” の数字は数字を構成する上での素材になっています。それでは自然数の BNF はどうなるでしょうか？この場合は複数の非終端記号の定義行を組合せた表現になります：

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”

この自然数の定義はさきほどの数字の定義と異なり帰納的な定義を含みます。まず、「**自然数**」は 0 から 9 までの「**数字**」か「**0 以外の数字**」と「**自然数**」を左から順番に並べて構成されるものと第一行で記述されています。この定義は演算子 “::=” の左右に「**自然数**」

が現われる帰納的な定義です。ここで「**数字**」は第二行目で「“0”」または「**0以外の数字**」から構成され、最後の行で「**0以外の数字**」は“1”, “2”から“9”までの数字と記述されています。

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BNFによる「**命題論理式の定義**」を次に示します：

命題論理式の定義

- (1) 真理値の真 \top は命題論理式である。
- (2) 真理値の偽 \perp は命題論理式である。
- (3) 論理記号 P は命題論理式である。
- (4) A, B が命題論理式であれば $\neg A, A \wedge B, A \vee B, A \rightarrow B$ も命題論理式である。
- (5) 上の (1), (2), (3), (4) で構成されたもののみが命題論理式である。

ここで定義(4)の ' $\neg A$ ' は命題論理式 ' A ' の否定, ' $A \wedge B$ ' は命題論理式 ' A ' と ' B ' の論理積を取る操作, ' $A \vee B$ ' は命題論理式 ' A ' と ' B ' の論理和を取る操作で, 最後の ' $A \rightarrow B$ ' は論理式の含意 (A ならば B) を取る操作に対応します。ここで(5)に帰納的な定義(既存の要素を使って新たな命題を生成する手順)が入っていることに注目して下さい。では、この定義をBNFで書換えてみましょう：

命題論理式のBNF

命題論理式	$::=$	$\top \mid \perp \mid$ 論理記号 $\mid \neg$ 命題論理式 \mid
\quad 命題論理式 \wedge 命題論理式 \mid 命題論理式 \vee 命題論理式 \mid		
\quad 命題論理式 \rightarrow 命題論理式		

こここのBNFで、記号“::=”の右辺から順番に命題論理式の(1)から(4)が現われています。そして、命題論理式の定義の(5)の帰納的な論理式の定義はBNFの構造で表現されています。このように構文の定義にもBNFは使えます。

なお、「Python言語マニュアル」のBNFは正規表現を追加した「**拡張BNF(EBNF)**」を採用しています。以下に拡張の要旨を纏めておきます：

EBNF の特徴

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

以下に EBNF の具体例を示しておきましょう:

整数の EBNF

整数 ::= ["-"] 自然数

これは整数の構成を EBNF で書換えたもので、角括弧 “[]” を使った例になります。ここでの角括弧 “[]” はあってもなくてもよいオプションに相当する部位の表記になります。また、自然数が定義されているという暗黙の仮定がありますが、この EBNF は整数は自然数、あるいは自然数の頭に記号 “-” を追加したものという字句的な定義です。こうすることで整数の表記上の定義がより簡潔で明晰なものへと置換えられています。

もう一つの例を示しておきましょう。「Python 言語リファレンス」では Python の「**名前 (name)**」という非終端記号の EBNF を以下で表現しています:

Python での EBNF の実例

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

このように Python の EBNF では文字を二重引用符 ("") で括るために文字を “_”, “a”, “z” と二重引用符で括った表記にして一行目で非終端記号 name がアルファベットの小文字: lc_letter と記号 “_” で構成されることを示します。ここで ‘(lc_letter | "_"")*’ は正規表現で ‘(...)*’ によって括弧を使ってグループ化を行い、さらに括弧内の表記が0回以上出現するという意味を持たせています。このことから name は lc_letter に対応する文字の羅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ということを意味します。では、lc_letter は何でしょうか？これを定義しているのが二行目で lc_letter が Python の文字 “a” から “z” までの小文字で構成されることを

記号 “...” を用いて表記しています。ここで表記は正規表現から外れた表記で、通常の正規表現であれば ‘a-z’ となるところです。このように「Python 言語リファレンス」の EBNF では記号 “...” が正規表現の “.” に対応します。この BNF から非終端記号 name は ‘a’, ‘a_bc’ のように頭文字がアルファベットの小文字、以後はアルファベットの小文字や記号 “_” で構成された文字列であって、‘_a’ のように先頭がアルファベットでない文字列、具体的には ‘a1’ や ‘A_v0.1’ のように BNF に記述のない文字が入った文字列は Python の名前に適合しないことが判ります。この BNF を利用することで Python の構文がより明瞭に表記されます。

3.5 字句解析について

3.5.1 トークン (token)

Python はプログラム入力時に構文解析器 (parser) によるプログラムの字句解析でプログラムをトークンの列に変換します。ここで「トークン (token)」はプログラム内で意味を持つコードの最小単位で、自然言語の「語彙素」に相当します。Python のトークンには「NEWLINE」、「INDENT」と「DEDENT」の他に「識別子」、「キーワード」、「リテラル」と「演算子」があります。ここで NEWLINE は後述の論理行の区切になるトークン、INDENT, DEDENT は INDENT-DEDENT のトークン対で後述の「複合文」で用いられて Python のコードの大きな特徴である字下げに関わります。また、「空白文字 (blank character)」^{*19}とよばれる文字の中で Space, TAB や FF にはトークンを区切る作用があります。たとえば ‘ab’ と文字の間に欧文間隔 Space を入れた ‘a b’ は前者が ‘ab’ の一つのトークン、後者が ‘a’ と ‘b’ の二つのトークンです。

3.5.2 行構造

プログラムは字句解析によってトークンの列に変換され、それからトークン列を NEWLINE を区切として分割することでプログラムは複数の論理行に、論理行はさらに物理行に分解されます。

■論理行 (logical line): 先頭に「字下げ/インデント (indentation)」と呼ばれる ASCII 文字の Space や TAB による空白文字の列の末端に NEWLINE を持つトークン列です。

^{*19} 空白文字と呼ばれる ASCII 文字に水平タブ (TAB), 垂直タブ (VT), 改行 (LF/NL), 改ページ (FF), 行頭復帰 (CR), 欧文間隔 (Space) の 6 文字があり、Python ではメソッド isspace() で空白文字かどうか判断できます。なお、空白文字には各言語の間隔文字もありますが、ここでの空白文字と区別します。

■物理行 (physical line): 論理行をプログラムの記述の状態から分割したもので、行末端文字で区切られている文字の列です。ここでの行末端文字は計算機環境で異なり、UNIX 環境で ASCII 文字の **LF(行送り)**、Windows 環境では **CR+LF**、MacOS で **CR(復帰)**ですが、Python のプログラムに物理行を埋め込むときは計算機環境を問わず C と同様に行末端文字として ASCII 文字の **LF** に対応する文字 ‘\n’^{*20}を用います。また、特殊な物理行に「**注釈**」、「**エンコード宣言**」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^{*21}。なお、注釈は論理行を終らせる働きがあります。そのために後述の行継続を行うときに注意が必要です。
- **エンコード宣言:** PEP-263 に規定があり、注釈と同様に記号 “#” から開始して物理行の行末端文字で終える論理行で、プログラムの先頭の一行目か二行目に置かれて正規表現 ‘coding[=:]\s*([-\\w.]+)’ に適合します。たとえば文字コードを UTF-8 に設定するのであれば ‘# coding = UTF-8’ や ‘# coding : UTF-8’ といった宣言行が該当します。ちなみに前者が GNU Emacs のエンコード設定書式、後者が vim のエンコード設定に適合します^{*22}。このエンコード宣言によってプログラムで用いられる文字リテラルを構成する文字がどの言語のどのエンコーディングであるかが明示的に指定されますが、この宣言がなければプログラムに記載された文字はエンコーディングを指定する接頭辞を持たない限り ASCII 文字として扱われます。
- **空行 (blank line):** 空白文字の **Space**, **TAB**, **FF(改ページ, Form Feed)**, あるいは注釈だけで構成された論理行です。これらの空行に字句解析で **NEWLINE** が生成されません。そして、これらの空行はプログラムの内容的な区切になりますが、それ以上の意味は持ち得ません。

■字下げ/インデント (indentation): 論理行の先頭に置かれた空白文字の **Space**, **TAB** の個数で字下げの水準が計算され、この水準で入力文が纏められます。プログラミング様式を規定する PEP-8 で字下げは一段 4 個の Space のみで TAB を混在させないことが推奨されています。実際、これらが混在していると両者の判別が容易でなく、ノートブック形式のユーザインターフェイスで TAB に入力補完機能を持たせることもあるため、Space のみで間隔を開けるように統一することが現実的です。

^{*20} 日本語 Windows で用いられている文字コード SHIFT_JIS(その実装の CP932) で記号 “＼” が勝手に記号 “¥” で置換えられています。そのために多くの書籍でこれらの記号を同一視していますが、UTF-8 等の文字コードで全く別の記号のため、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 MS-Windows 環境では適宜、記号 “＼” を記号 “¥” で読み直して下さい。

^{*21} 記号 “#” は後述のリテラルには含まれていません。

^{*22} vim は vi から派生したエディタで、GNU Emacs と vi はいわゆる Editor War の二大陣営です。

■物理行の明示的/非明示的な分割: 注釈とエンコード宣言を除く物理行を複数の物理行に置換できます。物理行の分割を明示的に行うときは行の継続を示す継続文字として記号“\”を物理行の末尾(行末端文字の直前)に置きます。このときにPythonの構文解析器は継続文字の直後の行末文字を削除して一つの物理行に変換します。ただし、継続文字“\”に続けて註釈を追記することができません。なぜなら、註釈は論理行を終える働きがあるために註釈の前後で二つの論理行に分割されるからです。また、註釈自体も継続文字を使って分割することができません。ここで改行の例外的な規則として丸括弧“()”，角括弧“[]”，波括弧“{ }”の内部で改行を行った際に継続行文字“\”の併用を必要としません。同時にこれらの記号で括られたその内部で註釈を続けられます。

3.5.3 識別子とキーワード

「**識別子(identifier)**」は「**名前(name)**」に用いられます。そして、オブジェクトへの参照はこの名前を介して行われます。識別子のEBNFは次のとおりです：

——— 識別子の EBNF ———

```

    識別子 ::= (文字 | "_")(文字 | 数字 | "_")*
    文字    ::= 小文字 | 大文字
    小文字  ::= "a" ... "z"
    大文字  ::= "A" ... "Z"
    数字   ::= "0" ... "9"

```

識別子はアルファベットと数字、それと記号“_”のみで構成されたASCII文字の列で、日本語の“**三毛猫**”やいわゆる全角文字の“**A B C**”といったASCII文字以外の文字は上述の識別子のEBNFの文字に該当しないため、これらの文字を含む文字の列はエンコード宣言の有無と無関係に識別子になりません²³。ただし、次に示す「**キーワード**」は予約語になって識別子として使えません：

——— キーワード ———

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	exec	global	is	pass	try	

²³ Python 3.x では文字コードとして UNICODE が採用された結果、漢字、ギリシャ文字、キリル文字等の非 ASCII 文字も識別子として使えます。詳細は PEP-3131 Supporting Non-ASCII Identifiers を参照。

ここで挙げたキーワードは Python の文や式の構成で用いられる特別な識別子で、これらのキーワードを除いた識別子が「**名前**」として使えます。Python では名前によってオブジェクトとの対応付けが行われ、オブジェクトへの参照は名前を介して行われます。この名前とオブジェクトとの対応付けは「**名前空間**」と呼ばれ、具体的には Python の連想配列である「**辞書 (dictionary)**」として表現されています。ここで何らのオブジェクトとの対応関係がない名前で参照を行おうとすれば「**NameError 例外**」が送出されます。

3.5.4 リテラル

「リテラル (literal)」は「**文字どおり**」、「**字義どおり**」を意味する言葉です。記号論理学で用いられるリテラルは命題記号 (原子論理式) や命題記号の否定といった論理式を構成する上で根本となる要素を指し、プログラミングのリテラルはソースコード内部で定数値となる文字列や数値といった値の記述を指します。ここで Python のリテラルは「**文字列リテラル**」と「**数リテラル**」の二種類に大きく分類できます：

リテラルの分類

リテラル ::= 文字列リテラル | 数リテラル

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

文字列リテラル

Python の文字列リテラルの EBNF を以下に示します：

文字列リテラルの EBNF

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

「**文字列リテラル**」はプログラムにエンコード宣言が含まれず、リテラル自体にも接頭辞がなければ ASCII 文字として解釈されます。逆に文字列に接頭辞を用いることで、その文字列リテラルにエンコードを明示的に指示することができます。たとえば文字列の先頭に “u” や “U” といった接頭辞を配置すると直後の文字列が UNICODE 文字列型として扱われます。ただし、文字列のエンコーディングを指定する際に接頭辞と後続する 文字列の間に空白文字を入れてはいけません^{*24}。

それから文字列には引用符の個数による型の区別があります。Python の引用符には单引用符 (') と二重引用符 ("") の二種類があり、文字列はこのどちらかの引用符で対として括られなければなりません。そのために "abc 'def' hij" のように文字列リテラル内部に文字列リテラル全体を括る引用符対と別の引用符対で文字列リテラルを括ることができます。そして「**短文字列 (shortstring)**」は一重に引用符で括られた文字の羅列の型、「**長文字列 (longstring)**」は三重に引用符で括られた文字の羅列の型です。具体的には「**三毛猫**」や「**"虎猫"**」が短文字列の例、「**''' 三毛猫'''**」と「**"""虎猫"""**」が長文字列の例です。なお、長文字列からは後述の「**文書文字列 (docstring)**」が構成されますが、この文書文字列はオンラインマニュアルとしての性格を持ちます。この文書文字列の中では改行や单引用符 (') や二重引用符 ("") を入れることも可能で、このことを利用して例題等を含む長い文字列を記述することができます。このときに注意することは長文字列を構成する際に用いた引用符と同じ引用符を三回連続して配置することで三重引用符の対が構成さ

^{*24} Python 3.x では文字列リテラルのエンコーディングが UNICODE になったために接頭辞の扱いが Python 2.x と異なります。

れた時点で長文字列が終了することです:

```
"""だからこんな使い方  
""やこんな使い方  
'それに改行をこんな風に複数入れたり引用符も長文字列を構成する際に用いた  
''引用符でなければ続けて三回使っても構いません  
''といったことを記入しても構わないのです  
. .  
'''
```

と、この例のように単引用符や二重引用符、それと行末端文字を含む文字列は一つの長文字列になります。なお、プログラムにて改行の出力を表現するときは C と同様に、出力したい文字に対応する文字列リテラル内にて改行コードの箇所を文字リテラル “\n” で置き換えることで出力文字の改行が行えます。この長文字列を reST の書式で記述することで Python のプログラムの文書としての性格を向上させることができます。

最後に「エスケープシーケンス」は通常の ASCII 文字では表現できない特殊文字や機能を Python で表現するための文字の並びです:

エスケープシーケンス

\\"	文字 “\”
\'	文字 (')
\"	文字 (")
\a	ベル (BEL)
\b	バックスペース (BS)
\f	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

ここで示すように Python 2.x のエスケープシーケンスは接頭辞に ‘r’ や ‘R’ が含まれていなければ C と同様の表記が使えます。ちなみに接頭辞 ‘r’, ‘R’ がある文字列はこれらの接頭辞に続く文字列に含まれる記号 “\” をエスケープシーケンスの一部ではなく、単なる文字として処理することを意味します。ここで UNICODE 文字名は Unicode Character Database^{*25}に Unicode 文字に対して記載された名前のこと、ASCII 文字で記載されています。たとえば文字 “[” の Unicode 符号点は 005B で、Unicode 文字名は ‘LEFT SQUARE BRACKET’ になります。

^{*25} <http://www.unicode.org/Public/UCD/latest/charts/CodeCharts.pdf> を参照。

数リテラル

数リテラルの EBNF を以下に示します:

—— 数リテラルの EBNF ——

```
数リテラル ::= 整数リテラル | 長整数リテラル | 浮動小数点数リテラル | 虚  
数リテラル
```

この EBNF で示すように数リテラルには「**整数 (plain integer) リテラル**」, 「**長整数 (long integer) リテラル**」, 「**浮動小数点数リテラル**」と「**虚数リテラル**」の 4 種類があります。ここで整数リテラルと長整数リテラルが整数の表現で、整数リテラルが符号付き 32bit 整数^{*26}, 長整数は計算機の記憶容量に依存するものの任意桁数の整数を表現します^{*27}。そして浮動小数点数リテラルが実数の表現であり,. なお、複素数は浮動小数点数リテラルと虚数リテラルの和という「式」で表現されるために数リテラルに該当しません。

以下に整数リテラルと長整数リテラルの EBNF を示します:

—— 整数リテラルと長整数リテラルの EBNF ——

```
長整数リテラル ::= 整数リテラル ("l"|"L")
整数リテラル ::= 10 進表示 | 8 進数表示 | 16 進数表示 | 2 進数表示
10 進表示 ::= 零以外の数字 数字* | "0"
8 進数表示 ::= "0" ("o"|"O") 8 進数 + | "0" | 8 進数 +
16 進数表示 ::= "0" ("x" | "X") 16 進数 +
2 進数表示 ::= "0" ("b" | "B") 2 進数 +
数字 ::= "0" | 零以外の数字
零以外の数字 ::= "1" ... "9"
8 進数 ::= "0" ... "7"
16 進数 ::= 数字 | "a" ... "f" | "A" ... "F"
2 進数 ::= "0" | "1"
```

この EBNF で示すように Python の整数リテラルには 10 進数の他に 2 進数, 8 進数と 16 進数があります。そして長整数のリテラルでは末尾に “l” や “L” といった長整数であることを示す文字を置くことになっていますが、小文字 “l” は数字 “1” と非常に紛らわしい

^{*26} 土214743647 の範囲の数で、Python 2.x では変数 sys.maxint に整数型の上限としてこの値が束縛されています。ただし、Python 3.x では整数型が実質的に長整数型で統一されるために sys.maxint は廃止されています。

^{*27} Python 3.x の整数のリテラルは Python 2.x の 32bit 整数リテラルの書式に統一されています。

ために大文字“L”的使用が推奨されています。また、Pythonの長整数による処理は高速ではありません。そのためにSageMathではPythonの長整数よりも高速演算が可能なGMP(GNU Multiple Precision Arithmetic Library)を用いた整数クラスInteger^{*28}で整数リテラルが処理されます。

次の例^{*29}でSageMathのGMPに基づく整数がPython標準の長整数よりも高速に処理されることを確認しましょう。そのために次の函数をあらかじめ定義しておきます：

```
def factorial(n, stop=0):
    o = 1
    while n > stop:
        o *= n
        n -= 1
    return o

def choose(n, k):
    return factorial(n, stop=k) / factorial(n - k)
```

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

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

ここで函数time.clock()はプロセッサ処理時間を秒数で返す函数で、timeモジュールに包含されています。そのためあらかじめ‘import time’でtimeモジュール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に束縛されるので、この変数elapseの値で小さなほうがより高速であると判断することができます。この式の評価をSageMath Cloud(SMC)で確認すると、SMC上のPythonで1.943、SageMathで0.594という結果を得て圧倒的にSageMathの整数処理が高速であることがわかります。このようにSageMathにはGMP由来の整数が使われていますが、SageMath上でプログラムを構築するときに数リテラルそのままはPythonの数リテラルとして処理されるために注意が必要になります。これはPythonの数クラスとSageMathの数クラスのメソッドに違いがあるためです。たとえば式‘2^3’はSageMathでは2の3乗を意味し、その値は8になります。しかし、Pythonでは演算子“^”は排他的論理和であるために‘2^3’の値は1になります。

^{*28} 正確にはsage.rings.integer.Integerクラスで、CythonでGMPのmpz_t integer型のラッパーとして実装されています。

^{*29} <http://jasonstitt.com/c-extension-n-choose-k> の記事: GMP bignums vs. Python bignums: performance and code examples の例です

ります。そして、SageMath プログラム内部の数リテラル 2 と 3 は Python の整数として扱われるため ‘ 2^3 ’ は排他論理和の計算になります。もしも、幂乗の計算を期待しているのであれば ‘Integer(2)’, ‘Integer(3)’ で SageMath の整数クラスのインスタンスとして処理しなければなりません。

次に浮動小数点数リテラルの EBNF を示します:

— 浮動小数点数リテラルの EBNF —

浮動小数点数リテラル	$::=$	小数表 指数表示
小数表示	$::=$	[10 進表示] ‘.’ 10 進表示 10 進表示 ‘.’
指数表示	$::=$	(10 表示 小数表示) 指数部
指数部	$::=$	(“e” “E”) [“+” “-”] 10 進表示 +

ここで浮動小数点数リテラルには符号を含みません。なぜなら符号は演算結果として得られるもの、つまり、式だからです。この点は次に言及する複素数でも同様です。

最後に虚数リテラルの EBNF を示しておきます:

— 虚数リテラルの EBNF —

虚数リテラル	$::=$	(浮動小数点数リテラル 10 進表示) (“j” “J”)
--------	-------	-----------------------------------

この虚数リテラルは「**浮動小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ることになります。実際、SageMath では代数的数の純虚数 $\sqrt{-1}$ に対しては I, あるいは i が定義されています。この Python の虚数リテラル 1J は浮動小数点数を基にしているために SageMath の代数的数の I と異なるので注意が必要です。このことを SageMath で確認しておきましょう:

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

函数 type() で I の型を調べると ‘sage.symbolic.expression.Expression’ と結果が返されることで I が多項式の仲間の数であることが判ります。ここで純虚数 i が多項式 $x^2 + 1$

の零点となる数であることは良いでしょう。代数学ではこのように整数係数の多項式の零点になる数のことを「**代数的数**」と呼び、その数を零点として持つ最少多項式と呼ばれる最小次数の1変数多項式に対応させることができます。純虚数の場合は $x^2 + 1$ がその最小多項式になります。それから1変数の整数係数の多項式全体、つまり、整数係数の多項式環 $\mathbb{Z}[x]$ に $x^2 + 1 = 0$ という関係を入れてできた世界で変数 x を単純に i と置いてしまえば $\{a + bi : a, b \in \mathbb{Z}\} (\subset \mathbb{C})$ になることが容易に理解できます。これが SageMath の I が多項式の仲間になる理由です。ところでもう一方の 1J を函数 type() で調べると ‘sage.rings.complex_number.ComplexNumber’ になって先程の I と型が異なります。この虚数リテラルは近似値である浮動小数点数から構成するために虚数リテラル自身も近似値としての性格を持ちます。そのために SageMath で近似的な数として虚数リテラルを用いるのか、代数的数である純虚数を用いるかを計算目的に応じて使い分ける必要があります。

3.5.5 演算子と区切文字

以下のトークンは Python 組込の演算子として用いられます:

Python 組込の演算子

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

なお、演算子 “ $<>$ ” と演算子 “ $!=$ ” は同じ意味ですが、演算子 “ $<>$ ” は時代遅れの表記のために演算子 “ $!=$ ” の利用が推奨されています。

ここで演算子 “ $^$ ” は多くの数式処理システムでは冪乗の演算子として扱われ、SageMath では多項式や数を表現するクラスで演算子 “ $^$ ” が冪乗の演算子として上書きされ、数や多項式の入力は自動的に整数環や多項式環等のその時点で扱っている数学対象のクラスのインスタンスとして処理されます。そのために対話処理を使っているだけでは演算子 “ $^$ ” が冪乗の演算子として書き換えられたように見えます。しかし、SageMath 上のプログラム内部の数リテラルは SageMath の整数環等のクラスのインスタンスとして明示的に指示しない限り Python の数リテラルとして解釈されるために、それらのインスタンスに対して演算子 “ $^$ ” は冪乗の演算子ではなく本来の XOR 演算子になります。このように SageMath の対話的処理で問題がなく動作しても、同じ処理を SageMath のプログラムに記載するとメソッドがないといったエラーが生じたり、期待した結果と異なるときは該当するインスタンスがどのオブジェクトに属しているかを確認しましょう。

次のトークンは「**区切文字 (delimiter)**」として用いられます:

Python の区切文字 (delimiter)

括弧:	()	[]	{	}	@
区切記号:	,	:	.	'	=	;	
累算算術代入演算記号 (1):	+=	-=	*=	/=	//=	%=	
累算算術代入演算記号 (2):	&=	=	^=	>>=	<<=	**=	

表の上段は括弧の類で、中段のコンマ “,”、コロン “:” は後述のスライス処理と呼ばれる添字を用いた処理で使われます。そして、下段の累算算術代入演算子は区切文字としても振舞います。また、单引用符、二重引用符、記号 “#” と記号 “\” といった ASCII 文字は他のトークンの一部に用いられて特殊な意味を持ちます。最後に “\$” と “?” は Python では用いられず、文字リテラルと注釈以外に現われたときは無条件にエラーになります。ただし、Python のシェルである IPython やその後継の Jupyter では記号 “?” に函数 help() を拡張した演算子としての働きを持たせているためにエラーになりません。

3.6 オブジェクトについて

3.6.1 オブジェクトの概要

ここでは Python のオブジェクトについて概要を述べます。なお、オブジェクトの生成に関する EBNF は §3.13 で詳細を述べることにします。まず、Python で扱う対象は全てオブジェクトであり、オブジェクトは「識別値 (identity)」、「型 (Type)」と「値 (Value)」をその属性として持ります。ここで最初の「識別値」はオブジェクト生成時に定まり、変更ができないオブジェクト固有の値です。具体的にはオブジェクトのメモリ上の番地に対応する整数値として表現されます。「型」はオブジェクトの生成時に指示され、こちらも変更できません。そして型 (type) 自体もまた型です。オブジェクトはその持つ得る値に関して、その値が「変更可能 (mutable)」であるオブジェクトと「変更不可能 (immutable)」であるオブジェクトに分類することができます。ただし、この変更不可能なものでも別のオブジェクトへの参照を持つコンテナと呼ばれるオブジェクトであれば、そのオブジェクトの参照先を変更することで実質的に変更可能になりますが、オブジェクトの構造自体をえるものではありません。そのために変更不可能なオブジェクトでは、その生成時に一定のメモリを割り当てることができます。さらに組込の変更不可能なオブジェクトは全て要約可能という性質を持ります。ここでオブジェクトが「要約 (ハッシュ) 可能, hashable」であるとは、そのオブジェクトが生成されて、それが存在している間に一定の整数型の値、すなわち、「要約値 (hash value)」を持つときです。この要約値の重要な性質は、二つのオブジェクトが同じ値を持っているときに一致するという性質です。ただし、要約値が一致することとオブジェクトの識別値が一致することは別です。要約値

はオブジェクトの持つ値が等しいことをオブジェクトの詳細を調べなくても判別できるようになる値で、このオブジェクトの要約値を計算するためのメソッドは`__hash__()`です。なお、要約不可能なオブジェクトでは`__hash__`にオブジェクト`None`が割り当てられています。これは`None`の説明でも行いますが、`None`は空集合`Ø`に対応するために空集合`Ø`を始域に持つ写像も空集合`Ø`になるという事実に符合します。

オブジェクトの参照で「**名前空間**」が使われます。Python の名前空間はオブジェクトと識別子の間に対応関係を与える仕組で、名前空間で扱われる識別子を「**名前**」と呼び、オブジェクトの識別値、型、値の参照は名前を介して行います。たとえば、オブジェクト固有の値である識別値の参照は名前を引数に函数`id()`で、オブジェクトの型の参照はその名前を引数にして函数`type()`を使って調べることができます。また、異なる二つの名前で指示されるオブジェクトの同一性の判断は演算子“`is`”に二つの名前を引き渡すことで行えます。このように名前はオブジェクトに取り付ける名札と言え、オブジェクトへのさまざまな参照は名前を介して得られます。この名前空間の実装は Python の連想配列である辞書と呼ばれる型のオブジェクトが用いられています。

Python ではオブジェクトの生成や破棄でメモリの割当や解放が自動的に行われます。そして、不要になったオブジェクトを利用者が直接、破棄できません。オブジェクトが破棄されるためには、オブジェクトと名前との関係が途切れた状態、すなわち「**到達不能の状態**」になることが必要です。この到着不能の状態でオブジェクトの破棄が許可され、やがて「**GC(塵回収, garbage collection)**」によるオブジェクトの破棄が行われます。この GC の実行や停止を利用者が行えますが、通常はオブジェクト総数があらかじめ設定された閾値を越えた時点で GC が自動的に実行されます。

3.6.2 オブジェクトの生成と廃棄を行うメソッド

オブジェクト指向プログラミング言語では、オブジェクトの生成を行う函数やメソッドを「**構築子/コンストラクタ (constructor)**」と呼びます。このオブジェクトの生成ではオブジェクトへの「**メモリの割当 (allocation)**」とオブジェクトの属性の設定等の「**初期化 (initialization)**」が同時に行われます。逆にオブジェクトを消去するときに呼び出される函数やメソッドのことを「**消滅子/デストラクタ (destructor)**」^{*30}と呼びます。

Python では、構築子に相当するメソッドに`__new__()`と`__init__()`の二つのメソッドがあり、メソッド`__new__()`でオブジェクトの生成、メソッド`__init__()`でオ

^{*30} Java ではファイナライザ (finalizer) と呼びます。

オブジェクトの初期化を行います。オブジェクトの生成でメソッド `__new__()` が呼び出されると自動的にメソッド `__init__()` が呼び出されます。これらのメソッド二つで構築子としての機能を持ち、特にメソッド `__init__()` が C++ の構築子に最も類似した働きをしていますが厳密には異なります。なお、Python ではクラスの定義で構築子に対応するこれらの二つのメソッドを記載する必要はありません。これらのメソッドがクラスに記載されていなければより上位のクラスのメソッドが継承によって用いられます。

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

3.6.3 GC(garbage collection)

について

Python は前述のように「**GC(塵回収, garbage collection)**」がある時点で自動的に実行されて不要になったオブジェクトの破棄とメモリの解放を行います。ここでは Python の GC について概要を述べることにします。まず、CPython の GC に「**参照カウント方式 (reference counting)**」が採用されています。この方式は全てのオブジェクトに「**参照カウント**」と呼ばれる整数値を付与し、オブジェクトの生成時に参照カウントに 0 が設定され、オブジェクトがある名前に束縛されたり、他のオブジェクトからの参照があれば参照カウントが 1 増やされます。逆に、その名前に別のオブジェクトが束縛されることで参照関係が解消されると参照カウントが 1 減じられ、最終的に参照カウントが 0 になった時点でオブジェクトの破棄が許可されてメソッド `__del__()` が呼び出されます。なお、明示的な消滅子 (= フィル `del()`) の実行は、それによってオブジェクトが直接破棄されるのではなく、名前との参照関係を外さるために参照カウントが 1 減じるだけで、消滅子で指示されている処理は参照カウントが 0 になった時点で実行されます。なお、オブジェクトの参照カウントは `sys` モジュールの函数 `getrefcount()` で確認することができます。このときに函数 `getrefcount()` で調べた値は本来の参照カウントよりも +1 になります。これは函数 `getrefcount()` による参照も追加されるためです。ここで簡単な例で確認をしておきましょう：

```
>>> import gc  
>>> gc.set_debug(gc.DEBUG_STATS)
```

```
>>> class A():
...     def __del__(self):
...         print "Nyao"
...
>>> for i in range(5):
...     print i
...     a = A()
...     a = 1
...
0
Nyao
1
Nyao
2
Nyao
3
Nyao
4
Nyao
>>>
```

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

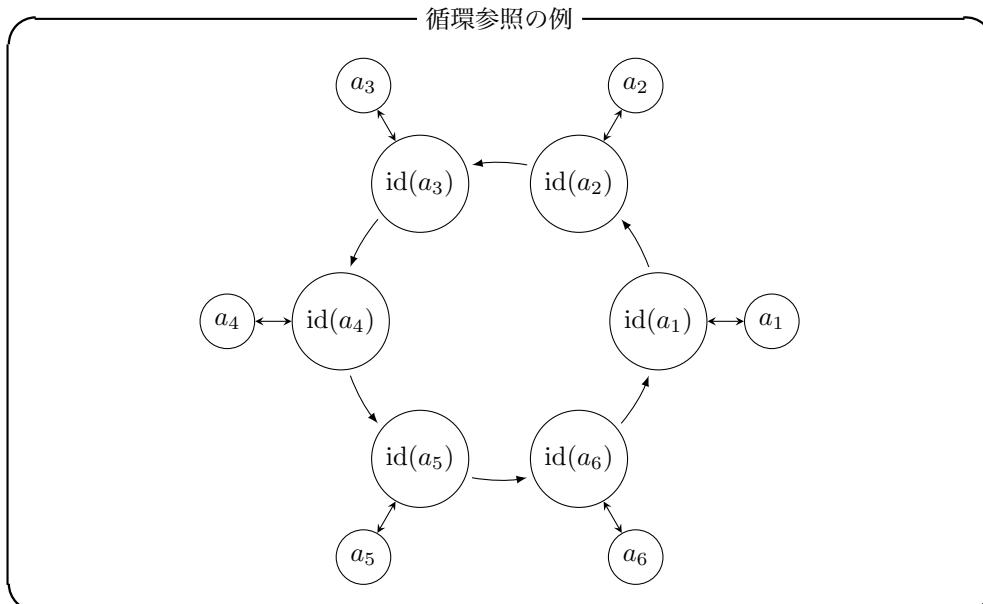
参照カウント方式では参照カウントが 0 になるとオブジェクトの破棄が許可されるだけで、実際にオブジェクトが破棄されるのはオブジェクトの総数が閾値に到達して GC が自動的に起動したときか利用者が `gc` モジュールの函数 `collector()` を起動したときです。この閾値は `gc` モジュールの函数 `get_threshold()` で確認することができます:

```
>>> import gc
>>> gc.set_debug(gc.DEBUG_STATS)
>>> gc.collect()
gc: collecting generation 2...
gc: objects in each generation: 249 3282 0
gc: done, 0.0017s elapsed.
0
>>> gc.get_threshold()
(700, 10, 10)
>>>
```

この例では `gc` モジュールを読み込み、`gc` 向けのデバッグ設定を行って函数 `collect()` でオブジェクトの回収を行っています。函数 `collect()` を実行すると、函数 `set_debug()` で統計量

の出力が設定されているためにその情報が表示されています。また、函数 `gc_threshold()` は GC を行う閾値を表示する函数で、ここで表示されたタブルの意味は、最初の 700 が第 0 世代、以降の 10, 10 が第 1, 第 2 世代の閾値となっています。この世代は到達可能なオブジェクトを乗り越えた GC の回数で分類したもので、最初に生成したオブジェクトを GC の 0 世代とし、第 1 世代が一度、GC を乗り越えた到達可能なオブジェクト、第 2 世代が二度以上 GC を乗り越えた到達可能なオブジェクトです。GC の自動実行は第 0 世代の総数が閾値を越えた時点で行われ、第 2, 第 1, 第 0 世代と古い順にオブジェクトの回収が行われます。

ところで、単純な参照カウント方式には大きな弱点があります。それは参照関係でオブジェクトを節点（ノード、node）、参照関係を線分（エッジ、edge）で置き換えることで得られるグラフで考えると明瞭になります。まず、得られたグラフに円環（=閉道）がなれば問題ありませんが、グラフに閉道ある場合に問題が生じます。たとえば $i \in \{1, \dots, 5\}$ のときにオブジェクト a_i が a_{i+1} を参照し、オブジェクト a_6 が a_1 を参照するといった自分の定義のために他者を参照し、その他者が最終的に自分を参照するために自分自身の参照が生じるときの参照関係をグラフ化すると以下のように円環が現れます：



この例では名前 $a_{i,i \in \{1, \dots, 6\}}$ に束縛されたオブジェクトを $\text{id}(a_i)$ と表記します。オブジェクトの表記に函数 `id()` を用いた理由は函数 `id()` で返却されるオブジェクトの識別値がオブジェクト固有の値になるからです。この例のように参照関係を辿ると自己に戻る参照のことを「**循環参照 (circular reference)**」と呼びます。この例のオブジェクトは名前を介して相互に参照があるために全てのオブジェクトと名前との参照関係が途絶えても

オブジェクト間の参照関係(内側の矢印)が残ります。このことは各オブジェクトの参照カウントが0にならないことを意味します。したがって、参照カウントが0のオブジェクトを回収する方法では、これらの到達不能のオブジェクトが回収できないことになります。

この弱点を克服する手段の一つがオブジェクトを世代別に分ける方法です。Pythonでは生成したばかりのオブジェクトを第0世代とします。そして、第0世代のオブジェクト総数が閾値を超えるか、gcモジュールの函数collect()を起動することでGCが開始されます。このGCではオブジェクトが世代毎に到達可能であるかどうかを調べます、このときに第0世代のオブジェクトで到達可能であれば第1世代、第1世代と第2世代のオブジェクトで到達可能なものは第2世代と世代の繰り上げを行います。その結果、到達不能なオブジェクトだけが残されるため、そのあとはこれらのオブジェクトを回収すればよいことになります。しかし、厄介なのはオブジェクトにメソッド__del__()が定義されている場合です。これは循環参照を行っているオブジェクトで、どちらのオブジェクトのメソッド__del__()を利用すべきかが判断できないためです。そのためPythonは循環参照に陥っている到達不能なオブジェクトでメソッド__del__()を持つものをGCで回収しません。

このことを例で確認しておきましょう。GCがどのように実行されているかを観察するためにgcモジュールの函数set_debug()を使って設定を行います。ここで設定しているのはgc.DEBUG_STATSで、統計情報を出力させるためです。まず最初に循環参照があってもメソッド__del__()を持たないクラスの場合です:

```
>>> import gc
>>> gc.set_debug(gc.DEBUG_STATS)
>>> class A():
...     pass
...
>>> for i in range(200):
...     print i
...     a = A()
...     b = A()
...     a.x = b
...     b.y = a
...
0
1
— 略 —
116
gc: collecting generation 0...
gc: objects in each generation: 717 3282 0
```

```
gc: done, 460 unreachable, 0 uncollectable, 0.0002s elapsed.  
117  
— 略 —  
199  
>>>  
>>> gc.collect()  
gc: collecting generation 2...  
gc: objects in each generation: 339 3496 0  
gc: done, 336 unreachable, 0 uncollectable, 0.0023s elapsed.  
336  
>>> gc.collect()  
gc: collecting generation 2...  
gc: objects in each generation: 4 0 3469  
gc: done, 0.0019s elapsed.  
0  
>>> gc.garbage  
[]  
>>>
```

閾値が 700 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 が行われますが、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るので手動で 2 回、関数 `collect()` を実行して GC を実行しています。ここで最初の GC の実行で 112、最後の GC の実行で 0 と表示されていますが、これは回収したオブジェクトの総数です。GC で回収できないオブジェクトの情報は `garbage` にリストとして蓄えられ、このリストの長さが回収不能なオブジェクトの個数に一致します。この例では `gc.garbage` を入力して空リストが返却されているために回収できないオブジェクトが発生していないことが分かります。したがって、ここでの循環参照は上手くオブジェクトの回収ができるので問題がありません。次にメソッド `__del__()` を有するクラスで同じことをしてみましょう：

```
>>> import gc  
>>> gc.set_debug(gc.DEBUG_STATS)  
>>> class A():  
...     def __del__(self):  
...         print "Nyao"  
...  
>>> class B():  
...     pass  
...  
>>> for i in range(200):  
...     print i  
...     a = A()
```

```
...     b = B()
...     a.x = b
...     b.y = a
...
0
1
2
— 略 —
114
gc: collecting generation 0...
gc: objects in each generation: 714 3282 0
gc: done, 452 unreachable, 452 uncollectable, 0.0001s elapsed.
115
— 略 —
198
199
>>> >>> gc.collect()
gc: collecting generation 2...
gc: objects in each generation: 347 3951 0
gc: done, 344 unreachable, 344 uncollectable, 0.0023s elapsed.
344
>>> len(gc.garbage)
199
```

この例は先程のクラス A にメソッド `__del__()` を追加し、もう一つのクラス B は消滅子を持たないクラスとして定義しています。ここでクラス A に追加したメソッド `__del__()` は文字列 'Nyao!' を表示するだけです。ところが、この場合は GC で到達不能のオブジェクトを認識しても、メソッド `__del__()` を持たないクラス B のインスタンスの回収は行っても、クラス A のインスタンスについては参照カウントを 0 にしてメソッド `__del__()` を呼び出して占有していたメモリを解放しません。その結果、不要になつたクラス A のインスタンスが溜る一方になります。このことは最後に函数 `collect()` を実行した後で `gc.garbage` のリスト長が 199 であることから判ります。

これらの例は GC がいつ起動するかその様子を見たかったために多くのオブジェクトを生成しています。最後の例として最初の a_1, \dots, a_6 の相互参照で構成される循環参照の例を挙げておきましょう：

```
>>> import gc
>>> gc.set_debug(gc.DEBUG_STATS)
>>> class A(object):
...     def __del__(self):
...         print "Nyao!"
```

```
...
>>> a1 = A()
>>> a2 = A()
>>> a3 = A()
>>> a4 = A()
>>> a5 = A()
>>> a6 = A()
>>> a2.x = a1
>>> a3.x = a2
>>> a4.x = a3
>>> a5.x = a4
>>> a6.x = a5
>>> a1.x = a6
>>> a1 = a2 = a3 = a4 = a5 = a6 = 1
>>> gc.collect()
gc: collecting generation 2...
gc: objects in each generation: 279 3282 0
gc: done, 12 unreachable, 12 uncollectable, 0.0018s elapsed.
12
>>> len(gc.garbage)
6
>>>
```

この例では函数 collect() で GC を実行しても円環を構築する 6 個のオブジェクトが残され、メソッド __del__() が実行されることもありません。また、この例では全ての名前に 1 を割り当てて参照関係を解消していますが、これを函数 del() で置き換えたとしても、名前とオブジェクトとの参照関係が切断されて参照カウントが 1 減じられるだけで、循環参照の問題である「どちらの消滅子を使えばよいのか?」のために、その肝心の処理が行われません。このように名前との参照関係を外しただけでメソッド __del__() の内容が必ず実行されるとは限らず、また、インタプリタを終了したときにメソッド __del__() が実行される保証もありません。そのためにファイル等の外部リソースを参照する場合はメソッド __del__() で解放するのではなく、適宜、close() 等の解放するためのメソッドを使ってリソースを開放することが妥当です。

3.6.4 オブジェクトの値

Python のオブジェクトは値が「**変更可能 (mutable)**」なものと「**変更不能 (immutable)**」ものの二種類に分類できます。この性質はオブジェクトの型で決定される性質で、変更することができません。

ここで簡単な例を示しておきましょう:

```
>>> a = b = []
>>> id(a)
3073670732L
>>> b.append(128)
>>> a is b
True
>>> id(a)
3073670732L
>>> id(b)
3073670732L
>>> c = [128]
>>> id(c)
3073670700L
```

この例では ‘`a = b = []`’ でリスト型のオブジェクト ‘`[]`’ を生成して名前 `a`, `b` に同時に束縛させています。そのため名前 `a` と `b` で参照されるオブジェクトが共通のためにその識別値が一致します。ところでオブジェクト ‘`[]`’ は変更可能なオブジェクトのために `b.append(128)` で名前 `b` で参照されているオブジェクトに 128 を追加できます。ところで、このオブジェクトを名前 `a`, `b` の双方で参照しているために名前 `a` で評価しても当然、128 が追加されています。このことは ‘`a is b`’^{*31}で調べても `True` が返却され、オブジェクト固有の識別値を返却する函数 `id()` で双方が同じ値になることからも判ります。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
```

^{*31} 二項演算子 “`is`” はオブジェクトが同一であるかどうか、二項演算子 “`==`” はオブジェクトの値が等しいか、どうかを判断するという違いがあります。

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。この時点では函数 `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’ のままになります。このことは識別値を函数 `id()` の結果からも判ります。ここでさらに名前 `d` に別のオブジェクトを束縛させると名前 `d` に束縛されていたオブジェクト 128 への参照が途切れますが、オブジェクト 128 への参照が名前 `c` と `d` 以外に存在しないのであれば、このオブジェクトへの参照が途切れた状態になります。この状態を「**到達不能の状態 (unreachable)**」と呼び、オブジェクトが到達不能な状態になると「**GC(塵回収, garbage-collection)**」でオブジェクトの回収処理が自動的に行われます。

Python のオブジェクトには他のオブジェクトへの参照を持つものがあります。このようなオブジェクトのことを「**コンテナ (container)**」と呼びます。Python のコンテナには「**集合**」、「**タプル**」、「**リスト**」と「**辞書**」があります。コンテナは値の変更不能な型であっても変更可能な型のオブジェクトへの参照が行われていれば参照先のオブジェクトの値の変更に伴ってコンテナの実質的な値が変化します。要するにコンテナはアパートみたいなものと思えばよく、住人が入替っても部屋番号はそのままのために部屋番号を介して住人への問合せができるが、コンテナの構造を変えることはアパートの改築に相当し、こちらは建物自体が旧来のものと異なって以前と同様と言えないでしょう。コンテナの例としてタプルとリストを使って違いを確認しておきましょう：

```
>>> a=[1]; b=[2]; c=[3]
>>> l1 = [a, b, c]
>>> t1 = (a, b, c)
>>> l1
[[1], [2], [3]]
>>> t1
([1], [2], [3])
>>> id(l1)
26005416
>>> id(t1)
25557832
>>> c.append(128)
>>> id(l1)
26005416
>>> id(t1)
```

```

25557832
>>> t1
([1], [2], [3, 128])
>>> l1
[[1], [2], [3, 128]]
>>> t1[2]=[128]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> l1[2]=[128]
>>> l1
[[1], [2], [128]]
>>> id(l1)
26005416
>>> c = [129]
>>> print l1, t1
[[1], [2], [128]] ([1], [2], [3, 128])

```

この例では三成分のリストとタプルを生成して中身の入れ替えを行っています。タプルは変更不可能のためにリストのように成分を直接変更できませんが、コンテナであるために参照しているオブジェクトの値が変更可能であれば、そのオブジェクトの値を変更することで結果として中身の変更ができます。ところで最後に名前 `c` にオブジェクト [129] を束縛させた場合はリストやタプルの側の旧来の参照が新しい参照に切り替わることがないために、そのまま名前 `c` に束縛されていたオブジェクト [3, 128] への参照が継続されます。つまり、オブジェクトの生成で名前を用いた場合、その名前に対応するオブジェクトが参照されることで新しいオブジェクトとの参照関係が発生しますが、名前はそのオブジェクトの名札であって入れ物でないため、名前に別のオブジェクトを束縛したからといって以前構築したオブジェクトが更新されることはありません。

3.6.5 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		

ここで `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 組込の定数にはこの他

にブール型の True, False と `__debug__` がありますが、これら None, NotImplemented と Ellipsis は条件文で True や False のいずれかと同じ働きをする真理値として扱うことができます。ただし、真理値と違って数オブジェクトでないために演算ができません。そして、数は後付けになりますが、抽象基底クラス (ABC, Abstract Base Class) で実装された numbers と前述のブール型を含みます。この抽象基底クラス numbers の具象クラスとして整数の int 型と long 型、実数の float 型、複素数の complex 型が表現されます^{*32}。数は Scheme の「**数値塔 (Numerical Tower)**」にしたがったもので、複素数に実数が、実数に整数が含まれるという状況を、複素数を基盤に実数を、実数を基盤に整数を載せた塔にたとえ、塔を逆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 §3.13 で述べます。

■None: LISP の nil に似た値を持つオブジェクトの型で、そのオブジェクトが意味のある値を持たないことを指示するために用います。None は組込の名前 None で参照され、その値は None そのもので if 文等の条件分岐で真理値 False として扱われますが False と同値ではありません。そして、この型を持つオブジェクトは None 以外に存在しません。このことを簡単な例で確認しておきましょう:

```
>>> def neko(x,y):
...     return None
...
>>> neko(1,2)
>>> zz = neko(1,2)
>>> zz
>>> type(zz)
<type 'NoneType'>
>>> xx=None
>>> zz is xx
True
```

この例では None を返す函数 neko() を定義していますが、この函数の返却値を名前 zz に割り当てています。同時に名前 xx にも None を割当てていますが、演算子 “is” で両者の同一性を調べると None 型を持つオブジェクトは一つのみが存在するために両者の識別子が一致し、そのため True が返却されます。このオブジェクト None の挙動は (小) 集合で構成される圈 Set にて空集合 \emptyset を始域とする矢 \emptyset と同様の性質です。実際、圈 Set の矢は通常の写像が対応し、 $a, b \in \text{Obj Set}$ に対して矢 $f : a \rightarrow b$ が存在したときに f を順序対の集合 $\{(x, f(x)) | x \in a\}$ と見なせます。ここで $a = \emptyset$ であれば $x \in \emptyset$ になる x が存在しないので $f = \emptyset$ となつて \emptyset は圈 Set の対象であると同時に矢にもなります。この

^{*32} PEP 3141 A Type Hierarchy for Numbers.

ようにオブジェクト `None` は空集合 \emptyset と同様の性質を持っています。

■`NotImplemented`: この型は单一の値しか持たず、この値を持つオブジェクトは `NotImplemented` のみです。条件文では真理値 `True` として扱われますが、`True` と同値ではありません。この `NotImplemented` の値は `NotImplemented` そのものです。

■`Ellipsis`: この型は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あり、`None` のように何もないことを意味する型と異なります。この型を持つオブジェクトは `Ellipsis` のみで、その値は `Ellipsis` そのものです。条件文で真理値 `True` と同じ働きをしますが `True` と同値ではありません。この `Ellipsis` を用いたスライスの例を以下に示しておきます：

```
>>> from numpy import arange
>>> a = arange(16).reshape(2,8)
>>> a
array([[ 0,  1,  2,  3,  4,  5,  6,  7],
       [ 8,  9, 10, 11, 12, 13, 14, 15]])
>>> a[0, ...]
array([0, 1, 2, 3, 4, 5, 6, 7])
>>> a[1, ...]
array([ 8,  9, 10, 11, 12, 13, 14, 15])
```

ここで示すスライス操作は配列要素の取り出し操作で MATLAB 系の言語でお馴染の添字操作です。例では最初に NumPy パッケージから函数 `arange()` の読みを行い、それから名前 `a` で参照される NumPy の一次元配列を生成してメソッド `reshape()` で 2×8 の配列へと大きさを変換しています。この配列は 2 次元で、話を簡単にするために 2×8 の行列として話を進めましょう。さて、それから二つの拡張スライス操作 ‘`a[0, ...]`’ と ‘`a[1, ...]`’ を行っていますが、これらの処理は行列の一行目と二行目の成分に相当する配列の取り出しを行っています。つまり：

$$a = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \end{pmatrix}$$

に対して

$$\begin{aligned} a[0, \dots] &\Rightarrow \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{pmatrix} \\ a[1, \dots] &\Rightarrow \begin{pmatrix} 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \end{pmatrix} \end{aligned}$$

という処理です。この拡張スライス操作に現われるリテラル ‘...’ は該当する添字の取り得る値の全てを意味する MATLAB の記号 ‘:’ に相当する省略記号で、この記号が `Ellipsis` です。ただし、Python や MATLAB の記号 ‘:’ は一次元の添字の省略記号に対し、この

Ellipsis は一次元に限定されず, Yorick の「ゴム添字 (rubber index)」のように多次元である点で大きく異なります^{*33}. この Ellipsis は明記された箇所以外の添字が配列の大きさに応じて「省略」されていることを示すオブジェクトで, 「何もない」ことを示す None とは違います. 実際, スライス操作で 1 次元的に記号 ":" を用いた操作で始点や終点を省略した表記が可能ですが, 実際は None が記載されたものと同値です:

```
>>> a = [0, 1, 2, 3, 4]
>>> a = [:4]
[0, 1, 2, 3]
>>> a = [None:4]
[0, 1, 2, 3]
```

この例の '[4]' という式と '[None:4]' という式が同じ意味であることから, None が Ellipse が意味する「省略」ではなく, 「何もない」ことを意味していることが明瞭になるかと思います.

なお, Ellipsis は Python 2.x 系では式や文を構成するオブジェクトで, 独立したオブジェクトではありません. 実際, Python 2.x のインタプリタ上で '...' のみを入力するとエラーになりますが, Python 3.x 系ではオブジェクトとしてリテラル '...' が許容され, Ellipsis 型のオブジェクトが生成されます.

■数 (numbers.Number): 数リテラルで生成されたり, 算術演算や組込の算術関数から返却されるオブジェクトです. 数の EBNF は数リテラルを参照してください. また, オブジェクトとしての数は変更不能のオブジェクトです. この数オブジェクトには整数, 浮動小数点数と複素数の型に分類することができます. ここで numbers.Number はモジュール numbers で定義されたクラス Number という意味で, このクラス Number を継承する形で整数, 実数と複素数が抽象基底クラスを用いて順序付けられています:

- 整数型 (plain integer) : 整数リテラルで生成され, 32bit 符号付き整数 (-214783648 から 2147483647 まで) が扱えます. そして, この型のオブジェクトの構築子は int() で, 関数 type() で調べると int が返却されます. なお, 演算結果や整数リテラルの入力が整数型の範囲を越えると自動的に後述の長整数型に切替えられます. このことは構築子 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生成されます^{*34}.

^{*33} Yorick は C に類似した言語で, 数値テンソル処理, 要するに高次元の配列処理を簡潔に行うために MATLAB 表記を使うという軽快な言語です.

^{*34} 数値処理言語によっては補数表現になるものがありますが, Python では自動的に長整数型に切り替わるために安全な言語といえるでしょう. なお, Python 3.x で整数型は長整数型に統合されます. なお, 抽象基底クラスの順序付で, このクラスは長整数型と共にクラス Integral の具象化クラスになります.

- 長整数型 (long integer): 長整数リテラルで生成され, 計算機のメモリに依存する任意桁数の整数が扱えます. そして, このクラスのオブジェクトの構築子は long() で, 函数 type() でこの型を調べると long が返却されます. なお, SageMath では通常の演算で用いる整数や任意精度の整数として Python 標準の整数型や長整数型ではなく GMP 由来の整数 (Integer) 型を用いています. この Integer 型では任意桁の整数も整数リテラルで末尾に “L” や “T” を付ける必要がありません. 逆に言えば SageMath で整数を扱うときは, それが Python の整数型で十分なのか, それとも SageMath の整数型 Integer でなくても良いものか注意を払ってください. 少なくとも数論関連の処理であれば SageMath の整数型 Integer にしなければ, 処理速度や使えるメソッドも含めて無意味なことになります. なお, 3.x では整数型は実質的に長整数型に統合されますが, そのリテラルは整数型のもので, 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() *³⁵ 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.
- 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. なお, 抽象基底クラスによる順序付では, このクラスはクラス Real の具象化クラスとされ, クラス Real はクラス Complex の子クラスになります.
- 複素数型 (numbers.Complex): 浮動小数点数リテラルと虚数リテラルを演算子 “+” で結合することで生成されます. 構築子は complex() で, 函数 type() で調べると complex が返却されます. また, 複素数 z の実部は z.real, 虚部は z.imag で取り出せますが, 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, そのことによって虚部が

*³⁵ 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.

0.0 の複素数を構築子 `int()`, `long()` や `float()` で変換することができません。なお、SageMath で多項式の型を導入しているために代数的数としての純虚数 `I`, `i` が虚数リテラルと別にあります。ここで代数的数は代数方程式の厳密解そのもので、浮動小数点数と違って近似の数ではありません。なお、抽象基底クラスによる順序付では、このクラスはクラス `Complex` の具象化クラスとされ、クラス `Complex` がクラス `Number` の子クラスになります。

■列 (sequence): 有限濃度の順序集合を表現する型です。ここで集合 S が順序集合とは、集合 S の各成分が順位を持ち、その順位に対して自然数の大小関係のような関係で比較できる集合です。より正確には次の順序関係を充す関係 “ \geq_{order} ”，すなわち「順序」を持つ集合 S のことです：

順序関係

反射律: $x \geq_{\text{order}} x \quad x \in S$

対称律: $x \geq_{\text{order}} y \quad \text{かつ} \quad y \geq_{\text{order}} x \quad \Rightarrow \quad x = y \quad x, y \in S$

推移律: $x \geq_{\text{order}} y \quad \text{かつ} \quad y \geq_{\text{order}} z \quad \Rightarrow \quad x \geq_{\text{order}} z \quad x, y, z \in S$

列の濃度、すなわち、列の長さは函数 `len()` を使って調べることができます。列 S の濃度が n であれば 0 から $n - 1$ までの n 個の自然数の集合を I とするとき、この I から列 S の成分への一対一写像が得られます。この写像を添字写像と呼びましょう。すると列の順序 “ \geq_{order} ” は添字写像の逆像から得られる自然数の大小関係で定めることができます。この様に列の順序は 0 から開始する整数と結び付けられ、列の成分の順序もこの整数で順序付られ、さらに列の成分の指定は ‘`a[2]`’ のように列の名前に対して括弧 “[]” に対応する整数を指定することで行えます。

列に対する特殊な操作に「スライス操作」と呼ばれる部分列の生成操作があります。この操作は MATLAB 系の言語でお馴染の操作で、長さ n の列 a に対して $i < j$ を充す二つの正整数 $i, j \in \{0, \dots, n - 1\}$ を添字として $a[i : j]$ で列 a の $i + 1$ 番目から j 番目の成分を持つ部分列を生成します。列の型によっては「拡張スライス操作」と呼ばれる刻幅指定のスライス操作が行えることもあります。たとえば、文字列 ‘123456789’ に対して刻幅 2 で先頭の文字から 8 番目までの文字で構成される部分列を取り出すときに ‘123456789’[0:8:2] で部分列 ‘1357’ を取り出することができます。つまり、[0:8:2] によって 0 から 8 までの間隔 2 の自然数の列 0, 2, 4, 6 が生成され、これらの自然数に相当する位置の文字が文字列 ‘123456789’ から取り出された部分文字列 ‘1357’ になるのです^{*36}。また、列を逆向きに取り出す場合は刻幅を負の整数にします。たとえば ‘123456’[5:2:-1] で

^{*36} Python では列の開始は 1 ではなく C と同様に 0 から開始します。

は 5 から開始して 2 を越えない-1 間隔の自然数の列 5, 4, 3, 2 に対応する文字列 '6543' になります。この拡張スライス操作で用いられる文字リテラル "... " は Ellipsis 型のオブジェクトへの参照になります。またスライス操作で添字の省略も可能です。これは添字が列の端部を指定するときに限って可能です。たとえばリスト [1,2,3,4] で先頭から 3 成分を取り出したいときには a[0:3] と通常は記述しますが、列の端部 0 を省略して a[:3] と記述することができます。

列は変更可能な型のオブジェクトと変更不能の型のオブジェクトの二種類に分類され、変更可能なものがリスト型と ByteArrays 型、変更不能なものが文字列型、UNICODE 文字列型とタプル型になります：

- リスト型 (list): 記号 “,” で区切られた任意の Python オブジェクトの列を角括弧 “[]” で括ることで生成されるオブジェクトの型です。この型の構築子は list() で、この型のオブジェクトを函数 type() で調べると list が返却されます。
- ByteArray 型: 構築子 bytearray() で生成され、0 から 255 までの整数の列をバイナリ形式の列として蓄えることができます。
- 文字列型 (string): 接頭辞に ‘u’, ‘U’ を含まない文字列リテラルで生成される型です。構築子は str() で、函数 type() でオブジェクトを調べると str が返却されます。
- UNICODE 文字列型: 接頭辞に ‘u’, ‘U’ を含む文字列リテラルから生成される型です。構築子は unicode() で、この型のオブジェクトを函数 type() で調べると unicode が返されます。
- タプル型 (tuple): 任意の Python オブジェクトと記号 “,” を並べたものを一組とし、これらを一列に並べて丸括弧 “()” で括ったものから生成されるオブジェクトです。成分が一つのタプルは「シングルトン (singleton)」と呼ばれます。この型の構築子は tuple() で、この型のオブジェクトを函数 type() で調べると tuple が返されます。

■集合 (set): 有限個のオブジェクトから構成されるオブジェクトです。構成するオブジェクトの間には順序や対応付けを持たないために列や対応付け集合と異なり、添字を用いた書式で成分の取出や参照ができません。なお集合の濃度は函数 len() で調べることができます。Python の集合には以下に示す型があります：

- 集合型 (set): 変更可能な型で、組込の構築子 set() で生成されます。
- FrozenSet 型: 変更不能な型で、組込の構築子 frozenset() で生成されます。集合型と違って要約可能 (hashable) であるために別の集合の成分や辞書の鍵にすることができます。

■連想配列 (mapping): LISP の「連想リスト (association list, a-list)」に相当するオブジェクトです。通常の列の成分の取り出しあは列 S の順位を表現する 0 から開始する自然数の部分集合を添字として指定することで行えましたが、連想配列では添字集合が自然数の部分集合に限定されずに Python の有限個のオブジェクトの集合で与えることができます。そして、連想配列 A の参照は k を添字の集合 I の元とするときに $A[k]$ で行えます。より簡単に言うならば添字として自然数以外のオブジェクトが使える配列です。また、連想配列の濃度は列と同様に函数 `len()` で調べることができます。なお、現時点では Python に組込まれている連想配列は「辞書 (dict) 型」のみです：

- 辞書型 (dictionary): 変更可能な型で、この辞書型の構築子は `dict()` ここで添字集合のことを「鍵 (キー)」、添字集合の元を「鍵値」と呼びます。なお、辞書は名前空間の実装で用いられています。

■呼出可能 (callable): 呼出が可能なオブジェクトの総称です。ここでの呼出とは、形式的に名前等を函数の書式で扱うこと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操作を行うことです。呼出可能なオブジェクトかどうかは組込函数 `callable()` で判断できます。呼出可能のオブジェクトには、その実装方法と挙動の違いから以下のものがあります：

- ユーザ定義函数型 (user-defined function): 利用者による通常の函数定義で生成されるオブジェクトです。ここでの「通常」は `yield` 函数を内部に包含しないという意味で、`yield` 函数を包含する利用者定義の函数は後述の生成函数型になります。ユーザ定義函数型のオブジェクトの呼出は函数定義で用いた引数の列と同じ長さ^{*37}の列を引数にして行われ、任意の属性の設定や取得ができます。

```
>>> def count(x):
...     return x + 1
...
>>> count(1)
2
>>> callable(count)
True
>>>
```

- ユーザ定義メソッド型 (user-defined method): クラスやクラス インスタンス、あるいは `None` を「一次語 (primary)」とし、記号`“.”` で任意の呼出可能なオブジェクトと結合させることで生成されるオブジェクトです。ここでの説明は `neko`

*37 引数の個数を函数のアリティ (arity) と呼びます。

という名前のクラスがあって、そのインスタンスマソッドに `CatchMouse()` があるときにそのインスタンスが `tama` であれば、一次語 `tama` と記号 “.” の結合 `tama.CatchMouse()` でインスタンスマソッド `CatchMouse()` を呼び出せるという意味です。

ユーザ定義のメソッドの簡単な例を示しておきます:

```
>>> class TEST(object):
...     def __init__(self, val):
...         self.val = val
...     def dbl(self):
...         return self.val * 2
...
>>> a = TEST(3)
>>> a.dbl()
6
>>> callable(TEST)
True
>>>
```

ここで示すようにメソッドの呼出はインスタンスの名前を一次語として記号 “.” に続けてメソッド名を記載することで行えます。このときにメソッドの定義で用いた引数 `self` は除きます。なお、メソッド `__call__()` を用いると、インスタンスを函数と同様の書式で記載することで、このメソッドの呼出が可能になります:

```
>>> class TEST(object):
...     def __init__(self, start):
...         self.count = start
...     def __call__(self):
...         count = self.count
...         self.count = count + 1
...         return count
...
>>> a = TEST(0)
>>> a()
0
>>> a()
1
>>> a.__call__()
2
>>> callable(TEST)
True
>>>
```

この例では、メソッド`__init__()`でインスタンス変数`count`の初期化を行い、メソッド`__call__()`を実行すると`count`の値が`+1`されるというものです。このクラスのインスタンス`a`に対して‘`a()`’と函数の書式でメソッド`__call__()`の呼出が行われています。

- 生成函数型 (generator function): 利用者による函数定義で内部に`yield`文を持つ函数です。

```
>>> def genos(start):
...     count = start
...     while True:
...         yield count
...         count = count + 1
...
>>> a = genos(1)
>>> next(a)
1
>>> next(a)
2
>>> type(a)
<type 'generator'>
>>> callable(genos)
True
>>>
```

この例で示すように`yield`文を用いた函数が生成するオブジェクトの型は`generator`で、函数`next()`で生成型のオブジェクトから次の値を取り出すことができます。この生成函数と同様の処理をメソッドで行うことも可能で、そのクラスのインスタンスが次に述べる反復子型になります。

- 反復子型 (iterator): 利用者によるクラス定義で、メソッド`__iter__()`とメソッド`next()`の一対を持つクラスです：

```
>>> class TEST(object):
...     def __init__(self, start):
...         self.count = start
...     def __iter__(self):
...         return self
...     def next(self):
...         count = self.count
...         self.count = self.count + 1
...         return count
...
>>> a = TEST(1)
>>> next(a)
```

```

1
>>> next(a)
2
>>> callable(TEST)
True

```

- 組込函数型 (built-in function): 組込函数オブジェクトは C の函数のラッパーになります。このような函数の例には組込函数 dir() や函数 math.sin()^{*38}があります。
- 組込メソッド型 (built-in method): 組込函数を隠蔽したもので、C の函数に引き渡されるオブジェクトを何らかの非明示的な外部引数として持ちます。
- クラスタイプ型 (class type): クラスタイプ型のオブジェクトはインスタンスオブジェクトを生成するために用いられ、このときに「**ファクトリクラス**」として振舞います。ここでメソッド __new__() の上書 (override) を行っても問題はありません。クラスを呼出したときの引数はメソッド __new__() に引渡され、このメソッドがクラスタイプ型のオブジェクトを返却するときにインスタンスオブジェクトの初期化メソッド __init__() に引数が渡されます。
- 古典的クラス型 (classic class): 呼出されたときに新たに「**クラスインスタンス型**」のオブジェクトが生成されますが、このオブジェクトはクラスタイプ型から生成されるインスタンスオブジェクトと別の型のために注意が必要です。この呼出で用いられる引数はメソッド __init__() に引渡されるため、このクラスにメソッド __init__() がないときはクラスを引数なしで呼び出さなければなりません。
- クラスインスタンス型 (class instance): 古典的クラスのクラスにメソッド __call__() があるときに限って呼出可能型になります。

■モジュール (module): Python の文を記載したファイルを import 文で読込むことで生成されます。つまり、Python スクリプトを記した一つ一つのファイルがモジュールになります。なお Python スクリプトは最初にインタプリタに読み込まれる際にコードオブジェクト翻訳され、コードオブジェクトは.pyc ファイルに蓄えられます。ただし、モジュールにコードオブジェクトは含まれていません。また、複数のモジュールに階層構造を入れて管理できるようにしたものがパッケージです。

■クラス (class): Python 2.x には古典的な「**クラスオブジェクト型**」と新しい様式の「**クラスタイプ型**」の二種類のクラスがありますが、どちらも辞書で実装された名前空間を持ち、オブジェクトやその各属性への参照で用いられます。なお、Python 3.x 系ではクラスオブジェクトが廃止されてクラスタイプ型のみになります。古典的クラス型は Python 2.x 系で基底クラスとして object を継承しないときに生成される型で、クラスタイプ型と

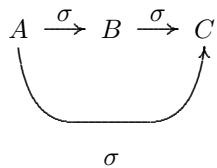
^{*38} 標準モジュール math に包含される函数 sin() を指示する名前になります。

違い、メタクラスが扱えず、引数なしにメソッド super() で基底クラスへの属性等の参照ができます:

```
>>> class TEST1:  
...     pass  
...  
>>>  
>>> class TEST2(object):  
...     pass  
...  
>>>  
>>> type(TEST1)  
<type 'classobj'>  
>>> type(TEST2)  
<type 'type'>  
>>> a=TEST1()  
>>> b=TEST2()  
>>> type(a)  
<type 'instance'>  
>>> type(b)  
<class '__main__.TEST2'>
```

Python 2.x のクラス定義で object を継承しないときだけ古典的クラスになります。この古典的クラスは函数 type() で classobj と表示されるクラスオブジェクト、そのインスタンスは函数 type() で instance と表示されるインスタンスオブジェクトです。ここで object を継承したクラスは函数 type() で type と表示されるクラスタイプ型で、そのインスタンスは instance ではなく、そのクラスのインスタンスになります。ここで type(b) の結果で '__main__.TEST2' とクラス名を含む文字列が返却されていることに注目して下さい。なお、函数 type() は動的にクラスタイプ型のクラスを生成することもできます。

ちなみに継承する側のクラスを「**派生クラス**」と呼び、継承される側のクラスを「**基底クラス**」と呼びます。基底クラスは派生クラスよりも普遍的なクラスで、その意味で上位のクラスです。この継承関係を親子関係にたとえるならば、基底クラスが「**親**」、継承する派生クラスが「**子**」にそれぞれ対応します。さらに、この継承関係を(素朴集合論の)集合の包含関係として捉えることも可能で、そのときに派生クラスが「**部分集合 (subset)**」にする「**サブクラス**」、基底クラスを「**スーパークラス**」と呼びます。さて、あるインスタンスがとあるクラスに包含されることと、あるクラスがとあるクラスを継承しているといった関係を矢で表現してみましょう。つまり $C_1 \xrightarrow{\sigma} C_0$ でクラス C_1 がクラス C_0 を継承したもの、すなわちクラス C_1 がクラス C_0 の派生クラスであるということを表現します。このとき、次の図式が可換になります:



まず、この図式は3個のクラスの派生関係を示すもので、 $A \xrightarrow{\sigma} B$ かつ $B \xrightarrow{\sigma} C$ であれば $A \xrightarrow{\sigma} C$ となること、つまり、この関係 $\xrightarrow{\sigma}$ は結合律を充します。ただし、反射律 $A \xrightarrow{\sigma} A$ に関しては、クラスの定義で自分自身が未定義の状態で自分自身を用いる循環的定義は許容されません^{*39}が、既存のクラスを上書きする形で自分自身を定義することは可能です。だから、反射律は循環的定義としては不可でも、既存の類定義の更新としては可能です。

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。この特殊属性に`__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: 古典的クラスを呼出すことで生成される対象で、辞書で実装された名前空間を持っているために最初の属性参照はここから開始します。属性参照にて辞書内で属性が見当らないものの、インスタンスのクラスに該当する属性名があるとき、そのインスタンスが含まれるクラス属性に検索領域が広げられます。このときの検索順序は基底クラスのタプルで、その左側のクラスから右側のクラスへと検索が行われます(MRO)。また、属性の代入や削除によってインスタンスの辞書が更新されますが、それに伴ってクラスの辞書が更新されることはありません。なお、Python 3.x では古典的クラスが廃止されるためにクラスインスタンスもなくなります。

■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): Python のインタプリタがその内部で用いる型で、これらの型の幾つかは利用者に公開されています。この内部型はインタプリタの仕様変更等で将来の変更が生じる可能性があります。ここではその概要を説明しておきます:

^{*39} 参照すべきオブジェクトがなければ例外が発生します。

- **コードオブジェクト (code object): 「バイトコード (bytecode)」を表現するオブジェクトです。** バイトコードはインタプリタ内部のデータ形式で, import 文等でインタプリタに読み込まれたプログラムはバイトコードに翻訳され, そのバイトコードを Python 仮想マシンで実行します。また, import 文で読み込まれたファイルは ‘.pic’ ファイルがなければモジュールとして解析とバイトコードへ翻訳され, このバイトコードは ‘.pyc’ ファイルに蓄えられます。一度 ‘.pyc’ ファイルが生成されると以後の import 文の読み込みで既存の ‘.pyc’ ファイルが利用されます。そのため ‘.pyc’ ファイルの更新は函数 reload() によるファイルの再読込か ‘python -m <ファイル名>’ で ‘.pyc’ ファイルの更新を行う必要があります。なお, このバイトコードは異なる Python 仮想マシンで互換性を保証しません。
- **フレームオブジェクト (frame object): 「実行フレーム (execution frame)」を表現するオブジェクトです。** プログラム実行時に呼び出し可能 (callable) なオブジェクトが呼び出され, そのオブジェクトの実行が終えた時点で呼び出した側のプログラムで呼び出し前の実行状態を保存しているオブジェクトから取り出す仕組になっています。このときの実行状態が保存されているときのオブジェクトをフレームオブジェクトと呼びます。
- **トレースバックオブジェクト (traceback object): 「例外スタックトレース (exception stacktrace)」を表現するオブジェクトです。** まず, traceback オブジェクトは例外が発生した時点で生成され, 例外ハンドラを検索して事項スタックを戻るときに, その戻ったレベル毎にトレースバックオブジェクトが, その時点の traceback の前に挿入されます。例外ハンドラに入るとスタックトレースをプログラムで利用できるようになります。
- **スライスオブジェクト (slice object): 「拡張スライス構文」を表現するために用いられるオブジェクトです。** この構文は MATLAB 系言語で用いられるベクトル成分の取出に類似した構文で, 配列処理で類似の機能を与えることになります。
- **静的メソッドオブジェクト (staticmethod object):** 組込の構築子 staticmethod() やデコレータ@staticmethod を使って生成されるオブジェクトです。インスタンスマソッドと違いインスタンス化しなくても利用可能です。また, メソッドの定義ではクラスメソッドと異なり引数としてそのクラス自体 ‘cls’ を取りません。クラス変数の参照ではクラスやクラス変数を直接呼び出すことで行います。またこのことから派生クラスで静的メソッドが定義されたクラスのクラス変数の参照が行われます。このように「静的」には「**継承において動的なクラス変数の参照が行われない**」という意味があります。なお, 静的メソッドはインスタンス変数を参照することができません。
- **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可能です。メソッドの定義では第一引数に‘cls’を必要とし、そのため継承で静的ではなく動的にクラス変数の参照が行われます。またクラスメソッドは静的メソッドと同様にインスタンス変数を参照することはできません。

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しょう：

```
>>>class A(object):
...     neko = 'mike'
...     def sm(x):
...         print "%s: neko = %s %%(A, A.neko, x)
...     def cm(cls,x):
...         print "%s: neko = %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 に静的メソッドである static_method() とクラスメソッドである class_method() を定義しています。ここで定義で示すように‘cls’を静的メソッドは引数として取らず、クラスやクラス変数への参照ではメソッド内部でそれらを直接呼び出しています。それに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 のままでです。しかし、クラスメソッド ‘B.class_method(1)’ ではクラスが B に切り替わっており、それに伴ってクラス変数の値もクラス B のものになります。

ここで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することができます:

```
>>>class A(object):
...     neko = 'mike'
...     @static_method
...     def static_method(x):
...         print "%s: %s %s"%(A, A.neko, x)
...     @class_method
...     def class_method(cls,x):
...         print "%s: %s %s"%(cls, cls.neko, x)
```

このように静的メソッドやクラスメソッドの定義の直前にデコレータを置くことで定義ができます。

リスト、集合や辞書の内包表現について

概念にはその外延 n の成員を列記した外延的表記と、その概念がどのようなものであるかを述べる文による内包的表記の二通りがあります。Python のリスト、集合や辞書には、それらに包含されるオブジェクトを直接、列記する外延表現と、その成員になるオブジェクトを列記するのではなく、Python の文で表記する内包表現の二種類があります。簡単にリストの例で説明すると、外延的表記は '[1, 2, 3, 4]' と具体的に記載する方法で、内包的表記の一例は '[i for i in range(1,5)]' といった方法です。このように内包的表現は列を生成する文を利用した記述であり、外延的表現は具体的に全ての成員を表記したり、リストであればメソッド append() で成員を追加する表記が対応します。この内包表現については、Python の式の節で詳細を述べますが、Python 言語の性格上、外延的表現は Python 言語のオーバーヘッドが生じ易いこともある、内包的表現の方が処理が高速になります。

3.7 特殊メソッド

3.7.1 特殊メソッドの概要

特殊メソッドは特定の演算や機能をクラスに実装するために Python であらかじめ用意された一群のメソッドのことです。これはメソッドの「**上書き (override)**」を利用したもので、定義するクラスに含まれない特殊メソッドは基底クラス側のものが用いられ、定義するクラスで特殊メソッドを新たに定義することで基底クラスから継承されたメソッドの上書きが行われます。その結果、定義したクラスで基底クラスのものとは別の改変されたメソッドが利用可能になります。たとえば、同じクラスのオブジェクトの比較では「**拡張比較**」のと呼ばれる一群のメソッドがあります。これらのメソッドは、これから定義す

るクラスの二つのインスタンスが等しいかどうか、同一クラスのインスタンス間の大小関係を判断するメソッドで、これらのメソッドを上書きすることで整数や実数で用いられる等号“==”とその否定“!=”，あるいは大小関係（“<”，“>”，“<=”，“>=”）に新たな意味を持たせることができます。ただし、ここでの拡張比較のメソッドは相反するメソッドのどちらか一方を上書きすることでもう一方が自動的に再定義されるというものではありません。たとえば、等号“==”の否定は“!=”ですが、等号“==”を再定義しても、その否定の“!=”が自動的に定義されません。そのために演算子“!=”を使う可能性があれば等号と併せて再定義を行う必要があります。

以下、項目別に特殊メソッドを列記しますが、クラス名を‘cls’、オブジェクト名を‘obj’、それからメソッドの引数をまとめて‘args...’と表記します。また、メソッドの引数に‘[, args...]’とある場合は鉤括弧“[]”で括った引数がオプションであることを指示します。それから二項演算子の引数は同一クラスに属するオブジェクトが二つ必要になりますが、引数として一つは前述の‘self’、もう一つは‘other’になります。

3.7.2 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するものです。

■`obj.__new__(cls [, args...])` クラス cls 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そのうち回収されて、C++ の消滅子のように不要になった時点でオブジェクトが回収されるというものではありません。そのために

C++ の消滅子と同様の働きをする消滅子を Python は持たないと言えます。一般的に基底クラスがメソッド `__del__()` 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。

3.7.3 表示に関するもの

通常、Python インタプリタでオブジェクトが束縛された名前を入力しても、そのオブジェクトが格納された番地以上の情報が返却されません。そうではなく利用者に適切なオブジェクトの持つ属性や値といった情報が表示されるようにあらかじめ定義しておくメソッドです。そのためにこれらのメソッドの引数は `self` のみになります。

■`obj.__repr__(self)` 組込函数 `repr()` や文字列への変換時に呼出され、オブジェクトを Python のフロントエンド側で表示する「公式の表示」の生成を行います。この章の有理数の定義にて整数対を有理数として表示する例のように、このメソッドを用いることで利用者にとって分かり易い表示や必要とされる情報の表示に変更することができます。

■`obj.__str__(self)` 組込函数 `str()` と `print` 文から呼出され、オブジェクトをフロントエンド側で表現する非公式の文字列の生成を行います。このメソッドの返却値は Python の文字列オブジェクトでなければなりません。なお、メソッド `__repr__(self)` が定義されていれば、このメソッドが定義されていなくても、その代わりにメソッド `__str__(self)` が利用されます。

3.7.4 比較に関するもの

クラスに「大小関係」を導入するためのメソッドで、これらの演算子は当然のことながら二項演算子です。したがって引数は `self` と `other` の二つのみになります。

比較の演算子

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

これらの演算子の書き換えは他の演算子に影響しません。すなわち、大小関係を新たに

定義したクラスに導入するために演算子“ \geq ”を定義したからといってその否定として演算子“ $<$ ”が自動的に定義されることはありません。大小関係の書き換えが一部でも生じた場合は全体を通して再定義が必要です。ただし、大小関係の演算子“ $<$ ”, “ $>$ ”, “ \leq ”, “ \geq ”の何れか一つと同値の演算子“ $==$ ”に相当するメソッドを定義していれば高階函数モジュール `functools` の函数 `functools.total_ordering()` を使って残りの比較演算子の自動定義を行うことが可能です。また、これらの演算子を書き換えていないときに用いられる比較の特殊メソッドが次の `obj.__cmp__()` です。

■`obj.__cmp__(self, other)` 二つの同一クラスのインスタンスを比較するときにインスタンスの識別子(整数型)を使って大小関係の判断が行われます。

3.7.5 整数演算に関するもの

整数オブジェクトの二項演算に関する特殊メソッドを以下にまとめておきます：

整数演算子(二項演算子)		
演算	Python の式	特殊メソッド
和	+	<code>obj.__add__(self, other)</code>
差	-	<code>obj.__sub__(self, other)</code>
積	*	<code>obj.__mul__(self, other)</code>
商	/	<code>obj.__truediv__(self, other)</code>
商	//	<code>obj.__floordiv__(self, other)</code>
剰余	mod	<code>obj.__mod__(self, other)</code>
幕	**	<code>obj.__pow__(self, other)</code>

ここで Python 2.x と 3.x では演算子“/”の挙動が異なります。Python 2.x で演算子“/”の二つの非演算子が整数のときは結果は整数が返却されます。しかし、Python 3.x で整数の除算は演算子“//”で、演算子“/”は割り切れない場合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二進数として表現したときの二項演算も加わります：

二進数整数演算子(二項演算)

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

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

3.7.6 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	Python の式	特殊メソッド
和	+	obj.__radd__(self, other)
差	-	obj.__rsub__(self, other)
積	*	obj.__rmul__(self, other)
商	/	obj.__rtruediv__(self, other)
商	//	obj.__rfloordiv__(self, other)
剰余	mod	obj.__rmod__(self, other)
冪	**	obj.__rpow__(self, other)

3.7.7 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)

演算	Python の式	特殊メソッド
$a = a + b$	$+=$	<code>obj.__iadd__(self, other)</code>
$a = a - b$	$-=$	<code>obj.__isub__(self, other)</code>
$a = a * b$	$*=$	<code>obj.__imul__(self, other)</code>
$a = a / b$	$/=$	<code>obj.__itruediv__(self, other)</code>
$a = a // b$	$//=$	<code>obj.__ifloordiv__(self, other)</code>
$a = a \bmod b$	$\bmod=$	<code>obj.__imod__(self, other)</code>
$a = a ** b$	$**=$	<code>obj.__ipow__(self, other)</code>

二進数累算算術代入演算子(二項演算)

演算	Python の式	特殊メソッド
$a = a <<= b$	$<<=$	<code>obj.__lshift__(self, other)</code>
$a = a >>= b$	$>>=$	<code>obj.__rshift__(self, other)</code>
$a = a \& b$	$\&=$	<code>obj.__iand__(self, other)</code>
$a = a ^ b$	$^=$	<code>obj.__ixor__(self, other)</code>
$a = a b$	$ =$	<code>obj.__ior__(self, other)</code>

ここで SageMath では数値や多項式の乗算に演算子 “ $**$ ” を用いていますが、演算子 “ $**$ ” を全ての対象に対して乗算になるように書き換えたものではありません。あくまでも SageMath で定義したクラスで利用可能な場合があるだけで、Python の数オブジェクトに対しては本来の XOR 演算子であることに注意が必要です。

3.7.8 単項演算子

単項演算子は二項演算子と異なり整数、浮動小数点数のメソッドの区別はありません。

単項演算子

演算	Python の式	特殊メソッド
$a \rightarrow -a$	$-$	<code>obj.__neg__(self)</code>
$a \rightarrow +a$	$+$	<code>obj.__pos__(self)</code>
$a \rightarrow a $	$\text{abs}()$	<code>obj.__abs__(self)</code>
$a \rightarrow \tilde{a}$	\sim	<code>obj.__invert__(self)</code>

ここで \tilde{a} は a の二進数表現での全てのビットの反転を意味します。

3.7.9 属性値の取得と設定に関連するメソッド

以下に説明するメソッドで 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。

■`obj.__setattr__(self, name, value)` 属性への値の束縛で呼出されます。ここで name が属性名, value が属性値です。なお、インスタンス側の属性に値を束縛させるとときは ‘`self.name = value`’ としてはいけません。‘`self.__dict__[name] = value`’ のようにインスタンス側の辞書に値を追加しなければなりません。

■`obj.__delattr__(self, name)` 属性に束縛した値の削除を行います。このメソッドの実装は ‘`del obj.name`’ に意味のあるときに限定すべきです。

■`obj.__getattribute__(self, name)` クラス型がクラスタイプのみに対して利用可能なメソッドで、指定した属性の値を返却するメソッドです。なお、メソッド `__getattr__()` が実装されていれば `AttributeError` 例外が送出されない限り呼び出されません。このメソッドの実装では再帰的な呼出を防止するために必要な属性全てへの参照で ‘`obj.__getattribute__(self, name)`’ のように基底クラスのメソッドと同じ属性名で呼び出さなければなりません。

3.7.10 その他

■`obj.__hash__(self)` 要約(ハッシュ)値を返却するメソッドです。要約値は整数値であり、同じ値を持つオブジェクトであればそれらの要約値は一致しなければなりません。つまり、‘`a == b`’ が `True` であるなら ‘`a.__hash__() == b.__hash__()`’ も `True` でなければなりません。なお、要約値は整数値になりますが、-1 をエラーフラグとして予約済にしている関係上、内部計算で-1 が得られると-2 を返却する仕様になっています⁴⁰。実際、整数オブジェクトでメソッド `__hash__()` は基本的にそれ自身を返却しますが、オブジェクトの値が-1 の場合のみ要約値として-2 を返却します。

このメソッドは組込の函数 `hash()`, `set()`, `frozenset()`, `dict()` のような要約(ハッシュ)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からです。逆に比

⁴⁰ <http://effbot.org/zone/python-hash.htm> 参照

較のメソッド `__cmp__(self)` と同値性検証のメソッド `__eq__(self)` が定義されていても、このメソッド `__hash__(self)` が定義されていなければ、そのインスタンスは要約可能 (hashable) にならないために辞書の鍵として使えません^{*41}。なお、利用者定義のクラスには `__hash__(self)` メソッドが継承されおり、このメソッドは識別値 `id()` を使って定義されています。ユーザ定義クラスを非要約可能 (unhashable) にするためには、「`__hash__ = None`」にすることで行えます。また Python 3.x 系では `__hash__` を未定義の状態で `__eq__(self)` を上書きすることで自動的に「`__hash__ = None`」になります。

■obj.__nonzero__(self) 真理値テストや組込演算 `bool()`^{*42} の実現のために呼出されます。このメソッドは真理値の ‘False’ か ‘True’、あるいはそれらと等価の整数 ‘0’ か ‘1’ 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__(self)` が呼出され、その結果が ‘nonzero’ であれば真: `True`、`nonzero` のときは偽: `False` になります。それからもしもメソッド `__len__(self)` と `__nonzero__(self)` の双方が実装されていなければ、そのクラスのインスタンスの真理値は全て ‘True’ とみなされます。

■obj.__unicode__(self) 組込函数 `unicode()` を実現するために呼出され、`unicode` オブジェクトを返却しなければなりません。このメソッドが定義されていないときは文字列リテラルへの変換が試みられ、その結果、既定値の文字エンコードを用いて UNICODE 文字列に変換されます。

3.8 記述子 (descriptor)

ある特定の性質を持つオブジェクトが実装すべきメソッドのことを「規約 (protocol)」と呼びますが、「記述子」はその規約の一つで、クラスタイプ (`object` や `type` の派生クラス) のみに対応し、属性の束縛に関わるメソッドです。記述子は「記述子規約 (descriptor protocol)」と呼ばれる 3 個のメソッド `__get__(self)`, `__set__(self)` と `__delete__(self)` の何れかを有するものです。ここで記述子はメソッド `__get__(self)` と `__set__(self)` の双方が定義されている「データ記述子」とメソッド `__get__(self)` のみが定義されている「非データ記述子」の二種類に大きく分類されます。また、データ記述子で、そのメソッド `__set__(self)` の呼出によって `AttributeError` 例外が送出されるものを「読み専用データ記述子」と呼びます。

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` でおこないますが、この `a.x` が基点になります。ここで引数がどのよ

^{*41} より正確には要約可能であるかどうかの判別にメソッド `__hash__(self)` が呼び出せ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变性はプログラマが保証しなければなりません。

^{*42} 何故か `bool` でない!

うに記述子に結合されるかはオブジェクト `a` がクラスのインスタンスであるか、あるいはクラスそのものであるかに依存します：

——記述子の呼び出し——

- 直接呼出：最も単純な呼出操作で ‘`x.__get__(a)`’ に変換されます。
- インスタンス束縛：クラスタイプのインスタンスに対する束縛で ‘`a.x`’ が ‘`type(a).__dict__['x'].__get__(a,type(a))`’ に変換されます。
- クラス束縛：クラスタイプのクラスに対する束縛で ‘`a.x`’ が ‘`a.__dict__['x'].__get__(None,a)`’ に変換されます。
- スーパークラス束縛：`a` が `super` のインスタンスのときに束縛 `super(b,obj).m()` を行うと最初に `a`、次に `b` に対して `obj.__class__.__mro__` を検索し、それから呼出：‘`a.__dict__['m'].__get__(obj,obj.__class__)`’ で構築子を呼出します。

まず、インスタンス束縛における構築子の呼出の優先順序は定義内容に依存します。そして構築子は上述の 3 つのメソッドの任意の組合せで定義することができますが、ここでメソッド `__get__()` が定義されていないときに該当する属性の参照が行われると構築子オブジェクト自体が返却されます。前述のようにメソッド `__set__` と `__delete__` のどちらか一方が定義され、データ構築子、双方が定義されていなければ非データ構築子になります。ここで組込函数 `property()` はデータ構築子として Python に実装されたもので、このときにインスタンスでは属性の上書きができません。その一方で `staticmethod()` と `classmethod()` を含む Python のメソッドは非データ構築子として定義され、そのためインスタンスでメソッドを再定義され、インスタンスでメソッドの再定義や上書きが可能になっています。このことを利用して同じクラスのインスタンスでも個々の挙動に違いを持たせることができます。また属性検索にてデータ記述子やインスタンスの属性辞書、非データ記述子の順番で検索が行われます。

3.8.1 記述子の実装

■`obj.__get__(self, instance, owner)` クラスの属性やインスタンスの属性への参照時に呼出されます。ここで ‘`owner`’ はオーナークラスで、`instance` は属性への参照を仲介するインスタンス属性が `onwer` を介して参照されるときは ‘`None`’ になります。

■`obj.__set__(self, instance, value)` オーナークラスのインスタンス `instance` 上の属性を新たな値: `value` に束縛する際に呼出されます。

■`obj.__delete__(self, instance)` オーナークラスのインスタンス `instance` 上の属性を削除する際に呼出されます。記述子は組込函数 `property()` と似た動作になります。

3.9 クラス属性の参照について

クラス属性の参照は、クラス名が C で属性が x のときに $C.x$ で行えます。この参照の実体は $C.__dict__["x"]$ で、目的の属性がクラス名で指示したクラスに見当らなければ、より上位にある基底クラスに対して参照が行われます。ここで属性の検索は検索しているクラスに無ければ継承関係が一つ上のクラスへと遡るという「**継承の深さ**」が関わる検索になります。たとえば、 $C_0 \xrightarrow{\sigma} C_1, C_1 \xrightarrow{\sigma} C_2, \dots, C_{n-1} \xrightarrow{\sigma} C_n$ という継承関係からはクラス C_0 から開始してクラス C_n に至るという継承関係の分解図式 (Resolution): $C_0 \rightarrow \dots \rightarrow C_n$ が得られ、この図式に現われるクラスの順に属性やメソッドの検索が行われます。つまり、この分解図式に現われるクラスを左側から並べることで得られたリスト (C_0, C_1, \dots, C_n) の並び順がクラス C_0 の「**MRO(Method Resolution Order)**」と呼ばれるメソッドの検索順序で、 $\mathcal{L}(C)$ と記述することにします。また、この検索順序を求める処理のことを「**線形化 (linealization)**」と呼びます。この MRO を説明するために幾つかの言葉を定義しておきます。まず、検索順序はクラスのリスト: (C_1, \dots, C_n) として表現されますが、これを語: $C_1 \dots C_n$ と表記することにします。ここで、リストの先頭にクラス C_0 を追加することは語の先頭に C_0 を追加することに対応し、この追加する操作を $C_0 + C_1 \dots C_n$ と表記します。次に語 $C_0 C_1 \dots C_n$ の先頭 C_0 を取り出す操作を **head**, 先頭以外の残りの $C_1 \dots C_n$ を取り出す操作を **tail** と表記し、語 L に対して $\underline{L} \stackrel{\text{def}}{=} \text{head}(L), \underline{\underline{L}} \stackrel{\text{def}}{=} \text{tail}(L)$ と略記します。それから語 $B_1 \dots B_m$ と語 $C_1 \dots C_n$ が与えられたとき、語 $B_1 \dots B_m$ に含まれる各 $B_i (1 \leq i \leq m)$ を語 $C_1 \dots C_n$ から取り除いた語を $C_1 \dots C_n \setminus B_1 \dots B_m$ と表記することにします。たとえば、 $abcd \setminus bd$ は ac になります。また、検索では一度出たクラスを再度検索する必要はありません。このことから語 ‘ $\dots W_{(i-1)} W_i W_{(i+1)} \dots W_{(j-1)} W_i W_{(j+1)} \dots$ ’ は語 ‘ $\dots W_{(i-1)} W_i W_{(i+1)} \dots W_{(j-1)} W_{(j+1)} \dots$ ’ と検索順序として一致することになります。このように後続の一貫する語を除いたものとの関係を ‘ $\dots W_{(i-1)} W_i W_{(i+1)} \dots \setminus \dots W_{(i-1)} W_i W_{(i+1)} \dots W_{(j-1)} W_{(j+1)} \dots$ ’ と表記します。そして関係 ‘ \setminus ’ 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 ‘ \setminus ’ の極限になる語をここでは \underline{L} と表記します。そして作用素 \mathcal{L} は次の性質を持ちます:

作用素 \mathcal{L} の性質

1. $\mathcal{L}(C) = C$
2. $\mathcal{L}(C_0(C_1)) = C_0 + \mathcal{L}(C_1)$
3. $L_1 \searrow L_0$ のとき $\mathcal{L}(L_1) = \mathcal{L}(L_0)$
4. $\mathcal{L}(C(B_1, \dots, B_n)) = C + \mathcal{M}(\mathcal{L}(B_1), \dots, \mathcal{L}(B_n))$

最初の 1. は継承関係を持たないクラス C のときに MRO は C のみであるということを示します。つぎの 2. は $C_0 \xrightarrow{\sigma} C_1$ のとき、つまり、クラス C_0 が C_1 の派生クラスのときに最初にクラス C_0 を検索し、それからクラス C_1 の検索順序に従うということを意味し、つぎの 3. は関係 “ \searrow ” で作用素 $*$ の値は不变であることを示し、最後の 4. が多重継承があるときの処理になります。次に作用素 \mathcal{M} を導入しておきましょう。この作用素 \mathcal{M} の働きは、継承関係を越す MRO を基本に多重継承のあるクラスにて継承の親達を示すタプルをそのまま用いて基底クラスに検索順序を入れるというものです。つまり、クラスの属性で基底クラスを示すタプルの左側から順番に先祖を辿る方法で行われるので「**深さ優先、左から右の順番規則 (left-to-right depth-first rule)**」と呼ばれる規則になります。

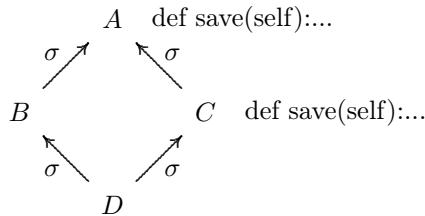
この操作を表現する作用素 \mathcal{L} は次の性質を持ちます：

作用素 \mathcal{M} の性質

- a. $\mathcal{M}(W) = \underline{W}$
- b. $\mathcal{M}(\dots, L, \dots) = \mathcal{M}(\dots, \underline{L}, \dots)$
- c. $\mathcal{M}(L_1, L_2, \dots, L_n) = \underline{\underline{L}_1} + \mathcal{M}(L_2 \setminus L_1, \dots, L_n \setminus L_1)$

最初の a. は検索順序を示す語 W 一つが引数であれば、関係 “ \searrow ” の極限 \underline{W} を返すという性質です。そして次の b. は関係 “ \searrow ” で作用素 \mathcal{M} の値は不变であることを示しています。そして最後の c. は引数の最も左側にある経路を外に出し、その語に含まれるクラス名を他の引数から除去していくという処理方法を示しています。もし、引数の語に共通するクラス名がなければ \mathcal{M} は語に対する和になるだけです。

Python の古典的クラス型でメソッド検索で用いられる順序は MRO です。しかし、この手法は多重継承があるときに有効に動作しない問題があります。このことを次のクラス A, B, C, D の関係が次の菱形状になる図式を使って解説しておきましょう：



この図式は、クラス D はクラス B と C の双方を継承する多重継承の関係にあり、同時にクラス B と C はクラス A の派生クラスであることと、クラス A と C でメソッド `save` が定義されていることを表現しています。ここで各クラスが古典的クラス型のときにクラス B やクラス C でメソッド `save()` を利用しようとなればクラス A のものがそのまま用いられ、クラス C で上書きされたメソッド `save()` はそのままではクラス D で用いられることがありません。ここで実際に MRO を計算してみましょう：

菱形状の継承関係の RMO

$$\begin{aligned}
 \mathcal{L}(B) &= BA \\
 \mathcal{L}(C) &= CA \\
 \mathcal{L}(D) &= D + \mathcal{M}(\mathcal{L}(B), \mathcal{L}(C)) \\
 &= D + \mathcal{M}(BA, CA) \\
 &= D + BA + \mathcal{M}(CA - BA) \\
 &= DBA + \mathcal{M}(C) \\
 &= DBAC
 \end{aligned}$$

このように古典的クラスの属性検索 (MRO) ではクラス D, B, A, C, A の順番で検索が行なわれます。ところがこの属性検索では D, B の次のクラス A のメソッド `save()` が発見された時点で検索が終了し、その結果、クラス A で定義されたメソッド `save()` が用いられ、クラス A の派生クラス C で再定義されたメソッド `save()` が用いられないということになります。すなわち、より近い側のメソッドが用いられないという問題が生じることになります。そのためには「C3 MRO(Method Resolution Order)^{*43}」と呼ばれる手法で検索が行われます。この C3 は多重継承にある場合に妥当な検出順序を提供するアルゴリズムで、最初に Dylan 言語に導入された手法です。この手法は先程の 3. の計算手順の作用素 \mathcal{M} を作用素 \mathcal{M}_{c3} で置換えて、次のようにまとめることができます：

^{*43} <https://www.python.org/download/releases/2.3/mro/> や PEP-253 を参照

C3 での作用素 \mathcal{L} の性質

1. $\mathcal{L}(C) = C$
2. $\mathcal{L}(C_0(C_1)) = C_0 + \mathcal{L}(C_1)$
3. $L_1 \searrow L_0$ のとき $\mathcal{L}(L_1) = \mathcal{L}(L_0)$
- 4'. $\mathcal{L}(C(B_1, \dots, B_n)) = C + \mathcal{M}_{c3}(\mathcal{L}(B_1), \dots, \mathcal{L}(B_n))$

ここで作用素 \mathcal{M}_{c3} は次の性質を持ちますが、最初の a., b. は作用素 \mathcal{M} の場合と同様です。また c. の計算手順は \mathcal{M} よりも階層を意識した検出方法に代ります：

作用素 \mathcal{M}_{c3} の性質

- a. $\mathcal{M}_{c3}(W) = \underline{W}$
- b. $L_0 \searrow L_1$ のとき $\mathcal{M}_{c3}(\dots, L_0, \dots) = \mathcal{M}_{c3}(\dots, L_1, \dots)$
- c. \mathcal{M}_{c3} の計算は後述の方法で計算される。

この \mathcal{M}_{c3} の計算手順を $\mathcal{M}(L_1, L_2, \dots, L_n)$ が与えられたときにどのように行われるのかを纏めておきましょう：

 \mathcal{M}_{c3} の計算手順

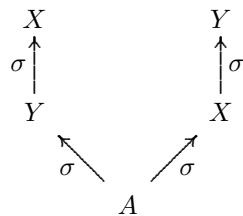
1. $i = 1$ とする。
2. $h_i = \overline{L_i}$ とする。
3. $j \neq i$ に対して $h_i \notin \underline{L_j}$ のときに $k \in (1, \dots, n)$ に対して $\overline{L_k} = h_i$ であれば、 \mathcal{M}_{c3} の引数にある L_k を $\underline{L_k}$ で置換し、 $h_i + \mathcal{M}_{c3}(L_1, \dots, L_n)$ を作用素 \mathcal{M}_{c3} の結果として返却する。
4. $h_i \in \underline{L_j} (i \neq j)$ のとき $i \neq n$ ならば $i = i + 1$ として 2. に戻る。もし $i = n$ であればエラーを出力して処理を終える。

作用素 \mathcal{M}_{c3} は引数の語に共通するものが何もなければ最初の作用素 \mathcal{L} を拡張したものと同様に語の和として作用します。しかし、共通する語が現われたときの処理がクラスの階層を合致させる働きになります。このことを先程の菱形状の継承関係で C3 MRO を計算することで確認してみましょう。

ダイアモンド状の継承関係の C3 RMO

$$\begin{aligned}
 \mathcal{L}(B) &= BA \\
 \mathcal{L}(C) &= CA \\
 \mathcal{L}(D) &= D + \mathcal{M}_{c3}(\mathcal{L}(B), \mathcal{L}(C)) \\
 &= D + \mathcal{M}_{c3}(BA, CA) \\
 &= D + B + \mathcal{M}_{c3}(A, CA) \\
 &= DB + C + \mathcal{M}_{c3}(A, A) \\
 &= DBCA
 \end{aligned}$$

C3 MRO では *DBCA* と MRO の *DBAC* と異なりクラス *C* の方が大本のクラス *A* よりも先に検索が行われるためにより新しいクラス *C* のメソッド *save()* が用いられ、古典的クラス型の MRO よりも妥当な結果が得られることになります。さらにこの C3 MRO の長所は間違った継承関係を検出することが可能ですが、たとえば次の継承関係を想定しましょう：



この継承関係は $X \xrightarrow{\sigma} Y$ かつ $Y \xrightarrow{\sigma} X$ と、クラスの定義では相互参照的な関係、いわゆる循環的な定義であり、このような定義は Python では間違った定義です。しかし、MRO ではエラーではなく *AYX* が得られ、一方の C3 MRO では

$$\begin{aligned}
 \mathcal{L}(Y) &= YX \\
 \mathcal{L}(X) &= XY \\
 \mathcal{L}(A) &= A + \mathcal{M}_{c3}(\mathcal{L}(Y), \mathcal{L}(X)) \\
 &= A + \mathcal{M}_{c3}(YX, XY)
 \end{aligned}$$

と計算が進むものの $\mathcal{M}_{c3}(YX, XY)$ の処理で *YX* の *Y* が *XY* に含まれるために *YX* の処理が行えず、今度は *XY* から *X* を取り出す処理に移ります。しかし、この *X* も前の *YX* に含まれるために *XY* からもクラスを取り出すことができずに作用素 \mathcal{M}_{c3} はエラーを出力しなければなりません。このように間違った継承関係の図式が与えられても C3 MRO では適切な処理が行えることを意味しています。

3.10 名前空間とスコープ

3.10.1 名前と名前空間

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

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

名前のBNF

名前 ::= [識別子 分離記号] 識別子

次にオブジェクトの参照を行う際に参照可能な範囲、すなわち、参照の有効範囲が問題となります。この参照の有効範囲のことを「**スコープ (scope)**」と呼びます。ここでPythonのスコープにはモジュールの大域的なスコープと函数内部の局所的なスコープの二種類しかありません。

3.11 名前付けと束縛

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

■**ブロック (block)**: プログラム内の一つの実行単位になる区画で、モジュール、クラスと函数定義はブロックになります。そして、Pythonのシェルを経由して対話的に入力された個々の命令もまたブロックになります。

■**コードブロック (code block)**: スクリプトファイル、スクリプト命令、組込函数eval()やexec()に引き渡した文字列、函数input()から読み取られて評価されるPythonの文で構成されたもので、これらコードブロックの実行は「**実行フレーム (execution frame)**」上で実行されます。

■スコープ (scope): 名前の参照可能な範囲/領域のことです。たとえば、局所変数があるブロック内部で定義されているとき、その変数のスコープはその変数に束縛されたオブジェクトが参照可能な範囲であるそのブロックを含みます。したがって函数やクラス内で定義された名前のスコープは、それらのブロック内に制限されます。とは言え、メソッドのコードブロックを含むような拡張は行われません。その例としてリファレンスマニュアルでは生成子を一例として挙げています：

```
class A:
    a = 42
    b = list(a + i for i in range(10))
```

この例では名前 b に束縛する構築子 list() の引数が生成子 (generator) で、ここではリストの成員の型を for 節を使って表記するリストの内包表現による生成に対応します。そして、この生成子内部で名前 a への参照がありますが、名前 a は構築子 list() のブロック外部にあるためにスコープ外になります。そこで、名前 a への束縛が行われていてもエラーになります：

```
>>> class A:
...     a=42
...     b=list(a+i for i in range(10))
...
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "<stdin>", line 3, in A
File "<stdin>", line 3, in <genexpr>
NameError: global name 'a' is not defined
```

名前がコードブロック内部で用いられているとき、その名前の参照では近傍のスコープを用いられます。ここで**ブロックの環境**とは、ある一つのコードブロック内で参照可能なスコープの全ての集合のことです。

名前があるブロック内部で束縛されているとき、その名前はそのブロックにおける局所変数となります。名前がモジュールで束縛されているときは大域変数になります。そして、コードブロックで用いられていても、そのブロックで定義や束縛が行われていないときに、その変数は自由変数となります。

ここで名前の参照を行った際に、その名前に束縛されたオブジェクトがないときに NameError 例外が出力されます。また、局所変数で、名前が束縛されていない変数の参照を行ったときには UnboundLocalError 例外が出力されます。この UnboundLocalError は NameError のサブクラス (NameError クラスの種の一つ) になっています。なお、del

文で指定された対象は、`del` 文の目的が対象の束縛の解除であるものの、束縛済みのものと見做されます。

`import` 文や代入文は、クラスや函数定義、モジュールレベル内で行われます。

`global` 文で指定された名前がブロック内にあるとき、その名前は名前空間の最上層で束縛された名前の参照を行うことになります。

3.12 例外

例外はコードブロックの通常の処理を中断して、エラー等のいわゆる「例外的な状況」を把握できるようにするための手段です。例外は何らかのエラーが検出された時点で送出されますが、`raise` 文を用いて意図的に例外の送出を行うことも可能です。また、例外ハンドラを `try ... except` 文で指定することもできます。そして、`try` 文にて `finally` 節を用いることでクリーンアップコードを指定することができます。

例外の識別はクラスインスタンスで行われ、`except` 節はこのインスタンスのクラスで選択されます。また、例外は文字列で識別することも可能です。

3.13 Python の式

3.13.1 原子要素

■原子要素 (atom): Python の式を構成する基本単位です。

原子要素 (atom)

原子要素	::=	識別子 リテラル 閉包
閉包	::=	括弧形式 リスト表現 生成式 辞書表現 集合表現
		文字列変換 Yield 原子要素

この EBNF からも判るように最も単純な原子要素は、識別子やリテラルと閉包の何れかです。ここで識別子は名前で、この名前にオブジェクトが束縛されているときに、その名前を評価することでオブジェクトの値が返却されます。しかし、名前にオブジェクトが束縛されていないときに評価を行うと `NameError` 例外になります。なお、リテラルは数リテラルと文字列リテラルの二種類があり、具体的なデータを構成し、閉包は括弧式、リスト表現、辞書表現、集合表現、文字列変換、生成式や `yield` 原子要素の何れかになりますが、各対象については後に詳細を述べることにします。

Python では名前の暗号化が可能です。これはクラスの定義内部に記述された識別子の名前が二つ以上の記号 “_” で開始し、末尾が二つ以上の記号 “_” がないものは、そのクラスでの隠蔽されるべき名前と見做されます。このときの隠蔽の方法は非常に安易で、別の新しい名前で変換することで行われます。この変換では、先頭に記号 “_” を一つ置き、それからクラス名を隠蔽すべき名前の前に置きます。マニュアルの例では、クラス名を Hom、その中の識別子を __spam とするときに名前は _Hom __spam に変換されることを意味します。

■リテラル: 文字列リテラルとさまざまな数リテラルで構成されます:

リテラル

リテラル ::= 文字リテラル | 整数リテラル | 長整数リテラル | 浮動小数点
数リテラル | 複素数リテラル

■括弧形式: 式の列を括弧“()”で括ったものです:

括弧形式

括弧形式 ::= "(" [式の羅列] ")"

ここで中身が空の括弧形式は空のタプルになります。また、タプル自体は式の列で、リストのように括弧“()”で括る必要がありません。

■リスト表現: 角括弧 “[]” で括られた式の列として表現されます:

リスト表現に関する構文

リスト表現 ::= "[" [式の列] | リスト内包表現 "]"
 リスト内包表現 ::= for 文リスト
 for 文リスト ::= "for" 標的リスト "in" 旧一般式リスト
 [リスト内包表現]
 旧一般式リスト ::= 旧一般式 [(",", 旧一般式)+[",",]]
 旧一般式 ::= 論理検証式 | 旧 λ-式
 リスト内包 ::= for 文リスト | if 文リスト
 if 文リスト ::= "if" 旧一般式 [リスト内包]

■内包表現: 集合と辞書の表現で用いられます。これらのオブジェクトはコンテナと呼ばれ、オブジェクトへの参照を伴うオブジェクトになって具体的に内容を列記したもの、あるいは内包表現の何れかになります。以下に内包表現の構文を示しておきます:

—— 内包表現の構文 ———

```

内包表現 ::= 式 for 節
for 節      ::= "for" 標的列 "in" 論理和検証式 [反復節]
反復節       ::= for 節 | if 節
if 節        ::= "if" 条件式 [反復節]

```

この内包表現は後述の生成式や if 節によるフィルター処理の組み合わせから、その外延が生成できる仕組となっています。

■**生成式 (generator):** ジェネレータオブジェクトを返却する式です:

—— 生成式の構文 ———

```
生成式 ::= "(" 式 for 節 ")"
```

丸括弧“()”で括られた for 節を伴う式です。ジェネレータオブジェクトにメソッド next() が呼び出された時点で、for 節が評価され、その時点の for 節が返す変数を用いた式の値が返却されます。

■**辞書表現:** 波括弧“{ }”で括られた鍵と値の対で構成された列です:

—— 辞書表現の構文 ———

```

辞書表現      ::= "{" [鍵リスト] | 辞書] "}"
鍵データリスト ::= 鍵データ (",," 鍵データ)* [",,"]
鍵データ       ::= 式 = ":" 式
辞書          ::= 式 ":" 式 for 節

```

■**集合表現:** 辞書表現と同様に波括弧“{ }”で括られていますが、鍵と値を区切る記号 “:” を持たないことで辞書表現と異なります:

—— 集合表現の構文 ———

```
集合表現 ::= "{" (式の列 | 内包表現) "}"
```

集合表現は列の左から式や内包表記が評価されます。なお、空集合 \emptyset を ‘{}’ で表現することはできません。‘{}’ は空のタプルになるためです。

3.13.2 一次語

Python の式は原子要素を基本単位として一次語が構成され、この一次語を用いて式が構成されます。この小節で解説する式が Python の最も基本的な単位になります。

■一次語、属性参照と添字表記: 名前、属性参照等のPython言語で最も基本となる表記です。一次語は名前、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を行う表記です:

一次語、属性参照と添字表記の構文定義

```

一次語 ::= 原子要素 | 属性参照 | 添字表記 | スライス表記 | 呼出
属性参照 ::= 一次語 "."
添字表記 ::= 一次語 "[" 式の列 "]"

```

ここで属性参照は、一次語で参照すべき属性を指定するときの表記方法で、添字表記はオブジェクトの参照先の型が列や辞書等、添字を必要とするときに用いられる表記です。

■スライス表記: MATLAB系の言語で行列や配列の添字操作を、その表記方法も含めてPythonで実現しています:

スライス表記の構文定義

```

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 "[" スライス列 "]"
スライス列 ::= 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 ::= "..."

```

このようにMATLABとほぼ同様の表記になっていますが、MATLAB系の言語の省略記号が“:”であるのに対し、Pythonの省略記号は“...”と刻幅を指定する長スライス表記の刻幅が短スライス表記のうしろに付けることがMATLAB系の言語の刻幅の表記と微妙に異なります。この省略記号はMATLABクローンで見られる省略記号よりもYorickのゴム添字に対応するもので、配列の構造を省略表記するものです:

```

sage: type(imat)
<type 'numpy.ndarray'>
sage: imat.shape

```

```
(509, 800, 3)
sage: imat[...,2].shape
(509, 800)
sage: imat[:, :, 2].shape
(509, 800)
sage: sum((imat[:, :, 2]==imat[...,2])==False)
0
sage: imat[:, 2].shape
(509, 3)
```

この例では NumPy の配列の `imat` を使っています。この配列の大きさは $509 \times 800 \times 3$ で、このことはメソッド `shape()` を適用することで判ります。さて、ここで ‘`imat[...,2]`’ は ‘`imat[:, :, 2]`’ と同等ですが、‘`imat[:, 2]`’ は明らかに違います。ここでの省略記号 ‘`...`’ は ‘`:, :`’ と同じ意味で、配列の構造を含めた記号となっています。つまり、記号 ‘`:`’ が一次元の省略であるのに対して ‘`...`’ は多次元で、添字で埋められている箇所以外の構造を保つものになります。

なお、MATLAB 系の言語では行列や配列の添字が 1 から開始しますが、Python では **C と同様に添字が 0 から開始する** という大きな違いがあります。この点については MATLAB 系の言語に使い慣れた人にとっては特に注意が必要です。

■呼出: フィルタやメソッド等の呼出で用いられる構文です:

呼出の構文定義

呼出	::= 一次語 "(" [引数の列 [","] 式 geneexpr_for ")"
引数の列	::= 定位引数 [","] キーワード引数

たとえば、引数が 1 個、つまり、arity が 1 の函数 `mike` の呼出では ‘`mike(a)`’ のようにすることで行えます。なお、Python の非組込のライブラリの函数を呼出すときは通常、ライブラリ名の属性として函数を呼出すことになります。たとえば、`sin` 函数は Python の `math` ライブラリに含まれるために ‘`math.sin(10)`’ のように一次語を (ライブラリ名). (函数名) として呼出が行われます。

3.13.3 演算式

ここでは Python の演算式について述べます。

■単項算術演算: 式 $-a$ や $+b$ での算術演算子 “`+`” 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に加え、幂乗 a^b を表現する Python の式を加えたもの

です:

———— 単項算術演算式の構文定義 ————

```
单項算術演算式 ::= 幂乗 | "-" 单項算術演算式 | "+" 单項算術演算式 |
                   "~~" 单項算術演算式
幂乗 ::= 一次語 ["**" 单項算術演算式]
```

単項算術演算子には、一次語そのものと、その一次語を用いて構築した幂乗、それに加えて算術演算子“+”や“-”に加え、二進数のビット反転演算子“~~”を先頭に置いた式があります。ここで幂乗 a^b を Python では ‘`a**b`’ で表現しますが、他の多くの数式処理で用いられている演算子“^”を SageMath では幂乗の演算子として利用することができます。ただし、幂乗になるのはあくまでも SageMath で定義されたクラスのインスタンスに対してであり、Python の数オブジェクトに対しては本来の排他的論理和 XOR の意味のままです。

■二項算術演算式: Python の二項算術演算式は乗法的と加法的の二種類があります:

———— 二項算術演算式の構文定義 ————

```
乗法的算術演算式 ::= 单項算術演算式 | 乗法的算術演算式 "**"
                     单項算術演算式 | 乗法的算術演算式 "//" 单項算術
                     演算式 |
                     乗法的算術演算式 "/" 单項算術演算式 |
                     乗法的算術演算式 "%" 单項算術演算式
加法的算術演算式 ::= 乗法的算術演算式 | 加法的算術演算式 "+" 乗法的算
                     術演算式 | 加法的算術演算式 "-" 乗法的算術演算式
```

加法的演算子としては、演算子“+”と“-”があります。乗法的演算子には積“*”，商“/”，剰余“%”，それと小数点以下の切捨を伴う商の演算子“//”があります。

■ずらし演算式: 整数値を二進数で表現したときに、ビットを左右に桁を移動させる演算で、通常の算術演算子よりも優先順位が低くなっています。

———— ずらし演算式の構文定義 ————

```
ずらし演算式 ::= 算術的演算式 | ずらし演算式 ("<<" | ">") 算術的演算
                     式
```

■ビット単位の論理演算式: ビット単位で論理積(AND), 論理和(OR)と排他的論理和(XOR)を表現する式です:

—— ビット単位の論理演算式の構文定義 ——

論理積式	::=	ずらし演算式 論理積式 "&" ずらし演算式
排他的論理和式	::=	論理積式 排他的論理和式 "^" 論理積式
論理和式	::=	排他的論理和式 論理和式 " " 排他的論理積式

被演算子を二進数で表現したときに、各桁のビット単位で演算が行われます。まず論理積は双方が 1 の場合のみ 1 で他が 1, 論理和は双方が 0 の場合のみ 0 で他が 1, 排他的論理和はどちらか一方が 1 の場合のみが 1 で、それ以外は 0 を返す演算子です。なお、Python で排他的論理和の演算子として演算子 “^” を用いていますが、SageMath では演算子 “^” を算術の演算子として用いています。この点は SageMath と Python の演算子の相違点として注意してください。

■比較の演算式: C と異なる優先順位を持っており、‘ $a > b > c$ ’ のような二項演算ではなく、複合的な表記が可能になっています。比較の演算子による結果は True か False の Boolean になります：

—— 比較の構文定義 ——

比較式	::=	論理和 (比較の演算子 論理和)*
比較の演算子	::=	"<" ">" "==" ">=" "<=" "<>" "!="
		"is" "not" ["not"]"in"

比較の演算子には、通常の大小関係の演算子に加え、合同性を示す演算子として演算子 “==” と演算子 “is”，それと包含関係を示す演算子 “in” があります。

この EBNF に示すように比較の演算式は幾らでも繋げることが可能です。ここで C や Fortran の比較の演算式は厳密に二項演算子で、2 以上のアリティを持つ演算子ではありませんが、Python では比較の演算子は 2 以上のアリティを持ち、さらに四則演算を表現する算術演算子のように扱えます。具体的には比較の式が ‘ $a_1 \text{ op}_1 a_2 \text{ op}_2 \dots a_{n-1} \text{ op}_n a_n$ ’ であれば ‘ $a_1 \text{ op}_1 a_2 \text{ and } a_2 \text{ op}_2 a_3 \dots \text{ and } a_{n-1} \text{ op}_n a_n$ ’ と式の評価が先頭から行われます。

■論理演算式: 比較の演算式で得られた結果を用いて論理演算を行う式です：

—— 論理演算式の構文定義 ——

論理和検証式	::=	論理積検証式 論理和検証式 "or" 論理積検証式
論理積検証式	::=	否定検証式 論理積検証式 "and" 否定検証式
否定検証式	::=	比較式 "not" 否定検証式

これらの演算式によって Boolean 値が返却されます。なお、Boolean の True は整数の 1, False は整数の 0 と等価で、このことを利用した算術演算で if 文の代用にすることが可能です。

■条件演算式: この演算式は if 節による条件分岐を含む式で、アリティが 3 の演算子とも言えます：

条件演算式の構文定義

```
検証式 ::= 論理和検証式 ["if" 論理和検証式 "else" 一般式]
一般式 ::= 條件式 | λ-式
```

この構文は ‘x if y else z’ の形式で、y が False または 0 の場合は z、それ以外は x が返却されます。なお、if 節には else 節が必ず含まれていなければなりません。

■λ-式: lambda-式は無名函数を構成する構文です：

条件演算式の構文定義

```
λ-式 ::= "lambda" [パラメータの列]: 一般式
旧 λ-式 ::= "lambda" [パラメータの列]: 旧一般式
```

■式の列: 式をカンマ “,” で区切った列はタプルになります。Python Reference Manual ではこれを「式の列」と呼びます：

式の列の構文定義

```
式の列 ::= 式 ( "," 式 )* [","]
```

このことを簡単な例で確認しておきましょう：

```
>>> True, True
(True, True)
>>> True, True,
(True, True)
>>> a=True, True,
>>> type(a)
<type 'tuple'>
```

函数 type() で確認した結果から判るように「式の列」はタプルになっています。

3.14 単純文

単純文は、その文全体を一つの論理行内に収めることができる文です。Python では複数の単純文をセミコロン “;” で区切って続けることができます：

単純文の構文定義

```
単純文 ::= 式文
          | 代入文
          | 引数付き代入文
          | pass 文
          | del 文
          | print 文
          | return 文
          | yield 文
          | raise 文
          | break 文
          | continue 文
          | import 文
          | global 文
          | exec 文
```

■**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：

式文の構文定義

```
式文 ::= 式の列
```

■**代入文**：名前にオブジェクトを束縛するために用いられます：

代入文の構文定義

```

代入文 ::= (標的列 "=")+ (式の列 | 生成式)
標的列 ::= 標的 ("," 標的)* [","]
標的   ::= 識別子
        | "(" 標的列 ")"
        | "[" 標的列 "]"
        | 属性参照
        | subscription
        | スライス

```

■**assert 文**: プログラム中にデバッグ用の仮定を仕掛けるための手法を提供します:

assert 文の構文定義

```
assert 文 ::= "assert" 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<>: raise AssertionError
```

また、引数が二つの assert 文 ‘assert <式1> <式2>’ は以下の if 文と等価です:

```
if __debug__:
    if not 式<1>: raise AssertionError 式(<2>)
```

■**pass 文**: null 操作の文で文字通り「何もしない」文です:

pass 文の構文定義

```
pass 文 ::= "pass"
```

この文は構文的に文が必要とされても何も実行したくないときに用います。

■**del 文**: オブジェクトの削除を行う文です:

del 文の構文定義

```
del 文 ::= "del" 標的列
```

標的列に対する削除では、指定した列の左端の対象で指示されるオブジェクトから右端の対象で指示されるオブジェクトへと再帰的にオブジェクトの削除を行います。

■**print 文**: オブジェクトの値を標準出力に対して出力する文です。

print 文の構文定義

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

■**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 文の引数として関数やメソッドが返却すべき値を記載します：

return 文の構文定義

```
return 文 ::= "return" [式の列]
```

■**yield 文**: 生成関数の定義のときのみに利用されます。このとき、yield 文を用いるだけで生成関数が定義されます：

yield 文の構文定義

```
yield 文 ::= yield 式
```

■**raise 文**: 例外の送出を行う文です：

raise 文の構文定義

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

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

break 文の構文定義

```
break 文 ::= "break"
```

■**continue 文**: break 文と同様に引数を持たない文で、for 文や while 文による反復処理の継続で用います。

continue 文の構文定義

```
continue 文 ::= "continue"
```

■**import 文**: モジュールの読み込みに用いられる文です：

import 文の構文定義

```

import 文      ::= "import" モジュール ["as" 名前]
                  ("," モジュール ["as" 名前])* 
                  | "from" 関係モジュール "import" 識別子
                  ["as" 名前] ("," 識別子 ["as" 名前])* [","] ")"
                  | "from" モジュール "import" "*"
モジュール     ::= (識別子 ".")* 識別子
関係モジュール ::= "." モジュール | ".+" 
名前          ::= 識別子

```

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

■**future 文**: 将来の Python のリリースで利用可能になるような構文や意味付けを行うための指示句です:

future 文の構文定義

```

future  ::= "from" "__future__" "import" 機能 ["as" 名前]
           ("," 機能 ["as" 名前])* 
           | "from" "__future__" "import" "(" 機能 ["as" 名前]
           ("," 機能 ["as" 名前])* [","] ")"

```

future 文はモジュールの先頭に置かなければなりません。ここで future 文よりも先行して置ける内容に、文書文字列、注釈、空行と他の future 文に限定されます。

■**global 文**: コードブロック全体で維持される宣言文で、後続の識別子を大域変数として扱われることを指示する文です:

global 文の構文定義

```

global 文  ::= "global" 識別子 ("," 識別子)*

```

ここで global 文で宣言する名前はプログラム中では global 文に先行して配置されではありません。また、for 文での反復処理制御用の変数の名前、class 文によるクラスの定義や関数定義、import 文内で global 文で宣言した名前を仮変数として用いてはなりません。

■**exec 文**: Python コードの動的な実行に関する文です:

exec 文の構文定義

```
exec 文 ::= "exec" 識別子 ("," 識別子)*
```

3.15 複合文

Python の複合文は他の文やそのグループが含まれる文の構造で、他の文の実行と制御に影響を及ぼします。複合文は通常、複数行で構成されますが一行に纏めた書き方が可能な場合もあります。

以下に複合文の構文定義を示しておきます：

複合文の構文定義

```
複合文 ::= if 文
          | while 文
          | for 文
          | try 文
          | with 文
          | funcdef 文
          | classdef 文
          | decorated 文
一揃いの文 ::= 文の列 NEWLINE
               | NEWLINE INDENT 文 + DEDENT
文      ::= 文の列 NEWLINE | 複合文
文の列 ::= 單純文 (";" 單純文)*[";"]
```

ここで示すように複合文は、条件分岐、反復処理、例外処理、関数定義とクラス定義で用いられます。

3.15.1 条件分岐に関する複合文

■if 文：条件分岐を行うための文です：

if 文の構文定義

```
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このif文のみです。

3.15.2 反復処理に関する複合文

■while文: 後述のfor文と並び反復処理を行うために用いる文で、与えられた条件の真理値がTrueのときだけwhile文内部の処理を行います:

————— while文の構文定義 —————

while文 ::= "while" 式 ":" 一揃いの文

■for文: 前述のwhile文と並び反復処理を行うために用意された文です。for文は式の列から標的列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文の構文定義 —————

for文 ::= "for" 標的列 "in" 式の列 ":" 一揃いの文

3.15.3 例外処理に関する複合文

■try文: Pythonの例外を処理するために用意された文です:

————— try文の構文定義 —————

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

3.15.4 隠蔽に関する複合文

■with文: ブロックの実行をコンテキストマネージャで定義されたメソッドで覆うために用いられます:

with の構文定義

with 文 ::= "with" with の項目 (", with の項目)* ":" 一揃いの文
 with の項目 ::= 式 ["as" 標的]

3.15.5 定義に関連する複合文

■**函数定義:** 函数やメソッドの定義のための文です.

函数定義の構文定義

函数定義 ::= "def" "(" [パラメータ列] ")" ":" 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 "(" [引数の列] [","] ")"
 NEWLINE
 付点名 ::= 識別子 ("." 識別子)*
 パラメータ列 ::= (パラメータ定義 ".")*
 ("*" 識別子 [, "*" 識別子] | "*" 識別子
 | パラメータ定義 [","])
 副リスト ::= パラメータ ("," パラメータ)* [","]
 パラメータ ::= 識別子 | "(" 副リスト ")"

函数定義は Python で実行可能な文です. ただし, 函数定義が函数本体を実行するものではなく, 函数が呼び出されたときのみ函数本体が実行されます. なお, 函数定義を行うと, 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:** クラスの定義ための文です.

函数定義の構文定義

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

このクラス定義文も Python で実行可能な文です. このクラス定義では最初に継承リストがあればリストの評価を行います. ここで継承リストの各要素の評価結果はクラスオブジェクト, あるいはサブクラス可能なクラス型でなければなりません. それから実行フレーム内部にて局所名前空間と大域名前空間を用いてクラス内の変数への束縛が行われま

す。すると実行フレームは無視されますが、局所名前空間は保持され、それから基底クラスの継承リストを用いてクラスオブジェクトが生成され、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。

第4章

プログラム開発の指針

4.1 はじめに

SageMath 上でプログラムを行う指針は Python のおおよそ指針に従います。Python でのプログラミングの指針は「**PEP, Python 拡張提案書**」と呼ばれる一連の文書で定められ、その規定は機能的なものに限定されず、プログラムの構造を視覚化するための字下げ、変数やクラス等の名前の書き方や文書文字列 (docstring) と呼ばれるプログラム内部に埋め込まれる文書の書き方等も包含されています。そして、SageMath 上で作成したプログラムを SageMath と一緒に配布するのであれば GPLv2+ か BSDL 等の制限の弱いライセンスの下で公開する必要があります。この章ではまず PEP について解説することから始めましょう。

4.2 PEP(Python Enhancement Proposal) について

Python の開発は **PEP(Python Enhancement Proposal: Python 拡張提案書)** と呼ばれる設計書に基いて開発が進められます。ここで PEP がどのようなものであるかは PEP-1 の「**PEP の目的と指針**」^{*1}にて規定されていますが、まず PEP は新機能の提案や課題に対するコミュニティの意見の集約、Python に取り込まれる機能の設計や決定事項の文書化の主要な機構として位置付けられ、PEP の作者がコミュニティの中での同意や反対意見の記録を取る責任を持ち、PEP 自体はテキストファイル形式でバージョン管理されたりポジトリに保管されます。PEP の種類には Python コミュニティに対して情報提供するもの、Python の新機能やプロセスと環境等を説明するものがあり、それらには技術的な仕様・機能に関する仕様、その機能が必要とされる論理的な理由が記載されていなければなりません。そして PEP はその目的や機能から次の三種類の文書に分類されます：

1. **標準化過程 PEP**: Python の新しい機能や実装に関する文書。
2. **情報 PEP**: Python の設計上の課題、一般的な仕様等の情報を解説する文書で、新機能の提案は行いません。また、この PEP に従う必要はありません。
3. **プロセス PEP**: Python を取り巻く工程を記述したもので、標準過程 PEP に似ていますが、Python 言語それ自体以外の領域に適用され、情報 PEP と違って推奨以上の拘束力を持ちます。具体的には手続、指針、意思決定方法等で、PEP を規定する PEP もプロセス PEP になります。

ここで Python がどのような言語で、何を目指しているかを最もよく体現している PEP があります。これが PEP-20，**The Zen of Python** と名付けられた詩で、Python の

^{*1} 原文は <http://www.python.org/dev/peps/pep-0001/>、その翻訳は <http://sphinx-users.jp/articles/pep1.html>

シェルで `import this` と入力することで読むことができます。とにかく面白いので下手ながら私が翻訳したものも載せておきますが、PyJUG に日本語訳があるので^{*2}、そちらの訳も参照して下さい:

美は醜に勝り。(Beautiful is better than ugly.)
直は喻に勝る。(Explicit is better than implicit.)
素は組に勝る。(Simple is better than complex.)
組は混に勝る。(Complex is better than complicated.)
平坦は入籠に勝る。(Flat is better than nested.)
疎は密に勝る。(Sparse is better than dense.)
読み易さは重要なり。(Readability counts.)
法を破らんが為の特例は特例に能わず。
(Special cases aren't special enough to break the rules.)
然れど実は理を破る。(Although practicality beats purity.)
エラーは黙認すべからず。(Errors should never pass silently.)
鎮めたるを除きて。(Unless explicitly silenced.)
五里霧中、恣意に囚われること勿れ。
(In the face of ambiguity, refuse the temptation to guess.)
只一無二の大道あらん。
(There should be one— and preferably only one —obvious way to do it.)
然れど、汝、彼の和蘭人にあづんば、先ずは道難し。
(Although that way may not be obvious at first unless you're Dutch.)
成すべきを直ちに成せ。(Now is better than never.)
然れど*拙速*よりも待つが果報もあり。(Although never is often better than *right* now.)
語り得ぬ実装は悪き觀念ならん。
(If the implementation is hard to explain, it's a bad idea.)
語り得る実装は善き觀念ならん。
(If the implementation is easy to explain, it may be a good idea.)
名前空間は轟々たる大乗なり、奮励して用うべし!
(Namespaces are one honking great idea – let's do more of those!)

さて、いかがでしょうか？ この詩から Python は簡素さを旨とする実務的な言語であることが伺えるでしょう。それでは幾つかの Python の PEP に従ってプログラム作成の流儀を細かく見ることにしましょう。

4.2.1 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

^{*2} <http://www.python.jp/Zope/Zope/articles/misc/zen>

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ということと、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、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： 字下げの一段は空白文字 (space) の 4 文字分とし、空白文字とタブ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 3.0 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くことで文字コードを明確に示します。

■import 文の書き方： import 文で読み込むパッケージは個別に読み込むべきです。パッケージ内部の import 文で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 *
```

このような 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何等の識別子が付かないために既存の名前と衝突が生じる可能性があります。もし既存の名前を置換えることが目的であれば問題はありませんが、そうでなければ名前の衝突が生じないように ‘import test’ でそのまま読み込もうか、あるいは ‘as’ を用いて識別子を変更すべきです：

良い例

```
import test as tst
```

例を補足すると、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*3。

■空白文字の利用について: 余計な空白文字の利用を避けること。演算子や式を合せるために無駄な空白を入れるようなことを避けるべきです。ただし、可読性を高めるために二項演算子の両端に空白文字を一つのみ入れたり、算術演算子の前後に空白文字を入れるべきです。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空白文字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 1 それ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 Subversion, 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わなくても良いことになっています。代表的な命名方法には以下のものがあります:

*3 as 節を付ける際、そのパッケージがプログラム内における立場を明瞭にするグループ（目的、機能、由来等）とすると良いでしょう。

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.

ここで挙げた命名法に加えて短い接頭辞を付けるという Python あまり利用されない様式もあります。たとえば X11 ライブラリの公開函数の命名法では函数名の先頭に ‘X’ を用います。ところで Python ではオブジェクト名が接頭辞、函数名でモジュール名が接頭辞になるために例外的な命名法になります。この命名規則は従来のプログラムとの互換性のための用いるべきで、名前空間をむしろ活用すべきです。

Python で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’ を一文字の変数名として利用することを挙げています。これらは ‘l’ や ‘0’ と混同し易いからです。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^{*4}や長い名前は短縮されるといった制約を受ける可能性があり、それらの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 “_” で語を区切れます。mixed-Case 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。

^{*4} MS-Windows の FAT、VFAT や NTFS、OSX の HFS+ は大文字と小文字の区別をし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 が区別している訳ではありません！

- フィードバック: インスタンスマソッドの第1引数を必ず `self` にし、クラスマソッドの第1引数を必ず `cls` にします。
- 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:** 継承を目的とした場合の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“`_`”を付けないようにします。
- 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るときは、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。

■**プログラミングにおける推奨案:**

- 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は書式を整える場合は括弧を用いる表記にすべきです。
- 例外を補足するときに ‘`except:`’ で受け取るのではなく、どの例外を補足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すること。バグの隠蔽が生じることを避けるためです。
- `string` モジュールではなく文字列メソッドを用いること。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ること。函数 `startswith()` や `endswith()` を用いるべきです。
- オブジェクト同士の型の比較では函数 `instance()` を用い、函数 `type()` 等を使って型を直接比較しないこと。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 UNICODE 文字列の可能性もあります。

- 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。その分、冗長な処理になります。

4.2.2 SageMath の様式

SageMath は PEP をそのまま継承する訳ではありません。この 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：

- プログラム内の字下げに空白 4 文字を用いてタブは使いません
- 関数名が全て小文字であれば文字 “_” を用いて切り分けます
- クラスや主要な関数名では「**らくだのこぶ記法 (CamelCase)**」を用います

最初の字下げですが、Python で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に字下げを行います。この字下げの文字数は Python と同じ空白文字 4 文字が推奨されています。次に、Python では関数名として小文字のみなら ‘set_some_value’ のように記号 “_” を区切記号として使った文字列とするか、‘SetSomeValue’ のように大文字を適宜利用する「**らくだのこぶ記法**」が使えます。ただし、クラスや主要な関数名では ‘PolynomialRing’ のように「**らくだのこぶ記法**」を用いることになっていますが、この基準は絶対的なものではなく、判り易さを重視するために関数名は「**らくだのこぶ記法**」ではなく「**大文字**」を用いることも許容させています。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。この SageMath の命名の指針はあくまでも名前から処理内容が明瞭となることを重視した結果であり、判り易さを犠牲にして守らなければならないような絶対的な指針にしていません。

4.3 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。

4.4 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/
#*****
```

SageMath に限らず Python で文書は非常に重要視されており, SageMath でも「一つの例題は幾千の言葉に優る」とあります。

4.5 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。文書文字列は Python, Lisp, Clojure, Matlab や Yorick では註釈として用いられますが, C や FORTRAN の註釈や javadoc のような特定の書式を持った註釈はプログラムの動作には無関係であるのに対し, 文書文字列はプログラムが動作している間も保持されて必要に応じて参照できる点で大きく異なります。たとえば, MATLAB や Yorick では文書文字列をオンラインヘルプとして活用しています。ここで 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。まず, 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ます。この文書文字列には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 SEE ALSO: 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

に単なる文字列のままにしても昔の計算機ならさておき現在の強力な処理能力を持つ計算機の能力を眠らせておくにはもったいない話です。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ることはできないでしょうか？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で可読性が高いものとは言えません。この点を改良したものとして Markdown や reStructuredText(reST 等と略記) と呼ばれる言語で、テキストファイルである程度の WYSIWYG を実現しており、Python では reST を文書文字列や PEP で標準的に利用することになっています。

4.6 reStructuredTextについて

4.6.1 reStructuredText の概要

reStructuredText^{*5}は名前から予想できるように組版指示（マークアップ）言語です。この reStructuredText は reST, RST, ReST と略称で表記されることがあります。ここでは略記する場合は reST と表記することにします。この reST は記述が容易であり、後述の Docutils を利用して整えた文書にしなくても読み易い文書になることを目的としたプレーンテキストのための言語 (WYSIWYG plane-text markup language) です。

reST は Python の文書の記述で標準的に用いられています^{*6}。ここで関連する PEP としては PEP-256, 257, 258, 287 で、特に PEP-287 が reStructuredText を Python の標準の文書文字列と PEP 等のファイルの標準的な書式として提案しています。この PEP-287 で文書文字列の書式として：

1. 一般人でも読み易いソースコードであること。
2. 通常のテキストエディタでも編集しやすいものであること。
3. モジュールの分析から演繹された情報を含む必要がないこと。
4. 他の主要な組版指示言語に変換するに十分な情報構造を持っていること。
5. 文書文字列でモジュール全体の書き換えがその組版指示言語を煩わしく思わずに行えること。

という要求があり、reST はこれらの要求をみたすものとして提示されています。この言語で記述した文書はプレーンテキストとして記述され、文書の構成も HTML や LATEX の

^{*5} “reStructured Text”ではなく一語の “reStructuredText”です。

^{*6} 文書が reST であることが必須ではありません。文書としてはプレーンテキストであればよいのです（参照：<https://docs.python.org/devguide/documenting.html> の冒頭の註）

文書をプレーンテキストの要素を用いて代用した表記です。この表示は古典的な数式処理、たとえば Maxima でテキスト端末向けに数式をアスキーアート風に表現する方法があります：

Maxima による数式の表示例

```
(%i1) integrate(f(x),x,-inf,inf);
          inf
          /
          [
(%o1)           I f(x) dx
          ]
          /
          -inf
(%i2)x*x-2*x+1;
              2
(%o2)           x  - 2 x + 1
```

この例では函数の積分 $\int_{-\infty}^{\infty} f(x)dx$ と多項式 $x^2 - 2x + 1$ を表示したのですが、共にアスキーワードを使って数式の表現を行っています。reST の文書はちょうどこのアスキーアート風に文書が本来あるべき姿をアスキーアート風に表示したものになっており、まさに WYSIWYG をアスキーアートで実現したものになっています。ただし reST は Docutils が提供する命令を利用することで HTML や TeX 等の組版指示言語の文書に変換が可能であり、それらをレンダリングすることでさまざまな媒体で著者が意図した文書を再現することが可能となっているのです。

なお SageMathCloud ではシェルに用いている Jupyter Notebook の機能を用いて reST に対応した WYSIWYG エディタが実装されており、LaTeX の場合と同様に reST 文書の編集が可能になっています。Docutils で LaTeX や HTML に変換することを必要とせずにその出来上がりを確認しながら文書の編集が可能であり、その利用価値は極めて高いものになるでしょう⁷。

⁷ SMC では組版指示言語 Markdown の編集も可能です。reST と Markdown の両者の比較は <http://www.unexpected-vortices.com/doc-notes/markdown-and-rest-compared.html> を参照のこと。

4.6.2 Docutils について

この reST の構文解析器は Python で記述された文書処理のためのツール群である Docutils に含まれており, reST を HTML, XML, LaTeX, ODF といった他の組版指示言語の文書に容易に変換することができます:

Docutils の命令の対照表

rst2html	HTML 形式への変換
rst2latex	LATEX 形式への変換
rst2man	UNIX の man ファイル形式への変換
rst2odt	ODF 形式への変換
rst2xml	XML 形式への変換

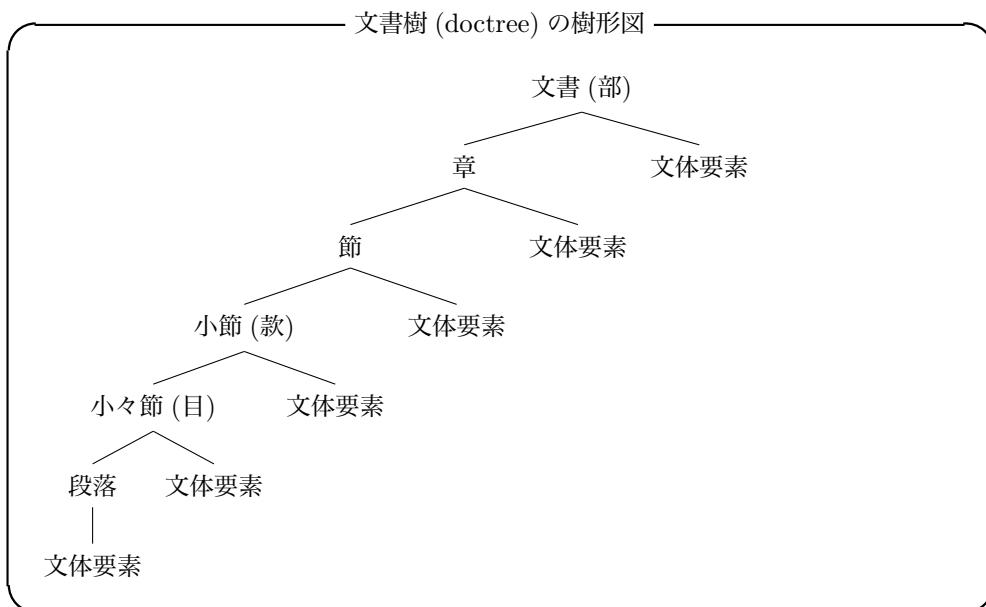
たとえば reST ファイル test.rst を HTML に変換するときは ‘rst2html test.rst’ で行えます. このときの出力は標準出力で行われるのでリダイレクトでファイルに落すといった工夫が必要です. とはいえ reST で一度文書を記述してしまえば, Docutils の命令を使って主要な組版指示言語のファイルに変換可能であり, さまざまな利用環境で最適の文書の表示を行うことが可能です. また通常の組版指示言語よりも大幅に簡易化された仕様で, テキストファイルとしても他の組版指示言語でレンダリングされた結果に近い表現となっていることは, どのような環境でも作成可能なだけではなく, 文書の作成の一元化, 過剰に華美にならずに無駄な労力を費す必要がないといった利点があります.

なお, SageMath には Docutils や後述の Sphinx がインストールされているために, これらの Docutils 等のパッケージや関連するアプリケーションを自分で入れる必要が全くありません. このように基本的な環境が一通り揃えられていることが SageMath の大きな利点なのです.

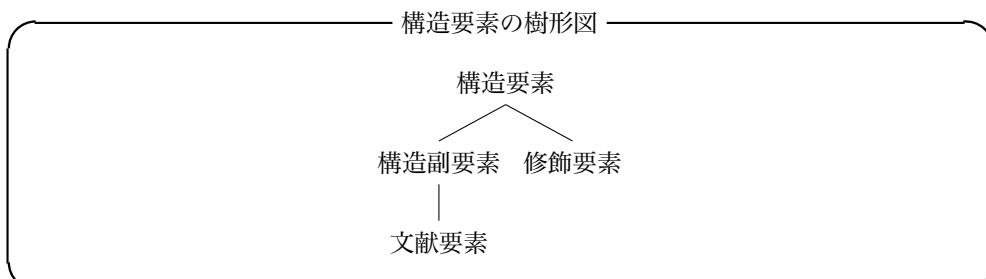
4.6.3 reST の文の構造

reST は HTML や LATEX のような組版指示を含むプレーンテキスト文書ではありません. ただ, WYSIWYG 的にプレーンテキストを構築すると構文解析器によって意図した文書が構成されるというものです. ここでは reST の文書の構造について述べることにします.

reST での文は Docutils の構文解析器によって「文書樹 (doctree)」として分析されます。この文書樹は reST の文の構造を示すもので、その樹形図は以下のものになります：



このように reST の文章は部, 章, 節, 小節 (款), 小々節 (目) という章立てに対応する「区分 (section)」があり、各区分には本文を包含する「文体要素」を配置することができます。また章立てに対応する区分はその文書の区分の入れ子の深さでその章立ての立ち位置 (部, 章, 節等) が定まります。そしてこの章立て自体にも、その章立ての表題等を表現するための構造、すなわち「構造要素 (structural element)」で構成されます。この構造要素の樹形図を次に示しておきます：



この構造要素の樹形図で示すように構造要素には構造副要素、文献要素と修飾要素から構築されます。そして、これらの構造要素として具体的には以下のものがあります：

構造要素

構造要素	document, section, topic, sidebar
構造副要素	title, subtitle, decoration, docinfo, transition
文献要素	address, author, authors, contact, copyright, date, field, organization, revision, status, version
修飾要素	footer, header

これらの要素の概要を述べると、まず「**構造要素**」が区分の構造を定めます。それから「**構造副要素**」が見出、表題や見出の飾り等を表現します。そして「**文献要素**」が著者の情報等を表記し、「**修飾要素**」が文書のヘッダーやフッターを表現します。このように構造要素は区分の枠組を定める働きを行います。

reST 文書の本文を構成する要素は「**文体要素 (body element)**」と呼ばれ、「**文体副要素**」、「**複合文体要素**」、「**単純文体要素**」から構成されます。ここで本文を直接含むことがあるのが文体副要素と単純文体要素であり、複合文体要素は構造要素と文体要素の中間的な存在です。また単純文体要素は必ず空行、あるいは幾つかのテキストデータを含みます。文体要素に含まれる要素を以下に示しておきます：

文体要素

文体副要素	attribution, caption, classifier, colspec, field_name, label, line, option_argument, option_string, termtitle, subtitle, decoration, docinfo, transition, definition, definition_list_item, description, entry, field, field_body, legend, list_item, option, option_group, option_list_item, row, tbody, tgroup, thead
複合文体要素	admonition, attention, block_quote, bullet_list, caution, citation, compound, container, danger, definition_list, enumerated_list, error, field_list, figure, footnote, hint, important, line_block, note, option_list, system_message, table, tip, warning
単純文体要素	comment, doctest_block, image, literal_block, math_block, paragraph, pending, raw, rubric, substitution_definition, target

行内 (inline) 要素は要素内部に直接本文を含むことができる要素で、単純文体要素に包含されます：

行内要素

abbreviation, acronym, citation_reference, emphasis, footnote_reference, generated, image, inline, literal, math, problematic, reference, strong, subscript, substitution_reference, superscript, target, title_reference, raw

この reST の文書樹は構文解析器向けの表現で、reST 文書の構造を明確にするものですが、利用者はこういった文書の構造を反映するように文書を作成している訳ではありません。むしろ、ワープロで文書を作成するような感覚で文書を生成しているのです。以降は具体的に reST の構成要素について述べることにします。

4.6.4 reST の構成要素

ここでは実際に reST 文書を作成するにあたって必要な機能について具体的に述べることにします。reST は HTML や L^AT_EX のような組版指示言語ですが、プレーンテキストで WYSIWYG 的な編集が行えます。この文書の構成は文書樹として構文解析器で分析されますが、実際の編集は非常にワープロ風です。ただし、そのプレーンテキストによる表現が reST の要素に対応してゆきます。この要素の区切として空行と空白文字が大きな意味を持ちます。

構成要素としての空行

空行は構成要素の区切りとして用いられます。したがって、構成要素の切り替えが必要な場合には必ず一つ空行を入れてから新たに構成要素の記述を行うことになります。この空行が有効に利用されていなければ、構成要素の区切が構文解析器によって認知されないことに繋り、HTML や L^AT_EX 等への変換を行った際に自分の意図したとおりの文書が得られない原因になります。この文書で要素の記述を示すときに水平線の破線で改行を表現します。水平線が二本存在する場合は上の行と下の行に二つの改行が存在することをから、一つの空行が存在することを示します。

構成要素としての空白文字

reST 文書では要素内部の区切として空白文字が用いられます。たとえば箇条書のときの番号と項目の区別のため、あるいは明示的な指示で用いられる記号“..”と具体的な指示や引数との同一行内での区別のために用いられます。また、文体要素間の構造を明確にするためにも空白文字を用いた字下げが用いられています。この構造を明瞭にするために用

いられる字下げでは同一要素内の各行の始点をこの字下げの水準に合わせなければなりません。ここで、この文書では要素の記述を示すときに縦線で揃えるべき水準を示し、破線の場合はその破線の前に空白文字が入ることを意味しますが、実線の場合は空白文字を入れる必要性はなく、単なる水準以上の意味を持ちません。

reST の構成要素

reST の構成要素を次に示します:

reStructuredText の構成要素

- 文体要素 (Body element)
 - 段落 (Paragraph)
 - 見出と章立
 - 簇条書 (List)
 - そのままの区画 (Literal block) と引用 (Quote)
 - 表 (Table)
 - 脚注 (Footnote)
 - 引用 (Citation)
 - ハイパーリンク (Hyperlink)
 - 指示 (directive)
 - 代入定義 (Substitution definition)
 - 註釈 (Comment)
- 行内組版指示 (Inline Markup):
 - 強調
 - ハイパーリンク参照
 - 脚注参照
 - 引用参照
 - 代入参照
 - 独立ハイパーリンク
- 単位 (Unit):
 - 長さの単位
 - 百分率の単位

reST は上のように文体要素 (Body Element), 行内組版指示 (Inline Markup) と単位 (Unit) に分類されます。基本的に文体要素が文章の大本を構築し、行内組版指示が reST 文書の行の強調等の飾りやハイパーリンクといった機能を与えることになります。以下、各項目について順番に解説します。

4.6.5 文体要素 (Body Element)

文体要素は reST 文書の中核を成す要素で、通常の文書作成で入力したり整形したりする文体そのものに該当します。ただし、文字の強調やハイパーリンクといった機能は文体要素ではなく行内組版指示要素が受け持ちはます。この文体要素はプレーンテキストを利用したアスキーアート風の WYSIWYG で編集を行うことを特徴としています。ただし、ここで明示的指示と呼ばれる組版指示があり、こちらはアスキーアートで表現が困難なもの、たとえば、画像の挿入といったものは先頭に “..” を配置し、それから具体的な指示を記載するという命令的な指示を採用しています。このなかで最も多機能なものが「指示 (directive)」と呼ばれるものです。このディレクティブについては別にその詳細を説明することにし、ここでは概要のみを述べることにしています。

段落 (Paragraphs)

reST 文書で最も基本的な要素です。一行以上の空行で区切られた文字列から構成され、文書樹では単純文体要素 paragraph として解釈されます。Python では字下げもプログラムの重要な要素ですが、このことは reST でも同様で同じ段落の全ての行の字下げの水準は左揃えで全て同一水準でなければなりません。次に一例を示しておきます：

段落はこのような文字の羅列です。そして、段落の区切りは空行です。だから、ただの改行だけでは段落の区切りにはなりません。||

この例では 7 行にわたって記述されていますが、間に空行が一つだけしかないために reST の構文解析器による解釈では段落 (paragraph) が二つの文書になります：

段落の解釈例

段落はこのような文字の羅列です。そして、段落の区切りは空行です。
だから、ただの改行だけでは段落の区切りにはなりません。

見出と章立

見出と章立は文書樹で title と section に分析されます。見出の記述は見出にする文字列の直上と直下の段に文字列長よりも長く記号 “=”, “-” 等の非英数字の 7 ビットアスキーワードから一つを選択し、その文字を見出行と同程度か、より長く挿入します。ここで reST の章立は、部、章、節、小節 (款)、小々節 (目) と段落で、見出もこれらに対応するものになります。したがって、この方法で指示できる見出は部、章、節、小節 (款)、小々節 (目)、段落に

対応する 6 種類であり、特に部と章の見出は中央揃えが行われます。この章立は内部的にはその文書の文書樹にそのまま対応することになります。だから指示自体は 7 種類以上でなくても章立の関係から意味のある指示にはなりません。このこともあって実質的に 6 種類のアスキー文字のみが使えます。さらに見出とアスキー文字との関係は reST 文書内部で見出として現われた順番で文書樹が定まるために、文字自体と章立が直接関係付けられている訳ではありません。

見出と他の reST の構成要素の間には空行を置きます。また見出に続けて置かれた組版指示のない行は通常の段落 (paragraph) として扱われます。以下に具体例とその HTML 文書への変換結果を示しておきましょう：

```
EXAMPLES
~~~~~
example1
=====
example2
=====
example3
!!!!!!!
example4
#####
example5
$$$$$$$
```

EXAMPLES
~~~~~
example1
=====
example2
=====
example3
!!!!!!!
example4
#####
example5
\$\$\$\$\$\$\$

図 4.1 reST の見出  
(HTML)

これを `rst2html.py` を使って HTML に変換すると図 4.1 の結果が得られます。ここで L^AT_EX への変換は `rst2latex` で行いますが、HTML と同等の結果が得られます。基本的に HTML と L^AT_EX への変換はほぼ同等の結果が得られますが、一部の機能で L^AT_EX が対応していない場合もあり、HTML の方がそのような問題は少ないようです。文書の再現性では SageMathCloud に実装された reST エディタに付属の viewer が最も明瞭な表示を行なっていると思います。reST 文書の「明示的な指示」、特に指示 (directive) によるものは、その reST 文書だけで目的とする文書の姿が判り難いために必要に応じて L^AT_EX への変換結果や SMC の reST エディタによるレンダリングの結果を示すことになります。

### 箇条書 (List)

reST の箇条書には先頭に記号付く数字なしの箇条書、数字が付く数字付の箇条書と定義箇条書と呼ばれる箇条書の三種類があります。文書樹で数字なしの箇条書の複合文体要

素 bullet_list, 同様に数字付きの箇条書が enumerate_list になります。これらの箇条書双方の項目と記載事項が文体副要素 list_item に対応します。reST では、行の先頭に記号 “-”, “*”, “+” のいずれかを置けば番号なしの箇条書、数字を置けば数字付の箇条書になります。ここで番号付箇条書では次の番号表記が選べます：

数字表記		
種類	番号表記	範囲
数字	0, 1, 2...	0 から上限なし。
アルファベット (大文字)	A, B, C...	A から Z まで。
アルファベット (小文字)	a, b, c...	a から z まで。
ローマ数字 (大文字)	I, II, III...	I から MMMMCMXCIX(4999) まで。
ローマ数字 (小文字)	i, ii, iii...	i から mmmmcix(4999) まで。

番号表記では通常の数字とローマ数字が選べます。開始番号は上記の表の範囲内であれば何でも構いません。また番号の飾りとしてピリオド “.”, 括弧 “()” と半括弧 “)” の何れか一つのみを併用することができます。そして番号の自動振当ても記号 “#” で行えます。ピリオドなどの番号飾りを行っている場合は文字飾りは表記し、番号のみを記号 “#” で置き換えるなければなりません。ただし、この略記方法は番号のみで数字なしの箇条書に用いてはなりません。ここで番号なしと番号付きの箇条書の基本的な記述を以下に示します：

番号なしと番号付の箇条書の記述	
‘表記記号’	‘項目’
	-----
	(文体要素)+

ここで縦破線が行の空白文字による区切、同様に横破線が改行を示します。また単引用符で括られた文字列は変数を意味し、実際の表記では引用符なしで文字列を記載することを意味します。ここでの表記記号は番号なしであれば “-” 等の番号なしで用いる記号、番号付きであれば “1” 等の番号記号になります。そして縦破線の直後の要素は字下げの水準を揃えることを意味します。この場合は項目と次の文体要素の先頭を揃えることを意味し、文体要素には ‘( )+’ と記載されていますが、この場合は文体要素を一つ以上配置することができる意味します。ただし、空白文字以外の要素は字下げの水準をその文体要素の構造に対応して合せることになります。

番号なしの箇条書と番号付を続ける場合、これらの箇条書は異なる区分になるために空行を入れる必要がありますが、番号付であれば自分で入れたものと自動の箇条書は続けて記述することができます：

```
* これは番号なしの箇条書の
* 一例です。

1. これが番号付の箇条書の
2. 一例です。
#. これが自動番号振りの箇条書の
#. 一例です。
```

箇条書も字下げを用いており、ある箇条書に含まれる項目は同じ字下げの深さを持つていなければなりません。そして、箇条書に包含される箇条書は空白行で区切られ、親の箇条書よりも深い字下げでなければなりません：

```
* これは
* 箇条書です。

    * これが子箇条書で、
    * このように字下げを
    * 必要とします。

* で、親箇条書をこのように
* 続けることもできるのです。
```

「**定義箇条書**」は定義で用いられる箇条書で、定義を記述して次に改行を入れ、それから字下げを行って定義内容を記載します。文書樹では複合文体要素 definition_list に相当し、記載事項は文体副要素 definition_list_item になります。なお、文体副要素は項 (term) と定義から構成されます：

#### 定義箇条書の記述

---

```
項
-----+
| 定義
-----+
| (文体要素)+
```

---

実例を示します：

```
| 白猫白い猫のこと
|
|. 黒い猫ではない
|. 三毛猫黒
,
| 茶, 白の三色の猫のこと.
```

ここで定義の記述は同一水準の字下げを行った複数の行で記述ができますが、reST 文書では一行として扱われます。次に LATEX への変換を示しますが、定義行の文字列が太文字で表示されます：

— 定義箇条書の LATEX への出力例 —

```
白猫 白い猫のこと。黒い猫ではない。  
三毛猫 黒、茶、白の三色の猫のこと。
```

なお、HTML への変換では reST 文書と同様に定義と記述に改行が入ります。ただし字下げはなくなります。このように多少の違いがあるものの HTML と LATEX でほぼ類似の結果が得られます。

フィールドリストは定義箇条書と類似しています。フィールドリストは複合文体要素 field_list として分析されます。ただし定義リストと異なる点は先頭の項（文体副要素 field_name に対応）をコロン ":" で括る点です：

field_list_item の表記

```
:'field_name': | field_body  
-----+-----  
| (文体要素)+
```

実例を以下に示しておきます：

```
白猫  
::: 白い猫のこと。黒い猫ではない  
     .三毛猫  
::: 黒、茶、白の三色の猫のこと。
```

これを LATEX でレンダリングした結果を示しておきます：

LATEX によるレンダリング結果

```
白猫: 白い猫のこと。黒い猫ではない。  
三毛猫: 黒、茶、白の三色の猫のこと。
```

と、このように定義箇条書に類似した結果が得られます。

オプションリストも定義リストに類似していますが、こちらは先頭項と後続項の間に空白文字を二文字以上配置することと、先頭項が定義箇条書のように太文字にされることが

ない点で異なります。また、空白文字が二文字現われるまではオプションの記述として構文解析器で解釈されます。またオプションリストは複合文体要素 option_list として解釈され、項目が文体副要素 option_list_item として解釈されます。この option_list_item の表記を以下に示しておきます：

option_list_item の表記	
option_group	記述
	(文体要素)+

ここで文体副要素 option_group は次の記述になります：

option_group の表記	
項	(引数)+

option_group は正確には option_string と呼ばれる項の部分と option_argument と呼ばれるオプションの引数で構成されます。ただ、これらの項の区切は空白文字一文字のみで行われます。空白文字二文字が現われた時点で option_group としての構造が終るからです。

### そのままの区画 (literal block) と引用 (Quote)

「そのままの区画」は、単純文体要素 literal_block として解釈され、区画の前の段落の末尾に文字列 ‘::’ を配置することで開始します。この区画の先頭行として空行をまず挿入し、それから区画の記述を行います。そのままの区画には字下げを伴うものと字下げは共なわないものの特殊記号を配置することで、引用として扱うものの二種類があります：

字下げ付きのそのままの区画	
段落 (paragraph)+::	
	literal_block

引用としてのそのままの区画の場合は、文字列 ‘::’ のある段落の字下げの水準にあわせて記号を配置します：

引用としてのそのままの区画	
段落 (paragraph)+::	
‘記号’	literal_block

---

‘記号’に配置できるもの

---

! " # \$ % & ' ( ) * + , - . / : ; < = > ? @ [ \ ] ^ _ ' { | } ~

---

いずれの場合にせよ、文字列 ‘::’ を末尾に配置する行が、文字列 ‘::’ に先行する段落との間に空白文字がない場合、空白文字がある場合、段落が空白文字だけの場合で段落の表示が異なります。具体的には文字列 ‘::’ の前に空白文字があればコロン ‘:’ の表示はありませんが、直前に空白文字が入らなければ先頭のコロン ‘:’ のみが残ります。また、そのままの区画内に組版指示が存在としても無視されます。そのためにプログラムの記述や文書の引用に向いています。ここでは最初に字下げのあるそのままの区画の例を示しておきます：

```
| 例題
|::
| 10 *(23 + 4*5)
| 3 **4** 5空白文字を入れないと例題
|   ‘::’になります。区画内では数式のアスタリスクが強調として解釈されることはありません。例題
|::
| 10 *(23 + 4*5)
| 3 **4** 5空白文字を入れると例題になります
|   ‘ ’ ‘:’ .
|::
| 10 *(23 + 4*5)
| 3 **4** 5    だけの行は表示されません
|::
|   ‘ ’ .
```

つぎに LATEX によるレンダリングの結果を示しておきます。文字列 ‘::’ と段落の間の空白文字の配置による違いに注目してください：

## ・ $\text{\LaTeX}$ によるレンダリングの結果

### 例題：

$$10 * (23 + 4*5)$$

3 **4** 5

空白文字を入れないと‘例題：’になります。

区画内では数式のアスタリスクが強調として解釈されることはありません。

## 例題

$$10 * (23 + 4 * 5)$$

3 **4** 5

空白文字を入れると「例題」になり「：」がありません。

$$10 * (23 + 4 * 5)$$

3 **4** 5

$\therefore$  だけの行は表示されません.

引用としては他に「区画引用 (block quote)」があります。これは複合文体要素 `block_quote` として解釈されるもので、段落 (paragraph) 等の次に空行を置き、直前の要素に対して適度な水準で字下げを行ったものです:

### block quote の表記

前の要素

このようにそのままの引用の文字列だけの形態に似ています。

## 表 (Table)

reST の表には二種類の表があります。どちらも複合文体要素 table になりますが、一つはアスキーアート風に罫線を描いた格子表 (grid table) と呼ばれるもので、こちらは非常に複雑な表もアスキーアート風に作成することができます。もう一つは列主体の単純な構造を持つ表で、簡素な表 (simple table) と呼ばれる表の二種類です。ここで表の罫線は記

号“-”を主体に用い、罫線同士の交差は記号“+”を用います。また罫線として記号“=”も使えますが、こちらは表の項目と表本体の分離記号として用いられます。

項目行, 列1	項目 2	項目 3	項目 4
項目行はオプション()			
+-----+-----+-----+-----+	+-----+-----+-----+-----+	+-----+-----+-----+-----+	+-----+-----+-----+-----+
行 1, 列 1	列 2	列 3	列 4
+-----+-----+-----+-----+	+-----+-----+-----+-----+	+-----+-----+-----+-----+	+-----+-----+-----+-----+
行 2 セルは複数列でも		OK.	
+-----+-----+-----+-----+	+-----+-----+-----+-----+	+-----+-----+-----+-----+	+-----+-----+-----+-----+
行 3	セルは複数   - 表のセルは		
+-----+-----+-----+-----+	+-----+-----+-----+-----+	+-----+-----+-----+-----+	+-----+-----+-----+-----+
行 4		- 文体要素を	
+-----+-----+-----+-----+	+-----+-----+-----+-----+	+-----+-----+-----+-----+	+-----+-----+-----+-----+

罫線で囲まれた表は等幅フォントで罫線が正確に閉じていなければなりません。また、  
LaTeXへの変換で複数のセルにまたがるセルを有する表の変換はサポート外になっています。  
ただし、HTMLへの変換は問題ありません。

罫線で囲まれた表に対して簡素な表は空白文字を使って列を明示的に分離するだけです。この簡素な表でもセルの結合は可能です。この表の例を以下に示しておきます：

===== ===== ======入力出力		
A	B	A and B
T	T	T
T	F	F
F	T	F
F	F	F
=====	=====	=====

レンダリングの結果を図 4.2 に示します。こちらは格子表のように見たままの表を構成するのとはいささか異なります。表の開始行で記号“=”を使って表の列を定義し、空白文字を使って列の区切りとします。この開始行は表の終了行でも用いられます。ヘッダ部分の区切りには記号“-”を用います。ヘッダ部分のセルの結合は開始行の列の区切りを表現する空白文字と同じ列に空白文字を置かなければなりません。

入力		出力
A	B	A and B
T	T	T
T	F	F
F	T	F
F	F	F

図 4.2 レンダリング結果

いようにヘッダの項目を記載します。ヘッダの終了は開始行の記号“=”を記号“”で置き換えた行を配置することで示します。項目に箇条書を含めることも可能です。この場合は1列目に関しては箇条書の先頭行の1列目のみに項目を記載して他は空白文字のみとし、2列目以降に行単位で箇条書の記述を行います。番号付の箇条書では記号“#”による番号の自動割当も可能です。

このように reST の表は HTML や L^AT_EX のような組版指示言語と比べてもほとんど「落書き」のように直観的な方法で比較的複雑な表さえも容易に記述することができるのです。

### ハイパーリンク (Hyperlinks)

ハイパーリンクには reST 文書外部に標的を持つ外部リンクと reST 文書内部に標的を持つ内部リンクの二種類があります。後述の「脚注」と「引用」は内部リンクに類似したものになります。

ハイパーリンクの表記

---

..		‘名称’	:	
- - +	- - - - -	(文体要素)	+ - - - -	

---

### 脚注

脚注は脚注をつけられる側と脚注の対で構成され、reST 文書中に ‘[ ]_’ で括られた番号や記号からの対応する註へのハイパーリンクを表現します、そして中からのリンクを受けるある種の箇条書がここで述べる本来の註で、複合文体要素 footnote として解釈されます：

脚注の表記

---

..		[ ‘ラベル’ ]	‘脚注’	
- - +	- - - - -	(文体要素)	+ - - - -	

---

ここでラベルに記載可能なものは数字、記号“*”と“#”を除く文字列です。記号“#”と“*”は構文解析器側による自動番号割当と記号割当に関連します。そして註と註へのリンクは ‘[ ]’ 内部の表記を合わせるか、註の箇条書としての位置を合わせるといった工夫が必要になります。

```
[2]_ will be "2" (manually numbered),  
[#]_ will be "3" (anonymous auto-numbered), and  
[#label]_ will be "1" (labeled auto-numbered).  
  
.. [2] This footnote is labeled manually, so its number is fixed.  
  
.. [#label] This autonumber-labeled footnote will be labeled "1".  
It is the first auto-numbered footnote and no other footnote  
with label "1" exists. The order of the footnotes is used to  
determine numbering, not the order of the footnote references.  
  
.. [#] This footnote will be labeled "3". It is the second  
auto-numbered footnote, but footnote label "2" is already used.
```

### 引用 (Citation)

脚注と同様の表記で類似のハイパーリンク機能を表現するのですが、脚注と違い引用のラベルが数字や記号以外の文字列で、複合文体要素 `citation` として解釈されます：

#### 引用の表記

```
.. + [‘文字列’] ‘引用’  
- - +-----+-----+-----+  
| (文体要素)+
```

```
Here is a citation reference: [CIT2002]_.
```

```
.. [CIT2002] This is the citation. It's just like a footnote,  
except the label is textual.
```

### 指示 (directive)

書式が定められたテキストや画像等のオブジェクトを包含するための機構の一つです。Sphinx が指示を多用しています。この指示については §4.6.7 で詳細を説明することにします。

## 代入定義

### 註釈

#### 4.6.6 行内組版指示

行内組版指示は文体要素とは異なり、行を構成する一部の文字列に対してイタリックや太文字等の強調を行うことを目的としたものと、脚注やハイパーリンク等の機能を実現させることを目的としたものがあります。

### 強調

文字列を特定の文字列（強調文字列）で括ることで挟まれた文字列の強調表示を実現します：

強調表示の記述			
‘強調文字列’	‘文字列’	‘強調文字列’	(他の文字列)*

ここで可能な強調表示はイタリック、太文字と固定長の三種類で、そのために用いる文字列は“*”，“**”と“`”です。また，“`”を使ってはハイパーリンクを表現することも可能です：

行内組版指示の一覧		
組版指示	レンダリング例	表記例
イタリック	<i>italic</i>	*italic*
太文字	<b>bold</b>	**bold**
固定長	<code>verbatim</code>	‘verbatim’
ハイパーリンク	<u>python</u>	‘python <www.python.org>’

このように強調すべき文字列を括る文字列としてアスタリスク“*”とバッククオート“`”が用いられていますが、もしも文中にこれらの文字が行内組版指示の要素以外、要するに指示ではなく単なる文字として用いる場合はバックスラッシュ“\”*8をそれらの文字の先頭に置くようにします。なお、この強調の仕方は日本語と英語との表記の違いが顕著になります。まず英語では語の区切りとして空白文字が用いられます。そして強調すべき語はこの空白文字で切り分けられた文字列になるために先頭のアスタリスクは強調しようとする語の直前に置かれることになります。それと同じ状況にしなければならないために日本語で記述するときに強調する文字列の前後に空白文字を入れ、その状態でアスタリスク

*8 日本語のWindows環境やMacOSXではいわゆる円記号“¥”を用います。

で括る必要があります。また強調すべき文字列は複数行、つまり空行を含むものであってはなりません。このreSTの行内組版指示の制約として次のものがあります：

—— 行内組版指示の制約 ——

- 入れ子にできません
- 強調すべき文字列の先頭や末尾に空白文字が配置できません。
- 行内組版指示では空白文字や括弧等の記号で区切る必要があります。空白文字を表示させたくない場合はバックスラッシュ“\”を用いると空白文字が表示されません。

#### 4.6.7 指示

指示はアスキー文字列で、ピリオド“.”二つに続けた空白文字の列“..”で開始し、それから指示型と呼ばれるアスキー文字列が続き、その後にコロン“:”が二つ続きます。それから空白文字を置いて指示の引数が続きます。これで指示の行は終り、次にオプションが指定可能なものであれば指示のオプション行が続くことになります。指示オプションも同様で、指示オプション名がアスキー文字列で、指示オプション行は先頭がコロン“:”一つで、それに続けて指示オプション名、それに続けてコロン“:”が一つ置かれ、指示オプションの引数が空白文字で続いている形になります。次に指示型と指示オプション名のBNFを以下に示しておきます：

**指示型と指示オプション名の EBNF**

---

lc_letter	::=	"a" ... "z"
指示型	::=	lc_letter   "-" lc_letter
指示オプション名	::=	lc_letter

---

この指示の一般的な構文を以下に示しておきます：

**指示の構文**

---

..   指示型::	指示の引数
- +-----+ :指示オプション名:	指示オプションの引数
段落、箇条書等の行	

---

ここで指示オプションや段落、箇条書等の行を「**指示ブロック**」と呼びます。指示オプ

ションは必要に応じて無指定や複数行になり、指示ブロックを構成する行は指示型の先頭を基準とした字下げを必要とします。

reStructuredText が提供する指示には次のものがあります:

- 警告: attention, caution, danger, error, hint, important, note, tip, warning, admonition
- 画像: image, figure
- 本体要素: topic, sidebar, line-block, parsed-literal, code, math, rubric, epigraph, highlights, pull-quote, compound, container
- 表: table, csv-table, list-table
- 文書構成: contents, sectnum, section-numbering, header, footer
- 参照: target-notes, footnotes, citation
- HTML: meta, title
- 代入定義: replace, unicode, date
- テキストロール: role, default-role, title, restructuredtext-test-directive
- その他: include, raw

また、多くの指示で用いられる共通のオプションに name と class があります:

共通のオプション

オプション名	取り得る値	概要
name	文字列	doctree 成分の name 属性.
class	文字列	doctree 成分の class 属性.

以降、順番に解説することにしましょう。

## 警告

警告には見出が固定の「**特定の警告**」と見出を自由に変更できる「**汎用の警告**」の二種類が用意されています。

**■特定の警告:** attention, caution, danger, error, hint, important, note, tip, warning があり、これらの指示には引数はなくて指示型を見出に使い、表示すべき文は指示型を含む行の直下に記載します。これらの警告にはオプションはありません。

**■汎用の警告:** 通常の警告では見出が指示型に対応するものになりますが、指示型が admonition の場合は任意の文字列で警告の見出を設定することができます。この場合は指示の引数として見出に用いる文字列を設定することになり、そのため他の警告と異な

り引数が必要になります また, admonition は次のオプションを探ることができます:

#### admonition のオプション

オプション名	取り得る値	概要
class	文字列	クラス属性を指定.

admonition の簡単な例を示しておきます:

```
... admonition :: ケイコク!ケイコク!! こんなふうに警告を自分でつくることもできるのです
```



図 4.3 admonition の実例

## 画像

reST で画像を扱う指示に image と figure の二通りがあります。なお, image は単に画像を文書に取り込むだけですが, figure はキャプションを付けるといった汎用のものとなっています。

■image: 画像ファイルの指定を行います。相対か絶対の何れかで画像ファイルへの経路を指定します。オプションとして以下のものがあります:

**image のオプション**

オプション名	取り得る値	概要
align	top   middle   bottom   center   left   right	画像の配置箇所を指定.
alt	任意の文字列	画像の代りに表示する代替文字列.
class	文字列	クラス属性を指定.
height	整数値	画素数による画像の表示高さ
scale	整数値	縦横比を変更せずに拡大縮小を行う場合の拡大率.
target	文字列	画像をハイパーリンクとして利用する場合に URI を指定.
width	整数値	画素数による画像の表示幅.

具体例を示しておきます:

```
.. image:: ScuolaDiAtene.png
:height:500
:width: 800
:scale: 50
:alt: アテネの学堂
:align: center
```

■**figure:** image と同様に画像ファイルの指定を行います。相対か絶対の何れかで画像ファイルへの経路を含めてファイル名の指定を行い、image と同じオプションが使えます。figure 固有のオプションを以下に示しておきます:

**figure のオプション**

オプション名	取り得る値	概要
align	center   left   right	画像の中央, 左寄, 右寄を指定.
figclass	文字列	クラス属性を指定.
figwidth	整数値 画像ファイル名	画素数による画像の表示幅.

image は基本的に画像の配置ですが、figure はキャプションの設定もできます。

```
.. figure:: ScuolaDiAtene.png
:height:500
```

```
:width: 800
:scale: 50
:alt: アテネの学堂
:align: center
```



図 4.4 figure の実例

### 引用 (quote)

### 表 (table)

reST では三種類の表が扱えます。これらの表の違いは主に入力データの書式によるもので、csv-table はコンマ等の特定の文字で区切られた文字や数字の列から表を作成するためのもので、ディレクトリ、あるいは URI で指定可能な外部ファイルを取り込んで表を生成することができます。

■table: 表を生成します。引数は文字列で表の題名になります。ここで表の水平の罫線は記号 “=” で指示します。この水平線で表のカラム幅が決定されるため、カラム毎で幅が一致するように揃える必要があります。

#### table のオプション

オプション名	取り得る値	概要
class	文字列	クラス属性を指定。

■**csv-table:** 表を生成します。引数は文字列で表の題名になります。csv-table は table のように罫線の指定は不要で、コンマ“,”を区切り文字とする文字列を指示ブロックに記載します。ただし、table のようにカラム数を揃える必要はありません。

table のオプション

オプション名	取り得る値	概要
header-rows	整数値	テーブルヘッダーに使用する列数。規定値は 0.
class	文字列	クラス属性を指定。
delim	文字   tab   space	文字列の区切り文字を指定。
encoding	テキストエンコーディングの指定	外部 CSV データのテキストエンコーディング
file	文字列	検索経路を含む CSV 形式のファイル名
header	CSV データ	テーブルヘッダーのための補足データ。メインの CSV データの前に追加。
quote	文字	引用符として用いる文字の指定。
stub-columns	整数値	タブとして使用するテーブルカラムの数。規定値は 0.
url	文字列	CSV 形式のファイルの URL による参照。
width	整数値,..., 整数値	カラム幅を指定。規定値は等幅カラム

■**list-table]** 表の定義が箇条書の流儀で行われます。

### 本体要素

■**topic:** 文字列を引数に持ち、この引数を表題とする一節を構成します。

topic のオプション

オプション名	取り得る値	概要
class	文字列	クラス属性を指定。

■**sidebar:** 文字列を引数に持ち、この引数を表題とする一節を構成します。ただし、topicと違い副題を持たせることができます。そして topic と異なり右側に寄せて表示されます。

#### sidebar のオプション

オプション名	取り得る値	概要
subtitle	文字列	副題を指定。
class	文字列	クラス属性を指定。

■**line-block:**

■**parsed-literal:**

■**code:** リテラルブロックを構成します。

#### code のオプション

オプション名	取り得る値	概要
number-lines	整数値	開始行に割り当てる番号。既定値は 1。
name	文字列	
class	文字列	クラス属性を指定。

■**math:** LATEX で記述された行で構成されたブロック向けの指示です。ここで利用可能な LATEX 命令には AMS 拡張が含まれています。

■**rubric:** 脚注を設定し、引数を一つとります。

#### rubric のオプション

オプション名	取り得る値	概要
name	文字列	
class	文字列	クラス属性を指定。

■**epigraph:** 引用句のために用います。epigraph の引数やオプションはありません。

■**highlights:**

■**pull-quote:**

■**compound:**

■**container:** HTML の div と等価です。

## 文書構成

■contents: 目次を生成します。

### contents のオプション

オプション名	取り得る値	概要
depth	整数値	セクションの深さを指定。
local		
backlinks	entry   top   none	
class	文字列	クラス属性を指定。

■sectnum と section-numbering: 篇条書の番号を制御します。双方ともに呼び名が違うだけでやることは同じです。

■header と footer: 文書のヘッダとフッタ。

## HTML

■meta:

■title:

## 代入定義

reST の文書で用いる定数の定義を行うために用いられる指示です。reST の定数は「定数名」で定義されますが、一度定義した変数の値の変更はここでの代入定義でおこなうことができません。ここでの指示では定数の置き換えを行う replace, UNICODE を UNICODE 文字に変換して変数に代入を行う unicode, 日時の代入を行う date があります。

■replace: 文字列をその値とする定数を定義する指示です。この指示には引数がありません。

■unicode: UNICODE 文字をその値とする定数を定義します。この定義では UNICODE 文字コードをそのまま指定します。解釈器において UNICODE が意図する UNICODE 文字に変換して定数項への代入を行います。そのために引数として UNICODE 文字コードを必要とします。

■date: 計算機環境の日時の代入で用います。引数は日時の書式です。

## テキストロール

■role: テキストロールを指定します。

role のオプション

オプション名	取り得る値	概要
class	文字列	クラス属性を指定.

■default-role:

## その他

■raw:

■include:

■class:

## 置換

### エンコーディング



## 第5章

# 数学的対象の表現

## 5.1 はじめに

SageMath はさまざまなアプリケーションやライブラリを Python で繋ぎ合せてできあがったシステムです。この繋ぎ合わせでアプリケーションやライブラリの入出力データを Python のオブジェクトとして取り込み、その結果、利用者には表の Python だけしか見えず、プロンプトが ‘sage:’ であることを除くと IPython を Python のシェルとして使っている状況と大差ありません。だからといって SageMath を各種アプリケーションやライブラリの入出力を整えただけの単純な拡張ではありません。SageMath は数学的対象を的確に、そして、より効率的に処理するための工夫があり、そのため Python と SageMath の両者に微妙な違いがあります。その具体的な違いの一つが演算子 “ $\wedge$ ” の扱いに現われています。Python とその上で動作するパッケージの SymPy で演算子 “ $\wedge$ ” は論理排他和の演算子として扱われ、FORTRAN 風の演算子 “**” が幕乗の演算子として用いられています。しかし、世の中の多くの数式処理では幕乗として演算子 “ $\wedge$ ” が用いられています。このことは数式を扱う分野で幅広く利用されている組版ソフト TeX の影響が挙げられるでしょう。実際、数式  $x^2$  は TeX で ‘ $x^2$ ’ と表記されるために数式処理で多項式  $x^2$  を ‘ $x^{**2}$ ’ と入力するよりも ‘ $x^2$ ’ と入力する方が自然です^{*1}。そのこともあって SageMath では数式全般の幕乗の演算子として演算子 “ $\wedge$ ” が用いられています。だからといって SageMath で演算子 “ $\wedge$ ” が幕乗の演算子として置き換えられているというものではありません。実際、SageMath 上で int 型や float 型の数に演算子 “ $\wedge$ ” を使うと従来の論理排他和として動作するからです。このことは SageMath では整数や実数といった数自体が本来の Python のそれとは別物であって、SageMath は SymPy を安易に拡張したものではない、それ以上のものであることを示唆しています。

実際にこの違いを確認してみましょう。Python でオブジェクトの型を調べるときに組込函数 type()^{*2}で調べることができます。このときは引数としてオブジェクトと参照関係にある名前を指定すれば十分です。では函数 type() を使って 1 と 1.0 がどのような意味付けをされた対象であるかを調べてみましょう：

---

```
sage: type(1)
<type 'sage.rings.integer.Integer'>
sage: type(1.0)
<type 'sage.rings.real_mpfr.RealLiteral'>
```

---

^{*1}多くの数式を扱わなければならないときに数式エディタを使うよりも線的にテキスト入力が可能な TeX は非常に便利です。

^{*2}組込函数 type() はオブジェクトの型を調べることができるだけではなく、指定した型を持つオブジェクトを生成する能力を持つ函数です。

このように SageMath で対象 1 は sage.rings.integer.Integer, 1.0 は sage.rings.real_mpfr.RealLiteral のインスタンスで、共に SageMath のクラスであることが判ります。ちなみに IPython で同じことを実行した結果を参考までに示しておきましょう：

---

```
In [1]: type(1)
Out[1]: <type 'int'>
```

```
In [2]: type(1.0)
Out[2]: <type 'float'>
```

---

今度は対象 1 は int 型で対象 1.0 は float 型であると返しており、SageMath の結果と異っています。このように SageMath は Python を基盤にして構築されていますが、より効率的な処理を行うことと、より統一的な数学的構造を導入するために、数を表現するオブジェクトについても SageMath の数学的構造に対応するオブジェクトに入れ替えていることが判ります。このことは一方で厄介な問題も孕んでいます。SageMath で入力した ‘1’ と SageMath 上で動作させるために組んだプログラムの ‘1’ が SageMath の代数的数としての ‘1’ であるとは限らず、Python の int 型の整数かもしれないからです。だから、プログラムを組む際にオブジェクトが Python 本来のオブジェクトなのか、SageMath のメソッドや函数を利用しなければならないものをあらかじめ明瞭にしておく必要があります。

このように Python と SageMath の数の表現に違いがあることが判りました。さて、ここで気になることは Python の int 型と SageMath の sage.rings.integer.Integer 型にある隙間です。この隙間にどのような定義や設定があるのでしょうか？また、どのような処理が行われているのでしょうか？この章では、Python での数の構成を含めて SageMath でどのような実装が行なわれているかを観察してみましょう。

## 5.2 Python の数の構成

### 5.2.1 Python における数オブジェクトの扱い

凡そ計算機言語では整数や浮動小数点数といった数の表現と処理は計算機の CPU の機能として最初から実装されています。だから、計算機言語で数そのものをフレーベの「算術の基礎」や集合論のように最初から全てを構成する必要がなく、むしろ、ハードウェアに実装されている機能をどのように使うかということが問題になります。ただ、オブジェクト指向言語では扱う対象が全てオブジェクトであり、それらの関係を利用して処理を行うという性格上、数に関してもそれらの関係を明確にしておいた方が全体の見通しが良くなります。たとえば、整数と実数であれば集合として包含関係があり、算術演算子も共

通のものを用います。だからこそ、個々の独立したオブジェクトではなく、これらの数オブジェクトの間に抽象的な関係、たとえば、継承関係を導入することで、これらのオブジェクトを捉えた方がより体系的な捉え方が可能になります。このような抽象的な関係を入れるために Python には抽象基底クラスと呼ばれるメタクラスが用意されています。

ここで Python の数のクラス (Numeric Class) の定義は PEP-3141^{*3}に沿って行われています。この PEP-3141 は PEP-3119^{*4}で導入された「**抽象基底クラス (ABCs: Abstract Base Classes)**」を Python の数の構成に適用しています。このことは「Python 言語リファレンス」の数の説明にて、「numbers.Number’, ‘numbers.Real’, ‘numbers.Complex’ といった型の表記からも伺え、実際、これらのクラスは数の抽象基底クラスです。これらのクラスに対応する Python の int, long, float, complex といった型が抽象クラスを現実化した具象化クラス (concrete class) と呼ばれるオブジェクトに該当します。とは言え、int 等の型はこれらの PEP よりも古くから存在する型で、モジュール numbers も通常の Python で最初に読み込まれるモジュールではありません。モジュール numbers は「**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？

### 5.2.2 numbers モジュールを眺めてみよう

モジュール numbers がどのようなものであるかを知るために Python の函数 help() を利用してみましょう。ここで Python のインタプリタを利用するのであればあらかじめ ‘import numbers’ で numbers モジュールを読み込んでから、SageMath ならそのまま ‘help(numbers)’ と入力してみましょう：

---

Help on module numbers:

NAME

numbers – Abstract Base Classes (ABCs) for numbers, according to PEP 3141.

FILE

/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>

---

^{*3} 原文:<http://www.python.org/dev/peps/pep-3141/>

^{*4} 原文: <http://www.python.org/dev/peps/pep-3119/>

日本語訳：<http://mft.la.coocan.jp/script/python/pep-3119.html>

**DESCRIPTION**

TODO: Fill out more detailed documentation on the operators.

**CLASSES**

```
-- builtin ___.object
    Number
        Complex
        Real
        Rational
        Integral
```

---

ここでは OSX 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 の順の階層があり、この階層がクラス間の継承関係を示しています。ここでの階層構造は「**数値塔**」と呼ばれる数オブジェクトの階層構造に対応します。

このオンラインヘルプからは FILE の項目にモジュール numbers.py の在処が掲載されています。だから、モジュール numbers がどのように定義されているかを知りたければ、実際にここに記載された numbers.py の中身を見てしまえば良いのです。では、最初に numbers.py からクラス Number の定義を抜き出しておきましょう：

```
# Copyright 2007 Google, Inc. All Rights Reserved.
# Licensed to PSF under a Contributor Agreement.

"""Abstract Base Classes (ABCs) for numbers, according to PEP 3141.
```

TODO: Fill out more detailed documentation on the operators.""""

```
from __future__ import division
from abc import ABCMeta, abstractmethod, abstractproperty

__all__ = ["Number", "Complex", "Real", "Rational", "Integral"]

class Number(object):
    """All numbers inherit from this class.

    If you just want to check if an argument x is a number, without
    caring what kind, use isinstance(x, Number).
```

```
"""
__metaclass__ = ABCMeta
__slots__ = ()

# Concrete numeric types must provide their own hash implementation
__hash__ = None
```

このクラス Number の定義では import 文を使ってモジュール abs からモジュール ABCMeta 等の読み込みとクラス属性 __metaclass__ に ‘ABCMeta’ を割り当てていますが、これらは 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 の役割は数という構造を提供することが目的で、数が何で、それがどのようなものであるかを指定することではありません。そこで数の本質的な属性やメソッドは、クラス Number を継承するクラス Complex 等のサブクラスに記載されます。そこで今度は複素数を表現する抽象基底クラス Complex の numbers.py からの抜粋を載せておきましょう：

```
class Complex(Number):
    """Complex defines the operations that work on the builtin complex type.

    In short, those are: a conversion to complex, .real, .imag, +, -,
    *, /, abs(), .conjugate, ==, and !=.

    If it is given heterogenous arguments, and doesn't have special
    knowledge about them, it should fall back to the builtin complex
    type as described below.
"""

__slots__ = ()

@abstractmethod
def __complex__(self):
    """Return a builtin complex instance. Called for complex(self)."""

# Will be __bool__ in 3.0.
def __nonzero__(self):
    """True if self != 0. Called for bool(self)."""
    return self != 0

@property
def real(self):
    """Retrieve the real component of this number.

    This should subclass Real.
```

```
"""
raise NotImplementedError

— 略 —

@abstractmethod
def __eq__(self, other):
    """self == other"""
    raise NotImplementedError

def __ne__(self, other):
    """self != other"""
    # The default __ne__ doesn't negate __eq__ until 3.0.
    return not (self == other)
```

Complex.register(complex)

---

抽象基底クラス Complex の定義からな特徴的なメソッドの抜粋をここに掲載していますが、このクラス Complex では複素数の四則演算を含む演算に関連するメソッドと同値性に関連するメソッド等がデコレータ付きで定義されています。ただ、それらの定義内容は本来メソッドで定義されるべき処理内容が長文書文字列で記載されているだけで、実際に行われる処理文は NotImplementedError 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記載し、これらのメソッドの具体的な処理は、この抽象基底クラスを継承する具象化クラスにて記載されます。

このモジュール numbers で定義されているクラスは Complex, Real, Rational, Integral があり、クラス A がクラス B のサブクラスであることを  $A \rightarrow B$  この構成方法は実利的な側面があります。まず、集合論の数の構成であれば  $\mathbf{N} \rightarrow \mathbf{Z} \rightarrow \mathbf{Q} \rightarrow \mathbf{R} \rightarrow \mathbf{C}$  と「塔」の上側から順に定義し、演算も順次拡張することで数値塔の継承の順序と逆の構成順序になります。しかし、これらのオブジェクトとその中で閉じた演算がメソッドとして実装されていればどうでしょうか？まず、複素数であれば複素数同士の演算はもちろん、複素数と実数、複素数と有理数、そして複素数と整数の演算は、実数、有理数と整数が複素数のサブクラスであることから問題なく行うことができますが、実数はどうでしょうか？複素数を継承していれば演算は可能ですが、双方が実数の場合、虚部が  $0 \times i$  の形式で現れる可能性があります。その意味で、双方が実数であれば実部のみを返すと演算を書き直すこ

となるでしょう。以降、同様のことが有理数と整数でも生じ、同じクラスのオブジェクトの処理を再定義することになります。このように数値塔は数と演算の存在を前提にした数の実装方法です。

そして、Python ではこれらの数と演算は、組込のオブジェクトとして個別に用意されており、抽象基底クラスを導入することで、後付的に数オブジェクトとして継承関係も含めて再定義しているといえます。もちろん、この構成は計算処理だけが目的であればあってもなくても良いものかもしれません。しかし、既存のオブジェクトに構造を導入することで抽象化した立場で考察が可能になること、さらには多項式等の数の拡張で、この抽象化が威力を発揮します。

## 5.3 SageMath の数の構成

### 5.3.1 SageMath の数

SageMath では、整数、有理数、実数と複素数といった数オブジェクトを Python の数オブジェクトから継承しません。この理由として、

- Python 2.x では整数が 32bit 整数と長整数の二種類があり、それぞれリテラルが異なる。
- 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任意精度でない。
- 演算が効率の良いものではない。

が挙げられます。これらの問題に対処するために SageMath は GMP(GNU MP, GNU Multi Precision Arithmetic Library) で整数、有理数、実数と複素数とそれらの基本演算を定義し直しています^{*5}。そのために SageMath で数式を構成する数は SageMath の数オブジェクトが用いられます。ただし、SageMath 上のプログラムの整数や実数のリテラルは型の変換を行わない限り、Python の整数や実数の型のままでです。このことは for 文や while 文でカウンターとして使っている整数が Python の数オブジェクトのままで良いものか、SageMath の数オブジェクトとしても利用するものであるかどうかを明瞭にしておく必要があることを意味します。

また、SageMath は既存の数式処理システムや数値計算ライブラリを組み合わせた巨大なシステムです。特に整数と有理数に関しては GMP 以外に数論専用の数式処理 PARI が組み込まれており、この PARI が output した数オブジェクトを SageMath の数として変換す

---

^{*5} GMP を利用しているものに商用の数式処理 *Mathematica* があります。

る機能もあります。

### 5.3.2 GMP の概要

GMP は C で記述された整数, 有理数と浮動小数点数の任意精度で演算を行うためのライブラリです。なお, ここでの任意精度とは計算機の記憶容量等のハードウェアに由来する制約を除いて必要な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, 一方で, 計算精度が向上することで収束計算の反復回数が減ることで却って全体の処理時間の増大を抑えることになったり, 適切な精度で数値計算が行えることから事象の解析が可能になることもあります。この GMP は単に任意精度の計算が可能なだけではなく, 通常の C が提供する精度以上の必要とされる精度で可能な限り最速の数値計算を実現することを目的にしています。そのために 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も含んでいます。

ここで SageMath の数の構成で利用される GMP のクラスを以下に示します:

#### SageMath の数の構成に必要な GMP のクラス

---

mpz_t	符号付整数とその演算. 名前が mpz_ で始まる 150 程度の函数を含む.
mpq_t	有理数とその演算. 名前が mpq_ で始まる 35 程度の函数を含む.
mpf_t	浮動種数点数とその演算. 名前が mpf_ で始まる 70 程度の函数を含む

---

なお, 有理数は整数の対として表現されるために, ここで挙げた 35 程度の函数に加えて整数演算の函数が必要になります。

SageMath はこの GMP を取り込むために Cython を利用しています。この Cython は C と Python を継ぎ目なしに融合することができる Python で記述された処理系です。この Cython について次の説で簡単に解説することにしましょう。

### 5.3.3 Cython の概要

GMP を SageMath に取り込むため, Python で記述された処理系の Cython が用いられています。この 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, ここでは C の函数も Python の函数等の定義を継ぎ目なしに Python 風に記述することができます。もう一方の pxd ファイル C の定義ファイルに相当するもので常に必要なものとは限りません。これらのファイルを基に

Cython のコンパイラが C のコードに変換^{*6}し、それをコンパイルして Python の拡張モジュールが得られます。その結果、C で記述したものと大差ない程度に高速化できます。

Cython の構文は Python 言語の一部に C の函数呼出と C の変数やクラスの型宣言が追加され、Python の文に交じって、C のライブラリに関連する変数や函数の宣言、ライブラリの読み込みに関する文が混在しています。そして、これらの文で Python の命令文の先頭の “c” が付いた命令文 (cdef, cimport 等) が C のライブラリ操作に関連する分になります。たとえば、C の変数と型定義では cdef 文が用いられます。cdef 文は C の宣言文の頭に “cdef” をそのまま載せた構文で、int 型の変数 i, j, k と float 型の変数 f, 配列 g, ポインタ g を宣言するときは

---

```
cdef int i, j, k
cdef f, g[42], *h
```

---

と記述します。また、cdef をブロックとして

---

```
cdef:
    int i, j, k
    float f, g[42], *h
```

---

とグループ化して記述することもできます。また、cdef は C の函数呼出でも同様の書式ですが、函数が output する型や引数の型を C 風に記述しなければなりません。

### 5.3.4 整数の実装

SageMath の整数の実装 GMP の mpz_t 型のオブジェクトを Cython で SageMath にクラス Integer として組込んでいます。クラス Integer は GMP の整数型クラス mpz_t のラッパー、すなわち、具象化クラスとして定義されています。この整数型の機能の一つに Python の int 型や long 型、numpy の整数オブジェクト、それと整数論向けの数式処理 PARI の整数を SageMath の整数 Integer 型に変換する機能もあります。整数の定義は src/sage/rings/integer.pyx にクラス Integer の実質的な定義があり、sage.structure.element.EuclideanDomainElement のサブクラスとして定義されています。

### 5.3.5 有理数の実装

有理数の実装も、GMP の mpq_t 型を Cython でクラス Rational として取り込んだものになっています。この Rational には PARI や numpy からの数オブジェクトの変換

---

^{*6} ソースの構文解析は Pyrex で行います。

機能を持っています。実質的な定義は /src/sage/ring/rational.pyx にクラス Rational が sage.structure.element.FieldElement のサブクラスとして定義されています。

### 5.3.6 実数の実装

SageMath の実数は浮動小数点数によって近似される数であるため、倍精度の浮動小数点数と任意精度の浮動小数点数が用意されています。倍精度の浮動小数点数は精度以上に計算速度を重視することもあってベクトルで用いられます。倍精度の実数のクラスは RealDoubleField_class, 略して RDF で、任意精度の実数クラスが RealField_class, 略して RealField で、共に Field のサブクラスになります。

### 5.3.7 複素数の実装

複素数の実装も実数と同様に倍精度と任意精度の浮動小数点数の二種類の数で近似されます。倍精度の浮動小数点数で近似した複素数が ComplexDoubleField_class で任意精度の浮動小数点数で近似した複素数がクラス ComplexNumber です。

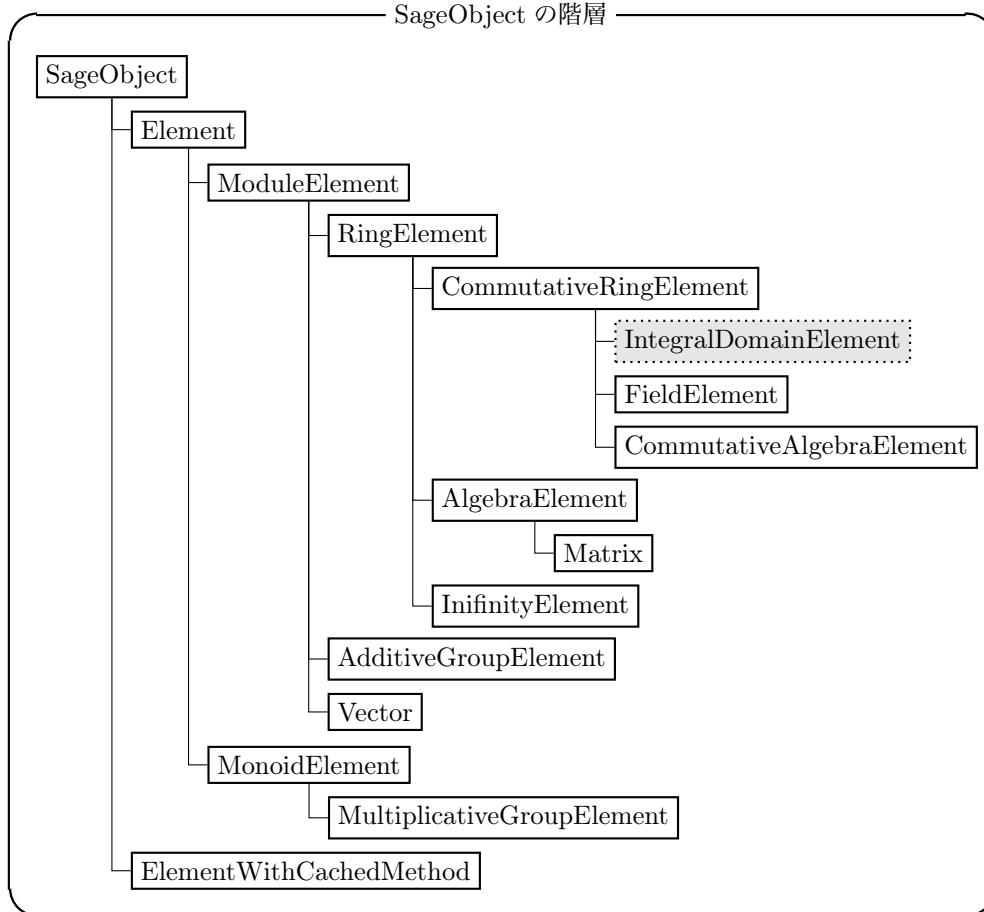
## 5.4 SageMath のオブジェクト

### 5.4.1 SageMath の抽象基底クラス

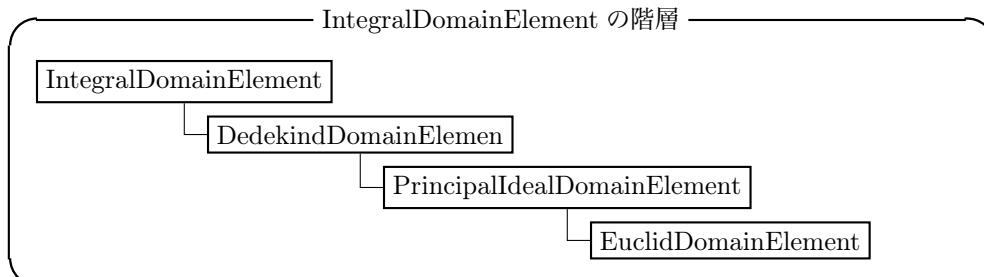
SageMath で利用者が認識できるオブジェクトで最上位のオブジェクトがクラス SageObject です。SageMath のオブジェクトで利用者が直接扱ったり返却されるオブジェクトが含まれるクラスはクラス SageObject と継承関係になければなりません。このクラス SageObject は注釈や文書で Python の抽象基底クラスと同様に “anstract base class” と呼ばれ、実際に Python の数オブジェクトで見られるような後付の階層構造が導入されていますが、抽象メソッドのデレータが @abstract_method と Python の抽象メソッドのデコレータが @abstractmethod と双方で異なり、@abstract_method が SageMath で新たに定義されたデコレータであることから、SageMath の “abstract base class” は Python のそれと異なる SageMath 独自のものであることが判ります。しかし、SageMath 上の整数、有理数などの数オブジェクトが GMP 由来で、該当するオブジェクトの具象化クラスとして現れており、Python の抽象基底クラスと類似の働きをしています。

この SageObject の継承関係図は src/sage/structure ある element.pyx から抜粋したもので最初に SageObject から六層までの階層を示します。それから六層目の灰色で塗り潰している IntegralDomainElement にはさらに三階層ありますが、継承関係による階層

構造が分かり難くならないように二つに分けて示しています:



次に煩雑になるために省略したクラス IntegralDomainElement の階層図を示します:



なお、SageObject がこれらの抽象基底クラスの何れかに分類されるという意味ではありません。あくまでも抽象基底クラスの継承関係を図示したものです。そしてクラス Element の継承関係は数学的な構造が反映されたものになっていますが、数学的な厳密さ以上に実装が容易になるように継承関係が導入されています。ここで SageObject を継承

する抽象基底クラスの概要を述べる前に簡単に、半群、群と環といった数学的構造について述べておきましょう。

### 5.4.2 数学的構造について

SageMath は数学的対象を扱いますが、数学的対象にはなんらかの代数的な構造、つまり、共通する性質を体系的にまとめたものがあります。たとえば、小学校で扱う自然数には足算、掛算に割算といった計算処理では割算と違って機械的に計算をしていますが、これらの演算は新しい自然数を生成する能力があります。実際、割算 “ $\div$ ” であれば、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

$$1 \div 2 = 0 \text{ あまり } 1$$

と答を書いて、 $1 \div 2 = \frac{1}{2}$  と書きません。つまり、 $1 \div 2$  で新しい数を生成しても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は商のみを結果として取るようにすれば足算、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一般的（普遍的）な方法で言い換えてみるとどうなるでしょうか。対象の自然数を集合  $S$ 、足算や掛算といった記号を記号 “ $*$ ” と表記し、この記号を「演算子」と呼びます、それから演算子 “ $*$ ” の影響を受ける集合  $S$  の元のことを「被演算子」と呼びましょう。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  $S$  の元を生成するの力で、この演算 “ $*$ ” が集合  $S$  に対して  $S \times S$  から  $S$  への写像になっていることです。この性質は「閉じている」と呼ばれる演算の性質で、この集合  $S$  と閉じた演算 “ $*$ ” の対  $(S, *)$  を「マグマ (magma)」と呼びます：

マグマの定義

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

- 演算 “ $*$ ” が閉じていること:  $a * b \in S$ .

この定義から自然数の集合  $\mathbb{N}$  の足算 “ $+$ ”，掛算 “ $\times$ ”，割算 “ $\div$ ” といった計算処理は全て閉じた演算で、 $(\mathbb{N}+)$ ,  $(\mathbb{N}, \times)$  と  $(\mathbb{N}, \div)$  はマグマになります。ここで  $(S, *)$  がマグマであるということは集合  $S$  に閉じた二項演算 “ $*$ ” が存在することを意味するだけで、他にどのような性質を持つものかは述べていません。

次に足算 “ $+$ ”，掛算 “ $\times$ ” に共通する性質に  $(1+2)+3 = 1+(2+3)$ ,  $(3 \times 4) \times 5 = 3 \times (4 \times 5)$

という性質があります。この括弧 “()” で指示された計算順序に依存しないという性質は「**結合律**」と呼ばれる重要な性質です。つまり、集合  $S$  の閉じた演算 “*” で任意の  $a, b, c \in S$  に対して  $(a * b) * c = a * (b * c)$  を充す性質です。ところで、割算 “÷” は  $(12 \div 6) \div 2 \neq 12 \div (6 \div 2)$  であるために結合律を充たしません。このことから割算が本質的に足算や割算と異なる演算であることが判ります。そこで、閉じた演算を持つという性質と結合律を充たすという性質を持つ集合に新しい概念を導入しましょう。これが「**半群 (semigroup)**」と呼ばれる概念です：

#### 半群の定義

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

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

自然数  $\mathbb{N}$  に対して  $(\mathbb{N}, +)$  と  $(\mathbb{N}, \times)$  は半群という概念で纏められますが、 $(\mathbb{N}, \div)$  はマグマであっても半群ではありません。この半群という概念を導入したことで割算が足算と掛算と本質的に異なる演算として分類できました。このように個々の数学的対象を普遍性を持った性質で区分することが数学的構造を入れることに他なりません。また、この区分では同じ集合でも演算によって入る構造が異なります。実際、自然数  $\mathbb{N}$  もその二項演算を足算 “+” にするか割算 “÷” にするかで半群やマグマといった異った概念へに分類されることになります。

ここで足算 “+” と掛算 “×” という演算には面白い性質があります。これは  $1 + 2 = 2 + 1 = 3$  や  $2 \times 3 = 3 \times 2 = 6$  と任意の  $a, b \in S$  に対して  $a * b = b * a$  と演算子 “*” の両側の集合  $S$  の元を入れ替えても結果が同じであるという性質です。このように二項演算子で左右の被演算子を入れ替えても演算結果が等しくなる性質のことを「**可換 (commutative)**」と呼びます。ところで割算 “÷” に関しては  $4 \div 2 \neq 2 \div 4$  であるために可換ではありません。このように可換でない演算子の性質を「**非可換 (noncommutative)**」と呼びます。

さらに  $(\mathbb{N}, +)$  の 0,  $(\mathbb{N}, \times)$  の 1 は共に面白い性質をもっています。0 については  $a + 0 = 0 + a = a$ , 1 については  $1 \times a = a \times 1 = a$  を充たしています。これらの関係式を演算 “*” , 0, 1 を元  $u$  で置き換えると任意の  $a \in \mathbb{N}$  に対して  $u * a = a * u = a$  という関係式が得られます。このように集合  $S$  の閉じた演算 “*” で任意の  $a \in S$  に対して  $a * u = u * a = a$  を充たす集合  $S$  の元  $u$  を「**単位元**」と呼びます。そして、単位元を持つ半群のことを「**单系 (モノイド, monoid)**」と呼びます。したがって、 $(\mathbb{N}, +), (\mathbb{N}, \times)$  は单

系で, 0 は  $(\mathbb{N}, +)$  の単位元, 1 は  $(\mathbb{N}, \times)$  の単位元です.

さて, 小学生も中学年になると分数が現れます. ここで分数  $\mathbb{Q}_+$  は分母と分子が 0 以外の自然数の数で  $\frac{n}{m}$  と表記され, 約分という操作が入った数です. この約分という操作は分数  $\frac{n}{m}$  の分子  $n$  と分母  $m$  が共通の約数  $a$  を持つときに  $m$  と  $n$  を  $a$  で割ったもので置換えるという操作で, 分母が 1 のときは分子のみの表記, 整数になります. この約分は  $m = a \times b$ ,  $n = a \times c$  のときに  $\frac{n}{m} = \frac{c}{b}$  とする関係⁷と言いたい換えることができます. それから分数は演算 “ $\times$ ” に対して 1 が単位元になります. また  $a, b, c \in \mathbb{Q}$  に対して  $a \times b \in \mathbb{Q}_+$ ,  $a \times 1 = 1 \times a = a$ ,  $(a \times b) \times c = a \times (b \times c)$  が成立するために演算 “ $\times$ ” は  $\mathbb{Q}_+$  で閉じた演算で, 1 がその単位元で結合律も成立することから  $(\mathbb{Q}_+, \times)$  は単系になります.

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

群の定義

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

では,  $(\mathbb{N}, +)$  と  $(\mathbb{N}, \times)$  は群でしょうか?  $(\mathbb{N}, +)$  では  $a + b = 0$  になる元は双方が 0 の場合のみで,  $(\mathbb{N}, \times)$  でも  $a \times b = 1$  になる元は双方が 1 の場合のみです. だから, 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のは逆元が存在しないからです. だから逆元を追加してしまえば群になるでしょう. 実際,  $(\mathbb{N}, +)$  に負の数を導入することで整数  $\mathbb{Z}$  を構築すると  $(\mathbb{Z}, +)$  は群になります. また,  $(\mathbb{N}, \times)$  に分数を導入して  $(\mathbb{Q}_+, \times)$  であれば群になります. 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

⁷ 同値関係と呼ばれる関係になります.

$-a$  と表記して「**加法群 (additive group)**」と呼ぶことがあります。また、加法群でない群を「**乗法群 (multiplicative group)**」と呼び、演算を記号“*”，単位元を 1,  $a$  の逆元を  $a^{-1}$  と表記します。これらの呼び名は自然数  $\mathbb{N}$  や整数  $\mathbb{Z}$  が足算“+”と掛算“×”のような二つの演算を持つ数学的対象で、それらの演算を区別するときに用いられます。

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

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

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

#### 環の定義

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

ここで乗法“*”にも単位元が存在すれば「**单位的環**」と呼びます。さらに乗法“*”も可換である環を「**可換環 (commutative ring)**」、逆に可換環でない環を「**非可換環 (non-commutative ring)**」と呼びます。可換環の代表的なものが整数  $\mathbb{Z}$  で、環としての構造を明確にするときは「**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  $M(n)$  で、体の代表的な例としては有理数  $\mathbb{Q}$ 、実数  $\mathbb{R}$  や複素数  $\mathbb{C}$  が挙げられます。

この環に次いで重要な対象が「**環上の加群**」で「**R-加群**」と呼ばれるものです。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です。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。この式は整数を左から有理数に掛けたもので減算を行っているという状況が現われています。つまり、左から環である整数  $\mathbb{Z}$  の元を加法群である有理数  $\mathbb{Q}$  の元と掛け合わせたもので計算を行っており、このときに有理数  $\mathbb{Q}$  の元に整数  $\mathbb{Z}$  の元を左側から掛けたものは有理数  $\mathbb{Q}$  の元で、マグマで見られる閉じた演算に類似した状況です。そこで、整数を左側から有理数に掛けるという操作を整数  $\mathbb{Z}$  の有理数  $\mathbb{Q}$  への「左から作用」と呼びましょう。これを一般化して、環  $(R, +, *)$  の元  $r$  による加法群  $(A, +)$  の元  $a$  への左側の作用を  $r * a$ 、右側の作用を  $a * r$  と表記します。

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

また、この左右の作用には環側と加法群側の加法に対する分配律が成立します：

左作用に関する分配律

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

右作用に関する分配律

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

さて、整数が有理数に作用するとき、整数の掛算 “ $\times$ ” の単位元 1 は有理数をそのままにする恒等写として作用しています。実際、 $1 = 1 \times 1 = 1 \times \dots \times 1$  なので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、環の作用の方向によって、左右の区別があります。これらの R-加群の定義を以下にまとめておきましょう：

——左 R-加群の定義——

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

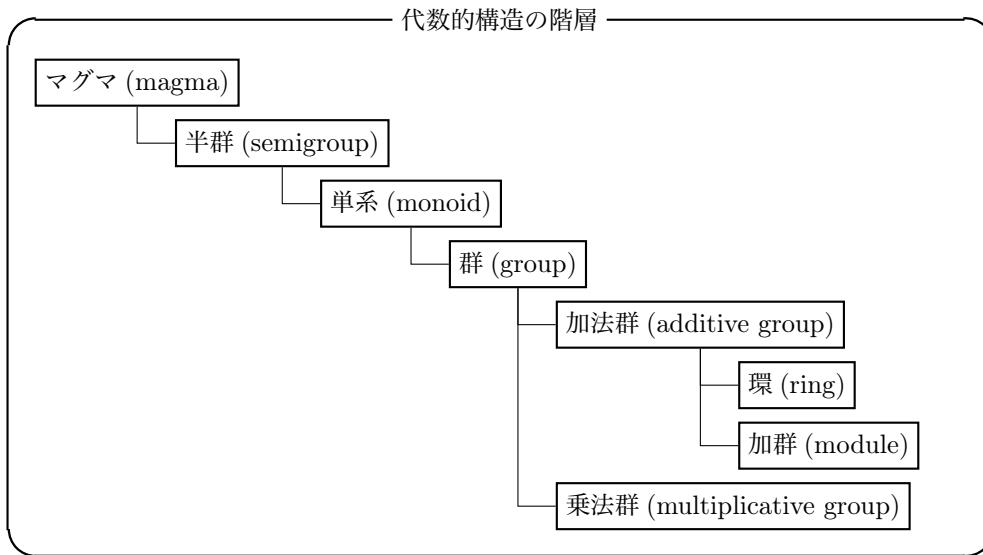
——右 R-加群の定義——

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

この左右の R-加群の定義で、この定義では環  $R$  の積と和と R-加群の環  $R$  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場合は両側 R-加群と呼びます。また、環  $R$  が可換環で、左右からの環  $R$  の作用が一致するとき ( $r * x = x * r$ ) に R-加群と呼びます。

この R-加群には先程の整数  $\mathbb{Z}$  の乗法  $\times$  による作用を持つ加法群  $\mathbb{Q}$ 、環を実数  $\mathbb{R}$ 、作用を通常の積  $\times$  による  $n$  次の実ベクトル空間が挙げられます。また、 $n$  次のベクトル空間も R-加群の例になります。実際、 $n$  次の(実)ベクトルの空間  $\text{Vect}(n) = \{(v_1, \dots, v_n) | v_1, \dots, v_n \in \mathbb{R}\}$  とし、係数環として実数  $\mathbb{R}$  の作用を各成分単位の積、つまり、 $v \in \text{Vect}(n)$ ,  $a \in \mathbb{R}$  に対して  $a \cdot v = (a \cdot v_1, \dots, v_n)$  で定めると、この作用は左右の R-加群の条件を満たすことから、この実ベクトル空間も R-加群であることが分かります。

さて、ここまでにマグマ、半群、群、環、体、そして、加群といった代数的構造を説明しましたが、これらの序列はどちらかと言えば発生順とです：



ところで SageObject の階層はどちらかと言えば定義のし易さです。このことは Element の直下にマグマに対応する抽象基底クラスがあるわけでもなく、抽象基底クラス Element の直下に環上の加群に対応する ModulElement と単系に対応する MonoidElement が配置されていることから明瞭です。なぜなら、加群は演算子 “+” と演算子 “*” の二つの演算子を必要としますが、単系では一つだけの演算子、この場合は積演算子 “*” の存在を前提にしているためです。その結果、ModuleElement にはメソッドとして積のメソッドと和のメソッドの二つがあらかじめ準備され、MonoidElement には積のメソッドのみが準備され、その結果、乗法群の抽象基底クラス MultiplicativeGroupElement が MonoidElement の直下に置かれ、和演算子 “+” を必要とする加法群の抽象基底クラス AdditiveGroupElement が ModuleElement の直下に置かれることになるわけです。

このように SageMath の抽象基底クラスと通常の数学的構造の階層との違いについて述べましたが、結局のところ、SageObject ではメソッドに重点が置かれ、数学的構造では数学的対象に重点が置かれたために生じたことです。結局のところ、オブジェクト指向プログラミングは対象の間との対応関係こそが主軸であり、このことは圏論で対象そのものよりも、対象間に存在する矢、すなわち、関係を重視する姿勢に対応します。

## 5.5 SageObject の各階層について

### 5.5.1 はじめに

ここでは SageObject の抽象基底クラスを階層ごとに解説します。この抽象基底クラスの最上位のクラス SageObject の階層を 0 とします。ここで解説するクラスは全て

SageMath の抽象基底クラスで、利用者がこれらのクラスの具象化クラスを構築する場合は継承によって定義することができます。また、通常の Python のクラスで演算や比較を実行するため用意された特殊メソッドそのものは基本的に具象化クラスの構築では用いず、抽象基底クラスに用意されたメソッドを利用します。

### 5.5.2 第0層

■**SageObject**: SageMath で利用者が直接扱い、見ることのできる継承関係で最上位のオブジェクトのクラスになります。このクラスのメソッドに計算結果をアスキーアートとして出力するメソッド、結果をファイルに保存するメソッド、SQLite 等の DB を操作するメソッドや SageMath が利用する数学アプリケーションとのインターフェイスが含まれます。

### 第1層

■**Element**: SageObject を親とする抽象基底クラスで、Element を継承するクラスは加群 (Module) と単系 (モノイド, Monoid) に対応する抽象基底クラスになります。

■**ElementByCachedMethod**: キャッシュを有するメソッドの抽象基底クラスです。計算結果を一時的に蓄えるメソッドといったものが対応します。

### 5.5.3 第2層

■**ModuleElement**: Element を継承する (R-) 加群 (Module) の抽象基底クラスです。R-加群の演算を表現するメソッドとしては、係数環  $R$  と可換群  $A$  の和 “+” については `_add_()`, 左作用のときは `_lmul_()`, 右作用のときは `_rmul_()`, 両側であれば `_mul_()` があります。また,  $a \in A$  の逆元  $-a$  をメソッド `_neg_()` で定めます。

■**MonidElement**: Element を継承する単系 (モノイド, Monoid) の抽象基底クラスです。ここで単系は可換であるとは限らない二項演算 “*” とその演算の単位元を持つ対象です。

このクラスの具象化クラスは積 “*” のメソッド `_mul_()` の書換えを必要とします。

### 5.5.4 第3層

■**RingElement**: ModuleElement を継承する環 (Ring) の抽象基底クラスです。環は加法 “+” と乗法 “*” の二つの二項演算子を持つ数学的対象です。ここで集合  $R$  が環であるとは次の条件を充たす場合です:

■**AdditiveGroupElement**: ModuleElement を継承する加法群 (Additive group) の抽象基底クラスです。可換な二項演算である和 “+” のみを演算として持つ群です。

■**Vector**: ModuleElement を継承するベクトル (Vector) の抽象基底クラスです。可換な二項演算である和の他に実数あるいは複素数との積 (スカラー積) を有します。なお、SageMath のベクトルは近似の数である浮動小数点数で表現され、そのために代数から独立して置かれています。

■**MultiplicativeGroupElement**: MonoidElement を継承する乗法群 (Multiplicative Group) の抽象基底クラスです。可換であるとは限らない二項演算子である積 “*” のみを持つ群を表現します。加法群と分けたのも二項演算の実装に由来し、Python では和に対応するメソッドが `__add__()`、積 “*” に対応するメソッドが `__mul__()` に対応するためです。

### 5.5.5 第4層

■**CommutativeRingElement**: RingElement を継承する可換環 (Commutative Ring) の抽象基底クラスです。可換環では乗法 “*” の可換性があります。

■**AlgebraElement**: RingElement を継承する代数 (Algebra) を表現する抽象基底クラスです。代数はベクトルを一般化した代数的構造です。この抽象基底クラスは数学的構造が加群に由来するということではなく、むしろ、和と積の二項演算を持つというメソッドの実装上の類似によるものです。

■**InfinityElement**: RingElement を継承し、無限遠  $\infty$  に対応する抽象基底クラスです。

### 5.5.6 第5層

この第5層には CommutativeRingElement を継承する抽象基底クラスのみです。

■**DedekindDomainElement**: CommutativeRingElement を継承する抽象基底クラスで、デデキント整域を表現します。ここでデデキント整域とは 0 生成されたイデアル (0) と異なるイデアルが有限個の素イデアルの積として表わせる整域のことです。

■**IntegralDomainElement**: 整域 (Integral Domain) に対応する抽象基底クラスです。ここで整域は環  $R$  で  $ab = 0$  となる  $a, b \in R$  が存在するときに  $a = 0$  か  $b = 0$  の何れかが必ず成立するときで、環  $R$  が零因子を持たないときとも言い換えられます。

■**FieldElement**: 体 (Field) に対応する抽象基底クラスです。 $R$  が体であれば、 $R$  はまず和演算 “+” と積演算 “*” を持つ可換環で、さらに  $(R - \{0\}, *)$  が可換群になる環です。た

とえば、整数  $\mathbb{Z}$  は和と積の二つの演算を持ちますが、積に関しては可換で単位元 1 を持っていても、 $\mathbb{Z} - \{0\}$  の任意の元が逆元を持たないために体にはなりません。有理数  $\mathbb{Q}$  は  $(\mathbb{Q}, +)$  が可換群、そして、 $(\mathbb{Q} - \{0\}, *)$  も可換群であることから体になります。

■CommutativeAlgebraElement: 可換代数 (Commutative Algebra) に対応する抽象基底クラスです。

### 第5層以下

整域を継承し、デデキント整域 (Dedekind Domain) を表現する抽象基底クラスです。

■PrincipalIdealDomainElement: デデキント整域 (DedekindDomainElement) を継承し、単項イデアル整域 (PID, Principal Ideal Domain) を表現する抽象基底クラスです。ここで PID は任意のイデアルが単項生成になる整域です。

■EuclidDomainElement: PrincipalIdealDomainElement を継承し、ユークリッド整域 (Euclid Domain) を表現する抽象基底クラスです。ここで整域  $R$  がユークリッド函数と呼ばれる写像  $f : R - \{0\} \rightarrow N$  が存在し、任意の  $a, b \in R$  に対して  $f(a) < f(b)$  であるときに  $r = 0$  かつ  $f(r) < f(a)$  かつ  $b = aq + r$  を充たす  $q, r \in R$  が存在するときに整域  $R$  をユークリッド整域と呼びます。

## 第6章

### SQLiteについて

## 6.1 SQLite 速習

### 6.1.1 SQLite 概要

SQLiteは「リレーションナルデータベース管理システム (RDBMS)」の一つで、その特徴としてCで記述されてサイズも軽量で動作も軽快であること、ソフトウェアパッケージとしても他のパッケージに無依存であること、利用する際には専用のサーバーを必要とせず、前もってさまざまな設定を行わずに手軽に使えること、これらの手軽さに加えて言語的に「ACID」^{*1}対応であること、SQL92版(SQL2)に準拠しているという特徴があります。SQLiteはその名前からも判るようにPostgreSQLやMySQLで構築するような大規模なシステムを組むことには向きですが、軽量で、事前設定(サーバーの設定、ユーザーの設定等々)が不要でただちに利用できるという性質から小規模なシステムに向いています。

この章の目的はSQLについて軽くおさらいをし、SQLliteのSageMathにおける簡単な利用についてその概要を述べることです。より具体的な事例はのちの章で述べることとし、ここではあくまでも簡単な入門とします。

### 6.1.2 SQLについて

まずデータベースシステムはデータを計算機に蓄え、必要に応じて引っ張り出したり更新したりすることができます。だからExcelシートにデータを記載してどこかのフォルダに保存しておくのも^{*2}データベース(DB)と言えなくもないでしょうが、システムと呼ぶには人手に頼りすぎたシステムです。さてSQLiteはRDBMSですが、ここでリレーションナルデータベース(RDB)とは、先頭の「リレーションナル」という言葉から見当がつくように「関係」を考慮したDBです。より正確には「関係モデル」に基づくDBで、あらゆるデータはn項の関係として表現されています。そして「表(テーブル、table)」はこの関係を2次元で可視化したものです。そしてxとyの関係はxがyで述語付けかれていると解釈できます。このことから集合に対する演算によって適合したデータを取り出す処理が行えます。またDBの検索、データの追加・削除や更新は「SQL」と呼ばれる言語が用いられます。このSQLという名前はIBMが開発したRDBMの実証機System Rの操作言語「SEQUEL(Structured English Query Language)」に由来し、その名前が示すようにDBに対して「問い合わせ」を行い、それに応じた処理を行う

^{*1} Atomicity(不可分性), Consistency(整合性), Isolation(隔離性), Durability(持続性)の略語で、トランザクションシステムが持つべき性質としてジム・グレイが定義したものです。

^{*2} どこかの出来の悪い自称IT企業のように!

ための構造化された言語です。このDBへの問い合わせでは、述語からそれに適合する外延を抽出するという側面があり、この抽出では「**関係代数 (relational algebra)**」や「**関係論理 (relational calculus)**」に基づいて処理が行われます。なお、SQLについては完全な形で関係論理を実装していないという批判もあります。またSQLは問い合わせを行うだけの言語ではなく、**表の構造 (schema)**の決定や検索結果の表示方法に至るまでのさまざまなDBの管理がおこなえるようになっています。計算機言語としてのSQLは各RDBMS毎に拡張が行われていましたが、近年はANSIやISOで言語仕様の規格化が行われており、SQLiteは1992年の規格化であるSQL92(SQL2)に準拠しています。

なお、SQLでは命令は大文字でも小文字で記載しても構いませんが、通常は読み易さのためにSQLの構文は大文字を用いて記載されています。ここでも構文として記載する場合は大文字で記載することにします。

### 6.1.3 SQL の文法について

このSQLの文法はデータ定義言語 (DDL: Data Definition Language), データ操作言語 (DML: Data Manipulation Language), データ制御言語 (DCL: Data Control Language) の三種類に大きく分けられます:

■DDL: DDLの文はRDBの構造、具体的には表の組(行)、属性(列)、関係(表)、索引といったDB固有の構造を定義したり、表の削除を行います:

#### データ定義言語 (DDL) の主要な命令

CREATE	DBの対象(表、索引等)の定義
DROP	既存のDBの対象の削除
ALTER	既存のDBの対象の定義変更
TRUNCATE	表からの不可逆的な削除

■DML: DMLの文はRDBに対して登録された対象の検索、削除、更新や対象のDBへの登録を行うための言語です:

#### データ操作言語 (DML) の主な命令

INSERT	データの挿入
UPDATE	表の更新
DELETE	表から指定した行を削除
SELECT	表データから指定した条件に合致するものの抽出

■DCL: DCL の文は利用者の DB に対するアクセス制御を行うことを目的としています:

---

データ制御言語 (DCL) の主な命令

---

GRANT	指定した利用者に DB への特定の作業権限を与える
REVOKE	指定した利用者から特定の作業権限の剥奪を行う
BEGIN	トランザクションの開始
COMMIT	トランザクションの確定
ROLLBACK	トランザクションの取消
SAVEPOINT	ロールバック地点の設定
LOCK	表等の資源を占有

---

■カーソル (cursor): DB 検索結果の集合は表として表現することができます。その表に対して表の行位置 (ポインタ) を指示するものがカーソルになります。また、機能としてはカーソルは検索条件とその結果集合に対応する表の行位置を保持するものと言えます。SQL にはカーソルに対して次の命令があります:

---

カーソルの主な命令

---

DECLARE CURSOR	カーソル定義
OPEN	カーソルのオープン
FETCH	行データを取得
UPDATE	行データの更新
DELETE	行データの削除
CLOSE	カーソルのクローズ

---

ここで FETCH, UPDATE, DELETE はその時点の行位置に対して処理を行い、行位置の変更はありませんが、FETCH の場合は行位置からデータを取り出すと行位置を一つ進める処理を行います。このように処理自体はファイル処理に類似しています。使い方としては

1. 検索条件を指定してカーソルを定義.
2. カーソルを開く.
3. 検索結果に対する処理.
4. 項目の追加等の表の変更処理.
5. 表に変更を行った場合は確定処理.
6. カーソルを閉じる.

といった手順になります。

#### 6.1.4 スキーマ (schema)

「スキーマ (schema)」は DB の構造を決定するものです。RDB の場合は表 (関係) と表内部の属性等の定義になります。このスキーマを次の概念、論理、物理の三層に分けて説明することができます：

##### 概念-論理-物理によるスキーマの分類

- 
- ・ 概念スキーマ 概念間の関係を定義
  - ・ 論理スキーマ 実体とその属性、および実体間の関係を定義
  - ・ 物理スキーマ 論理スキーマの実装
- 

ちなみに SQLite3 では DB のスキーマは `sqlite_master` という特殊な表にスキーマが記録されています。この `sqlite_master` は DB 単位で一つだけ生成され、表、索引、ビューとトリガーの情報が記録されます。この表の構造は次の通りです：

##### SQLITE_MASTER の構造

---

`type` オブジェクトの型、`table`, `index`, `view`, `trigger` の何れか

`name` オブジェクトの名前

`tbl_name`

`rootpage`

`sql` オブジェクト生成で用いられた SQL 文

ここで `table`, `index`, `view` と `trigger` は何れも CREATE 文で生成されるオブジェクトです。この表の `sql` にはそのオブジェクトの生成で実行された SQL 文が保存されています。したがって DB に含まれる表の情報を入手したければ

---

```
SELECT name FROM sqlite_master WHERE type='table';
```

---

で表の名前が入手可能で、表の構造は

---

```
SELECT name FROM sqlite_master WHERE type='sql';
```

---

で入手可能です*3。

#### 6.1.5 DB の有り難味

深く考えずに利用するのであれば、最初に表 (TABLE) を生成してしまえば `INSERT` でデータを指定した表に追加し、`SELECT` で条件に合致するデータを指定した表から取

---

*3 SQLite3 の命令に `.table` や `.schema` でその内容を調べることも可能ですが、Python から SQLite3 を使うときにこれらの命令が使えないため、この `sql_master` を検索することになります。

り出すといった処理が基本です。と書いてしまうと Excel のような表計算ソフトウェアを使って、そちらに表を構成してシートを管理すれば良いのではないかと思われるかもしれません。それも少々のデータで稀に個人が書き直す程度であればそれで十分でしょう。しかし、それなりの規模のデータでありながら参照する事項が僅かなことであったり、複数のシートを横断的に参照する必要がある場合、さらには複数の人間が表の管理に関わるのであれば Excel を利用することは効率が良くて安全な方法ではありません。無理に Excel シートで管理する暇があれば DB を活用すべきです^{*4}。そして SageMath に SQLite が付属しているので、ちょっとした計算結果の保持で、メモを取る暇があるのでならそれを使わない手はないのです。

## 6.2 SQLite のキーワード

表、データベースの名前は何でも使えるという訳ではありません。実際、SQL 文で用いられる言葉（キーワード、予約語）は使えません^{*5} とは言え、SQL で用いられている言葉は SQLだけの特殊な言語ではないために項目名等に使わなければならぬこともあります。その場合には次の表記にしなければなりません：

### キーワード

单引用符 ‘‘で括る	'keyword'
二重引用符 " "で括る	"keyword"
括弧 [ ] で括る	[keyword]
引用符 ‘‘で括る	'keyword'

ここで括弧 [ ] で括るのは ACCESS や SQL Server の流儀、引用符 ‘‘で括るのは MySQL の流儀で、SQLite では各 DB との互換性のためにこれらの流儀が使えるようになっていきます。

## 6.3 SageMath から SQLite を使う

### 6.3.1 sqlite3 について

ここでは SageMath から SQLite を扱う簡単な実例を挙げます。SQLite 単体を立ち上げて利用する実例ではありません。SageMath の実体は Python 環境であるた

^{*4} 急いでいるときにシートを開いて「読み専用で開きます」で悩まされた人、表が大きくなりすぎて開くのに時間がかかり過ぎることで困った人は手を上げて！

^{*5} キーワードの詳細については SQLite Keywords: [https://www.sqlite.org/lang_keywords.html](https://www.sqlite.org/lang_keywords.html) を参照のこと。

め Python から SQLite を扱うというのが実態です。しかし、ここでは SageMath から使うと安易に記載します。なお、Python から SQLite を扱う話の詳細については ‘<http://docs.python.jp/2/library/sqlite3.html>’ も参考にして下さい。

まず Python から RDB である SQLite を利用するためには Python Database API(DB-API) インターフェイスである `sqlite3` を用います。ここで Python Database API とは Python から RDB への共通のインターフェイスを策定したもので、現行の DB-API 2.0 は PEP-249 で定められています^{*6}。このように DB-API によって Python からの RDB の操作方法が定められているので、他の RDB についても共通する操作は同じ作法で行えることになります。この PEP-249 によると RDB への接続は `connect()`、SQL 文の実行は `execute()` で行ないます。より具体的には `connect()` が構築子であり、`Connection` オブジェクトを返します。この `Connection` オブジェクトに対しては次のメソッドがあります：

#### Connection オブジェクトに対するメソッド

---

<code>close()</code>	RDB への接続を切断します。
<code>commit()</code>	トランザクションの確定
<code>rollback()</code>	トランザクションの取消
<code>cursor()</code>	cursor オブジェクトを生成します。

---

実際の RDB への処理は `Connection` オブジェクトを構築したのちに `Cursor` オブジェクトを作つて、そのオブジェクトに対して行うことになります。次に `Cursor` オブジェクトに対する主要なメソッドを示しておきます：

#### Cursor オブジェクトに対するメソッド

---

<code>close()</code>	Cursor を閉じます。
<code>execute()</code>	一つの SQL 文を実行します。
<code>executemany()</code>	
<code>fetchone()</code>	SELECT 等による問合の結果を一件取り出します。
<code>fetchmany()</code>	SELECT 等による問合の結果を取り出します。
<code>fetchall()</code>	SELECT 等による問合の結果を全て取り出します。
<code>arraysize()</code>	<code>fetchmany()</code> で取得する行数の指定を行います。
<code>setinputsizes()</code>	<code>execute()</code> の実行前に操作のパラメータの記憶領域の大きさを設定します。
<code>setoutputsizes()</code>	列のバッファの大きさを指定します。

---

^{*6} PEP-248 は DB-API 1.0 と古い版

ここで重要なのは `execute()` と `fetchall()` でしょう。`execute()` で SELECT 文等の問合を発行し、`fetchone()` や `fetchall()` 等でその結果を取り出すという処理になります。次に簡単な実例を見ることにしましょう。

### 6.3.2 sqlite3 の利用例

SQLite の利用は幾つかの方法があります。まず一つが SageMath に含まれる SQLite を仮想端末から直接使う方法です。もう一つは SageMath を起動してそこから SQLite を利用する方法です。前者の場合はあらかじめ環境設定ができていなければなりません。ただし、OSX 上で Sage.App を起動している場合は SageMath メニュから「Terminal Session」の「Misc.」から「sh」を選択することで環境設定が行われたシェルが立ち上がる所以、そこから ‘sqlite3’ と入力するだけで利用を開始することができます。このことは後述の SageMathCloud でも同様です。

もう一つの方法は Python 向けの DB-API 2.0 インターフェイスである `sqlite3` ライブライアリを Sage 上で読み込んで、そこから利用する方法です。この場合は SageMath を立ち上げてから通常のライブラリと同様に `sqlite3` ライブライアリの読み込みを行い、それから SQLite が使えます。この様子を以下に示しておきましょう：

---

```
sage: import sqlite3
sage: conn=sqlite3.connect("/home/yokota/TEST.db")
sage: conn.execute("create table mycat (name text, age int, weight int)")
<sqlite3.Cursor at 0x4f56180>
sage: conn.execute("insert into mycat values (?,?,?)",('tama',int(4),int(15)))
<sqlite3.Cursor at 0x4f56420>
sage: x1 = conn.execute("select name from mycat")
sage: x1.fetchall()
[(u'tama',)]
sage: x1 = conn.execute("select * from mycat")
sage: x1.fetchall()
[(u'tama', 4, 15)]
sage:
```

---

この処理では `import` 文で `sqlite3` の読み込みを行っています。それからメソッド `connect()` によって ‘/home/yokota/’ 上に生成したデータベース TEST.db に接続します。この TEST.db は指定したディレクトリ上に存在していないければ新規に作成されます。これが PostgreSQL や MySQL であればユーザーの設定等が必要ですが、そういったことはお構いなしにデータベースに接続しているのです。それからメソッド `execute()` を使って SQL 文を実行させています。ここで最初の SQL 文の ‘create table ...’ では mycat という表を生成しています。この表には text 型の name、整数型の age と weight を項目として持つ

ています。それから insert 文で表 mycat に値の書込みを行いますが、このときはメソッド execute() に SQL 文を文字列で与えてしまうと引数の型が全て text 型になってしまふので引数の部分を ‘_’ で置換え、メソッド execute() の第二引数にタプル型として引き渡します。それからあとは select 文で検索を行っています。ここで文字コードが UTF-8 であるために文字列の先頭に UNICODE 文字列であることを示す記号 ‘u’ が付いていることに注目して下さい。なお、文字列として UTF-8 であることを指定せずに ASCII 文字以外の文字を入力するとエラーが発生するので注意が必要です。

このように前処理を行なうことなく簡単に RDB の表を生成して、それを活用することができる所以で、大量の計算結果を配列等に記憶させるだけではなく、RDB の表に登録して、それらの処理を行うといったことも容易に行えるのです。



## 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を行います。SageMath には群論専用の数式処理システム GAP に由来する有限群のライブラリがあり、その中に自由群や組紐群が含まれています。そこで、結び目/絡み目を組紐で表現してしまえばあとは組紐群の話にすることができますが、もちろんそれだけではありません。結び目/絡み目を表現するリストを構築して、そこから不変量を構築することもできます。ところで、結び目/絡み目の不変量の計算では計算過程で膨大なデータが発生します。ここで SageMath は「**車輪の再発明**」を行わないためにさまざまなアプリケーションやライブラリを統合しているので、この計算途上で発生したデータを RDB に保存すると同時にその状態の可視化も行ってみましょう。

## 7.2 結び目/絡み目とは

結び目には「蝶々結び」等の色々な紐の結び方があります。ここで二つの結び目が与えられたときに同じ結び方かどうかを判別するためにはどうすれば良いでしょうか？現実問題としては紐を引いたりして同じ形に変形できるかどうかで判別することになりますが、このことを数学ではどう行けばよいのでしょうか？そこで問題を整理しておきます。まず最初に結び目はどこにあるでしょうか？ここでは適度な大きさのボール、つまり、3 次元球  $B^3$  の中にすっぽりと入っていると考えて良いでしょう。それから結び目は紐の両端を繋いだ円、すなわち 1 次元球面  $S^1$  で考えます。これは実用上の問題で、もしも結び目の両端が切れたままだと一つの端点から別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ることができますが、ここで紐の両端を繋いで輪にしてしまえば解けるものと解けないものが出てくるでしょう。だから端点を繋いで 1 次元球面  $S^1$  にして考察します。

つぎに結び目は 3 次元球  $B^3$  の中にどのように入っているでしょうか？これは自分自身が交わることのないようにきれいな形で入っているべきです。実際、自分自身が交わって複数穴の開いたドーナツみたいな結び目を解けるかどうか考える人はいないでしょう。このように自分自身が交差する事がない状態で 1 次元球面  $S^1$  がボールに限らず 3 次元空間の中にある状態のことを「埋め込み」と呼びます。そして「**一つの 1 次元球面  $S^1$  の 3 次元空間への埋め込み**」のことを「**結び目 (Knot)**」と呼びます。また「**互いに交差しない複数の 1 次元球  $\bigcup_{i=0}^n S_i^1$  の 3 次元空間への埋め込み**」を「**絡み目 (Link)**」と呼びます。当然、絡み目を構成する各成分は結び目になります。そして、円周に反時計回り、時計回りの二種類の向きを入れることができます。これは絡み目でも同様で、絡み目であれば

各成分に独立して向きを入れることができます。そして向きが入った絡み目/結び目のことを「**向き付けられた絡み目/結び目**」と呼びます。

ここで結び目を 3 次元球  $B^3$  の中に埋め込んだものにしましたが、ちょっと美的でません。結び目/絡み目が埋め込まれた空間からそれを除去した空間のことを「**結び目/絡み目の補空間**」と呼びますが、この空間は除去すべき結び目/絡み目を自分自身で交わらない程度に太らせた「**管状近傍 (tubular neighborhood)**」と呼ばれる空間  $N(K)$  を抜き出した空間と違いが位相幾何学でないために通常は  $B^3 - N(K)$  で考えます。この補空間の境界には太った結び目/絡み目のチューブ状の境界と 3 次元球  $B^3$  の表面である 2 次元球面  $S^2$  があり、結び目/絡み目の境界には意味があっても 2 次元球面は無駄です。そこで話を簡単にするために結び目/絡み目が入っている球面の外側に別の 3 次元球を貼って結び目のチューブ状の境界しか現れないようにします。その結果、3 次元球面  $S^3$  が得られますが、このことをもう少し詳しく説明しましょう。まず 3 次元球面  $S^3$  を 4 次元空間内の一定の距離の点の集合  $\{(x, y, z, w) | x^2 + y^2 + z^2 + w^2 = 1\}$  とすると  $B_+^3 = \{(x, y, z, w) | x^2 + y^2 + z^2 \leq 1, w \geq 0\}$  と  $B_-^3 = \{(x, y, z, w) | x^2 + y^2 + z^2 \leq 1, w \leq 0\}$  の二つに均等に分けられます。これらの式からも判るよう  $B_+^3$  と  $B_-^3$  は 3 次元球です。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。これは 2 次元球面も同様で二つの 2 次元球をその境界の 1 次元球面の貼り合わせでできます。また、3 次元球の貼り合わせには別の考え方があります。つまり、境界の球面を 1 点に潰すという考え方です。そうすると 3 次元球面  $S^3$  は通常の 3 次元空間  $\mathbb{R}^3$  に無限遠点  $\infty$  を追加した空間として考えることができます。そのために、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

結び目/絡み目は3次元球面  $S^3$  に埋め込まれた対象です。ただ3次元のままで考察するには流石に難しいために対象を平面に射影して考察します。これが結び目/絡み目の「射影図」と呼ばれる図です。この考え方を SageMath を使って説明しておきましょう：

```
var('t,u,v');
a, b = 3, 1
x = (a + b*cos(u))*cos(v)
y = (a + b*cos(u))*sin(v)
z = b*sin(u)
T = parametric_plot3d([x,y,z],(u,0,2*pi),(v,0,2*pi),opacity=0.3,aspect_ratio=1)
c = (3*t,2*t)
s =[_.subs(dict(zip((u, v), c))) for _ in (x, y, z) ]
K = parametric_plot(s, (t,0,2*pi),color='red',thickness=20,plot_points=200 )
sb =[_.subs(dict(zip((u, v), c))) for _ in (x, y, -10) ]
Kb = parametric_plot(sb,(t,0,2*pi),color='blue',thickness=20,plot_points=200)
(T + K + Kb).show()
```

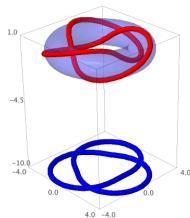


図 7.1 3次元空間内の三葉結び目

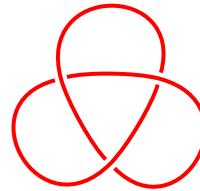


図 7.2 結び目の射影図

図 7.1 に示す絵がこのスクリプトで生成したもので、半透明のトーラス（ドーナツ）T を `parametric_plot3d()` で描き、それからトーラス T とは別個に結び目 K とその  $z=-10$  にある XY 平面への射影図 Kb を描いて三つのグラフィックス・オブジェクトを重ね合わせてメソッド `show()` で表示させています。この図 7.1 の結び目は「**三葉結び目 (trefoil)**」と呼ばれる結び目で、図に示すようにトーラス表面上の曲線として表現されます。この描画では変数として t, u, v を持つ式を用いるためにあらかじめ `var('t, u, v')` でこれらの変数の宣言を行います。そして、この結び目のトーラスへの巻きつき具合は変数 c に束縛したタブル  $(3*t, 2*t)$  で決定されます。ここでトーラスにはその穴の周りを一周する「**緯線 (ロンジチュード, longitude)**」とそれに直交しトーラスの腕を一周する「**経線 (meridian)**」の二つの座標がありますが、このタブルは緯線側を 3 回転する間に経線側

を 2 回転することを意味します。このようにトーラス上の結び目は互に素の整数対  $(p, q)$  で表現することが可能で、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。この図の三葉結び目は「 $(2, 3)$  型のトーラス結び目」になります。さて、この結び目を XY 平面に投影したものが図左下の絵です。先程の緯線側を 3 回転することは読み取れます、この射影図では紐が交差する点でどちらの紐が上で下かという Z 座標に関する情報がこの射影図では欠落しています。流石にこの絵では困るので地図で立体交差を表現する要領で上を通る紐で下を通る紐が切断されたように描きます。そのように描いた射影図が図 7.2 で、これなら交差点の状況も一目瞭然です。さて、この射影図で交差点の上側の紐を「上道」、下側の紐を「下道」と呼びます。これら上道・下道は後述の基本群の構築で生成元として用いられ、交差点の状況で生成元の間で充たすべき関係式が決定されます。ただし、射影にはもう一つ注意しなければならないことがあります。それは交差点では二つの紐だけが交差し、交差するのであれば紐の向こう側に渡りきってしまう性質、つまり、「横断的」と呼ばれる性質を持つべきです。つまり 2 本以上の紐が長々と重なっていたり、3 本以上紐が 1 点で交差すること、それと折れ線の頂点が紐の射影図と重なることを除外します。と言うのも、これらの性質は考察する結び目/絡み目が後述の順であれば紐を局所的に動かせば容易に実現できるためです。このように射影の方向や紐を微調整することで得られた横断的な二重点のみを持つ射影図を「正則射影図」と呼びます。そして絡み目/結び目が向き付けられているときに交点で分断された紐(道)を矢印で表現します。

正則射影図が出たところで結び目/絡み目に現実的な制約を入れましょう。まず、ここで扱う結び目/絡み目はその正則な射影図が有限個の折線で近似可能なものに限定します。この有限個の折線で近似され得る性質を「順 (tame)」と呼び、逆に無限個の折線がどうしても結び目/絡み目の近似で必要なときに「野性的 (wild)」と呼びます。順な結び目/絡み目はその性質から交差点は有限個で、野性的なものは(可算)無限個の交差点を持ちます。順な結び目/絡み目しか扱わないということは現実の結び目/絡み目に限定して特殊なものや病的なものを排除していることを意味します。

## 7.4 結び目/絡み目の同値性

次に与えられた二つの結び目/絡み目が同じ形であるとはどのようなことでしょうか? ここで結び目は 1 成分の絡み目なので、絡み目として話を進めますが、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、紐をに交差させずに変形することで互いの絡み目に移り合えるものが同じ形であると言っても良いでしょう。このことは二つの同じ成分数  $n$  の絡み目  $L_1$  と  $L_2$  の間に「アンビエント・イソ

トピー (ambient isotopy)」と呼ばれる連続写像  $F : S^3 \times [0, 1] \rightarrow S^3 \times [0, 1]$  が存在することに対応します:

——アンビエント・イソトピーの性質——

- $t \in [0, 1]$  に対して  $F_t(x) (= F(x, t))$  は 3 次元球面  $S^3$  の同相写像である
- $F_0 : S^3 \rightarrow S^3$  は恒等写像である
- $t \in [0, 1]$  に対して  $F_t(L_1)$  は共通な部分集合を持たない  $n$  個の 1 次元球面  $S^1$  の和集合と同相である
- $F_0(L_1) = L_1$ かつ  $F_1(L_1) = L_2$

そして、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  $t = 1$  に絡み目  $L_2$  にする映画が存在することと言い換えられます。

ここで結び目/絡み目を正則射影図で考察するためには正則射影図に対してもそのような写像を構築しなければなりません。そこで目的の写像を構築するために式を弄くり倒すよりも良い方法はないのでしょうか？たとえば正則射影図に対して局所的な操作を有限回繰り返すことで互いに移りあえるという操作はどうでしょう？そのような結び目/絡み目の正則射影図に対する操作で「**ライデマイスター移動 (Reidemeister Move)**」と呼ばれる局所的な紐の変形操作で同じ正則射影図が得られる結び目/絡み目は同値で、逆に、同値な結び目/絡み目であればそれらの正則射影図は有限回のライデマイスター移動で移り合えることが知られています：

——ライデマイスター移動——

TYPE I



TYPE II



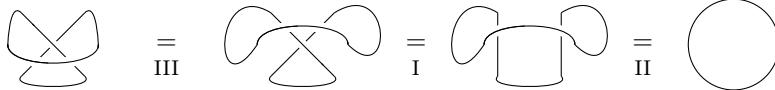
TYPE III



最初の TYPE I は紐に捩れがあったときに、それをまっすぐにしても同じ結び目/絡み目になるという操作で、TYPE II は絡んでいない紐の交差点を解消する操作です。最後の TYPE III は TYPE II を複雑にしたもので交差点付近で絡まっていない紐を並

行移動させる操作です。これらの三種類の操作で 3 次元球面  $S^3$  内部の結び目  $K$  が  $\{(x, y, z) \in S^3 | x^2 + y^2 = 1, z = 0\}$  で与えられる 1 次元球面  $S^1$  に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。つまり、輪になった結び目を切らずに輪ゴムや穴ひとつの大ナットの形に連続的に変形できるものが自明な結び目です。次にライデマイスター移動を使った簡単な例を示しておきましょう：

——ライデマイスター移動の例——



ここではライデマイスター移動の TYPE I, II, III の全て用いて正則射影図の変形を行っています。最初に TYPE III で横紐を上に動かし、下側の捩れを TYPE I で解消します。それから最後は TYPE II で上側の横紐を下に動かして自明な結び目が得られます。なお、最後の TYPE II は TYPE I を左右の捩れに施す操作に置き換えられます。

二つの結び目/絡み目が与えられたときに、その同値性を確認するために知恵の輪よろしく射影図間のライデマイスター移動を試行錯誤して、見つかれば同値と結論付けられることが判りました。では、二つの結び目/絡み目が同値でないことをどう示せば良いでしょうか？「**同値であればライデマイスター移動が必ず存在する**」の対偶から「**ライデマイスター移動が存在しなければ同値でない**」からといって、ライデマイスター移動が二つの同じ成分数の絡み目に存在しないことを示すためにどうすれば良いのでしょうか？そこで結び目/絡み目に固有の値を求めて、それらの比較の話に落とし込むことができないでしょうか？当然、その固有の値はライデマイスター移動で不变な値で、機械的な操作で求まるものでなければなりません。この特徴付けを行うために「**群**」という概念を結び目/絡み目に導入します。

## 7.5 群について

ここでは最初に群について簡単に解説しておきましょう。「**群**」を天下り的に説明すれば「**群は性質の良い二項演算を持った集合**」です。まず、集合  $A$  には「**演算**」と呼ばれる集合  $A$  の二つの元  $a_1$  と  $a_2$  から新しい元 “ $a_1 * a_2$ ” を生成する能力を持ちます。この演算の演算記号 “ $*$ ” を「**演算子**」と呼びます。そして、群を集合と演算の対 ‘ $(A, *)$ ’ で表記し、演算を省略しても問題がないときは群  $A$  と簡単に記述します。

$(A, *)$  が群であれば演算 “ $*$ ” が「**良い性質を持っている**」と述べました。そこで群の

良い性質を列記しておきましょう。まず、集合  $A$  の任意の二つの元  $a, b$  に対して  $a * b$  は必ず集合  $A$  の元になります。これは演算子 “ $*$ ” が集合  $A$  の新しい元を生成する能力を持つことを示しています。この演算子 “ $*$ ” の能力を「**集合  $A$  は演算 “ $*$ ” で閉じている**」と呼びます。たとえば整数の集合  $\mathbb{Z}$  に対して演算を和 “ $+$ ” のときに和 “ $+$ ” は  $\mathbb{Z}$  で閉じていますが、演算が商 “ $/$ ” のときに  $\frac{1}{2}$  が整数でないために閉じていません。このように二項演算は何らかの対象を表記上は生成しますが、演算で閉じていなければ生成した対象が集合  $A$  に含まれる保証はありません。さらに  $\frac{1}{2}$  の例と  $\frac{0}{0}$  は意味が違います。 $\frac{1}{2}$  は集合を有理数  $\mathbb{Q}$  に拡張すれば実在する数ですが  $\frac{0}{0}$  は不定値と呼ばれ、一定の値を持たない表記上の存在です。

つぎの良い性質ですが、この演算 “ $*$ ” が閉じているときに集合  $A$  の任意の 3 個] の元  $a, b, c$  に対して  $(a * b) * c$  と  $a * (b * c)$  が同じ集合  $A$  の元でしょうか？残念ながらこの保証もありません。この ‘ $(a * b) * c = a * (b * c)$ ’ という式は「**結合律**」と呼ばれる性質です。この結合律を充してよいことは最初に  $a * b$  を計算して  $c$  との演算を計算する方法の  $(a * b) * c$  と  $b * c$  を計算して  $a$  との演算を計算する方法の  $a * (b * c)$  の両者に違いがないことが保証されることです。実際、日常の言葉では括弧 “ $()$ ” がなければ微妙な解釈の違いが生じことがあります。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)**」[20] と解釈するかで意味が異なります¹。群であればこのような式の構造上の曖昧さがなくなるために  $(a * b) * c$  なのか  $a * (b * c)$  なのか悩まずに  $a * b * c$  と括弧を外して表記できます。ここで集合  $A$  が演算 “ $*$ ” に対して閉じていて結合律を充すときに「**半群**」、あるいは「**準群**」と呼びます。この「**半**」から予測されるように集合  $A$  が群になるためには閉じた演算  $*$  が結合律を充たさなければなりません。では残りの半分は何でしょうか？それは集合  $A$  の元に関わることになります。

集合  $A$  が群であれば演算 “ $*$ ” に対して特殊な元があります。その一つが「**単位元**」で、 $u \in A$  が単位元であれば集合  $A$  の任意の元  $a$  に対して ‘ $a * u = u * a = a$ ’ を充します。そして単位元は存在すれば一つだけです。なぜなら単位元として  $u$  の他に  $v$  も存在すれば  $u * v$  は  $u$  が単位元なので ‘ $u * v = v$ ’ になります。ところで  $v$  も単位元なので ‘ $u * v = u$ ’、両方を併せて ‘ $u = v$ ’ が得られるからです。この単位元は 1, あるいは  $e$  や  $u$  と表記されます。

では、ここで単位元を持つ半群の例として、絡み目/結び目の集合を挙げておきましょう。ちなみに半群とか群といった数学的構造を持つものは整数や行列といった数から構成され

---

¹ 「太った」の参照範囲（スコープ）が猫だけなのか、猫と犬の両方なのかが分からぬという文の構造のためです。

た対象とは限りません。絡み目や結び目といった幾何学的対象でも、その性質を吟味することで数学的構造を引き出せます。

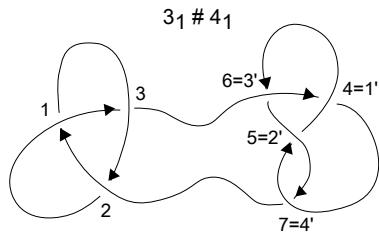


図 7.3 連結和の例

さて、絡み目/結び目の集合には「**連結和**」と呼ばれる操作があります。この連結和という操作は二つの結び目（絡み目であればその一つの成分）の紐でまっすぐな箇所を選び、その箇所を取り外して二つの絡み目/結び目を繋ぎあわせて新しい絡み目/結び目を作るという操作です。図 7.3 に向き付けられた三葉結び目（左側）と八の字結び目（右側）の連結和を示しておきます。連結和は向き付けられた絡み目/結び目に対しては一意に定まり、しかも、左右の被演算子の入れ替えが可能です。実際、結び目  $K_1, K_2$  の連結和  $K_1 \# K_2$  にて  $K_1$  を小さく潰して  $K_2$  上の好きな位置に動かして  $K_2 \# K_1$  に変形することができるためです。また、自明な結び目が連結和の単位元 0 になります。このことから「（向き付けられた絡み目/結び目、#）」は単位元 0 を持つ半群になることが判ります。また、「**二つの結び目の連結和として表現できない結び目**」のことを「**素な結び目 (prime knot)**」と呼びますが、この連結和では延々と結び目を生成するだけで自明な結び目同士でなければ自明な結び目が得られることはありません。実際、 $K_1 \# K_2 = 0$  とするときに  $K_1$  を小さく潰して動かしたところで結び目が解けるのは  $K_2$  が自明な結び目のときだけです。これは  $K_1$  についても同様で  $K_1 \# K_2 = 0$  になるのは双方が自明な結び目 0 のときに限られます。なお、連結和で構成された結び目はスケイン多項式と呼ばれる結び目不変量が連結和を構成する結び目の積になることが知られており、自明な結び目がスケイン多項式で 1 になることから素な結び目は結び目のスケイン多項式の既約性に関係します。そして多項式不変量が 1 になる結び目が自明な結び目に限定されるかという重要な問題があります。

この半群に漂う半端感の原因の「**半**」を外して群になるために必要な条件が単位元と逆元の存在です。つまり、集合  $A$  の元  $b$  が集合  $A$  の元  $a$  の逆元になるのは半群  $(A, *)$  の単位元  $u$  に対して ' $a * b = b * a = u$ ' を充すときです。この元  $b$  を  $a^{-1}$  と表記し、逆元を持つ元  $a$  のことを「**正則元**」と呼びます。単位元を持つ半群  $(A, *)$  が晴れて群になるためには集合  $A$  の全ての元が演算 “ $*$ ” に対して逆元を持つこと、すなわち、集合  $A$  の元が全て正則元でなければなりません。したがって、演算 “ $*$ ” を持つ集合  $A$  が群であるためには、まず、集合  $A$  が半群であり、単位元 1 を持ち、各元が正則元でなければなりません。さきほどの結び目の連結和 “ $\#$ ” は全ての結び目が正則元にならないために結び目は演算

この半群に漂う半端感の原因の「**半**」を外して群になるために必要な条件が単位元と逆元の存在です。つまり、集合  $A$  の元  $b$  が集合  $A$  の元  $a$  の逆元になるのは半群  $(A, *)$  の単位元  $u$  に対して ' $a * b = b * a = u$ ' を充すときです。この元  $b$  を  $a^{-1}$  と表記し、逆元を持つ元  $a$  のことを「**正則元**」と呼びます。単位元を持つ半群  $(A, *)$  が晴れて群になるためには集合  $A$  の全ての元が演算 “ $*$ ” に対して逆元を持つこと、すなわち、集合  $A$  の元が全て正則元でなければなりません。したがって、演算 “ $*$ ” を持つ集合  $A$  が群であるためには、まず、集合  $A$  が半群であり、単位元 1 を持ち、各元が正則元でなければなりません。さきほどの結び目の連結和 “ $\#$ ” は全ての結び目が正則元にならないために結び目は演算

が連結和 “#” のときに単位元を持つ半群であっても群になりません。

以下に群の条件を纏めておきましょう:

群の条件

- 演算  $*$  に対して閉じている:  
 $a, b \in A \rightarrow a * b \in A$
- 結合律が成立:  
 $(a * b) * c = a * (b * c)$  を充す.
- 単位元 1 の存在:  
 $a * 1 = 1 * a = a$  となる  $1 \in A$  が存在する.
- 逆元の存在:  
任意の  $a \in A$  に対し,  $a * b = b * a = 1$  を充す  $b \in A$  が存在する.

ここで最初の二つの条件が半群の条件です。また、演算 “*” が常に ‘ $a * b = b * a$ ’ を充すときに演算 “*” を「可換 (commutative)」, 群  $(A, *)$  を「可換群 (commutative group)」と呼びます。また、演算が可換でなければ「非可換 (non-commutative)」, 非可換な演算を持つ群を「非可換群 (non-commutative group)」と呼びます。可換群の例としては整数  $(\mathbb{Z}, +)$  や有理数  $(\mathbb{Q}, \times)$ , 非可換群の例としては  $n$  を 2 以上の自然数とするときの  $n$  次正方行列  $(M(n), \cdot)$  を挙げておきます。

では群がどのようなものか具体的に表記する方法はないでしょうか? 一つの方法として群の表示というものがあります。ここで群の表示の説明のために語の集合の例を挙げて説明しましょう。まず集合  $A$  を  $a$  から  $z$  までのローマ小文字を並べた文字列(「語」と呼びます)と空白 “ ”(ここでは 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  $mike * neko$  の結果は  $mikeneko$  と演算子 “*” の左右の語を繋いだ文字列です。また  $WWWWW \dots$  のように同じ語  $W$  が  $n$  回続くときに  $W^n$  と表記します。ここで空白 “ ” を文字として考えると、この空白を語の左右に繋いでも元の語になるので空白 “ ” が演算  $*$  の単位元になることが判ります。そして、語  $W_1 W_2 W_3$  は  $(W_1 * W_2) * W_3$  と  $W_1 * (W_2 * W_3)$  の双方から構成されること^{*2}から演算子 “*” は結合律を充します。このように単位元が存在して結合律を充すので  $(A, *)$  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

^{*2} 「太った猫と犬」の例のようにその意味を考えることはなく、単に文字の羅列としてです。

$W^{-1}$  は語  $W$  の逆元になり、以上から  $(A, *)$  は群になることが判ります。ここで群  $A$  の元は  $a$  から  $z$  までのローマ小文字で構成されるために、そのことが判るように  $\langle a, \dots, z \rangle$  と記述します。ここで用いた記号 “...” は Python の拡張スライス構文で省略を意味する Ellipsis というオブジェクトになりますが、ここでの表記も同様の省略を意味する記号です。このアルファベット小文字の例と同様に  $n$  個の対象  $a_1, \dots, a_n$  を連結することで生成されるものも同様に群になるので、これも  $\langle a_1, \dots, a_n \rangle$  と表記することにしましょう。そして、この群を「**自由群**」と呼びます。そして  $\langle a_1, \dots, a_n \rangle$  の中の対象  $a_1, \dots, a_n$  を「**群の生成元**」と呼び、生成元が  $n$  個の自由群を  $F_n$  と表記します。このように自由群は単純にその群の元を繋ぎ合せる処理と削除を持つ集合であると言えます。また、積の順序を入れ替えることは語順を入れ替えることと同値で、その結果、もとの語とは別の語ができてしまうために自由群は通常は非可換群になります。実際、最初の語の例で ‘inu’ と ‘uni’ は別物です。ただし、生成元が一つだけのときのみ可換群になります。実際、一成分  $a$  だけであれば、結合律から  $a^m * a^n = \underbrace{a * \dots * a}_m * \underbrace{a * \dots * a}_n$ 、結合律によって括弧 “( )” の付け替え操作が可能なので  $\underbrace{(a * \dots * a)}_n * \underbrace{(a * \dots * a)}_m$  であることが判ります。このことで可換性:  $a^m * a^n = a^n * a^m$  が証明できます。

なお、自由群は統々とその元を演算 “*” を使って生成できる群ですが、全ての群がそのようなものではありません。たとえばスイッチボタンによる ON/OFF 操作も群としての構造を持ちます。実際、スイッチボタンを押すという操作を  $a$  とすると生成元はこの  $a$  だけになりますが、状態は ON と OFF の二つだけで、さらにボタンの二度押しで元に戻ることから  $a^2 = 1$  という「**規則**」があることが判ります。このように群の持つ規則はその群の元で構成される式、すなわち、「**関係式**」で表現されます。スイッチの例では  $a^2 = 1$  がその関係式になります。だから集合の表記にならって  $\langle a | a^2 = 1 \rangle$  とこの群を表示します。この方法に倣って、群の生成元が  $a_1, \dots, a_n$  で関係式が  $r_1, \dots, r_m$  であれば群を次で表示します:

群の表示

$$\langle a_1, \dots, a_n | r_1, \dots, r_m \rangle$$

ここで関係式を変形して単位元 1 に等しい式で置き換え、さらに ‘= 1’ を自明なものとして省略することができます。スイッチの例では関係式が  $a^2 = 1$  のために  $\langle a | a^2 \rangle$  という群の表示にします。そのような 1 に等しくなる式  $r_1, \dots, r_m$  のことを「**関係子 (relator)**」と呼びます。この関係子で群  $G$  の表現を与えると群  $G$  の生成元で生成される自由群  $F_n$  から群  $G$  への至つて自然な写像が考えられます:

$$\begin{array}{ccc} f & : & \langle a_1, \dots, a_m \rangle \rightarrow \langle a_1, \dots, a_m | r_1, \dots, r_n \rangle \\ & \Downarrow & \Downarrow \\ a_i & \mapsto & a_i \end{array} \quad i \in \{1, \dots, m\}$$

この写像  $f$  は自由群  $F_n$  の生成元  $a_i$  をそのまま群  $G$  の生成元  $a_i$  に写すだけの写像なので自由群での積  $ab$  はそのまま群  $G$  での積  $ab$  に対応させ、自由群  $F_n$  の単位元  $1$  もそのまま群  $G$  の単位元  $1$  に写します。つまり、 $f(ab) = f(a)f(b)$  と自由群側の演算を書いた側の群でも保ち、 $f(1) = 1$  と単位元を単位元に写す性質があります。この積と単位元を保つ性質を持つ群から群への写像を「(群) 準同型写像」と呼びます。

この写像  $f$  によって自由群  $F_n$  の項である関係子  $r_{i,i \in \{1, \dots, m\}}$  は全て  $1$  に写され、さらに、それらの逆元も積も群  $G$  の単位元  $1$  に写されます。また写像  $f$  は準同型であることから、自由群  $F_n$  の単位元  $1$  も写像  $f$  で群  $G$  の単位元  $1$  に写されます。このことから関係子からも群が構成されることが分かります。この関係子の集合のように群の部分集合で群の性質を充たすものを「部分群」と呼びます。そして、関係子が構成する群のことを特に「帰結群」と呼びます。

一般的に群  $G_1$  から群  $G_2$  への準同型写像  $f: G_1 \rightarrow G_2$  で群  $G_2$  の単位元  $1$  に写される群  $G_1$  の元の集合を  $\text{Ker}(f)$  と表記して準同型写像  $f$  の「核 (kernel)」と呼びます。この核  $\text{Ker}(f)$  は帰結群の議論で見たように群  $G_1$  の部分群になります。ここで見たように関係子を記述することは自由群からの自然な準同型写像の核を記載することに対応します。また、基本群の関係子はそれ自体に幾何学的な意味を持っています。基本群の生成元は共通の基点を持つ閉じた道が対応し、3次元空間の基本群の関係子は、関係子に対応する道に自己交差のない滑らかな2次元円盤が貼れることを意味します。

## 7.6 結び目/絡み目を表現する群

### 7.6.1 基本群と組紐群

結び目/絡み目に固有のものに、それらの補空間の「**基本群 (fundamental group)**」と呼ばれる群があり、これらの群は単に「**結び目/絡み目群**」と呼ばれます。また、結び目/絡み目を組紐と呼ばれるものに変形することで「**組紐群**」と呼ばれる群が構築できます。これらの群の構成は正則射影図から容易に行うことができます。ここでは基本群と組紐群について概要を述べましょう。

### 7.6.2 基本群について

ここでは結び目/絡み目の基本群の構成手順を述べます。結び目/絡み目の基本群の構成では向き付けをした正則射影図を用います。

この正則射影図の向き付けは結び目/絡み目を構成する各1次元球面に向きを入れ、その向きを正則射影図で矢印で表現したものです。たとえば三葉結び目に向きを入れた正則射影図は図7.4のように各道が矢印になった図です。このときに各交点での交差の状況が基本群の関係子を決定しますが、絡み目/結び目の穂空間の基本群の構成方法には「Wirtinger表示」と「Dehn表示」の二種類の群の表示方法に関連した手順があります。ここではWirtinger表記による群の構成を解説します。このWirtinger表示による結び目/絡み目群の表示は次の形になります：

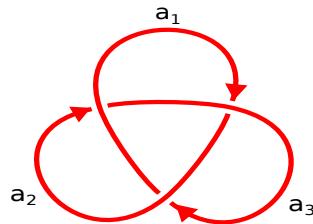


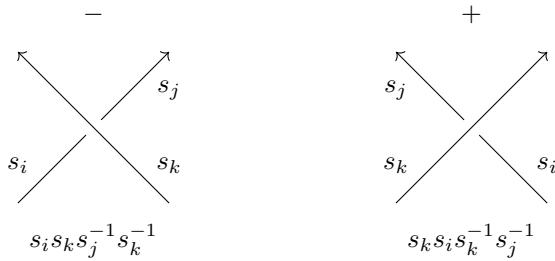
図7.4 向き付けられた正則射影図

#### 結び目/絡み目群の Wirtinger 表現

$$\text{結び目/絡み目群} = \langle \text{上道}_1, \dots, \text{上道}_n \mid \text{関係子}_1, \dots, \text{関係子}_n \rangle$$

ここで  $n$  は正則射影図の交点の数、すなわち道の総数になります。それから関係子は交点で定まります。ただし  $n$  番目の関係子は他の  $n - 1$  個の関係子からも生成できるために不要です。これらの関係子は次で与えられます：

#### Wirtinger 表示の関係子



各交点の上にある  $+1$  と  $-1$  は「交点の符号」と呼ばれます。なお、この結び目/絡み目の正則射影図  $\mathcal{D}$  の交点の符号の総和を「捻れ」と呼び、 $w(\mathcal{D})$  と表記します。

このように結び目/絡み目の基本群の表示の構成自体は非常に機械的に行うことができます。ただし、群の表示が得られたからといって、その群が同じかどうかを確認すること

は一般的に簡単なことではありません。また、基本群は幾何学的な構造を反映する群ですが、誰もが直感的・視覚的に理解できる群ではありません。そこでより視覚的に分かりやすいもう一つの結び目/絡み目を表現する群である組紐群 (Braid group) について解説しましょう。

### 7.6.3 組紐について

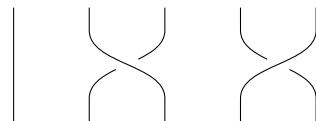
結び目/絡み目に関連する群には基本群だけではなく組紐群もあります。この組紐群は組紐と呼ばれる図形に関する群で、基本群よりも群としての構造が誰にでも判るように可視化することができます。まず数学上の「**組紐 (Braid)**」は  $n$  本の紐 (閉区間  $[0, 1]$  と同相) の 3 次元球  $B^3$  への埋め込みの一種です：

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  $B^2 \times 0$  での紐の交点は等間隔に並ぶ。

ここでの定義から判るように組紐は酒樽みたいな 3 次元球  $B^2 \times [0, 1]$  の上下の蓋 ( $B^2 \times 1$  と  $B^2 \times 0$ ) に取り付けられた紐で、その紐は互いに交差することなく上から下へと垂れた状態であり、紐が縦軸に対して互いに右回りや左回りで絡んでいます。このことから組紐は以下の基本的な 3 種類の成分で構成されることが分かります：

組紐の基本成分



組紐の一例として図 7.5 に SageMath の BraidGroup モジュールを使って描いた組紐を示しておきます。この例では 3 本の紐で構成された組紐です。このように組紐は非常に具体的なもので、その構成も基本成分をどのように積み上げるかで決まります。そして、紐の数が  $n$  本のときに「 $n$  糸の組紐」と呼びます^{*3}。なお、組紐の充たすべき性質で 2. を充たさない紐の埋め込みのことを「**タングル (tangle)**」呼びます。タングルは組紐のように紐が上蓋から出発して下蓋に向かって下がる一方でなく、紐に結び目を作ることも可能になるために組紐よりも複雑な图形に

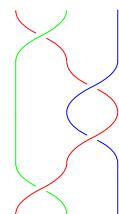
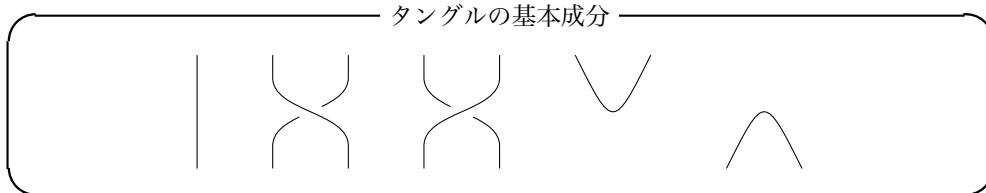


図 7.5 組紐の例

^{*3} 配置空間 (configuration space) を使った定義もあります [27]。

なりますが、基本成分は新たに「**最小(消滅)**」と「**最大(生成)**」の二つが加わるだけです：



ここで紐に向きを入れていないために 5 成分ですが、向きを入れたタングルでは組紐のように上から下への一方的な向きになるとは限らないために、この 5 成分から二つの交差を除いた成分を基に 8 成分になります。それに応じて正則射影図上の変形操作も組紐よりも操作が増えます。

ここで与えられた二つの組紐が同じであるということは、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です。このことは二つの組紐に「**アンビエント・イソトピー (ambient isotopy)**」が存在するときと結び目/絡み目のときと同様に言い換えられます。ただし、組紐は結び目/絡み目よりも図形的な制約が入っていることからより代数的な表現に訴えることができます。それが組紐群と呼ばれる群です。

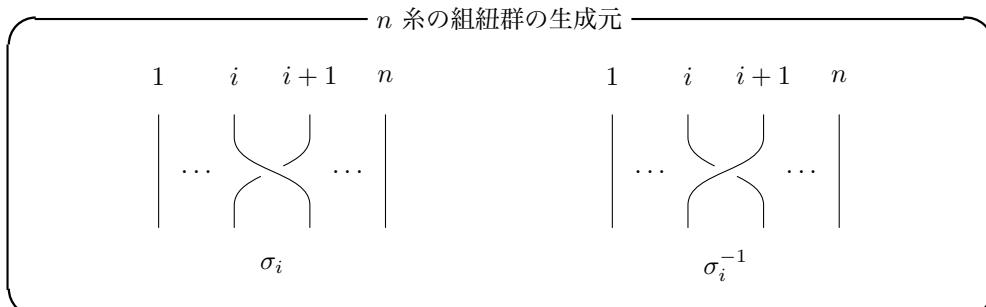
#### 7.6.4 組紐群

組紐からは「**組紐群 (Braid group)**」と呼ばれる群が構成されます。この群は紐が  $n$  本の組紐、すなわち  $n$  糸の組紐であれば次のように表示することができます：

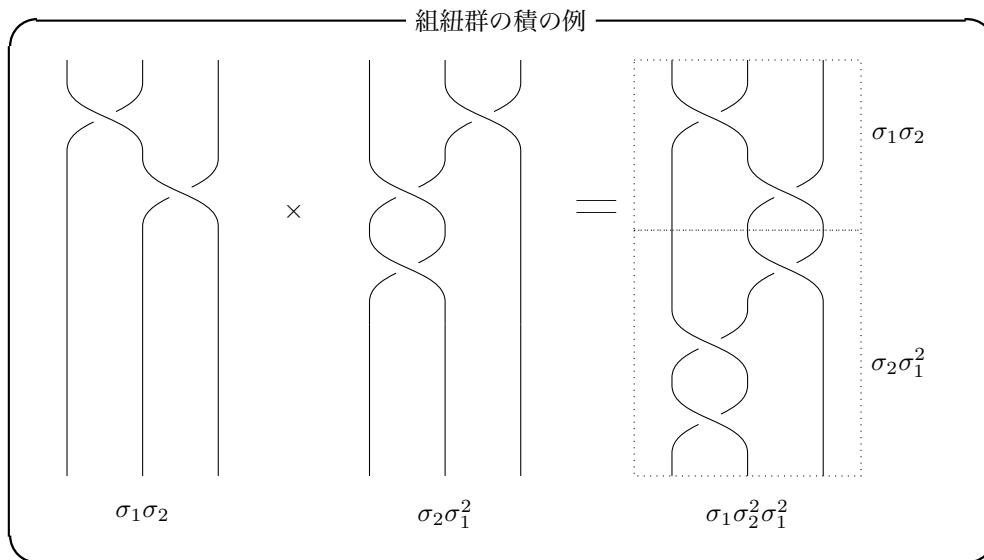
$n$  糸の組紐群

$$\left\langle \sigma_1, \dots, \sigma_{n-1} \mid \begin{array}{l} \sigma_i \sigma_k = \sigma_k \sigma_i, (|i - k| \geq 2, i, k \in [1, n-1]), \\ \sigma_i \sigma_{i+1} \sigma_i = \sigma_{i+1} \sigma_i \sigma_{i+1}, (i \in [1, n-2]) \end{array} \right\rangle$$

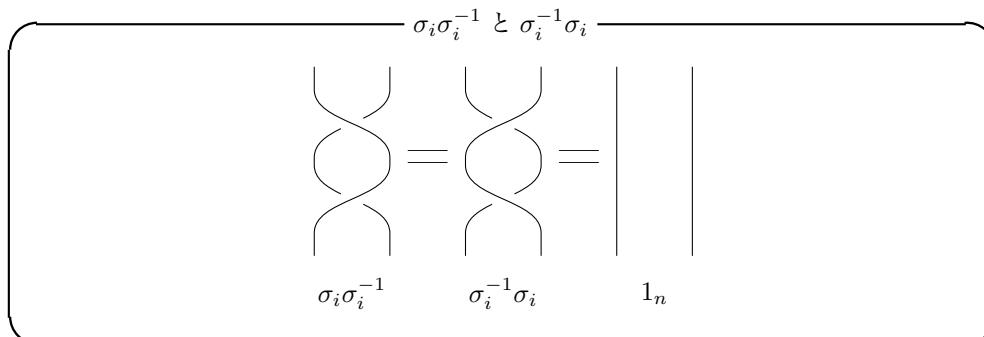
では生成元  $\sigma_i$  は具体的にどのようなものでしょうか？次に  $\sigma_i$  と  $\sigma_i^{-1}$  と対応する組紐を示しておきましょう：



ここで図示したように  $\sigma_i$  は左から  $i$  番目と  $i+1$  番目の紐を縦方向を  $Z$  軸として反時計回りに 180 度回すという操作です。それに対して  $\sigma_i^{-1}$  は紐を反対の時計回りに 180 度回す操作になります。ここで  $\sigma_i^{-1}$  の右肩の  $-1$  の意味は組紐群の積に対応する操作の結果から判ります。それから組紐群の積演算は二つの組紐  $a, b$  が与えられたときに組紐  $a$  を上、組紐  $b$  を下にしてこれらの組紐を縦に繋ぐ操作に対応します。具体的な例を以下に示しておきましょう：



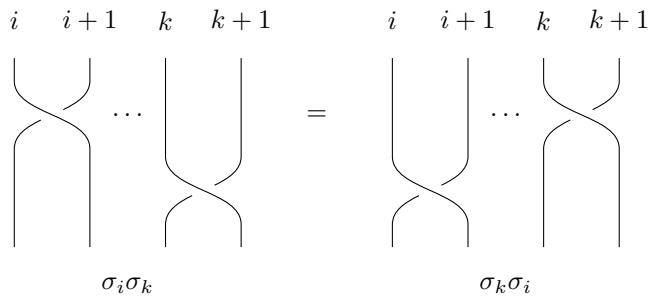
ここでの例では  $\sigma_1\sigma_2$  と  $\sigma_2\sigma_1^2$  の積  $\sigma_1\sigma_2\sigma_1^2$  を示しています。ここで右に示すように  $\sigma_1\sigma_2$  を  $\sigma_2\sigma_1^2$  の上に積み上げる形になります。また積の順番から言えば、左側から右にかけて対応する組紐を上から下に繋ぐ恰好になります。さて、ここで  $n$  本の紐がまっすぐに垂れた組紐を  $n$  級の組紐の上に付けても下に付けても各紐の長さを調整してしまえば元の  $n$  級の組紐に戻せます。つまり、このことはまっすぐに垂れた  $n$  本の組紐が  $n$  級の組紐群の単位元  $1_n$  であることを示しています。また  $\sigma_i$  に対する  $\sigma_i^{-1}$  の意味ですが  $\sigma_i\sigma_i^{-1}$  と  $\sigma_i^{-1}\sigma_i$  を描いてみると共に単位元  $1_n$  と同じ組紐になることが判ります：



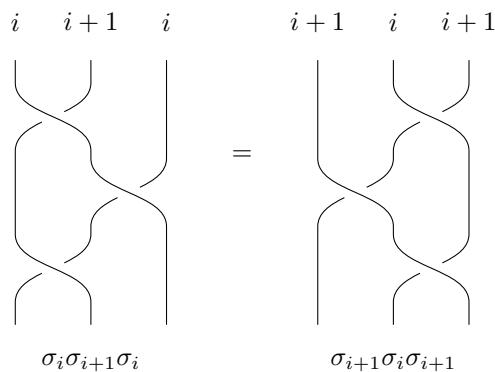
実際、左側の状態の紐を引っ張って長さを調整すると 2 本のまっすぐな紐にすることが

できますね. このことから  $\sigma_i^{-1}$  は表記どおりに  $\sigma_i$  の逆元になっていることが判ります. 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幂表示の正負も逆にしたもので与えられることも判ります.

ここまで群の表示の生成元は理解できたでしょう. 要するに組紐を重ねるという操作がそのまま組紐を表現する項同士の積なのです. それから群の表示には生成元の右側に二つの式:  $\sigma_i\sigma_k = \sigma_k\sigma_i$  と  $\sigma_i\sigma_{i+1}\sigma_i = \sigma_{i+1}\sigma_i\sigma_{i+1}$  があります. この関係式の意味は図示すると非常に明瞭なものになります. そこで最初に  $\sigma_i\sigma_k = \sigma_k\sigma_i$ ,  $|i - k| \geq 2$  を図示しましょう:

 $\sigma_i\sigma_k = \sigma_k\sigma_i$  の図示

この図から  $|i - k| \geq 2$  のときに  $\sigma_i\sigma_k$  と  $\sigma_k\sigma_i$  は互いの操作が影響する範囲にないために交点を上下を動かすことが可能だということに対応する式であることが判りますね. この上下に紐を移動させるという操作で互いの組紐が得られることから, これらの組紐を同じものとみなすということに問題はないでしょう. では次に  $\sigma_i\sigma_{i+1}\sigma_i = \sigma_{i+1}\sigma_i\sigma_{i+1}$  で表現される組紐の図示を行ってみましょう:

 $\sigma_i\sigma_{i+1}\sigma_i = \sigma_{i+1}\sigma_i\sigma_{i+1}$  の図示

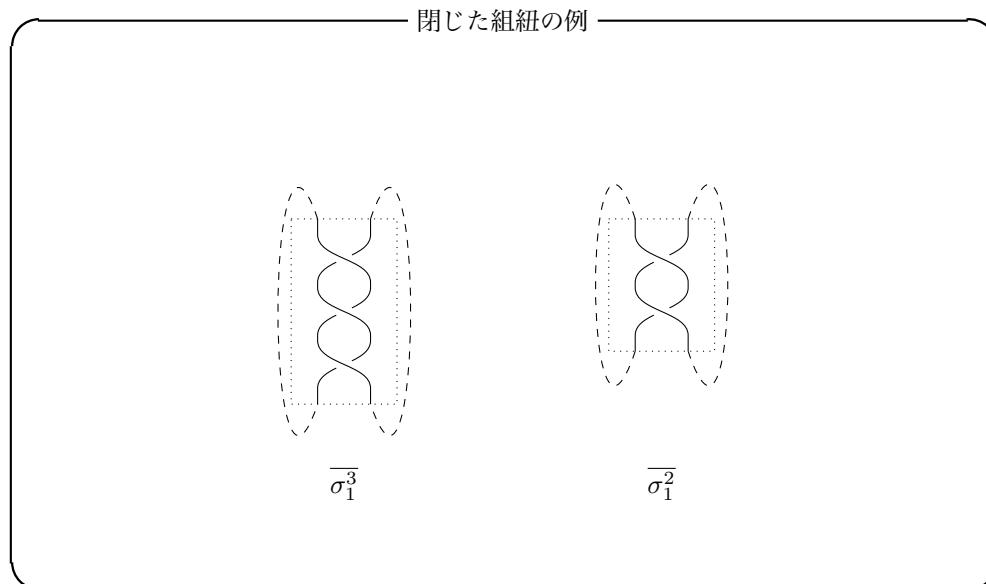
この関係式も図示してしまえば明確なものです. 実際, この関係は  $i$  番目の紐を上下に

動かすことで相互の組紐に変形することができることを意味します。この操作は同値な結び目/絡み目が得られるライデマイスター移動の TYPE III に対応する操作です。このように関係式は項の同値性を与える式で、この式を利用して項をより簡単なものにまとめることが可能ですが、この関係式を可視化することで図形的な解釈が明瞭になるのです。

### 7.6.5 組紐と結び目

組紐の上下の端点を交互に繋ぐ操作で得られる図形を「閉じた組紐 (closed braid)」と呼びます。そして閉じた組紐で幾つかの円が得られます。円が一つだけの場合は結び目、円が二つ以上の場合は絡み目になります。ここでは縦に閉じた  $n$  糸の絡み目  $b$  を  $\bar{b}$  と表記します。

ここで実際に例をみておきましょう：



この例は 2 糸の組紐を縦に閉じたもので、他に上で繋ぐ方法もありますが、その場合は紐の本数が偶数個必要になります。ここで採用した縦に閉じる方法は紐の本数と無関係に閉じることができます。例の左側の結び目は  $\sigma_1^3$  から得られる「**三葉結び目**」と呼ばれる結び目で、右側が  $\sigma_1^2$  から得られる「**ホップ絡み目 (Hopf Link)**」と呼ばれる絡み目です。

このように組紐を閉じることで結び目/絡み目が得られますが、逆に「**結び目/組紐は閉じた組紐として表現可能である**」ことが知られています [8]。この閉じた組紐としての結び目の記述を「**結び目の組紐表現**」と呼びます。このときに結び目の組紐表現で最小の組紐の本数を「**組紐指数 (braid index)**」と呼び、この組紐指数は結び目の不变量の一つであ

ることが知られています。

なお、二つの閉じた  $n$  糸の組紐がアンビアント・イソトピーで移りあえるのは次のマルコフ移動 M I, M II を許容するときに限ることが知られています：

マルコフ移動

$$\begin{array}{l} \text{M I. } \overline{ab} = \overline{ba} \\ \text{M II. } \overline{a\sigma_n} = \overline{a\sigma_n^{-1}} = \overline{a} \end{array}$$

これも図示すると明快になります。移動 M I は  $a$  を構成する生成元を閉じた組紐であることから  $a$  の上から順にまわして  $b$  の下に移すことができることに対応します。また移動 M II は最も右側の紐がライデマイスター移動の I と同じ状況になっていることに対応しますが、ただ、この場合は紐の数に変動が生じます。

## 7.7 組紐群と置換群

ここで  $n+1$  糸の組紐の上蓋で紐の端点に左側から番号を  $1, 2, \dots, n$  と振ります。それから紐をたどって下蓋に到着したところで紐に対応する番号を配置します。すると上蓋の  $1, 2, \dots, n$  は下蓋では並び替えられています。この並び替える操作を  $\sigma$  と表記すると  $\sigma$  は  $\{1, 2, \dots, n\}$  から  $\{1, 2, \dots, n\}$  への全単射写像になります。また上蓋の  $1, \dots, n$  が  $(x_1, \dots, x_n)$  に対応するときには

$$\sigma = \begin{pmatrix} 1 & \dots & n \\ x_1 & \dots & x_n \end{pmatrix}$$

と表記することにしましょう。このような  $\{1, 2, \dots, n\}$  から  $\{1, 2, \dots, n\}$  への全単射写像の集合に対し、その演算を函数の合成○とすると、この集合は群になります。この群を「置換群  $S_n$ 」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  $S_n$  への自然な写像が得られ、しかも、この写像は積を保つので準同型写像になります。ただし、この準同型写像は「情報の欠落」が生じます。たとえば  $\sigma_i$  と  $\sigma_i^{-1}$  は置換群としては  $i$  と  $i+1$  を入れ替える操作なので一致しますが、組紐群としては回転の方向が反対で別物なのです。このように紐の回転の向きの情報に欠落が生じているのです。

この置換群を使うと組紐を閉じたときに得られた絡み目の成分数が置換群の分析から判ります。この分析では置換群を巡回置換の積として表現しなければなりませんが、ここで置換  $\sigma$  と  $k \in [1, \dots, n]$  に対して  $k \xrightarrow{\sigma} \sigma(k), \sigma(k) \xrightarrow{\sigma} \sigma^2(k), \dots, \sigma^i(k) \xrightarrow{\sigma} \sigma^{i+1}(k) = k$  と  $k$  の置換  $\sigma$  による像を追いかけることで次の置換：

$$\begin{pmatrix} k & \dots & \sigma^{i-1}(k) \\ \sigma(k) & \dots & \sigma^i(k) \end{pmatrix}$$

が得られます。これが「巡回置換」と呼ばれる置換で、より簡潔に  $(k, \sigma(k), \dots, \sigma^i(k))$  と表記されます。一般的に任意の置換  $\sigma \in \mathfrak{S}_n$  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  $\sigma$  で写して巡回置換を求め、この巡回置換で現れなかった数に対して 1 の時と同様に置換  $\sigma$  で写して巡回置換を求めます。この作業で全ての数が出ると置換  $\sigma$  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  $\sigma$  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  $\sigma$  に写されたとします。ここで巡回置換は  $k$  番目の始点から出発して  $\sigma^{i+1}(k)$  で出発点に戻るために、その軌跡が一つの円周を構成していることが判ります。そして、置換  $\sigma$  が巡回置換の積として表現されることから、その巡回置換の数だけ円周が、すなわち、絡み目の成分が現われることになります。

### 7.7.1 ザイフェルト曲面

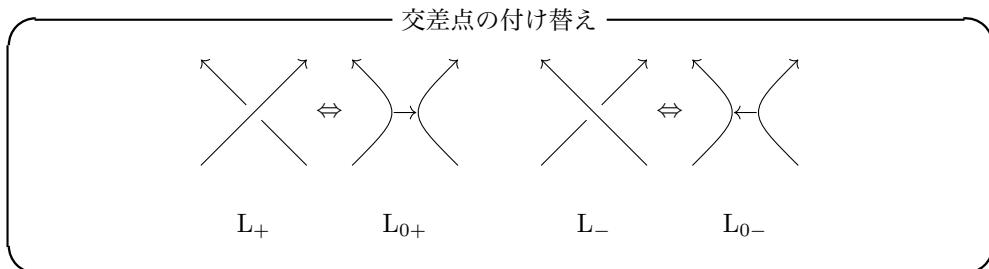
任意の結び目と絡み目には組紐としての表現を持つと述べました。この組紐への変換を機械的に行う処理がありますが、この処理は後述の結び目/絡み目の不変量の計算で用いるさまざまな操作が含まれているために、この組紐への変換操作について述べます。また、この変換では「ザイフェルト曲面 (Seifert surface)」と呼ばれる重要な曲面の作成手順と重なるために、この曲面の構成方法についても一緒に述べることにします。このザイフェルト曲面は、その境界が結び目/絡み目になる向き付け可能な曲面です。ここで向き付け可能という意味は「裏と表がある曲面」で、曲面を白と黒の二色で塗り分けることができるということです^{*4}。そして、向き付け可能な曲面は「種数 (genus)」と呼ばれる自然数で完全に分類されています。この種数は「ドーナツの穴の数」に対応し、種数 0 は 2 次元球面  $S^2$ 、種数 1 がトーラス  $T^2$ 、すなわち穴一つのドーナツの表面になります。

**■正則射影図の準備:** 結び目/絡み目の正則射影図を準備します。このときに結び目/絡み目に向きを入れて向き付けられた正則射影図にします。その結果、全ての道は向きの入った矢印です。ここでもしも交差点が一つも存在しなければ絡み目の全ての成分は自明な結び目であり、円盤を貼りつけることでザイフェルト曲面が得られます。しかし、通常は交点が存在するために各交点に対して次に述べる交点の解消処理を行います。

**■交点の解消:** 絡み目の向き付いた正則射影図の各交点を次の手順で書き直します：

---

^{*4} 向き付かない曲面の代表格が帶に 180 度の捻りを入れて繋いでできるメビウスの帯 (Möbius band) です。



これらの交点の書き直しは、交点に入る下道は交点から出る上道に繋ぎ、交点に入る上道と交点から出る下道に繋ぐという処理です。この処理を結び目の正則射影図で行うと成分の分離が生じますが、上道と下道が絡み目の別成分であれば二つの絡み目の成分が一つに融合します。この処理を全ての交点に対して行うと正則射影図から交点が消えて幾つかの孤立した円周とそれらを繋ぐ矢印だけが残っています。この最終的に得られる正則射影図の円周のことを「ザイフェルト円周」と呼び、ザイフェルト円周とそれらを繋ぐ矢印のことを「ザイフェルト系」と呼びます。この道の付け替え操作は後述のガウス・コードを利用すれば容易に SageMath のプログラムで表現することができます。ここでザイフェルト系の各円周を繋ぐ矢印の意味はそれらを繋ぐ橋として各交差点の符号に対応した捩れの帯を表現します。

言葉だけでは難しいので、実際に結び目を使って変形操作を説明しましょう。まず、結び目の正則射影図に向きを入れ、交点を線分で置換えたものを以下の図 7.6 に示します：

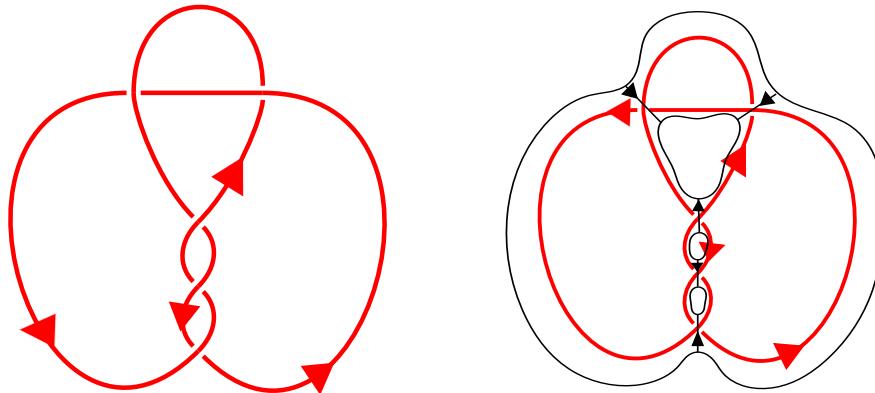


図 7.6 結び目と交点の変換

左側の向きを入れた結び目に対して交差点での変形操作を行うことで右図のザイフェルト系が得られます。このザイフェルト系では 4 個のザイフェルト円周と交点に対応する 5 個の矢印で構成されます。

**■ザイフェルト曲面の構成:** 上の交点の解消で得られたザイフェルト系のザイフェルト円周に円盤を貼り付け、矢印に本来の帶の正負に対応する捻りを入れた帶で置き換えることで得られる曲面がザイフェルト曲面です。この曲面は構築手順から向き付られた曲面で、その境界は結び目/絡み目になります。なお、曲面の構築を次の円の向きを揃えた時点で行つても構いませんが、円の向きを揃える操作によって後述のように線分が増え、種数が増加した曲面になります。だからと言って、この時点で得られた曲面が最少種数の曲面になるとは限らず、円盤の貼り方にも二通りの方法があります。これは結び目/絡み目を一点コンパクト化で3次元球面  $S^3$  内の対象として捉えたことから、円盤を無限遠点を含まないように貼る通常の貼り方と無限遠点を含むように貼る貼り方の二通りがあります。無限遠点を含まないように貼るとでき上がった曲面はホットケーキを数段重ねたような形になることがあります、無限遠点を含むように貼ると幾らか平面的な曲面ができ上がります。図7.7にこれら二通りの円盤を貼り付けたものを示しておきます：

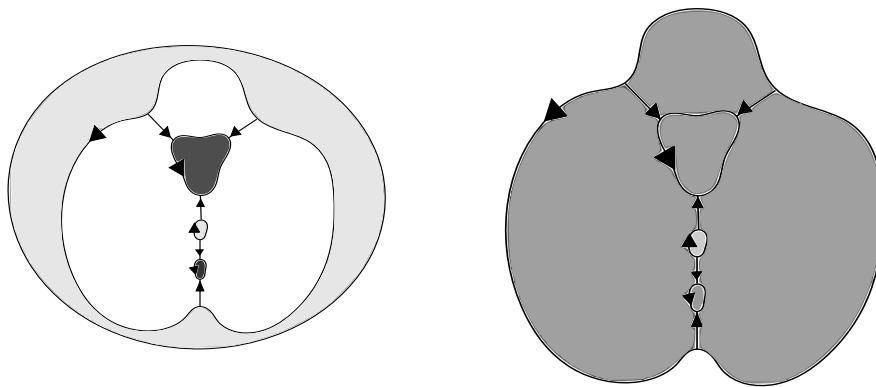


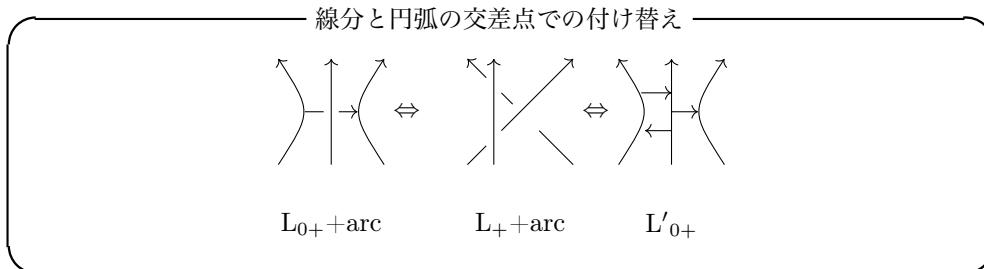
図7.7 結び目の円盤

左側が無限遠点を含むように円盤を貼ったもので、右が有界な円盤のみを貼り付けたものです。左側の方法では平面上に配置されて孤立した円盤を帶で結び付けるかたちになりますが、右側では有界な円盤を貼った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ることになります。このように円盤の配置が右図では同一平面上にありません。そして、これらの円盤を繋ぐ線分がそれぞれ対応する符号の捻りを入れた帶で置換えてザイフェルト曲面が構築できます。これでザイフェルト曲面の構築は終わりです。ところが、ここで得られたザイフェルト系に手を加えれば組紐表現に持ち込むことができます。そのため同心円状にザイフェルト円周を変形する必要があり、この手順を以下に示しておきます。

**■円の向きを揃える:** ここで円盤を貼る前のザイフェルト系の状態に戻って操作を行います。この時点ですべての円の中心が僅かな平行移動で一致し、円の向きが一致していれば線分を円の左側に平行移動します。円の中心と向きが揃っていないければ各円の中心を一致させて向きも揃えます。そのために以下の手順で作業を行います：

1. 基準となる円を一つ定める。作業を楽にするために複数の円の最も内側になっているもの、あるいは線分が一番多く出ているもののいずれかを基準にする。
2. 基準になる円に線分でつながった円で、その向きが一致した円は交差しないように中心点を揃える。逆向きの円があれば構築手順から基準にした円を内部に含むことがないために、その円から出ている線分を適宜、平行移動させて、線分のない側の円弧を動かして基準になる円を包含するようにできる。このときに基準にした円から出ている線分と交差が生じるが、この線分は捻りの入った帶であるために交点が新たに二つ生じる。そのため、この交点を線分で表現するが、図式では外側に向かう一つの線分を二つの線分で置き換える形になる。これらの交差点は有限個なので有限回の操作でこの処理を終えることができる。ここで基準にした円と線分で繋がっていない円があるときは基準にした円と向きと中心を合わせ、それからこの円を基準に線分で繋がった円に対して向きと中心を合わせる操作を行う。

ここで 2. の円弧の移動によって交点が増えます。この交点は前の交点の線分への置き換え操作と同様のことを行います。たとえば  $L_+$  の線分については次の置き換えが生じます。このことは  $L_-$  の交点でも同様ですが、この場合は本来の線分の向きを逆向きにするだけで、新たに加わる交点の置き換えになる線分の向きは一緒です：



この図の左が  $L_{0+}$  に他の円が線分に交差した状況で、これは本来は真ん中の絵の交差です。この交差を解消したものが右図の状態で、線分が二つに割れた状況になります。これらの処理によって複数の円は共通の中心点を持つ同じ向きの円として再構成できます。この様子を先程の結び目で説明しましょう。まず、ザイフェルト系に対して基準となる円を選んだ様子を以下に示します：

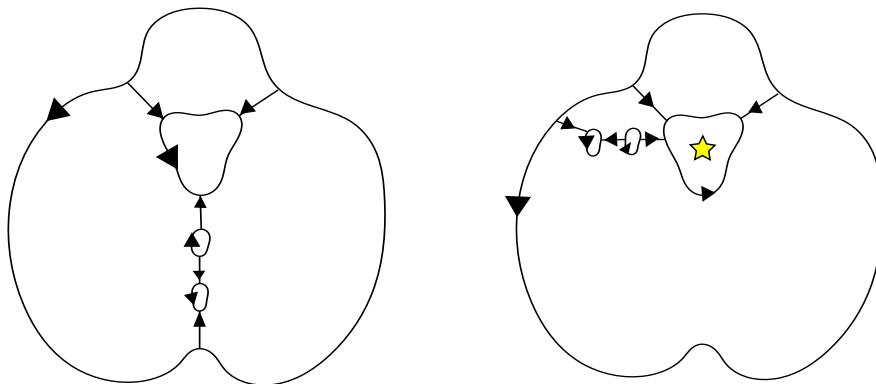


図 7.8 結び目のザイフェルト系と基準円の選定

この図で左側が結び目のザイフェルト系で、このザイフェルト系に対して基準となる円を選んで他の円を動かした結果が右図です。また、星印を含む最も小さな円が基準になる円です。この状態で基準になる円を含む円は最も外側の円だけで、残る二つの円は違います。だから、これらの二つの円に対して変形操作を行わなければなりません：

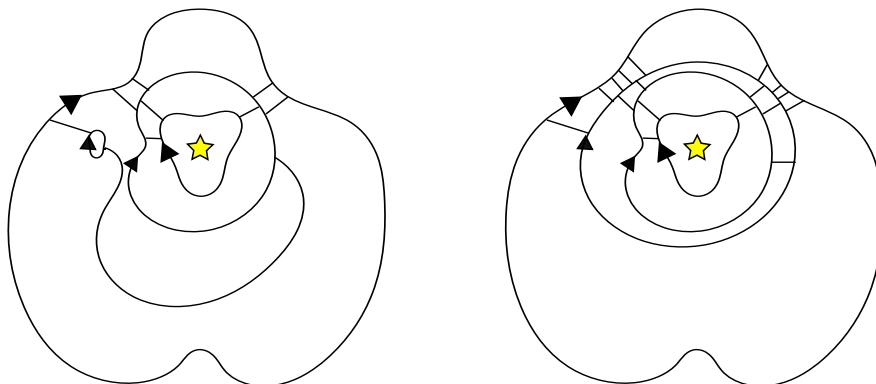


図 7.9 結び目のザイフェルト円周の変形操作

左図で基準円に線分で直接繋がっている円の向きを合せて中心を揃えています。このときに線分で繋がっている側の円弧はそのままで線分で繋っていない円弧を引いて基準円を包含するように移動させています。このときに他の円と基準円を繋ぐ線分との交差が生じます。これらの交差に対しても交差の置き換えを行うことで図では新たな二つの線分が外側に生じます。右図は残った円に対する操作で、この円は向きは一致しており、中心を合致さ

せるように平行移動させます。ここでも新たな線分との交差が生じるために、その交差を二つの線分で外側に置換えます。

**■閉じた組紐表現:** 全ての円が共通の中心点を持ち、同じ向きの円であれば、これらの円を繋ぐ線分を円に沿って円の左側に水平移動させて纏め、それから線分を本来の交差に戻す。これらの処理で閉じた組紐の表現が得られる。

この様子を先程の例に対して行ったものが図 7.10 です:

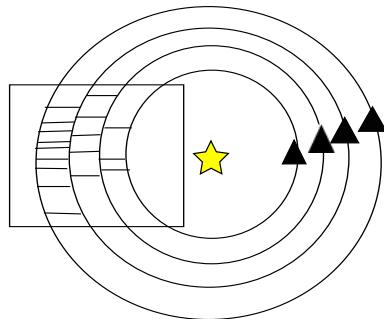


図 7.10 結び目の閉じた組紐としての表現

ここでは線分で表記していますが、これらは  $\pm$  の捩りに相当するので、ここで略記が閉じた組紐の表現になっていることに問題はないでしょう。また、ここでの変換は非常に機械的な方法で、その交差数が最小になるような「綺麗な組紐」を与える方法ではありません。

なお、閉じた組紐表現が得られた時点で各円に円盤を貼り、線分を帯で置き換えることでザイフェルト曲面が得られます。ただし、閉じた組紐表現にまで持つてゆくと構成された曲面はホットケーキを重ねたような複雑な曲面になります。実際、閉じた  $n$  糸の組紐が得られたのであれば  $n$  段重ねのホットケーキ状の曲面になります。そのためにここで得られたザイフェルト曲面が最小種数を持つ可能性はまずないでしょうが、円盤と 180 度の捻りを入れた帯だけで与えられた絡み目を境界とするザイフェルト曲面が機械的に構築できます。

## 7.8 ガウス・コード

ここまで正則射影図を基に絡み目/結び目を表現する群の構成について述べました。もちろん、絡み目/結び目を表現するためには群を持ち出さなくても射影図さえあればよいのです。この正則射影図をそのまま表現する一つの方法としてガウス・コードとよばれる数列で正則射影図を表現する方法があります。このガウス・コードは絡み目の各交点に番号を割り当て、それから各成分にも向きを入れた上で構築されます。ただし、正則射影図の考查状況をそのまま反映するものであり、交差点番号の割り当て方や基点の取り方次第で異なった値が返されますが、正則射影図が与えられると交差点番号と向きを正則射影図に与え、各成分に基点を置きさえすれば容易に記述することができます。

この表記の構成手順を順番に説明しましょう。まず最初に正則射影図に向きを入れます。それから各交差点に番号を配置します。それから絡み目上の各成分に基点を定めて結び目の向きに従って結び目上を移動します。基点から出発して交差点を通過する際に下道を通過するのであれば交差点番号に ‘-’ を付け、上道を通過するのであれば交差点番号だけにします。この操作を基点に戻るまで行います。ここで示した三葉結び目では基点の☆から開始するときに基点の乗った道が交差点 1 では下道になるので ‘-1’、交差点 3 では上道になるので ‘3’、それから交差点 2 は下道になるので ‘-2’、それから二度目に通過する交差点 1 では上道なので ‘1’、同様に交差点 3 では下道になるので ‘-3’、次の交差点 2 では上道なので ‘2’ となって出発点に戻ります。このように得られた数列を最初から並べると  $[-1, 3, -2, 1, -3, 2]$  になります。こうして得られた数列が「ガウス・コード (Gauss code)」と呼ばれる結び目/絡み目の情報になります。この三葉結び目の例のようにカウス・コードで表記したときの数列が正負を交互に繰替えす結び目を「交代結び目 (alternating knot)」と呼びます。

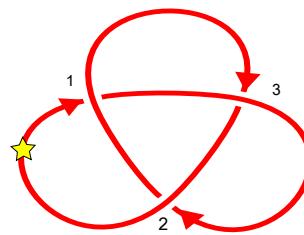


図 7.11 三葉結び目でのガウス・コードの生成

このガウス・コードには交点の符号の情報は含まれておらず、道の繋がり具合から交点の符号が別途判明するというものです。ただ、この数列の構成では交差点を眺めている訳で、だからこそ交差点の符号の情報を追加しておきたいものです。そこで、この交差点の符号の情報を追加したガウス・コードのことを「拡張ガウス・コード」と呼びます。このガウス・コードには二種類あり、一つは交差点番号の前に ‘p’, ‘m’ 等の正負を表現する記号を付与する表記です。この表記であれば絡み目への対応も比較的容易です。なぜなら絡み

目の場合は一つの成分のガウス・コード上に別成分との交差点が一度しか出ないために他の成分への参照が必要になりますが、この方法であれば絡み目一つの成分だけで関係子生成のための情報が得られるからです。もう一つの方法は二度目に通過した時点に交点の符号を付ける表記です。こちらは整数リストとしてガウス・コードが得られることと、結び目であれば最初の符号によって二度目の符号が決定されているために交差点に関する十分な情報が得られます。実際、絡み目  $L$  の成分  $K_i, K_j$  が交差しているときに横断的にこれらの成分が交差するので交点は必ず偶数個になります。ここで絡み目  $L$  の成分に順序が入っていて  $K_i > K_j$  となっているときに  $K_i$  の交点では通常の上下関係による符号、 $K_j$  の拡張ガウス・コードに交点の符号に対応する符号を添付することになります。ただし、交点番号  $p$  が一度しか成分  $K_i$  に現れなかつたときに  $p$  が含まれる成分  $K_j$  を探し、絡み目の成分の順序から符号の意味を判断するという操作が必要になります。ここで図 7.11 の三葉結び目に対してこれらの拡張ガウス・コードを示しておきましょう。まず符号に対応する記号を付与する方法であれば  $[-m1,m3,-m2,m1,-m3,m2]$ 、後者の二度目に符号を付与する方法であれば  $[-1,3,-2,-1,-3,-2]$  になります。なお、後者の拡張ガウス・コードは SageMath 7.2 から結び目理論パッケージに向けられたガウス・コード (oriented gauss code) として提供されています。

この交差点での符号が追加されている拡張ガウス・コードであれば容易に絡み目/結び目の交差点を復元することができます。そして、この結果から Wirtinger 表現で基本群の表現が計算できます。まず、基本群の生成元は交点番号が上道に対応するので簡単に得られます。残りは関係子で交点が  $n$  個であっても基本群の関係子としては  $n - 1$  個あればよいので  $n - 1$  個の交点に対して関係子を求めればよいことになります。ここで用いる拡張ガウス・コードは二度目に交点番号が現れた時点で交点の符号を付与する方法にします。その理由は処理が多少煩雑になってしまっても、データ処理は整数リストの処理に限定されるからです。では手順を以下にまとめておきましょう：

#### 拡張ガウス・コードの処理手順

1. 交差点の抽出
2. 交差点の符号の抽出
3. 正規ガウス・コードへの変換
4. 交差点情報の復元

ここに挙げた手順について順番に考察してゆきましょう。

**■交差点の抽出:** 交差点の番号の集合をガウス・コードから抽出します。ここでの処理は自然数を選択し、それらの並び替えを行ったリストを生成するだけです。

**■交差点の符号の抽出:** 結び目であれば二度目に現れたガウス・コード中の交差点番号の符号から判ります。絡み目の場合は交差点番号を含む二つのガウス・コード  $G_i, G_j$  で先頭の  $G_i$  から交差の上下関係,  $G_j$  から交差点での符号が得られます。

**■正規のガウス・コードへの変換:** この処理は最初に現れた交差点番号  $i$  の符号を覚えておき、拡張ガウス・コードで二度目に  $i$  が現れた時点で覚えていた符号を逆にすること、すなわち  $-1$  を掛けたもので置き換えればよいことになります。

**■交差点情報の復元:** 交差点の番号  $i$  がガウス・コードで最初にどのように現れるかという場合分けが必要になります：

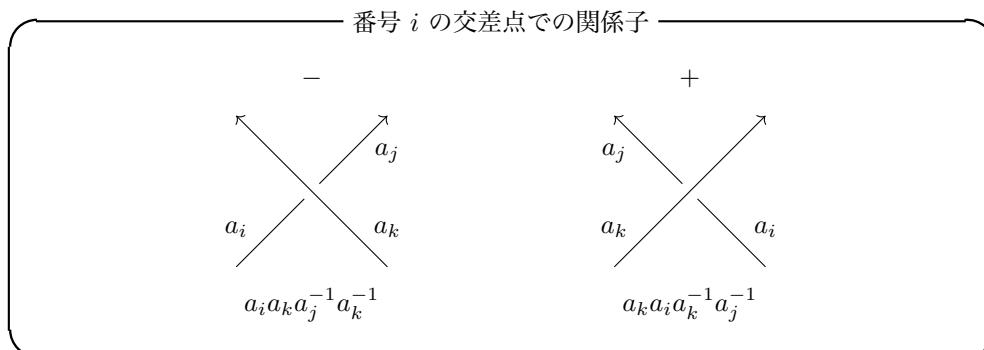
- 負の数の場合:

まず負の数として最初に現れる場合は生成元  $a_i$  が  $i$  の交差点の下道の一つであることが判ります。それから  $a_i$  に続く下道はガウス・コード内の  $-i$  から右側の数で最初に現れる負の数を探します。もし、リストの右端で見つからなければリストの左端から負の数を探します。このときに最初に負の数  $-j$  が見つかると  $a_j$  が  $a_i$  に続く道になります。そして交差点  $i$  の上道はガウス・コードの  $i$  から右側にある最初の負の整数  $-k$  から  $a_k$  であることが判ります。

- 正の数の場合:

正の数として現れた場合は、ガウス・コード内の  $i$  から右側の数で最初に現れる負の数  $k$  から  $a_k$  が交差点  $i$  の上道になります。それからガウス・コードで  $-i$  を探し、その  $-i$  から右側に最初に現れる負の数  $-j$  から  $a_j$  が判ります。

これらの情報から交差点が次のように復元できます：



このことを図 7.11 に示す三葉結び目の拡張ガウス・コードで説明しておきましょう。まず三葉結び目の拡張ガウス・コードは図中の☆を基点に進めて  $[-1, 3, -2, -1, -3, -2]$  になります。このことから交点集合は  $\{1, 2, 3\}$  になります。次に各交点の符号を調べます。二度目に現れる番号は  $[-1, -3, -2]$  であることから全ての符号が  $-1$  であることが判ります。

そこで  $[[1, -1], [2, -1], [3, -1]]$  と交差点の符号の情報を交差点番号との対のリストとして表現することにします。また、最初に現れる番号が  $[-1, 3, -2]$  になることから二番目に現れる箇所の  $[-1, -3, -2]$  は  $[1, -3, 2]$  で置き換えられるので本来のガウス・コードは  $[-1, 3, -2, 1, -3, 2]$  であることが判ります。それから交差点の復元に入ります。まず交差点 1 はガウス・コードの最初が  $-1$  なので交差点 1 に入り込む下道が  $a_1$  になります。それから  $-1$  の右側で最初に現れる負の整数が  $-2$  であることから交差点 1 から出る下道が  $a_2$  であることが判ります。さて交差点 1 を通過する上道は二度目の  $1$  をガウス・コードから探します。すると  $[1, -3, 2]$  に二度目の  $1$  が含まれており、この二度目の  $1$  の右側の負の数は隣の  $-3$  なので上道が  $a_3$  であることがわかります。このことから交差点 1 から得られる関係子は  $a_1 a_3 a_2^{-1} a_3^{-1}$  が得られます。次に交差点番号 3 に関しては、ガウス・コードでは最初に  $3$  が現れているために、最初に上道を探すことになります。この場合は  $3$  の右側に最初に現れる負の整数が  $-2$  であることから上道が  $a_2$  であることが判ります。それから  $-3$  をガウス・コードから探します。すると  $[-3, 2]$  がありますが、 $-3$  の右側には  $2$  と負の数がありません。そこでガウス・コードを繋いで  $[-3, 2, -1, 3, -2, -1, 3, 2]$  で考えますが、 $-3$  の次に現れる負の数は  $-1$  であることから交差点番号 3 から出てゆく下道が  $a_1$  であることが判ります。このことから  $a_2 a_3 a_2^{-1} a_1^{-1}$  が関係子になります。以上からこの三葉結び目の群の表示として

$$\langle a_1, a_2, a_3 \mid a_1 a_3 a_2^{-1} a_3^{-1}, a_2 a_3 a_2^{-1} a_1^{-1} \rangle$$

が得られます。

前述のように SageMath では 7.2 から結び目理論パッケージで拡張ガウス・コードが提供され、この拡張ガウス・コードで結び目や絡み目を定義すると、それらの描画まで行えます。しかし、単純にパッケージを使うだけでは面白くないので上述の手続きを SageMath で表現してみましょう。ただし、ここでは話を簡単にするために結び目に限定しておきます。また、SageMath で行わせる処理は群の表示までは基本的に Python のリストの処理が中心です：

---

```
def check_plusOnly(list_0):
    if True in map(lambda(x):x<0,list_0):
        return(False)
    else:
        return(True)

def Generators(ExGaussCode):
    list_crossings = list(set(map(lambda(x):abs(x),ExGaussCode)))
    n = len(list_crossings)
    gns = ""
```

---

```

for i in list_crossings:
    if gns=="":
        gns = "a_" + str(i)
    else:
        gns = gns + ", a_" + str(i)
return ([gns, list_crossings])

```

---

最初の check_plusOnly() は与えられたリストの成分が全て正であれば True, それ以外で Faulse を返す函数です。それから函数 Generators() は拡張ガウス・コードから結び目群の生成元を文字列として返す函数です。ここでの処理は交差点と道が一対一に対応し, さらにそれらの道が結び目群の生成元になることを利用しています。なお生成元は ‘a_i’ の形になるようにしています。

---

```

def GaussCode(ExGaussCode, list_crossings):
    gauss_code = []
    sign_crossings = []
    for x in ExGaussCode:
        i = abs(x)
        sgnx = 2*(x>0)-1
        if i in list_crossings:
            list_crossings.remove(i)
            gauss_code.append(x)
        else:
            if x in gauss_code:
                gauss_code.append(-x)
                sign_crossings.append([i, sgnx])
            else:
                if -x in gauss_code:
                    sign_crossings.append([i, sgnx])
                    gauss_code.append(x)
    return ([gauss_code, sign_crossings])

def RelatorCode(gauss_code, sign_crossings):
    list_relatorCode = []
    for x in sign_crossings:
        upth = 0
        [i, s] = x
        rel = []
        n = len(gauss_code)
        "Set a_i"
        p0 = gauss_code.index(-i)
        p1 = gauss_code.index(i)
        "Set a range to find a_j, next of a_i"

```

```

gcx = gauss_code[p0+1:n]
"Extends the range of Gauss code to be cyclic"
if check_plusOnly(gcx):
    gcx = gcx + gauss_code
"searching a_j"
for j in gcx:
    if j<0 and rel==[]:
        rel = [i, -j]
"Search a upper path"
if p1==n:
    gcx = gauss_code
else:
    gcx = gauss_code[p1+1:n]
    "Extends Gauss code to be cyclic"
    if check_plusOnly(gcx):
        gcx = gcx + gauss_code
    for k in gcx:
        if k<0 and upth==0:
            rel.append(-k)
            rel.append(s)
            upth = 1
list_relatorCode.append(rel)
return(list_relatorCode)

```

それから函数 GaussCode() は拡張ガウス・コードから通常のガウス・コードと交差点の符号の情報を分けます。それから函数 RelatorCode() では通常のガウス・コードと交差点の符号の情報から交差点での交差の状況を 4 成分のリストとして表現します。まず、ガウス・コードと生成元の対応では交差点  $i$  に向かう下道  $a_i$ , 交差点  $i$  を出る下道  $a_j$  と交差点  $j$  の上道  $a_k$  の番号  $i, j, k$  と交差点  $i$  の符号を順番で並べたリストが交差点  $i$  の交差の情報になります。この函数 RelatorCode() から関係子を容易に求めることができます。

---

```

def representation_KnotGroup(generators, list_relatorCode):
    relators = []
    "generates a free group F"
    F = FreeGroup(generators)
    "generates relators from relatorCode"
    for x in list_relatorCode[0:-1]:
        sgn = x[-1]
        i = x[0]
        j = x[1]
        k = x[-2]
        if sgn>0:

```

```

        relators.append(F([k,i,-k,-j]))
    else:
        relators.append(F([i,k,-j,-k]))
return(F/relators)

def KnotGroup(ExGaussCode):
    list_crossings = []
    sign_crossings = []
    gauss_code = []
    list_relatorCode = []
    relators = []
    "Define generators"
    [gns, list_crossings] = Generators(ExGaussCode)
    "Generate a free group F"
    F = FreeGroup(gns)
    "Redefine Gauss code and a list at each crossings"
    [gauss_code, sign_crossings] = GaussCode(ExGaussCode, list_crossings)
    "Get relatorCode of each crossings"
    list_relatorCode = RelatorCode(gauss_code, sign_crossings)
    "calculate the knot group"
    G = representation_KnotGroup(gns, list_relatorCode)
    return(gauss_code, sign_crossings, G)

```

---

この関係子の計算は `representation_KnotGroup()` で行いますが、ここで注意すべきことは、関係子は通常の多項式環で定義するのではなく自由群上で定義しなければならないことです。SageMath では無指定であれば多項式環が用いられます（ただし、実際の利用では変数の宣言を函数 `var()` であらかじめ行う必要があります）。この多項式環は可換環であり、SageMath では項や項を構成する変数を与えられた順序にしたがって並び替えてしまいます、その結果、関係子を勝手に簡約化してしまい、関係子の図形的な意味を壊してしまうからです。このことを防ぐために自由群 `F` とその商群を定義し、その中で関係子を設定する必要があります。ここでは自由群 `F` を内部で定義し、その `F` の元として関係子をリストの形で定めています。そして結び目群は自由群 `F` を関係子の帰結群の商群としてあたえられます。それから最後の函数 `calc_KnotGroup()` はこれらの函数をまとめて拡張ガウス・コードからガウス・コード、交差点での符号と結び目群を返却する函数にしたもので

す。

実際の計算例を以下に示します：

---

```

sage: EGC_Trefoil = [-1, 3, -2, -1, -3, -2]
sage: KG_Trefoil = KnotGroup(EGC_Trefoil)
sage: KG_Trefoil

```

---

```
([-1, 3, -2, 1, -3, 2],
 [[1, -1], [3, -1], [2, -1]],
 Finitely presented group < a_1, a_2, a_3 | a_3*a_1*a_3^-1*a_2^-1,
 a_2*a_3*a_2^-1*a_1^-1 >
sage: EGC_FigureEight = [-1, 3, -2, 4, 3, 1, 4, 2]
sage: KG_FigureEight = KnotGroup(EGC_FigureEight)
sage: KG_FigureEight
([-1, 3, -2, 4, -3, 1, -4, 2],
 [[3, 1], [1, 1], [4, 1], [2, 1]],
 Finitely presented group < a_1, a_2, a_3, a_4 | a_2*a_3*a_2^-1*a_4^-1,
 a_4*a_1*a_4^-1*a_2^-1, a_3*a_4*a_3^-1*a_1^-1 >)
```

---

と、このように結び目の拡張ガウス・コードを与えると通常のガウス・コード、各交点での符号、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

これらのプログラムでは拡張ガウス・コードを基にして処理を進めていますがガウス・コードは正則射影図を基にしているので、拡張ガウス・コードを使って正則射影図のクラスを定義したくなります。そこで絡み目の正則射影図のクラスを拡張ガウス・コードを使って定義してみましょう。そしてこのクラスを C の構造体のように拡張ガウス・コードを保持することを中心としたクラスとするのではなく、関連するメソッドや函数を擁するものにしてみましょう。以下に正則射影図のクラス LinkDiagram とその解説を載せてゆきましょう：

---

```
import sqlite3
import networkx as nx
import matplotlib.pyplot as plt
from sage.structure.element import RingElement

class LinkDiagram(RingElement):
    ExGaussCodes = []
    GaussCodes = []
    SeifertCircles = []
    Crossings = []
    Components = 1
    SQLite3_DB = ''

    def __init__(self, ExGaussCodes=None):
        if ExGaussCodes is not None:
            self.ExGaussCodes = ExGaussCodes
            self.GaussCodes = []
```

```

        self.Crossings = []
    for ExGaussCode in ExGaussCodes:
        [gssc, self.Crossings] = self.gauss_code(ExGaussCode)
        self.GaussCodes.append(gssc)
    self.Components = len(self.GaussCodes)
    self.SeifertCircles = self.seifert_circles()

def __repr__(self):
    ExGaussCodes = []
    "The case that the first operand is a trivial knot"
    if self.ExGaussCodes[0]==[]:
        return(LinkDiagram(other.number_shift(1)))
    else:
        "The case that the second operand is a trivial knot"
        if other.ExGaussCodes[0]==[]:
            return(LinkDiagram(self.number_shift(1)))
        else:
            fa = map(abs, flatten(self.ExGaussCodes))
            bs = max(fa) - min(fa) + 2
            "1st link starts from 1."
            a = self.number_shift(1)
            "2nd link starts bs."
            b = other.number_shift(bs)
            tb = b[0][-1]
            rb = b[0][0:-1]
            ab = abs(tb)
            arb = map(abs, rb)
            if ab in arb:
                n0 = arb.index(ab)
                cb = rb[n0]
                rb[n0] = tb
                tb = -cb
            ExGaussCodes.append(a[0] + [tb] + rb)
            for i in a[1:]:
                ExGaussCodes.append(i)
            for i in b[1:]:
                ExGaussCodes.append(i)
    return(LinkDiagram(ExGaussCodes))

def __add__(self, other):
    ExGaussCodes = []
    "The case that the first operand is a trivial knot"
    if self.ExGaussCodes[0]==[]:
        return(LinkDiagram(other.number_shift(1)))

```

```

else:
    "The case that the second operand is a trivial knot"
    if other.ExGaussCodes[0]==[]:
        return(LinkDiagram(self.number_shift(1)))
    else:
        fa = map(abs, flatten(self.ExGaussCodes))
        bs = max(fa) - min(fa) + 2
        "1st link starts from 1."
        a = self.number_shift(1)
        "2nd link starts bs."
        b = other.number_shift(bs)
        tb = b[0][-1]
        rb = b[0][0:-1]
        ab = abs(tb)
        arb = map(abs, rb)
        if ab in arb:
            n0 = arb.index(ab)
            cb = rb[n0]
            rb[n0] = tb
            tb = -cb
        ExGaussCodes.append(a[0] + [tb] + rb)
        for i in a[1:]:
            ExGaussCodes.append(i)
        for i in b[1:]:
            ExGaussCodes.append(i)
return(LinkDiagram(ExGaussCodes))

def number_shift(self, n=None):
    ExGaussCodes = []
    a = copy(self.ExGaussCodes)
    m = min(map(abs, flatten(a)))
    if n is not None and n>0:
        bs = m - n
    else:
        bs = m
    for xgausscode in self.ExGaussCodes:
        ExGaussCode = []
        for i in xgausscode:
            ExGaussCode.append(i - sign(i) * bs)
        ExGaussCodes.append(ExGaussCode)
    return(ExGaussCodes)

def __mul__(self, other):

```

```

ExGaussCodes = []
if len(self.ExGaussCodes)==0:
    ExGaussCodes.append([])
    b = other.number_shift(1)
    for i in b:
        ExGaussCodes.append(i)
else:
    if len(other.ExGaussCodes)==0:
        b = self.number_shift(1)
        ExGaussCodes.append([])
    else:
        a = self.number_shift(1)
        fa = map(abs, flatten(a))
        bs = max(fa) - min(fa) + 2
        b = other.number_shift(bs)
        for i in a:
            ExGaussCodes.append(i)
        for j in b:
            ExGaussCodes.append(j)
return(LinkDiagram(ExGaussCodes))

```

---

最初にクラスが必要とするパッケージとモジュールの読み込みを import 文で行います。ここでは SQLite3, NetworkX, そして matplotlib からは plot, それから sage.structure.element から RingElement の読み込みを行っています。まず, SQLite3 は RDBM で RDB を不变量の計算で現れる中間的な正則射影図の保管に用いたいからです。実際,  $n$  個の交差点のある絡み目/結び目であれば、一つの交差点で 2 通りの状況が発生するための中間データは鼠算的に増大します。その上、不变量の計算ではこの中間データ全てが必要で、それも表形式で保持しておくことが不变量の計算で有利なためです。それから NetworkX はグラフ理論のためのパッケージですが、ここではザイフェルト系をグラフとして表現するために用います。また plot は NetworkX を使って定義したグラフの可視化で利用します。これらのパッケージに関してはザイフェルト系の可視化で詳細を説明します。つぎに ‘class LinkDiagram(ieldElement)’ はクラス RingElement を継承することを意味する class 文です。この LinkDiagram クラスの定義では SageMath で定義されたクラス RingElement を継承することで代数的な構造を手に入れようとしているからです。

### 7.9.1 導入すべき代数的構造

ここで入れようとする「代数的構造」について解説をしておきましょう。まずクラス LinkDiagram でクラス RingElement を継承しようとしています。このクラス RingElement は SgaeMath で抽象基底クラスと呼ばれるクラスです。SageMath の抽象

基底クラスは Python の同名のクラスと類似の機能を持つクラスで、我々が扱おうとする対象が持つ性質をより大局的な側面から扱おうとするものです。つまり、SageMath の抽象基底クラスを継承させるということで、その数学的対象に適合した性質を表現するメソッド等をそのまま雛形として継承し、利用者はそれらのメソッド等を、実際の対象に適合するように上書きすることで SageMath に組込み、それらの対象の解析を行います。

ここで適切な「構造」を与えるためには「対象が何であるか」、「演算がどのようなもの」であるかを適切に「語ること」をしなければなりません。では、対象は何でしょうか？ここで定義するクラスは絡み目の正則射影図を表現します。そして前述のプログラムでは拡張ガウス・コードを基にしてプログラムを記述しています。だから対象は拡張ガウス・コードであるべきです。これで対象が決まりました。さて、正則射影図には前述のように連結和と直和の二つの演算があります。どちらも「和」という言葉がありますが、最初に連結和はどのように表現すべきでしょうか？ここで絡み目の連結和は一つの成分に対して行われる処理のために結び目の連結和として捉えられます。結び目の連結和  $K_1 \# K_2$  は双方の起点で処理を行うのであれば最初に  $K_1$  を回って次に  $K_2$  を回る形になるので単純にリストの結合として捉えることができます。そして単位元もリストで表記することができます。実際、空リスト [] を使えば上手く単位元としての性質が表現可能で、さらに連結和の単位元である自明な結び目は交差点をもたないことから、連結和の単位元を空リスト [] で表現することに問題が全くないことが分かります。また、絡み目については、どの成分で連結和を取るかということが問題になりますが、ここで絡み目  $L = \cup_{i=1}^m L_i$  を単なる円周の集合ではなく順序対  $\langle L_1, L_2, \dots, L_m \rangle$  として考え、連結和はその第一成分に対して行うものとします。このときにもう一つの演算の直和を

$$\langle L_1^1, L_2^1, \dots, L_m^1 \rangle \oplus \langle L_1^2, L_2^2, \dots, L_n^2 \rangle = \langle L_1^1, \dots, L_m^1, L_1^2, \dots, L_n^2 \rangle$$

で定めますが、この結果、直和については可換性が充たされません。また、この直和の単位元として空集合  $\emptyset$  があることが判ります。さて、この絡み目の集合は二つの演算を持ちますが、これに類似したものに何があるでしょうか？演算が二つあるものにはたとえば自然数  $\mathbb{N}$  があります。実際、自然数  $\mathbb{N}$  には式 ‘ $1+1$ ’ にある和 “+”，式 ‘ $2 \times 3$ ’ にある積 “ $\times$ ” の二つの演算がありますが全て可換です。では、2 次の正方行列の集合  $M(2)$  はどうでしょうか？こちらは和 “+” は可換ですが積 “.” は可換ではありません。そして、2 次の正方行列の集合は環と呼ばれる代数的構造を持ちますが、この 2 次の正方行列がモデルになりそうです。そこで絡み目の連結和は和演算 “+”，直和は可換性がないために積演算 “*” として表現しましょう。そうなると代数的構造として和と積の二つの演算を持つ環が妥当と判断できるでしょう^{*5}。これが抽象基底クラス RingElement を継承することにした理由で

---

^{*5} より正確には絡み目の集合は半環 (semiring) の構造を入れることができます。

す。これは演算それ自体の定義ではなく、その演算が持つ性質を利用するときに威力を發揮します。

抽象基底クラス `RingElement` を継承することで環として必要な性質は一通り継承されますが、具体的なことはそれだけではまだ何も決っていません。抽象基底クラスは一通りのメソッドや属性が定義されていても、これらを利用者がきちんと定義しなければなりません。この絡み目の集合に対しても、連結和や直和といった演算の具体的な処理内容を記載しなければ、単に形式的な二つの演算があって、それらが環としての性質を充しているというだけです。それではこれらの演算をどのように入れればよいでしょうか？これが数式処理 Maxima であれば自由自在に連結和 “#” や直和 “⊕” を定義することができますが、このように万事が気軽に見えるのも継承すべきクラスというものがないからです。ところが Python では和や積演算は特殊メソッドで表現されており、それらの上書きを行えば良いようにできています。つまり、Python では、和演算 “+” の定義は特殊メソッド `__add__()` で行い、積演算 “*” の定義は特殊メソッド `__mul__()` の上書きを行えば、和や積を定義し直すことができます。絡み目の連結和と直和の表現も、これらの特殊メソッドを利用すれば良さそうですが、ここで注意が必要になります。まず、演算の表現は要するにリストの処理に過ぎないので、その記載に問題はなさそうです。そして、連結和の可換性ですが、二つの演算結果を比較することを現時点では考えておらず、それで実用上の問題がないために不間にしましょう。実用上の問題になるのは結合律です。まず、この結合律が充たされれば  $a \# b \# c$  のように二項以上の計算式の記載が可能になりますが、Python の特殊メソッド `__add__()` や `__mul__()` を上書きしても、これらの二項演算子は結合律を充しません。そのため  $a + b + c$  や  $a * b * c$  のような式はエラーになります。だから結合律を入れなければ、このクラスのためだけに自力で結合律を構築するか、既存の SageMath のクラスを流用するかを選択しなければなりません。ところで、自力で結合律を構築するのであれば「**車輪の再発明**」を行う妥当性が問われますが、ここでは単に結合律を入れたいだけで、SageMath にあるものを利用しない理由にはなりません。そこで SageMath のどの代数的構造を入れるかという問題になりますが、ここでは演算が二つで一方が可換、もう一方が非可換であり、各演算に対して半群になることから、環を表現する抽象基底クラス `RingElement` で十分と判断できます。この抽象基底クラス `RingElement` を継承するために `import` 文で `sage.structure.element` から `RingElement` モジュールをあらかじめ読み込んでいる理由です。では、SageMath の代数的構造を与える抽象基底クラスを継承してしまえば、和や積の特殊メソッドを上書きしてしまえば良いでしょうか？SageMath の抽象基底クラスには書き換えるべきメソッドがちゃんと用意されています。ただ、その書き換えを行うメソッドは Python の特殊メソッドの名前とは文字 “_” が一つ少なくなっています。

### 7.9.2 書換るべきメソッドについて

これで LinkDiagram クラスの骨子が定まりました。では、与えられた 絡み目の拡張ガウス・コードをこのクラスのインスタンスとするにはどうすればよいでしょうか？このクラスのインスタンス化で拡張ガウス・コードを引数として指示すれば、その拡張ガウス・コードに対応するインスタンスを生成するようにすると良いでしょう。そのためには LinkDiagram クラスは正則射影図に対応するインスタンス生成で、その正則射影図の拡張ガウス・コードを成分とするリストを一つ取ることにします。ここでの拡張ガウス・コードは上述の方法で構築した整数のリストで、結び目は各成分の拡張ガウス・コードから構成されることになります。そこでインスタンス化の際の引数としては拡張ガウス・コードのリストを与えるようにすれば結び目の場合でも問題ありません。通常、Python のインスタンスの初期化では特殊メソッド `__init__()` が用いられます。結び目の半群としての構造を入れたいがためにクラス `RingElement` を継承することにしています。このクラスには利用者が書換るべき特殊メソッドの代用として文字 `_` が一つだけのメソッドが準備されており、インスタンスの初期化では `_init_()` を用います。このメソッド `_init_()` の記載内容はメソッド `__init__` を用いるときと記述に違いはありません。今回は拡張ガウス・コードのリストを一つだけ取るために `self` 以外の明示的な引数として拡張ガウス・コードのリスト `ExGaussCodes` を記載しておきます。ところで自明な結び目も一々 ‘`[]`’ で与えて初期化するのもどうでしょう？それよりも引数が与えられなければ自明な結び目のデータを生成するようにしましょう。このことはメソッド `_init_()` の引数 `ExGaussCodes` に既定値として `None` を設定することを ‘`ExGaussCodes=None`’ と記載することで行えます。それからメソッド内部で変数 `ExGaussCodes` の値で自明な結び目か、まともな処理を必要とするかに分岐するようにすれば良いのです。そして、このクラスの初期化で与えられた拡張ガウス・コードのリストから交差点の上下関係のみの正規カウス・コードと符号付き交差点のリスト、結び目の成分数、ザイフェルト円周のリストを生成するようにしましょう。このように変数 `GaussCodes`, `Crossings`, `Components` にはインスタンス化の時点で計算した値がそれぞれ割り当てられます。ただし、変数 `SQL` は多項式不变量の処理で SQLite3 の RDB の情報を載せるために準備したもののためにメソッド `_init_()` では指示しません。

それからインスタンスの情報を得るために一々、メソッドに訴えるよりもインスタンスの名前を指示したときに分かり易い書式で表示させたいものです。このようなことを実現させる Python の特殊メソッドが `__repr__()` です。ただし、SageMath の `sage.structure.element` のクラスを継承するときは Python の特殊メソッドではなく、あらかじめ準備されたメソッドの上書きを行うことがあります。ここでは環のクラス

`RingElement` を用いますが、この場合はメソッド `_repr_()` の上書きで対処します。ここで表示させる内容はガウス・コードとその交差点のリスト、成分数とザイフェルト円周のリストを表示させることにしますが、単純に指定した書式の文字列を `return` 文で返せばよいのです。

次に具体的な演算を定義しなければなりません。実際、代数的な構造を入れたとはいえ、具体的な演算は何一つ決まっていないからです。ここで定義すべき演算は連結和と直和です。まず連結和に関しては代数的構造について考察したように、ここでの処理は与えられた拡張ガウス・コードのリストの先頭の成分に対して行うものとします。この演算については前述のようにリストの結合を基とすればよいことが分かっていますが単純な結合ではありません。図 7.3 に三葉結び目と八の字結び目の連結和の例を示していますが、連結和は直感的には左右に結び目を並べて双方の自明な紐の箇所を外して結びつける操作で、それを忠実に表現しようとすると和 “+” 右辺の被演算子の結び目に関して交差点番号の付け直しが必要になります。そこで番号の振替には内部的な函数として函数 `numberShift()` を構築し、左右の被演算子に対して番号の付け替えを行った上で拡張ガウス・コードを構築するようにします。ここで和演算の実装は通常は二項演算子 “+” を定義するメソッド `__add__()` の上書きで実現できます。ただし、このメソッドは単純に二項演算子を定めるだけで、結合律までも定めるものではありません。ここでは `RingElement` クラスを継承するようにしたためにメソッド `_add_()` に演算を定義します。ただし、メソッド `_add_()` で記載する内容はメソッド `__add__()` に記載するものと違いはありません。このメソッドには二つの被演算子に対応する引数が必要になります。被演算子の一つはメソッドの定義で必要な `self`、もう一つは同一クラスの別インスタンスに対応する `other` で、`self` と `other` の `ExGaussCodes` の第一成分に対してリストの和 “+” を行い、それで得られた拡張ガウス・コードを使って新たなインスタンスを返すようにしています。これらのこととは直和についても同様です。この直和はその非可換性から積演算を用いることにします。ここで積演算は特殊メソッド `__mul__()` の上書きで一般的には行いますが、`RingElement` クラスではあらかじめメソッド `_mul_()` が用意されているのでそちらを用います。記述内容はメソッド `__mul__()` と同様です。このように `RingElement` クラスのメソッドを上書きすることで労なく結合律を充たす演算にすることができました。

### 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 link_component(self, n=None):
    ExGaussCode = []
```

```

if n is not None:
    k = 0
    if n<self.Components:
        xgssc = self.ExGaussCodes[n]
        axgssc = map(abs, xgssc)
        crssngs = list(set(axgssc))
        chk = map(lambda (x):axgssc.count(x)==2, axgssc)
        for i in chk:
            if i:
                ExGaussCode.append(xgssc[k])
            k = k + 1
    return(Diagram([ExGaussCode]))


def del_Link_component(self, n=None):
    dcrssng = []
    xgsscs = self.ExGaussCodes
    if n is not None:
        if n<self.Components:
            exgssc = xgsscs[n]
            xgsscs.pop(n)
            if len(exgssc)>0:
                aexgssc = map(abs, exgssc)
                chk = map(lambda (x):aexgssc.count(x)==1, aexgssc)
                k = 0
                for i in chk:
                    if i:
                        dcrssng.append(aexgssc[k])
                k = k + 1
            for i in dcrssng:
                for x in xgsscs:
                    if i in x:
                        x.remove(i)
                    if -i in x:
                        x.remove(-i)
    return(LinkDiagram(xgsscs))

def mirror_image(self, n=None):
    ExGaussCodes = []
    if n is None or n<0 or n>self.Components:
        for x in self.ExGaussCodes:
            ExGaussCodes.append(map(lambda(i):-i, x))
    else:
        xgausscode = map(lambda(z):-z, self.ExGaussCodes[n])
        xcrssngs = map(lambda(x):abs(x), xgausscode)

```

```

crssngs = list (set(xcrssngs))
cnt = map(lambda(z): xcrssngs.count(z)==1, crssngs)
w = range(0,len(cnt))
if sum(cnt)>0:
    crps = list (set(map(lambda(z):cnt[z]*crssngs[z],w)))
    for x in self.ExGaussCodes[0:n]:
        ExGaussCodes.append(x)
        for i in crps:
            if i in xgausscode:
                n = xgausscode.index(i)
                xgausscode[n] = -i
            else:
                if -i in xgausscode:
                    n = xgausscode.index(-i)
                    xgausscode[n] = i
        ExGaussCodes.append(xgausscode)
    for x in self.ExGaussCodes[n+1:]:
        ExGaussCodes.append(x)
        for i in crps:
            if i in xgausscode:
                n = xgausscode.index(i)
                xgausscode[n] = -i
            else:
                if -i in xgausscode:
                    n = xgausscode.index(-i)
                    xgausscode[n] = i
        else:
            for x in self.ExGaussCodes[0:n-1]:
                ExGaussCodes.append(x)
            ExGaussCodes.append(map(lambda(z):-z, xgausscode))
    for x in self.ExGaussCodes[n+1:]:
        ExGaussCodes.append(x)
return (LinkDiagram(ExGaussCodes))

```

これらのメソッドは整数  $n$  をオプションとするものです。整数  $n$  が指示されなかったときの処理はそれぞれ異なります。まずメソッド `link_component()` は整数で指定した番号の絡み目の成分を取り出します。この番号は拡張ガウス・コードのリストの添字に対応します。整数  $n$  が与えられないときと成分数を超過したときは自明な結び目に対応する空リスト [] を返します。つぎのメソッド `del_link_component()` は指定した番号の成分を絡み目から削除します。無指定のときは削除を一切行なわずに元と同じ拡張ガウス・コードを持つインスタンスを返却します。メソッド `mirror_image()` は指定した成分のみを鏡像にしたインスタンスを返し、整数が指示されていなければ全ての成分を鏡像にしたインス

タンスを返します。

さて、次に拡張ガウス・コードから交差点の符号の情報の無い正規のガウス・コードと交差点の情報を生成するメソッドを構築しておきましょう：

---

```
def gauss_code(self, ExGaussCode):
    GaussCode = []
    crssngs = copy(self.Crossings)
    listCrossings = list(set(map(abs, ExGaussCode)))
    for x in ExGaussCode:
        i = abs(x)
        s = sign(x)
        if i in listCrossings:
            listCrossings.remove(i)
            if -i in self.Crossings:
                crssngs[crssngs.index(-i)] = s*i
                GaussCode.append(i)
            else:
                if i in self.Crossings:
                    crssngs[crssngs.index(i)] = s*i
                    GaussCode.append(-i)
                else:
                    crssngs.append(x)
                    GaussCode.append(x)
        else:
            if x in GaussCode:
                GaussCode.append(-x)
            else:
                GaussCode.append(x)
                if -i in crssngs:
                    crssngs[crssngs.index(-i)] = s*i
                else:
                    if i in crssngs:
                        crssngs[crssngs.index(i)] = s*i
    return ([GaussCode, crssngs])
```

---

このメソッド `gauss_code()` は拡張ガウス・コードをリストとして先頭から解釈し、初めて現れた交差点はそのままにし、二度目に現れた交差点で符号と番号を交差点のデータに追加し、ガウス・コードには前回現れたときの逆の符号を与えるという処理を行っていますが、この処理は結び目でも絡み目でも違ひはありません。

---

次にザイフェルト系の円周を求めるメソッドを定義します：

```

def seifert_circles(self):
    GaussCodes = self.GaussCodes
    Crossings = self.Crossings
    newCrossings = copy(Crossings)
    for i in Crossings:
        newGaussCodes = []
        newCrossings.remove(i)
        gcds = []
        gpr = []
        sgn = sign(i)
        ai = abs(i)
        mi = -ai
        for x in GaussCodes:
            if not(i in x or -i in x):
                gcds.append(x)
            else:
                gpr.append(x)
        if len(gpr)==1:
            GaussCode = gpr[0]
            p0 = GaussCode.index(mi)
            p1 = GaussCode.index(ai)
            if p0<p1:
                newGaussCodes.append(GaussCode[0:p0]+[i]+GaussCode[p1+1:])
                newGaussCodes.append(GaussCode[p0+1:p1]+[i])
            else:
                newGaussCodes.append([i]+GaussCode[p1+1:p0])
                newGaussCodes.append(GaussCode[p0+1:]+GaussCode[0:p1]+[i])
        else:
            if len(gpr)==2:
                if ai in gpr[0]:
                    ga = gpr[1]
                    gb = gpr[0]
                else:
                    ga = gpr[0]
                    gb = gpr[1]
                p0 = ga.index(mi)
                p1 = gb.index(ai)
                px = [i]+gb[p1+1:]+gb[0:p1]+[i]+ga[p0+1:]+ga[0:p0]
                newGaussCodes.append(px)
            for j in gcds:
                newGaussCodes.append(j)
    GaussCodes = newGaussCodes
    return(GaussCodes)

```

---

このメソッド `seifert_circles()` は各交点での解消処理を行います。のちの不变量の計算では上道と下道の繋ぎ替えで向きを変更する処理が入りますが、ここでの処理は向きの逆転ではなく単純な繋ぎ替え処理が中心です。そして出力はザイフェルト円周を構成する交差点番号に交差点の符号を付けたもののリストです。この番号は絡み目の各成分に与えた向きに従って並んだものです。

#### 7.9.4 ザイフェルト系の可視化について

そしてザイフェルト系の可視化を行えるようにメソッドも定義しておきましょう。このメソッドではグラフの定義に NetworkX、グラフの描画で matplotlib から `plot` を利用します：

```
def draw_seifert_circles(self, file=None, layout=None):
    G = nx.MultiDiGraph()
    G.clear()
    scs = self.SeifertCircles
    nodes = []
    nodelist = []
    fsc = []
    edges = []
    a = ''
    b = ''
    asc = 97
    for x in scs:
        b = ''
        tmp = []
        for i in x:
            a = b
            ai = abs(i)
            fsc.append(ai)
            b = chr(asc) + str(ai)
            nodes.append(b)
            tmp.append(b)
        if len(a)>0:
            edges.append((a,b))
            G.add_edge(a,b,weight=0.5, sign=0)
        a = chr(asc) + str(abs(x[0]))
        nodelist.append(tmp)
        edges.append((b,a))
        G.add_edge(b,a,weight=0.5,sign=0)
        asc = asc + 1
```

```

G.add_nodes_from(nodes)
for j in self.Crossings:
    sj = sign(j)
    aj = abs(j)
    p0 = fsc.index(aj)
    p1 = fsc[p0+1:].index(aj) + p0 + 1
    G.add_edge(nodes[p0], nodes[p1], weight= -1, sign=sj)
"The classifications of edges into 3 types with sign."
sc=[(u,v) for (u,v,d) in G.edges(data=True) if d['sign']==0]
pb=[(u,v) for (u,v,d) in G.edges(data=True) if d['sign']==1]
mb=[(u,v) for (u,v,d) in G.edges(data=True) if d['sign']==-1]
plt.clf()
if layout is None:
    pos = nx.shell_layout(G)
else:
    if layout=="circular":
        pos = nx.circular_layout(G)
    else:
        if layout=="spring":
            pos = nx.spring_layout(G)
        else:
            pos = nx.shell_layout(G)
for i in nodelist:
    nx.draw_networkx_nodes(G, pos, nodelist=i, node_size=200,
                           alpha=0.8, color='r')
    nx.draw_networkx_edges(G, pos, edgelist=sc, width=1.5)
    nx.draw_networkx_edges(G, pos, edgelist=pb, style='dashed',
                           edge_color='b', width=1)
    nx.draw_networkx_edges(G, pos, edgelist=mb, style='dotted',
                           edge_color='g', width=1)
    nx.draw_networkx_labels(G, pos, font_size=8,
                           font_family='sans-serif')
plt.axis('off')
plt.show()
if file is not None:
    plt.savefig(file)
return(G)

```

このメソッドで行う可視化は絡み目そのものの可視化ではありませんが、ザイフェルト系は絡み目を境界とする曲面の設計図そのものになるので、その中心的な存在であるザイフェルト系の可視化によって絡み目の構造が把握できることになります。ただし、ザイフェルト系の可視化で真面目に各交点の位置を決めて描くことは、絡み目の交点数や成分数が増えれば非常に困難になることが予想されます。そこで、ここでは「車輪の再発明を

しない」という SageMath の基本方針を尊重して SageMath に含まれているパッケージで解決します。ここで、ザイフェルト円周が基本的に孤立した円周であることから、ザイフェルト系を有向グラフとして定義して表示すればどうでしょうか？SageMath には都合の良いことにグラフ理論向けのパッケージ NetworkX を標準で含んでいるのでこれを使わない手はありません。

まず、ザイフェルト円周を有向グラフとして可視化するためには、交差点をグラフの節点 (node) として定義し、それから各交差点を繋ぐ道を辺 (edge) として定義しなければなりません。ここで注意することはザイフェルト円周の系では同じ交点が二つの円周上に別々に現われるので、番号をそのまま節点にすると何を描いているのか不明になる恐れがあります。そこでザイフェルト系の円周単位で節点をグループに分けします。この分類はザイフェルト曲面の構成で貼られる円盤に直接対応しますが、たとえば円盤 ‘a’ の境界となるザイフェルト円周に属する節点の先頭に ‘a’ を配置すると節点の名前とその意味がより判り易くなります。そこで、函数 `char()` を使って整数から ASCII 文字を生成し、この文字を交差点番号を函数 `str()` で文字列として先頭に追加することで節点の名前を定めましょう。あとは節点のリストをまとめてグラフの節点として定義するためにメソッド `add_nodes_from()` を用います。ただ、これではでは節点を定義するだけです。そうではなく節点をザイフェルト円周単位で扱いたいので、`nodelist` に節点を円周単位のリストとして保存しておきます。一方のグラフの各辺は一括処理を行うよりもグループ単位で処理する方が効率的なので個別にメソッド `add_edge()` で定義します。この辺の定義では辺に重みを ‘weight’ で、種類を明記するために ‘sign’ を指定し、ザイフェルト円周上の辺、符号 +1 の交差点、符号 -1 の交差点が区分できるように設定しておきます。ここでの重みは可視化で節点配置を行う `spiral_layout` で意味があり、負の数であれば排斥、正の数であれば引力として系が安定するように節点の配置計算を行います。また `sign` は辺の色の設定で用います。さてザイフェルト円周を構成する節点と辺がこれで定義できたので今度は交差点自体も定義しておきましょう。この交差点はザイフェルト円周系では向きを持った線分として表現されています。そこでこの線分を重さを持った辺として表現します。つまり、符号  $\pm 1$  の交差点を表現する辺なら -1、ザイフェルト円周を構成する辺には 0.5 を設定しておきます。そして、ザイフェルト円周を構成する節点を節点リストとして纏めておきます。それから NetworkX ではグラフ可視化で節点の配置をレイアウトで指定することができます。ここで可能なレイアウトには `circular_layout`, `shell_layout`, `spring_layout` と `special_layout` の 4 通りあり、`circular` は全ての節点を円周上、`shell` は螺旋上に節点を並べるために重みはさほどの意味を持ちませんが、`spring` は辺の重みを利用して節点の配置を行います。ただし、`spring` では初期節点配置をランダムに配置して、位置エネルギーの計算を行う方法のために毎度、図式の配置が異なります。その結果、絡み目の節点の良い配置が得られたり、不可解な配置になったりと安定しない問題があります。この点は非常

に悩ましいことです。そのためにこのメソッドでは `spring`, `shell`, `circle` を必要に応じて選択可能にし、無指定の場合は `shell` を選択するようにしています。また、グラフを構成する辺に関してもグループ分けを行いますが、ここでは `sgn` でグループ分けを行い、このグループ分けした辺に対してメソッド `draw_network_edges()` で辺の線の太さ、色、形状を指定しています。ここでは-1の符号を持つ交差点を表現する辺を緑の点線、+1の符号を持つ交差点を表現する辺を青の破線で表現し、ザイフェルト円周を構成する辺は黒の実線とします。それから NetworkX のグラフ可視化では `matplotlib` の `plot` モジュールを用います。このことが `LinkDiagram` クラスの冒頭で `matplotlib` から `plot` を import していた理由です。このメソッドを定義する際に注意することは `plot` のメソッド `clf()` で描画の初期化を行わないで古い描画が残って再度表示されてしまうことです。そのためにここではレイアウトの指定を行う前に `plt.clf()` で描画の初期化を行っています。

ここで実例を示しておきます。最初に半群としての性質を見ておきましょう:

---

```
sage: load("LinkDiagrams.py")
sage: K3_1 = LinkDiagram([-1,3,-2,1,3,2])
sage: K3_1
GaussCodes:[-1, 3, -2, 1, -3, 2]
Crossings:[1, 3, 2]
SeifertCircles:[[3, 2, 1], [1, 3, 2]]
Components: 1

sage: Duo_K3_1 = K3_1 + K3_1
sage: Trio_K3_1 = Tri_K3_1 = K3_1 + K3_1 + K3_1
sage: Duo_k3_1
GaussCodes:[-1, 3, -2, 1, -3, 2, 5, -4, 6, -5, 4, -6]
Crossings:[1, 3, 2, 5, 4, 6]
SeifertCircles:[[4, 6, 5], [1, 3, 2, 5, 4, 6], [1, 3, 2]]
Components: 1

sage: Trio_K3_1
GaussCodes:[-1, 3, -2, 1, -3, 2, 5, -4, 6, -5, 4, -6, 8, -7, 9, -8, 7, -9]
Crossings:[1, 3, 2, 5, 4, 6, 8, 7, 9]
SeifertCircles:[[7, 9, 8], [1, 3, 2, 5, 4, 6, 8, 7, 9], [4, 6, 5], [1, 3, 2]]
Components: 1
```

---

最初の `load` 文で `LinkDiagram` クラスの読み込みを行った後、結び目 `K3_1` を定義し、その後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。ここでザイフェルト円周の例は図 7.12 に示す絡み目を使います。この絡み目は符号が +1 の三葉結び目と -1 の捻りが入った自明な結び目が絡んだものです。この絡み目のザイフェルト系を手書きで構築したものを図 7.13 に示しておきますが、このザイフェルト系は 4 個の円周で構成され、特に交差点番号 6 を持つ円周では交差点 6 を一つだけ持つ円周が現われます。この図では交差点を置換える帶は負の交差であれば緑の点線、正の交差であれば青の破線として表現しています。この例では三葉結び目側の交点は全て +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：

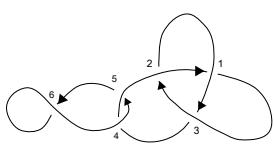


図 7.12 扱っている絡み目

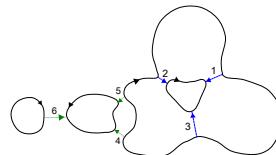


図 7.13 手書きのザイフェルト系

---

```
sage: d2 = LinkDiagram([[-1,3,-2,1,3,-4,5,2],[-4,-5,-6,-6]])
sage: d2
GaussCodes:[[-1, 3, -2, 1, -3, -4, 5, 2], [4, -5, -6, 6]]
Crossings:[1, 3, 2, -4, -5, 6]
SeifertCircles:[[-6, -4, -5], [-6], [-4, -5, 2, 1, 3], [3, 2, 1]]
Components: 2
sage: g2 = d2.draw_seifert_circles(file="d2_test.png", layout=spring)
```

---

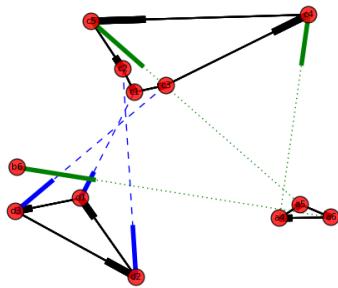


図 7.14 layout=spring による描画

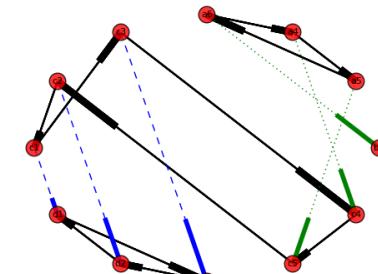


図 7.15 layout=circular による描画

LinkDiagram クラスでは絡み目の拡張ガウス・コードをその引数として与えると絡み目に応する LinkDiagram クラスのインスタンスを生成します。ここでインスタンス名

を入力するとガウス・コードやザイフェルト円周等の情報が表示されますが、この機能は RingElement クラスのメソッド `_repr_()` の上書きで実現しています。ザイフェルト円周の可視化はメソッド `drawSeifertCircles()` で行います。ここでは引数にファイル名と節点のレイアウトを指定しているために画像ファイルの出力と節点のレイアウトを行います。ここで出力した画像を図 7.14 と図 7.15 に示します。ザイフェルト円周は黒で交差点を表現する線分よりも太く表示し、交差点は符号が +1 であれば青の破線、符号が -1 であれば緑の点線としています。それからザイフェルト円周はそれら円周のリストから順番に ‘a’, ‘b’, ‘c’, ‘d’ … と割り当て、各節点にはグループを表現するアルファベットを交差点番号の先頭に置いた名前にしています。レイアウトの指示は多少の試行錯誤が必要になるかもしれません。図 7.14 では `spring` を設定したためにザイフェルト円周はグループ単位で適度に散らばった形で表示されます。これが `circular` や `shell` であれば円周上に接点が配置されます。実際、図 7.15 ではレイアウトとして `circular` を選択しているので接点は全て单一の円周上に載ります。どのレイアウトがザイフェルト系の構造を分かり易く示しているかは実際に幾つか試して最適なものを選ぶことになります。ただし、節点が円周上に並ぶものであれば `circular`、ザイフェルト円周が共通の中心点を持つものであれば `shell`、それ以外の一般的なものは `spring` を試すと良いでしょう。

この LinkDiagram クラスでは拡張ガウス・コードを基本としているので、先程の基本群を計算する函数もこのクラスに追加することが容易です。さて、このように絡み目固有の群の構成や、ザイフェルト系の可視化ができるようになりました。そうすると絡み目の構造をザイフェルト系で観察し、それらの分類を群で把握したいものです。具体的には三葉結び目と八の字結び目の基本群を構築したので両者が同じ結び目なのかどうかを調べたいところですが、そのときに問題になるのが次に述べる「語の問題」です。

### 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)**」と呼ばれるものです。ここで二つの群に対する同値性の検証で次に示す「**Tietze 変換**」と呼ばれる操作で移り合える群は同じものであることが知られています：

## — Tietze 変換 —

I	$\langle x_1, \dots, x_n \mid r_1, \dots, r_m \rangle$	$\Rightarrow \langle x_1, \dots, x_n \mid r_1, \dots, r_m, u \rangle$
		$u \in R$
I'	$\langle x_1, \dots, x_n \mid r_1, \dots, r_m, u \rangle$	$\Rightarrow \langle x_1, \dots, x_n \mid r_1, \dots, r_m \rangle$
		$u \in R$
II	$\langle x_1, \dots, x_n \mid r_1, \dots, r_m \rangle$	$\Rightarrow \langle x_1, \dots, x_n, y \mid r_1, \dots, r_m, y\zeta^{-1} \rangle$
		$y \notin \{x_1, \dots, x_n\}, u \in F$
II'	$\langle x_1, \dots, x_n, y \mid r_1, \dots, r_m, y\zeta^{-1} \rangle$	$\Rightarrow \langle x_1, \dots, x_n \mid r_1, \dots, r_m \rangle$
		$y \notin \{x_1, \dots, x_n\}, u \in F$

ここで群  $F$  の表示を  $\langle x_1, \dots, x_n \mid r_1, \dots, r_m \rangle$  とし、関係子  $\{r_1, \dots, r_m\}$  が自由群  $\langle x_1, \dots, x_n \rangle$  内で生成する帰結群を  $R$  と表記しています。ここでは Fox の本 [14] の P.59 にある例題を使って、二つの群の表示  $\langle x, y, z \mid xyz(yzx)^{-1} \rangle$  と  $\langle x, y, a \mid xa(ax)^{-1} \rangle$  が同じ群の表示であることを確認してみましょう：

## — Tietze 変換の例 —

$$\begin{aligned}
 \langle x, y, z \mid xyz(yzx)^{-1} \rangle &\xrightarrow{\text{II}} \langle x, y, z, a \mid xyz(yzx)^{-1}, a(yz)^{-1} \rangle \\
 &\xrightarrow{\text{I}} \langle x, y, z, a \mid xa(ax)^{-1}, a(yz)^{-1}, xyz(yzx)^{-1} \rangle \\
 &\xrightarrow{\text{I}'} \langle x, y, z, a \mid xa(ax)^{-1}, a(yz)^{-1} \rangle \\
 &\xrightarrow{\text{I}} \langle x, y, z, a \mid xa(ax)^{-1}, z(y^{-1}a)^{-1}, a(yz)^{-1} \rangle \\
 &\xrightarrow{\text{I}'} \langle x, y, z, a \mid xa(ax)^{-1}, z(y^{-1}a)^{-1} \rangle \\
 &\xrightarrow{\text{II}'} \langle x, y, a \mid xa(ax)^{-1} \rangle
 \end{aligned}$$

このように有限回の Tietze 変換の列によって二つの群の表示が同値であることが示せますが、この手法の難点は Tietze 変換を見つけなければ二つの群の同値性が主張できず、そのような Tietze 変換の列を見付けられないと Tietze 変換が構築できないこととが別問題であるため、Tietze 変換が判らないことが別の群の表示であると結論付けられない点です。この点はライデマイスター移動と似ていて、いずれにせよ計算と比較が容易な不变量が望まれる理由になります。

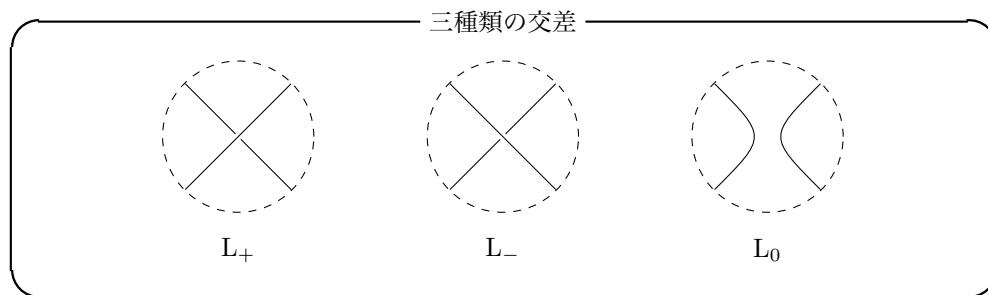
## 7.10 多項式不变量

このように群の表示を使って結び目/絡み目の同値性を判断することは簡単なことではありません。そこで多少情報が落ちても使い勝手の良い値で結び目/絡み目の分類がどこまでできるかやってみることになります。そこで、結び目/絡み目からさまざまな多項式を

生成し、これらの多項式を使って判別するということが行われています。これらの多項式の計算方法では補空間の基本群を直接使う代数的な方法、結び目/絡み目のザイフェルト曲面を貼って求める幾何学的な方法がありますが、正則射影図の局所的な交差点の状態を変更することで得られる正則射影図の多項式の間に成立する「**スケイン関係式**」と呼ばれる関係式から求める手法があります。なお、自明な結び目（正則射影図で交点を持たないもの）の多項式不变量は 1 になるように正規化されます。

### 7.10.1 スケイン関係式

向き付けられた結び目/絡み目の正則射影図に対して「**局所的な交差の変更**」として次の三種類の交差を考えます：



この三種類の交差に基づく関係式のことを「**スケイン関係式 (skein relation)**」^{*6}と呼びます。この関係式から得られる多項式は  $z^{-3}$  等の負の次数を許容するもので、より正確には「**ローラン多項式 (Laurent polynomial)**」と呼ばれる多項式です。そして、ライデマイスター移動でその多項式が不変になるものを結び目/絡み目のスケイン多項式と呼びます。スケイン多項式としては以下の関係式を充たす多項式がその代表として挙げられます：

- **ジョーンズ多項式 (Jones polynomial):**  $t^{-1}V_{L_+}(t) - tV_{L_-}(t) = (t^{\frac{1}{2}} - t^{-\frac{1}{2}})V_{L_0}(t)$
- **コンウェイ多項式 (Conway polynomial):**  $\nabla_{L_+}(z) + \nabla_{L_-}(z) = z\nabla_{L_0}(z)$
- **アレキサンダー多項式 (Alexander polynomial):**  $\Delta_{L_+}(t) + \Delta_{L_-}(t) = (t^{\frac{1}{2}} - t^{-\frac{1}{2}})\Delta_{L_0}(t)$
- **ホンフリー多項式 (HOMFLY polynomial):**  $xP_{L_+}(x, t) - tP_{L_-}(x, t) = P_{L_0}(x, t)$

これらの多項式は自明な結び目なら 1 になる性質がありますが、1 に等しいときに自明な結び目になるかどうかは別問題です。また、コンウェイ多項式とアレキサンダー多項式は、変数については  $z \leftrightarrow t^{\frac{1}{2}} - t^{-\frac{1}{2}}$  で互いに移りあえるという性質があります。またホンフリー多項式の由来は多項式を発見した人の名前の頭文字 (Hoste, Ocneanu, Morton,

^{*6} skein は縫糸の意味です。

Freyd, Likorish, Yetter) を並べたものです。これらのスケイン多項式は絡み目  $L_1, L_2$  の連結和  $L_1 + L_2$  に対しては  $L_1$  と  $L_2$  のスケイン多項式の積になることが判ります。

なお、SageMath の BraidGroup パッケージで実装されている多項式はジョーンズ多項式とアレキサンダー多項式です。歴史的にはアレキサンダー多項式が古く、ジョーンズ多項式は 1980 年代に現われた強力な不变量です。

### 7.10.2 ジョーンズ多項式

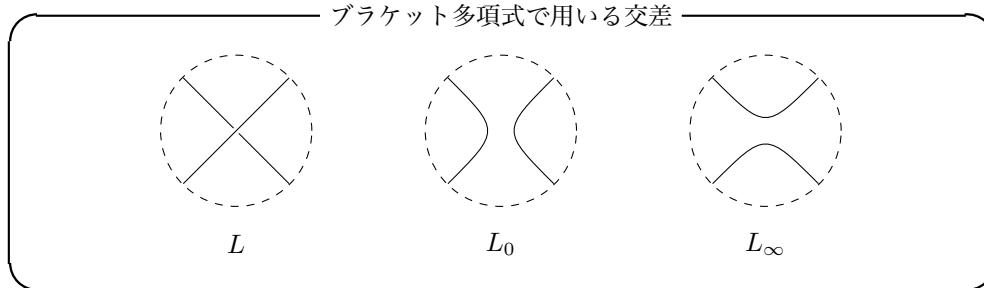
ジョーンズ多項式は作用素環の研究から得られたもので、やがてカウフマンのブラケット多項式で表現され、それからコンウェイ多項式と同様のスケイン関係式からも求められるように定式化が行われました。ジョーンズ多項式は後述のアレキサンダー多項式よりも結び目の分類で強力な多項式で、2016 年の現時点でも非自明な結び目で多項式が 1 になるものが発見されていません。なお、SageMath の BraidGroup パッケージは、その関係からオプションの指定がなければカウフマンのブラケット多項式の流儀で表示され、オプションの指定があればスケイン形式から得られる多項式を出力します。

### 7.10.3 アレキサンダー多項式

アレキサンダー多項式は古典的な結び目多項式です。スケイン多項式として認知されるまでは結び目群の表示から計算したり、ザイフェルト曲面を使って求めていました。この求め方からも判るように基本群と密接な関連を持っていますが、機械的に求めることでできる手法が、結び目群の表示から求める方法で、結び目群の  $n$  個の関係子をフォックス微分と呼ばれる特殊な「**微分**」を使って  $n \times n + 1$  の大きさのヤコビ行列を計算し、このヤコビ行列から  $n \times n$  の行列を取り出して、それらの行列式の最小公約数として(1次の)アレキサンダー多項式が得られます。この多項式の幾何学的解釈は補空間をザイフェルト曲面で切り開いた空間の「**普遍被覆空間**」と呼ばれる空間の構成に密接に関係します。ただし、このアレキサンダー多項式は他の結び目多項式と比較して非力で、たとえばアレキサンダー多項式が 1 になる非自明な結び目として「**樹下・寺坂結び目**」が著名です。

### 7.10.4 カウフマンのブラケット多項式

カウフマンのブラケット多項式は上述のスケイン関係式と別の交差関係を用います。なお、この多項式では結び目に向きを入れる必要はありません:



カウフマンのブラケット多項式は次の性質を充します:

—— カウフマンのブラケット多項式の性質 ——

1.  $\langle \text{O} \rangle = 1$
2.  $\langle L \sqcup \text{O} \rangle = (-A^2 - A^{-2})\langle L \rangle$
3.  $\langle L \rangle = A\langle L_0 \rangle + A^{-1}\langle L_\infty \rangle$

カウフマンのブラケット多項式はライデマイスター移動の TYPE II と TYPE III に関して変化がありませんが、TYPE I の移動については  $A^{\pm 3}$  だけ違いが生じます。実際に TYPE I の計算をしてみましょう:

$$\langle \text{O} \text{ (p)} \rangle = A\langle \text{O} \rangle + A^{-1}\langle \text{ (p)} \rangle = A(-A^2 - A^{-2})\langle \text{O} \rangle + A^{-1}\langle \text{ (p)} \rangle = -A^3\langle \text{O} \rangle$$

$$\langle \text{ (p) O} \rangle = A\langle \text{ (p)} \rangle + A^{-1}\langle \text{ O} \rangle = A\langle \text{ (p)} \rangle + A^{-1}(-A^2 - A^{-2})\langle \text{ O} \rangle = -A^{-3}\langle \text{ O} \rangle$$

ここで  $\text{ (p)}$  の交点の符号が 1,  $\text{ (p)}$  の交点の符号が -1 と幕の次数の符号が交点の符号に対応していることがわかります。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ので絡み目  $L$  の不変量になることが判ります。以上から、次で多項式  $J(L)$  を定めます：

$$J(L) \stackrel{\text{Def.}}{=} (-A^{-3})^{-w(\mathcal{D}_L)} \langle \mathcal{D}_L \rangle$$

この多項式  $J(L)$  は変数  $A$  の置換によってジョーンズ多項式  $V_L$  に変換することができます。つまり次の関係式を充します：

$$J(L) = V_L(A^4)$$

## 7.11 カウフマンのブラケット多項式を計算するプログラム

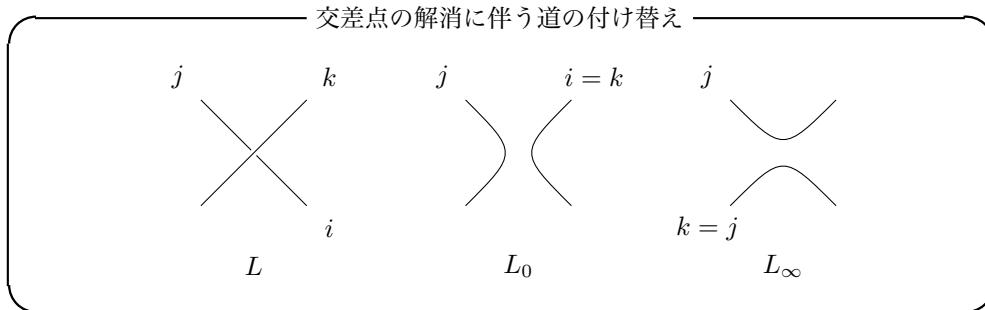
カウフマンの多項式の計算は非常に機械的に行うことが可能ですが、実際、 $L$  から  $L_0$ ,  $L_\infty$  への移行で交差が一つ解消しており、この交差一つの解消で二つの正則射影図が得られます。この操作を次の交差点で行うと  $2 \times 2$  で 4 個の正則射影図が、さらに次の段階では  $4 \times 2$ , すなわち 8 個の正則射影図が得られます。そして  $n$  段目では  $2^n$  個の正則射影図が得られます。その一方で、各段で交差点が消去されるので交点が  $n$  個の正則射影図であれば最終的には  $n$  段目で  $2^n$  個の互いに交差しない自明な結び目を成分とする絡み目の正則射影図が得られます。ここで自明な結び目を  $-A^2 - A^{-2}$  で置き換え、その段に至るまでの経路から  $A$  と  $A^{-1}$  を乗じて総和を計算すればカウフマンのブラケット多項式が計算できることになります。この点はスケイン関係よりも非常に明瞭で機械的な処理になっているので、その意味では計算し易いものになっています。実際、スケイン関係で交差を入れ替えた  $K_-$  の交差が減るのかどうかは実際にそれを判別するプログラムを構築しない限りは機械的に判断はできません。だから処理を行うたびに全体がより簡易なものになる保証がなく、それを確認する手続きが別途必要になるためです。とはいえて隣り合った交点の符号を合わせてしまえばライデマイスター移動の Type II によって二つの交差点が外せるので、このような戦略性を持った処理を行わなければなりません。ところがカウフマンのブラケット多項式の計算では交差点であればどこでも必ず交差点を減らすことができるのです。

では、カウフマンのブラケット多項式を計算するために必要なものは何でしょうか？ これはガウス・コードと各交点での符号の情報だけで十分です。実際、 $L$  から  $L_0$ ,  $L_\infty$  の構築は実質的に指定した交差点番号をガウス・コードから削除し、その付近での道の付け替えで済むからです。そして前述の LinkDiagram クラスではザイフェルト系を求めるメソッドを構築しており、このメソッドも交差点の解消で道の向きを変更しない場合の処理に対応しているのです。だから残りを構築すれば交差点の解消によって生成される絡み目や結び目のガウス・コードが構築できることになります。そこで、この多項式を計算するプログラムを LinkDiagram クラスのメソッドとして定義することにしましょう。まず最初に上道と下道の繋ぎ合わせで道の向きを変更することで交差点の符号が逆転するときのプログラムを構築しておきましょう。

そこでまず、正規のガウス・コードのリストと交差点の符号を保持したリストで構成されたリストを正則射影図を代表する対象として「Diagram」と呼ぶことにします。そして Diagram を成分とするリストを「Diagrams」と呼びます。ただ Diagrams はカウフマンのブラケット多項式の計算過程で生じるもので基本は Diagram です。また LinkDiagram

クラスのインスタンスとの違いは、Diagram は LinkDiagram クラスの正規ガウス・コードと交差点リストで構成されたもので、多項式の計算で大量に生成される中間的な絡み目を表現する中間的なデータであるということです。

次に正則射影図の交差を解消する函数を作成します。ここで重要なのは交差点の解消によって結び目が絡み目に変形される可能性があることです。この状況を次の図で説明しましょう：



この図では交差点番号  $i$  の解消を示していますが、その実態は交差点番号  $i$  における上道と下道の繋ぎかえです。ガウス・コードでは  $a_i$  の下道が右下から交点  $i$  に向かい、交点  $i$  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、交点  $j$  が絡み目の別成分との交差で生じるのであれば  $i$  から  $j$  を通過しても  $k$  を通過することはありません。

さて、 $L_0$  の交差点の解消では上道  $a_k$  を交点  $i$  で二つに分割し、下道  $a_i$  と上道  $a_k$  の上側半分を繋ぎ、下道  $a_j$  と上道  $a_k$  の下半分を繋ぎます。このように上道と下道を繋ぎ合わせる操作であるために上道と下道が別成分であれば一つの成分に融合される処理であることが分かります。問題になるのはこれら上道と下道が同じ成分にある場合です。このときは交点  $i$  の符号で事情が異なります。まず交点  $i$  の符号が +1 であれば上道  $a_k$  は左下から右上に向かいます。そして右上からやがて下道  $a_i$  を通って交点  $i$  に戻るので  $L_0$  の処理では左右に成分が分離することが分かります。ところが  $L_\infty$  の処理では  $a_i$  を通って上道  $a_k$  の下半分を逆走するとやがて下道  $a_j$  を逆走し、それから上道  $a_k$  の上半分を通過して下道  $a_i$  に戻るために成分の変化が生じません。ところが交差点  $i$  の符号が -1 であれば上道  $a_k$  は右上から左下に向かうために左下側と下道  $a_i$  が繋り、 $L_0$  にて成分の変化は生じないものの  $L_\infty$  にて上下の成分の分割が生じます。このように成分の変化が交差点の符号に依存して生じることと、交差点の解消で上道を逆走する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::-1]
    sb = toggle_crossings(sb, x)
    a0 = [a2 + x]
    pa = [[a1, a2], sa]
    ma = [a0, sb]
    if s==1:
        z = [pa, ma]
    else:
        z = [ma, pa]
    return(z)

def toggle_crossings(Crossings, GaussCode):
    newCrossings = []
    list_crossings = list(map(abs, GaussCode))
    ordr = map(abs, Crossings)
    for n in Crossings:
        i = abs(n)
```

---

```

m = n
if i in list_crossings:
    if list_crossings.count(i)==1:
        m = -n
newCrossings.append(m)
return (newCrossings)

```

---

ここで注意すべきことは削除する交差点の符号が  $+1$  のときに  $L_0$  で絡み目の成分が一つ増え、削除する交差点の符号が  $-1$  のときには  $L_\infty$  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  $L_\infty$  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。

次に交差点の後に絡み目の二つの成分に交差点番号が配置されているときに交差点の解消を行う函数 `crossing_change_b()` を示します。この函数による処理では絡み目の成分変化はありませんが、一部の交差点で符号の反転が生じます：

---

```

def crossing_change_b(disjointDiagram):
    [GaussCodes, Crossings] = disjointDiagram
    z = []
    n = Crossings[0]
    s = sign(n)
    i = abs(n)
    sa = Crossings[1:]
    sb = copy(sa)
    if -i in GaussCodes[0] and i in GaussCodes[1]:
        CodeA = GaussCodes[0]
        CodeB = GaussCodes[1]
    else:
        CodeA = GaussCodes[1]
        CodeB = GaussCodes[0]
    p0 = CodeA.index(-i)
    p1 = CodeB.index(i)
    px = CodeB[p1+1:] + CodeB[:p1]
    pa = [[CodeA[0:p0] + px + CodeA[p0+1::]], sa]
    x = px[-1::-1]

```

---

```

sb = toggle_crossings(sb, x)
pb = [[CodeA[0:p0] + x + CodeA[p0+1::]], sb]
if s==1:
    z = [pa, pb]
else:
    z = [pb, pa]
return(z)

```

---

そしてこれらを統括する交差の解消を行う函数 crossing_change() を次に示します。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ときは函数 crossing_change_b() で処理を行うようにすれば良いのです：

---

```

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  $L_\infty$  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だからです。

この交差点の解消は与えられた絡み目の交差点のリストに従って一斉に行うことになり、この操作によって生成される正則射影図を整理して上手く並べると、各正則

射影図から二本の分枝が生じる樹形図として整理することができます。そして、函数 crossing_changes_a() と crossing_changes_b() が output 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  $L_\infty$  を置く表記に意図的に合わせたためです。さらに Diagram のリスト Diagrams の添字を二進数で表示すると添字 0 の箇所で  $A$  を乗じ、添字 1 の箇所で  $A^{-1}$  を乗じることに対応していることが分かります。そして、自明な結び目では交差点が存在しないために拡張、正規ともに自明な結び目のガウス・コードは空リスト ‘[ ]’ が対応します。このことから最後に函数 crossing_change() で得られたリスト Diagrams は空リストのみで構成され、各 Diagram の添字を二進数表示したものが  $A$  と  $A^{-1}$  の積の情報で、各 Diagram のガウス・コードの空リストの総数から 1 を減じたものが  $-A^2 - A^{-2}$  の幕の次数の情報になります。これらを利用して Diagrams からカウフマンのブラケット多項式の計算を行う函数 calc_Kauffman_Bracket() が次のように構築されます：

---

```

def kauffman_bracket(linkDiagram, LinkName=None, DB=None):
    var('A')
    Diagrams = [[linkDiagram.GaussCodes, linkDiagram.Crossings]]
    dtable = "diagrams"
    kbptable = "kauffman_bracket_table"
    As = []
    Fs = []
    KBIK = []
    Stage = 0
    if DB is not None and LinkName is not None:
        connect_db(DB, [[dtable, "LinkName text", "Stage int", "Position text",
                         "GaussCodes text", "Crossings text"]])
        insert_diagrams2table(DB, dtable, LinkName, Stage, Diagrams)
    cps = linkDiagram.Crossings
    for i in cps:
        Stage = Stage + 1
        tmp = map(crossing_change, Diagrams)
        Diagrams = []
        for j in tmp:
            Diagrams = Diagrams + j
        Crossing = abs(i)
        if DB is not None and LinkName is not None:
            insert_diagrams2table(DB, dtable, LinkName, Stage, Diagrams)
    for i in Diagrams:
        j = len(i[0]) - 1
        KBIK.append((-A**2-A*(-2))**j)
    n = len(KBIK)
    m = len(Integer(n-1).digits(2))
    Polynomial = 0

```

```

for i in range(0,n):
    As.append(sum(Integer(i).digits(2)))
for i in As:
    Fs.append(A**(m-2*i))
for i in range(0,n):
    Polynomial = Polynomial + KBIK[i]*Fs[i]
Polynomial = expand(Polynomial)
if DB is not None and LinkName is not None:
    connect_db(DB, [[kbptable, "LinkName text", "CrossingNumber int",
                     "GaussCodes text", "Crossings text", "Polynomial text"]])
    insert_kauffman_bracket2table(DB, kbptable, LinkName, Diagrams, Polynomial)
return(Polynomial)

def kauffman_bracket_polynomial(LinkDiagram, KnotName=None, DB=None):
    var('A')
    Polynomial = kauffman_bracket(LinkDiagram, KnotName, DB)
    w = sum(map(sign, LinkDiagram.Crossings))
    return(expand(((A)^3)^(-w)*Polynomial))

```

---

この処理では樹形図の情報が最終的な計算結果リストの位置と対応が、その位置情報を2進数表示したときの次数リストと一致していることを利用しています。この2進数への変換はメソッド digits() を用いますが、このメソッドは SageMath の Integer 型のものであるのに対し、関数 range() が生成するリストの成分は Python の int 型であるため、そのままでは利用できません。同一表示で型が異なっているのも厄介ですが、ここでは Integer() で型を int 型から Integer 型に変換して処理を行っています。SageMath では整数の利用でこのようなことが生じ易いので注意が必要です。

計算例を以下に示しておきましょう：

---

```

sage: K3_1 = LinkDiagram([-1,3,-2,1,3,2])
sage: Duo_K3_1 = K3_1 + K3_1
sage: Trio = K3_1 + K3_1 + K3_1
sage: Ans = map(factor, map(kauffman_bracket, [K3_1, Duo_K3_1, Trio_K3_1]))
sage: for i in Ans:
....:     print i
....:
-(A^12 + A^4 - 1)/A^7
(A^12 + A^4 - 1)^2/A^14
-(A^12 + A^4 - 1)^3/A^21

```

---

と、ガウス・コードと交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ね。

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ケット多項式を属性とする表、もう一つの表をロルフセンの結び目表の名前、消去した交差点番号、樹形図での絡み目の位置、絡み目のガウス・コードと交差点の情報を属性とする表とします。ここで前者の表が計算結果、後者の表は計算過程に関するものになります。では SQLite3 で最初に DB と上記の表をあらかじめ生成しておきましょう。ここで SQLite3 を Python から利用するためのモジュール sqlite3 を読み込んでおく必要がありますが、SageMath にはパッケージに含まれているので import 文で読み込むだけです。

それからデータベースを開いて表を生成する函数 connect_db() を次で定義します:

---

```
def connect_db(DB, tables):
    if len(tables)>0:
        cursor = sqlite3.connect(DB)
        sql = "SELECT * FROM sqlite_master WHERE TYPE='table'"
        csr1 = cursor.execute(sql)
        ans = flatten(csr1.fetchall())
        for i in tables:
            tname = unicode(i[0])
            if not(tname in ans) and len(i)>2:
                sql0 = "CREATE TABLE " + tname + " "
                sql1 = "(" + unicode(i[1])
                for k in i[2:]:
                    sql1 = sql1 + "," + unicode(k)
                sql1 = sql1 + ")"
                cursor.execute(sql0 + sql1)
            cursor.commit()
            return(1)
    else:
        return(0)
```

---

この函数では指定された DB に接続し、SQLite3 の DB 単位に唯一作成される表

sqlite_master から表を検索し、リスト tables に同一名の表が無いか検索し、表が存在しなければ表を生成する処理を行います。なお、cursor オブジェクト情報を一旦蓄え、メソッド commit() を実行することで初めて処理が確定されます。この commit を忘れていると cursor オブジェクトをメソッド close() で閉じた途端に情報が消えてしまうので注意が必要です。

---

```
def insert_diagrams2table(DBName, TableName, LinkName, Stage, Diagrams):
    cursor = sqlite3.connect(DBName)
    sql = "insert into " + TableName + " values (?, ?, ?, ?, ?, ?)"
    m = 0
    for i in Diagrams:
        Position = number2position(m, Stage)
        m = m + 1
        GaussCodes = str(i[0])
        Crossings = str(i[1])
        cursor.execute(sql,(LinkName, int(Stage), Position, GaussCodes, Crossings))
    cursor.commit()
    cursor.close()

def insert_kauffman_bracket2table(DBName, TableName, LinkName, Diagrams, Polynomial):
    cursor = sqlite3.connect(DBName)
    sql = "insert into " + TableName + " values (?, ?, ?, ?, ?, ?)"
    GaussCodes = str(Diagrams[0])
    Crossings = str(Diagrams[1])
    CrossingNumber = int(len(Diagrams[1]))
    KBP = str(Polynomial)
    cursor.execute(sql, (LinkName, CrossingNumber, GaussCodes, Crossings, KBP))
    cursor.commit()
    cursor.close()
```

---

まず、位置を定める函数が number2position() です。この函数は引数として Diagrams の位置と交差点を消去する段階にそれぞれ対応する int 型の整数を取ります。この函数は位置情報として Diagrams での絡み目の情報の位置を 2 進数表記した文字列として返し、そのときの表示桁数を交差点を消去する段階を表現する数値が対応します。たとえば三葉結び目の段階は ‘0’ で一つの交差点の消去を行うと段階は ‘1’。このときの  $L_0$  が ‘0’,  $L_\infty$  が ‘1’ になり、次の段階 ‘2’ では  $L_0$  派生のものが ‘00’, ‘01’,  $L_\infty$  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'[]')

sage: bf1 = cur1.execute("select * from diagrams where Stage==2")
sage: ans=bf1.fetchall()
....: 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:
```

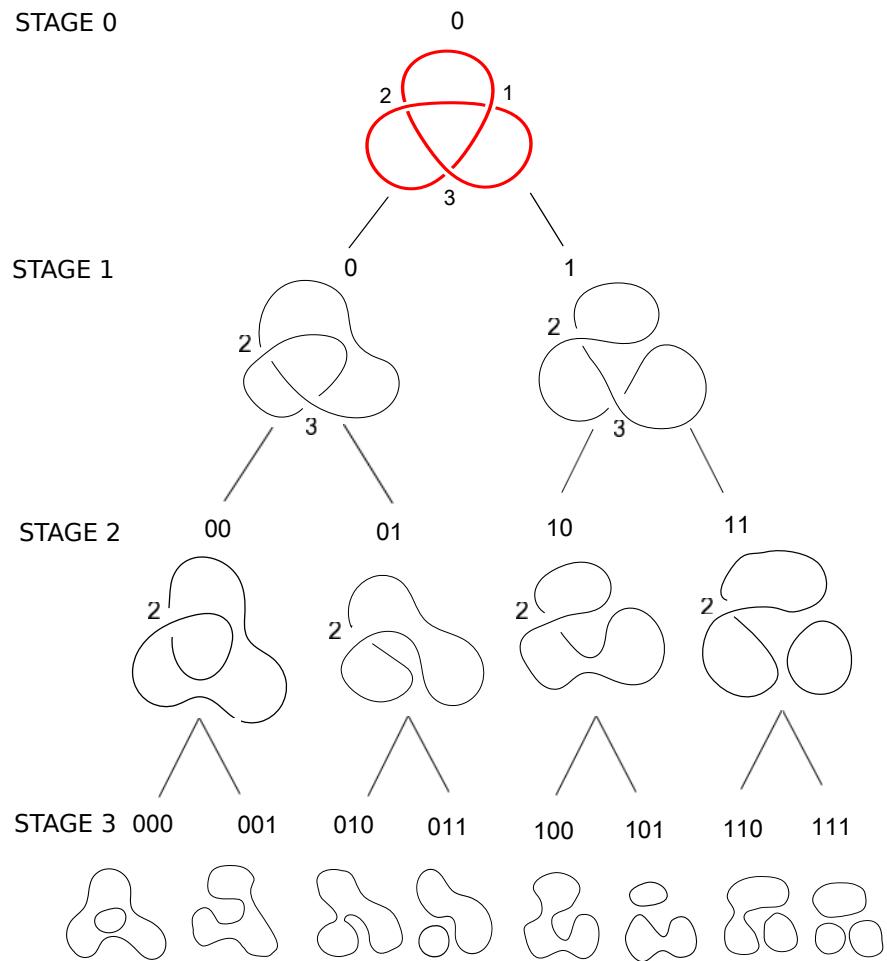


図 7.16 三葉結び目での処理

```
....:     print i
(u'Trefoil', 2, u'[[[], []], []]', u'[[[], []]', u'-A^5 - 1/A^3 + 1/A^7')
```

と、このようにデータの検索が行えるのです。ここでやっていることの詳細を解説すると、`kauffman_bracket()` では DB と LinkName にデータベースの情報と絡み目の名前を指定するとカウフマンのブラケットをデータベースに書き込みます。ここで `kauffman_bracket()` フункциでは書き込むテーブルは現時点では固定しています。このテーブルは分解の様子を書き込む `diagrams` テーブルと多項式を書き込む `kauffman_bracket_table` の二つです。`kauffman_bracket()` フункциではテーブルを初期化することなく、繰り返し書き込むだけです。

## 第8章

# SageMathで画像処理

## 8.1 画像の読み込み

SageMath は Python を中核とした総合数学環境であると述べましたが、Python で画像を扱う場合には二つの方法があります。一つが PIL (Python Imaging Library) を用いる方法で、もう一つが Matplotlib を用いる方法です。双方とも SageMath に含まれています。なお、PIL を用いて読み込んだ画像は Python の instance になりますが、Matplotlib を用いて読み込んだ画像は numpy.ndarray という numpy の配列になります。これは matplotlib が数値配列ライブラリ NumPy を基に構成されたライブラリであるため、逆に言えば Matplotlib を適宜用いることで商用の数値行列処理システムの MATLAB ④ と同等の処理が可能になるという副作用があります。

前述のように SageMath には画像処理ライブラリの PIL (Python Imaging Library) が標準ライブラリとして含まれています^{*1}。PIL を用いた画像の読み込み、書き出しや表示では、それらの処理を行うモジュールがサブモジュールの Image に纏められているので、あらかじめ `from PIL import Image` でモジュールの読み込みを行う必要があります。それから画像の読み込みは函数 `open()` でファイルを開いて行います。このとき JPEG 形式の画像ファイルを扱う場合にはあらかじめ `libjpeg` が必要です。ところで SageMath にはこのライブラリは標準で含まれていないため、JPEG 画像を扱う際にはあらかじめインストールしておくか、SageMath の追加パッケージからインストールしておく必要があります。さて、ファイルを開いて、インスタンスにしてしまえば、あとは各種ライブラリのメソッドや函数を画像ファイルの処理が行なえることになります。

たとえばディレクトリ Documents 上に `ScuolaDiAtene.png` という名前の画像ファイルを開いて表示する場合は以下の処理となります：

---

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

---

この例では最初にモジュールの読み込み、それから画像の読み込みと外部アプリケーションを用いた画像の表示と画像データの保存を行っています。まず、最初のライブラリの読み込みは先程説明したものです。それから Image ライブラリの中の函数 `open()` で画像ファイルの読み込みを行なって SageMath のオブジェクトとして取り込んでいます。この際に画像

---

^{*1} PIL から分枝 (fork) したものに Pillow (<http://python-imaging.github.io/>) があります。PIL は Python 2.x のみに対応しており、setuptools も対応していませんが、Pillow は 3.x や setuptools に対応し、PIL と同様の使い方もできます。

オブジェクトは名前 `im` に対して束縛されています。それからインスタンス `im` のメソッド `show()` を用いて読み込んだ画像の表示を行なっています。ちなみに、このメソッド `show()` は外部アプリケーションを用いて画像の表示を行うメソッドで、表示に用いる外部アプリケーションを切り替えることができます。ちなみに Linux 版の SageMath では表示アプリケーションとして `xv` があらかじめ指定されているので、`xv` がインストールされていない環境や `xv` がインストールされておらず、その上、他のビューアを用いるような設定が行われていなければエラーになります。`xv` 以外のビューアの指定も容易に行えます。たとえば、ImageMagick の `display` を表示用の外部アプリケーションとして用いたければ `im.show(command='display')` のようにメソッド `show()` の `command` オプションを用いればよいのです。この `command` オプションでは、検索経路上にあるアプリケーションならそのアプリケーション名、そうでなければアプリケーションに至るまでの経路も含む Python の文字列として指定すれば良いのです。なお、OSX 版では函数 `open()` が用いられるように設定されています。また SageMath をノートブック形式のフロントエンドで利用していればメソッド `save()` で PNG 形式や BMP 形式で画像で保存するとノートブック側にも表示されます。

さて、画像オブジェクト `im` に付随するメソッドの一覧を見たければ、「`im.`」と入力して TAB キーを押すこと、つまり、「`im.`」TAB」でメソッドの一覧が表示されます。この機能は Python そのものの機能ではなく、ターミナル版であれば IPython、ノートブック形式であればユーザ・インターフェイス側の機能です。さて、PIL には画像データを RGB の行列や配列等に変換して処理する函数はありません。そのために画像データを配列データに変換して MATLAB 系の言語が行うように配列上の処理で済ませなければ、Matplotlib, SciPy や NumPy といったライブラリが別途必要になりますが、これらのライブラリも SageMath には最初から含まれています。

まず、Python 上で効率的に多次元の数値配列を扱うために NumPy を用います。この NumPy は、Python で多次元の数値配列を扱うためのライブラリ Numeric と後述の数値計算を行うためのライブラリ SciPy の数値配列ライブラリ Numer を統合したものです。そのこともあって NumPy は SciPy や Matplotlib の基底になっており、SciPy や Matplotlib で扱うデータも NumPy の配列であることが前提になっています。

この numpy ライブラリのメソッド `array()` を用いることで、PIL の函数 `open()` で読み込んだ画像データを RGB の配列データに変換することができます。ただ、この NumPy は数値配列ライブラリであるために数値配列そのものの可視化は行えません。この可視化では後述の Matplotlib ライブラリのモジュールを用いることになります：

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

---

この例では最初に NumPy を import で読み込みますが、その際に接頭辞として numpy ではなくより短い np を用いるように ‘as’ を用いて指定しています。画像データの配列データへの変換は NumPy の array() フィルを用います。この array フィルは Python のリスト等のデータを配列に変換するフィルです。さて、PIL ライブラリを使って読み込んだ画像の本来の大きさはメソッド size() で調べられます。このインスタンスを array() で NumPy の配列に変換したとき、その配列の大きさはメソッド shape() で調べられます。最初に im.size の結果から画像は縦 509、横 800 画素の画像であることが判ります。それから imat.shape の結果では im.size で現われなかつた ‘3’ という数がありますが、これは読み込んだ画像がカラー画像であることを意味し、画像データが  $509 \times 800$  の大きさの赤(R)、青(B)、緑(G)に対応する配列に分解されていることを意味します。さて、NumPy の配列は MATLAB 系の言語と違い、その添字は 1 ではなく 0 から開始します。つまり、NumPy の配列 imat はその第一成分の添字が 0 から 508 までの値を取り、第二成分の添字が 0 から 799 までの値を取り、第三成分の添字が 0, 1, 2 を取り得ることになります。

さて、SageMath に読み込んだ画像を函数 array() で NumPy の配列に変換すると、MATLAB や Yorick のような数値行列処理言語のような数値配列に対する操作で画像の処理が可能となります。まず、RGB を個別に取り出してみましょう。まず、NumPy の配列に変換した画像データは画像の大きさの数値配列が 3 個含まれていることがメソッド shape() の結果から判っています。さて、これら 3 個の配列は第三の添字を指定することで取出が可能となります。つまり、第三の添字が 0 の配列が画像の赤の強さ(輝度)に対応し、同様に第三の添字が 1 の配列が画像の青の強さ、そして第三の添字が 2 の配列が青の強さに対応します。これを SageMath で取り出すときは imat[:, :, 0] で赤、imat[:, :, 1] で青、imat[:, :, 2] で緑の輝度に対応する配列が得られます。ここで用いた添字記号 ‘:’ は、その添字記号が置かれた場所で添字が取り得る値の全てを意味し、MATLAB やそれに類似する行列処理言語で広く用いられている表記で、Python ではスライスオブジェクトと呼ばれる構文です。

さて、実際の配列で  $i$  から  $j$  までの  $j - i$  個の添字を取り出したければ、MATLAB では  $a(i:j)$  と表記します。Python の NumPy 配列でも同様の表記が可能です。ただし、Python では配列が 0 から開始するために  $a[i-1:j]$  と表記することになり、これで添字

$i - 1$  から添字  $j - 1$  までの  $j - i$  個の部分配列を返却することになります。また、増分を 1 以外に設定する場合は ‘ $10 : 0 : -1$ ’ のように ‘始点’ : ‘終点 + 1’ : ‘増分’’ と表記します。これは MATLAB 系の言語では ‘始点’ : ‘増分’ : ‘終点’’ と始点と終点の間に挟むので MATLAB 系の言語に慣れている方は注意が必要です。なお、些細なことですが、MATLAB 系の言語と異なり配列の添字は “[ ]” で括らなければなりません。“( )” は関数を構成する要素なので SageMath(Python) の配列で間違って使わないように注意して下さい。

ここで画像データ `imat` は  $509 \times 800$  の 3 個の配列データになっています。では ‘`imat[200:300,300:500,0]`’ は何になるでしょうか？ 実は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 軸の正方向になります。したがって ‘`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 の赤なので赤色の強度を示す画像になります。この箇所を表示すると次の図 8.1 が得られます：

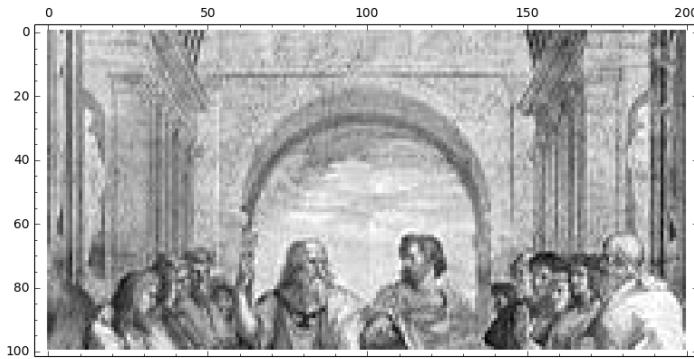


図 8.1 画像の一部切り取り

NumPy はそもそも数値配列を扱うためのライブラリであるために画像の表示といった

画像の処理のためのモジュールや函数を持ちません。このような画像処理を行うために Matplotlib に付随する pyplot といった NumPy の配列に対応した画像表示用のライブラリを利用する必要があります。また、2次元配列、すなわち行列データの可視化であるならば SageMath には標準で matrix_plot 函数が用意されているのでそちらを使うことも可能です。

## 8.2 OpenCV の利用

OpenCV(Open Computer Vision) は Python で画像処理を行うためのパッケージです。この OpenCV は SageMath には標準で含まれていませんが、SageMathCloud ではあらかじめ用意されており、モジュールの読み込みを行うだけで利用可能です。ただ、この OpenCV を SageMath で利用することは難しいことではなく、計算機上でインストールした OpenCV の cv.py と cv2.so の二つのファイルを SageMath の Python に複製することで利用可能になります。

## 第 9 章

# SageMath の拡張

## 9.1 sagemath からのパッケージ入手

SageMath にはさまざまなパッケージが存在し、それで十分に思えるかもしれません。しかし、Python の GUI ライブライアリの wxPython や PyQt4 といったライブライアリ、数学関連のデータベースや CLISP 等のアプリケーションといったものを追加することも可能です。このようなパッケージは ‘<http://www.sagemath.org/download-packages.html>’ で公開されています。現在、Standard, Optional, Huge, Experimental の四種の範疇に分類されて FTP や BitTorrent 等の P2P からの入手が可能となっています。

そして入手したパッケージは SageMath をインストールした user-id で、仮想端末上で  
`sage -i <パッケージファイル>` と入力することで SageMath にインストールすることができます。

## 9.2 一般の Python パッケージのインストール

SageMath は Python 上で構築されたシステムであるため、SageMath 側の Python にライブライアリやパッケージをインストールすることが可能です。この場合は [sagemath.org](http://www.sagemath.org) からのパッケージを入手してインストールする方法と異なり、あらかじめ環境変数の設定を行う必要があります。なぜなら SageMath 側の Python やライブライアリ一式を利用しなければならないので、環境変数 PATH と LD_LIBRARY_PATH をインストールしてある SageMath の状況に合せておく必要があります。たとえば UNIX 環境で Sage が /usr/local/sage にインストールされ、仮想端末で Bash をシェルとして用いている環境なら `export PATH=/usr/local/sage/local/bin:$PATH` と  
`export LD_LIBRARY_PATH=/usr/local/sage/local/lib64:/usr/local/sage/local/lib:$LD_LIBRARY_PATH` と設定しておきます。あとは Sage に easy_install があるので、それを用いたり、setup.py 等を用いたりと、インストールしようとする Python パッケージ別の対処になります。

## 9.3 GNU R のパッケージのインストール

上記の設定を行っていれば、SageMath に付属する GNU R に CRAN で公開されているパッケージの導入も行えます。

## 第 10 章

SageMathCloud

## 10.1 SageMathCloud(SMC) とは

SageMathCloud^{*1}は SageMath 6.10 を基に Linux のディストリビューションの一つの Ubuntu 上で構築されたクラウド計算機環境です。SageMath はさまざまな数学関連のアプリケーションやライブラリ、それらを支える言語を含む膨大な Python で構築された環境です。だからアプリケーションとして SageMath だけを利用することや、SageMath から限定的に各アプリケーションやライブラリにアクセスすることはとても「**もったいない**」ことなのです。そのために、この SageMathCloud では SageMath だけではなくさまざまなアプリケーションを「Jupyter Notebook」^{*2}を使って利用することも可能になっています。この Jupyter Notebook は SageMath や SageMathCloud 標準のノートブック形式のユーザ・インターフェイスで利用されている IPython Notebook^{*3}の後継で、Python 以外の言語に対してもノートブック形式のシェルとしても利用できるように開発されたものです^{*4}。したがって SageMathCloud では単に SageMath を動かすだけの環境ではなく、Jupyter を用いて SageMath が本来備えているライブラリやアプリケーションをウェブ・ブラウザから直接利用できます。この意味することは非常に大きなことで、Android や iOS の端末からも SageMathCloud を経由することでさまざまな処理が行えるのです。これほどの御利益がありながら、SageMathCloud を利用するために利用者がすべきことは後述の利用者登録を行うだけで済むのです。なお、2016 年 7 月現在、SageMath は 7.2 がリリースされているため、SageMathCloud の SageMath は 6.10 とやや古い版で、そのため SageMath 7.2 からリリースされた結び目理論向けのモジュールがまた利用できないといった不利な点もありますが、ネットワークに接続されたスマートフォンや PC 上のウェブ・ブラウザから本格的な SageMath の環境が享受できるという御利益があるのです。

SageMathCloud を利用するにあたって利用可能なウェブ ブラウザは特に指定はありません。Firefox, Chrome, Safari 等でも動作可能です。さらに 3D グラフ表示も SageMath では Java が必要とされるために Java アプレットが使えない携帯電話で表示ができませんでしたが、SageMathCloud は iPhone のようなスマートフォンでも 3D グラフ



図 10.1 SageMath, Inc

^{*1} <https://clouds.sagemath.com>

^{*2} <https://jupyter.org>

^{*3} <http://ipython.org/notebook.html>

^{*4} 2015/10 の時点では 40 以上のプログラム言語に対応しています。

<https://github.com/ipython/ipython/wiki/IPython-kernels-for-other-languages> .

表示の利用が可能です。なお、Chrome には「The Sagemath Cloud」というプラグインがあり、このプラグインを使うと素早くログインすることができます。だからと言って必須と言えるほど便利至極なものではありません。

SageMathCloud は 2013 年の 4 月に立ち上げられ、その GUI も SageMath ノートブック形式のユーザ・インターフェイスよりも洗練されています^{*5}。なお、従来のオンライン上で SageMath が無料で利用できるサービスの Sagenb^{*6}がありました^{*7}が、SageMathCloud は無課金のサービスですら遙かに強力な環境を提供しています。

## 10.2 利用者登録について

SageMath を利用するためにはあらかじめ利用者登録を行う必要がありますが、この登録は無料です。図 10.2 に示すログイン画面で Google+, Facebook, GitHub, Twitter のアカウントを使ってログインすることができます：

ここでの登録は無料で、特に指定を行わないかぎり無課金の利用になります。ここで無課金で一つのプロジェクトに割り当てられるディスク容量は 3GB、メモリは 1GB、CPU の 1 つの core を有する形になります。またタイムアウトする時間が定められており、1 時間の無操作の場合になっています。この条件を変更することもできますが、その場合は課金制で 3 種類のプランが選べ、それぞれ月単位、あるいは年単位の課金が必要になります。なお、処理速度については無課金のものでも大雑把ですが MacBookAir 2013 と同程度で、そこそこ本格的な処理が可能です。だから無課金でもちょっとした計算や文書の作成、それらの共有といったことが十分に実用的な範囲で行えます。だからウェブベースであることも含め、教育や試計算には適しているのではないか。実際に 400+ の学生の学生の教育で用いている例もあるようです。詳細は <https://github.com/sagemath/cloud/wiki/Teaching> を参照して下さい。また SageMathCloud の FAQ が <https://github.com/sagemath/cloud/wiki/FAQ> に纏められていますが、このような教育事例が挙げられています。

ここではこの SageMathCloud を使う上で、どのようなアプリケーションがあって、どのように使えるのかといった簡単な事例を幾つか紹介することにします。

---

^{*5} GUI は専門家に開発させていることです。

^{*6} <http://www.sagenb.org>

^{*7} 2015 年 5 月に停止。

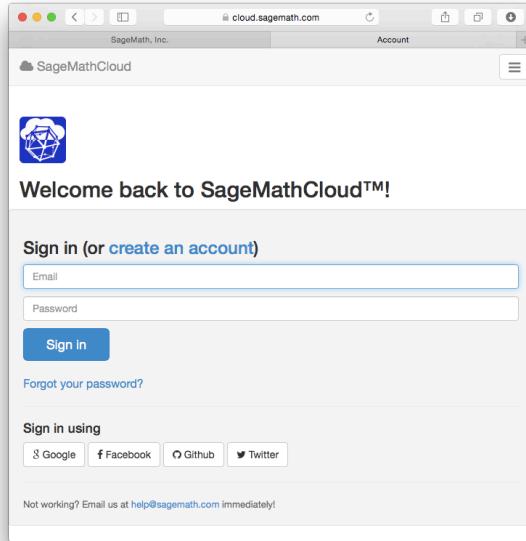


図 10.2 ログイン画面

### 10.3 基本設定

SageMathCloud を利用するうえで基本的な設定は SageMathCloud のサイトに接続した状態で、ブラウザ右上側に表示されている利用者の名前を押すと基本設定のページに遷移します。そこで「Settings」、「Billing」と「Upgrades」という名前のタグがありますが、通常の利用に関連することは「Settings」で行います。この「Settings」設定可能なことを以下に纏めておきます：

- 利用者情報の設定
- 仮想端末のフォントと画面と文字色の設定
- キーボードショートカットの設定
- エディタの設定

- その他の設定
- UI の外観の設定

図 10.3 に利用者情報と仮想端末の設定を示しておきます:

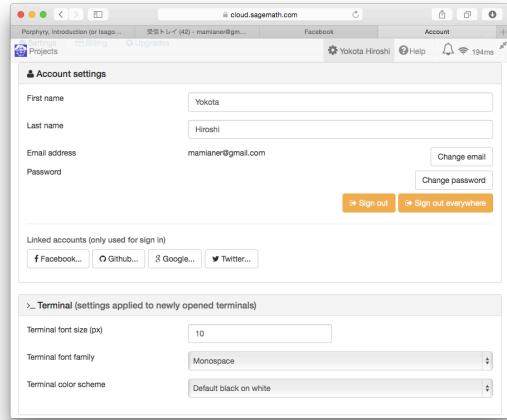


図 10.3 利用者情報と端末の設定

図 10.4 にキーボードショートカットの設定とエディタの設定の一部を示しておきます:

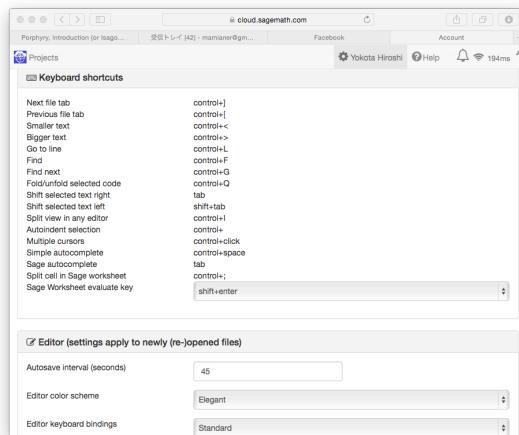


図 10.4 キーボードショートカットの設定とエディタの設定

次に「Billing」と「Upgrades」は有償で利用するときに請求先のクレジットカードの指定、それとどのような条件で利用するかといったことを設定する箇所で、後述のプロジェクト

クト側からはここで設定した有償利用の設定に基づいた設定以外のことはできません。現時点では有料利用のプログラムとして 3 通りの設定があり、支払いは月単位か年単位の双方が選べます。図 10.5 に Upgrades のページの一部を示しておきます：

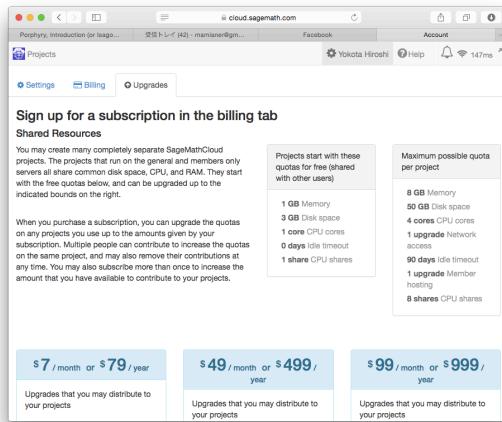


図 10.5 Upgrades のページ

## 10.4 プロジェクト

SageMathCloud ではプロジェクト単位でジョブやファイルの管理を行います。そのために利用者として登録したての場合は最初にプロジェクトを生成します。このプロジェクトの生成は「New Project...」とあるボタンを押すと「Create a New Project」というページに遷移します。ここでは表題と概要を「Title」と「Description」欄にそれぞれ記載して左下側の「Create project」ボタンを押せばプロジェクトが新規に生成され、最初の Projects のページに戻ります。プロジェクトでさまざまな作業を行うためには、この Projects ページに表示されたプロジェクトの名前を選択すればよいのです。プロジェクトの設定もプロジェクト単位で行うことができます。その場合、プロジェクトを選択し、そのプロジェクト画面の右上にある「Project settings and controls」から行うことができます。このボタンを押すと図 10.6 に示す「Settings and Configuration」ページに遷移します：

ここで設定できる項目として、もしも有償利用の設定をあらかじめ行っていれば、それを適用するための設定が可能です。それ以外にできることを以下に挙げておきます：

- プロジェクトの隠蔽と削除
- プロジェクトの制御

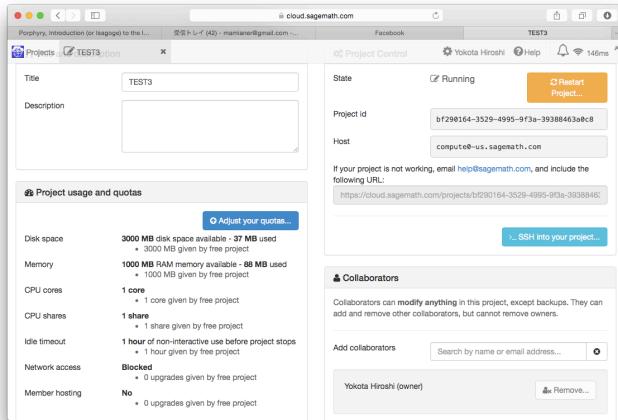


図 10.6 Project settings and controls のページ

- プロジェクトの共有設定
- Sage ワークシートサーバの再起動

ここでプロジェクトの制御は基本的にプロジェクトの再起動を行う程度で、その他はプロジェクトの ID やそのプロジェクトが起動しているホスト名の情報や SSH の公開鍵の情報があります。

次に SageMathCloud ではプロジェクト単位で共有、つまり、SageMathCloud の利用者で一つのプロジェクトを共有することで互いに閲覧や編集が自由に行えるのです。この設定を行うためには「Search by name or email address...」と表示された欄に名前か E-mail アドレスを記入して SageMathCloud の登録者の検索を行います。ここで条件に合致する利用者が複数存在すれば表示されたリストから相手を選択して招待するためのメールを送付します。それでプロジェクトの共有が行えるようになります。

## 10.5 ファイルのアップロードとダウンロード

まず PC 等の端末上のファイルを SageMath 側にアップロードすることも、逆にファイルをダウンロードすることも可能です。まずアップロードのときはファイルの送り先のプロジェクトで右上の「New」メニューを押して図 10.7 に示すファイル生成に移ります：

このページで「Upload files from your computer」とある項目で大きく「Drop files to upload」と表示された箇所にローカルにあるファイルをドラッグしてドロップするか、その箇所をクリックします。もしもクリックしたときはファイルのセレクターが現われる

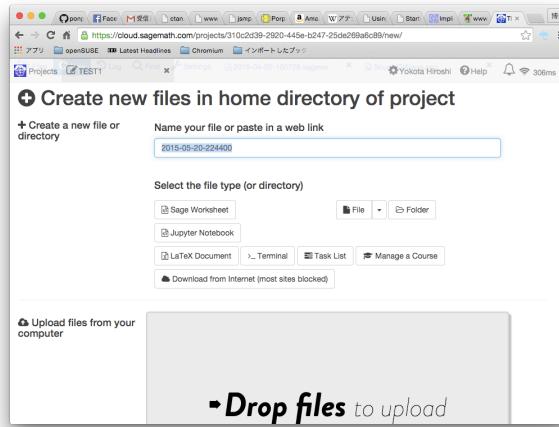


図 10.7 ファイル生成ページ

ので、そこでアップロードするファイルを選択します。またファイルをドロップしたのであれば、しばらく待つと図 10.8 に示すようにサムネイルがやがて表示されます。当然のことですが大きなファイルだと転送に時間がかかります：

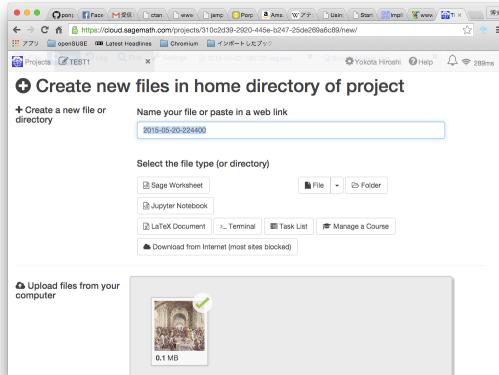


図 10.8 ファイルのドロップ後

なお、アップロードしたファイルや、プロジェクト内で生成したファイルの一覧は「File」メニューを押すことで表示される図 10.9 に示すページで確認することができます：

この図の右上に「Terminal command..」と記載のある入力欄があります。ここでは通常の UNIX コマンドが入力可能です。このページではファイルやディレクトリ操作（ファイル/ディレクトリの生成/削除、移動、名前変更等）が行えます。

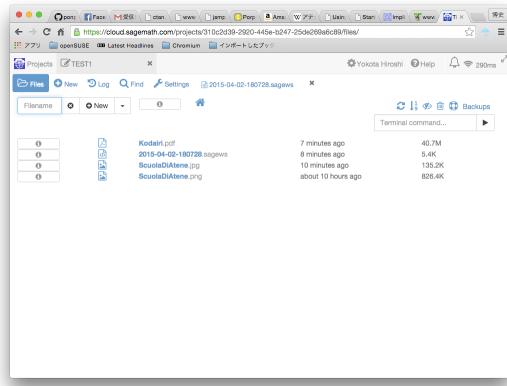


図 10.9 プロジェクト内のファイルの一覧

SageMathCloud ではプロジェクト単位にファイルが保管され、プロジェクト間のファイルの移動も複製することで行えます。また、同一プロジェクトであれば SageMathCloud が包含するアプリケーションからも通常の Linux 環境と同様に扱うことができます。たとえば SageMathCloud 上にアップロードしたファイルの読み込みの例を図 10.10 に示しておきます。アップロードした画像を SageMathCloud 上に読み込んで、函数 `matrix_plot()` で表示した例を示しておきます：

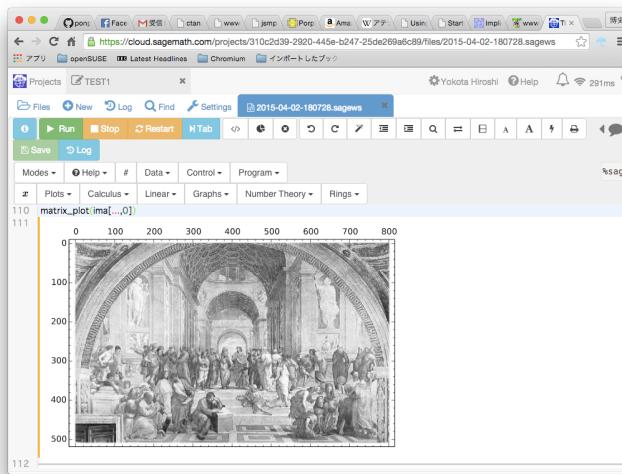


図 10.10 アップロードした画像の表示

## 10.6 端末, Jupiter, L^AT_EX の利用について

SageMathCloud は単に数式処理システム SageMath を単にクラウド環境で実現するだけではなく、SageMath が包含するさまざまなアプリケーションが利用できる環境です。アプリケーションの起動には二つの方法があります。一つは先程のファイルのアップロードで用いた New メニューから図 10.7 に示すファイル生成に移行し、そこで「Select the file type(or directory)」の項目以下に並んだ「Sage Worksheet」、「Jupyter Notebook」、「LaTeX Document」や「Terminal」を選ぶか、あるいは「File」メニューを押してプロジェクトのファイル一覧のページに移行し、そこで「New」メニューを押して図 10.11 に示すように「Sage Worksheet」等のメニューを出して選択する方法があります：

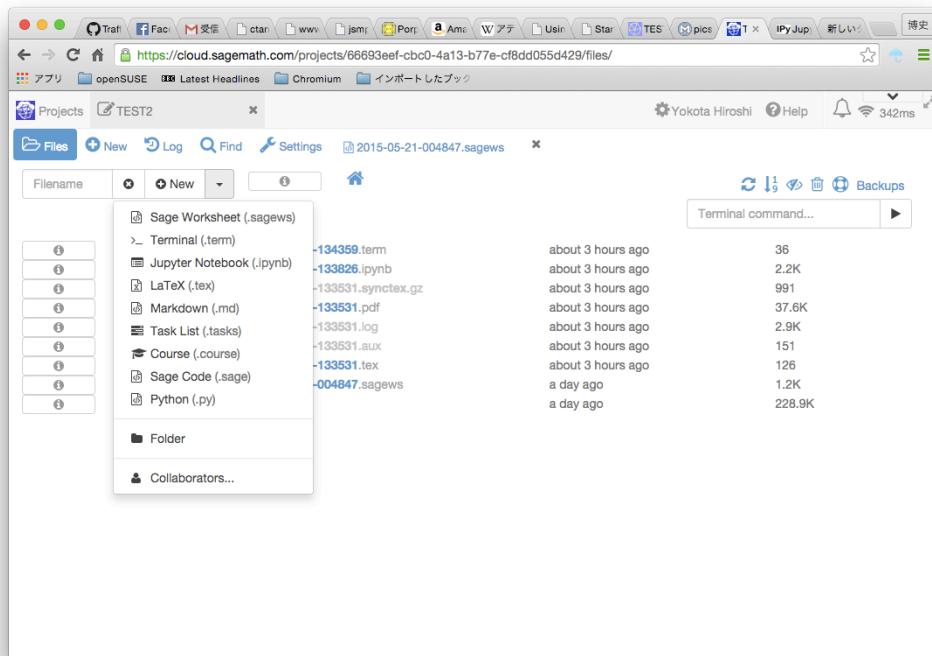


図 10.11 New メニューの一覧

ここで「Sage Worksheet」は説明するまでもなく SageMath のワークシートですが、通常の SageMath のワークシートよりも洗練されています。それから「Terminal」は起動するとウェブ・ブラウザを通常のテキスト端末として利用できます。ここで SageMathCloud というサービスを Ubuntu Linux 上で実現していることから、Terminal を起動するという

ことは Ubuntu という Linux 環境が直接利用できることを意味します。また, SageMath というシステムは雑多なアプリケーションやライブラリの集合体であるために, ターミナルを使うことで SageMath を構成するアプリケーションやライブラリを SageMath から必要に応じて呼び出すだけではなく, そのアプリケーションやライブラリを直接活用することが可能になります。なお, 端末で処理した結果はそのままプロジェクト単位でディスク上に残すことができます。

次の「Jupyter」は Jupyter notebook と呼ばれるノートブック形式のシェルを起動します。この Jupyter^{*8}は IPython の後継のプロジェクトで, Jupyter notebook は Python 専用のノートブック形式のシェルの IPython notebook の中核を他の言語のシェルとして利用できるようにしたもので。そのためにその他の言語, たとえば統計処理システムの GNU R, 近年注目されている数値計算言語 Julia 等に対応するカーネルを利用することで, ウエブ・ブラウザ上でこれらの言語のノートブック形式のシェルとして使えます。ちなみに <https://try.jupyter.org/> に Jupyter をシェルとして利用する GNU R, Julia と IPython のデモがあります^{*9}。なお, SageMathCloud では Jupyter Notebook で利用できる kernel として Python2, Python3, GNU R と Julia があらかじめ用意されており, カーネルの切替はノートブックのセル単位で行えます。つまり, Jupyter では一つのノートブックにこれらの言語の結果が混在可能です。この kernel の切替はメニューの「Kernel」から使いたい言語に対応する kernel を選択することで行いますが, この選択を行った時点でアクティブなセルから次に kernel を切替えるまでが指定した kernel に対応する言語環境が利用できることになります。ただし, kernel を切替えて再び戻したときに以前の処理結果はノートブックには記載されても言語環境が再起動されているために, それ以後の入力セルに以前の処理結果が引継がれません。なお, kernel の切替を行わなければ kernel を切替えてからの処理結果はそのまま後続のセルに引継がれます。

Jupyter+Python でノートブックにグラフや絵を表示させることができます。この場合はあらかじめおまじないとして `%matplotlib inline` を入力しておきます。すると Matplotlib に包含される函数で画像の表示を行うと図 10.12 に示すようにそのまま Jupyter 側のノートブックに画像が表示されます。このことは IPython Notebook も同様です:

それから L^AT_EX は幅広く利用されている組版処理環境です。こちらは図 10.13 に示すように, ブラウザ画面を左右に二分して L^AT_EX ファイルのエディタ画面が左半分, レンダリングした結果が右半分に表示されます。左半分に表示された L^AT_EX ファイルを直接編集

^{*8} <https://jupyter.org>

^{*9} <https://github.com/ipython/ipython/wiki/IPython-kernels-for-other-languages> には Jupyter で利用可能な kernel の一覧があります。

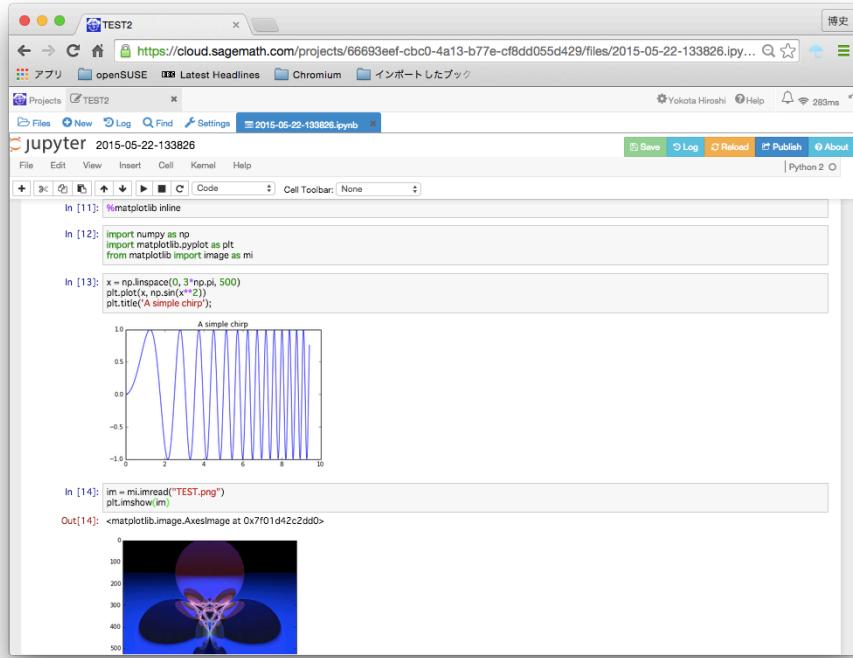


図 10.12 Jupyter+Python

すると早速コンパイルが行われて結果が右側に表示されるというもので、作成した文書のプレビューを行い、最終的に仕上げた文書を PDF に変換してダウンロードできます：

なお、クラウド上で LATEX のサービスを行うものに Cloud LaTex^{*10}があります。この Cloud LaTeX は SageMathCloud が提供する LATEX 環境よりも本格的なサービスで、原稿ファイルのドラッグ・ドロップによるアップロードが容易に行えること、日本語、中国語等のマルチバイト言語にも標準で対応しています。とはいっても SageMathCloud 上の LaTeX も TeXLive であるために一通りのスタイルファイルが準備されています。ただし、日本語文書の作成では幾らかの工夫が必要になります。まず SageMathCloud は通常、PDFLaTeX が呼び出されます。このときに SageMathCloud 側であらかじめ用意されたテンプレートでは日本語を扱うことができません。そのためにはクラスファイルやパッケージの設定を行う必要があります。また、PDFLaTeX ではなく pLaTeX や LuaTeX-ja を呼び出して利用することも可能です。ただし、SageMathCloud で利用できないクラスファ

---

^{*10} <https://clouddlatex.io/ja>

The screenshot shows the SageMathCloud interface with a project named 'TEST2'. The code editor contains the following LATEX code:

```

276 の
277 イデアのことを専ら\textgreek{>e'idos}と呼んでいます。), そして概
278 念は我々の
279 対象への理解が深まることで語られること、つまり、
280 「説明規定」(\textgreek{l'ogos}, logos, account) が更新されて
281 ゆく可能性が
282 あることも理解されるでしょう。そしてこの「\textbf{t}{それが何である
283 か?}」や
284 「\textbf{t}{それがどのようなものであるか?}」といった問に対する回答に
285 ついて
286 考察した人物がアリストテレス(\textgreek{>Aristot'elhs})
287 \index{ANTRO@人名!あ!ありすとれす@アリストテ
288 レス,\textgreek{>Aristot'elhs},Aristotle}
289 です。
290
291
292
293
294

```

右侧有日文注释和一幅古典画。

図 10.13 SageMathCloud 上の L^AT_EX

イルも一部あるようです¹¹。この SageMathCloud の L^AT_EX 環境で、この本の §2 の文書のように図や図式が多くなるとやや苦しくなりますが、一寸した論文の著作には問題にはならないでしょう。

**■PDFLaTeX で日本語文書を作成する場合:** PDFLaTeX が SageMathCloud の L^AT_EX 環境で既定値として使われています。そのため日本語の利用では次の設定で日本語文書の作成が可能となります:

```
\documentclass{scrartcl}
\usepackage[whole]{bxcjkljatype}
```

ただしスタイルファイルの bxjsarticle には対応していないことです。

**■LuaLaTeX(LuaLaTeX-ja) を利用する場合:** PDFLaTeX が既定値であるために、LuaLaTeX を利用するためには PDFLaTeX の代りに LuaLaTeX が動作するように指定する必要があります。そのために SMC の環境変数 latex_command に lualatex のオプショ

*11 ここでの情報は Mathlibre の WIKI¹²を参照しています。

図 12 アテネの学寮より: プラト  
ンとアリストテレス

ンとレンダリングすべき L^AT_EX 文書も含めて記載する必要があります:

```
%sagemathcloud={"latex_command ":"lualatex -synctex=1 -interact nonstopmode sample.tex"}\documentclass{ltjsarticle}
```

ここで例では L^AT_EX ファイルの名前が “sample.tex” のときのヘッダ部分を示しています。

このように SageMathCloud は Cloud 環境における Sage の動作に限定されるものでなく数学の研究に必要とされる一切合切を統合した、より大規模なシステムを目指しており、同列に論じられることの多い *Mathematica*, MATLAB や Maple といった数式処理や数値行列処理システムとの際立った違いを見せているのです。SageMath の核となる Python は Python(ニシキヘビ)」というだけあって一切合切を丸呑みしている傾向がありますが、Python 言語を中心としたさまざまなアプリケーションを統合した環境を構築しているという面で注目すべきことなのです。

## 第 11 章

### 手引 (ポルピュリオス)

## 11.1 概要

この章ではテュロスのポルピュリオス ( $\Piορφύριος$ ) のエイサゴーゲー ( $Eἰσαγωγή$ ) 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)」[28] で知られています。この翻訳は「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知られるボエティウス (Boethius) が行なったもので、忠実なラテン語の翻訳になっています。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そうです。

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)[26] がギリシャ語文献から翻訳したも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}されていました。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、より深く調べる為にはこちらの文献の方が良いと思われます。なお、ここでの訳はバーンズの訳と註釈を中心にオーエンの訳も必要に応じて参考して翻訳しています。

この著者のポルピュリオスに関して現代に伝わっている情報は意外にありません。まずポルピュリオスはフェニキアのテュロスで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^{*2}。ことから後にギリシャ語で「王」という意味のバシレウス ( $Βασιλεύς$ )^{*3}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

^{*1} [http://www.ccel.org/ccel/pearse/morefathers/files/po..._isagogue_02_translation.htm](http://www.ccel.org/ccel/pearse/morefathers/files/po..._isagogue_02_translation.htm)

^{*2} アラブ世界へのヘレニズム文明の伝播では、シリア人のキリスト教徒がアラビア語への翻訳で活躍しています。

^{*3} 東ローマ帝国では皇帝の意味です。

Purple) の生産で当時は有名だったのですが、この紫色がローマ皇帝の色であったことから「**ポルピュリオス**」という渾名^{*4} が付けられています。ポルピ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 ( $\Lambda\sigma\gamma\gamma\iota\nu\varsigma$ )^{*5} に修辞学、数学と哲学を学び、このロンギノスがポルピュリオスという渾名を付けたようです。それから 263 年にローマに行って新プラトン主義^{*6} の創始者として知られるプロティノス ( $\Pi\lambda\omega\tau\iota\nu\varsigma$ , 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ことから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を行ったことやマルセラ (Marcella) という女性と結婚したことがポルピュリオスが妻のマルセラに宛た手紙 [34]^{*7} から伺えますが、彼が何時、何処で死んだといった伝承は残っておらず不明です。なお、ポルピ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、ポルピュリオスは新プラトン主義の学者です。その彼が記述した手引書であるエイサゴーゲーが⁹ (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、ポルピ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るのです^{*8}。まず、ポリピュリ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たのです。このことからポルピュリオス以後の新プラトン主義の哲学教育ではエイサゴーゲーから教育を開始するというという教育課程になります。この点についてはプラトンの著作には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

^{*4} バーンズの本 [26] の表紙が紫色なのもこの洒落でしょう。

^{*5} 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処刑されています。

^{*6} 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。

^{*7} マルセラはポルピ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いものの、他は結婚適齢期を迎えていたようです。そして、その手紙によると結婚した理由も、ポルピュリオスが子供が欲しいわけでも妻に世話をしたかったわけでもなかったとか言い切っていたりし、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。

^{*8} 詳細についてはカテゴリー論 [1] 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一部でアリストテレス寄りの考えをもっていたことも関係するようです。

ることは「秘儀の漏洩を防ぐ」ためと考えられたこと^{*9}、それと新プラトン主義のライバルであるストア派^{*10}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は新プラトン主義的な解釈やそういった夾雜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に多くが翻訳され、そこから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)が新プラトン主義の影響下にあったアリストテレスの哲学への解釈に対して、より新プラトン主義思想と融合させてしまいます。結局、12世紀になってイブン・ルシュド(ibn rušd, 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)が「純正アリストテレス」を唱え、その夾雜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)されることになるのです。

このエイサゴーゲーをポルピュリオスが著述したきっかけの一つにポルピ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が挙げられます。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ものです。これが一番知られている伝承ですが、それと別の伝承ではエトナ山の火炎の調査のためにシチリア島を行ったとも言われています^{*11}。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(Chrysanthus, 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ます。ところがさっぱりわからないので、ポリピュリオスにローマに戻って指導するか、それができないならせめて手引を書いて送って欲しいとポルピ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。

^{*9} ある意味で深読みのし過ぎですが。

^{*10} ヘレニズム哲学の一学派でキティオンのゼノン(Zénon)が創始者。アカデマイア学派、逍遙学派、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的なもの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を導入しています。

^{*11} このアンモニオス(Ἀμμώνιος, 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(Ἐμπεδοκλῆς, Empedocles)のや、その噴火を船で観察したというプラトンのことを思い出させるものです。

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、西ヨーロッパには古代のヘレニズム文明との大きな断絶があり、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ったことです。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なると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翻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であった地域は蛮族が割拠し、言語的にも文化的にも東西に分断されつつあったのです。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、そして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な文献になり、これらを基に西ヨーロッパの哲学が発展することになりますが、こういった著作がそのまま埋もれたり、散失してしまう可能性も十分にあったのです。実際、ボエティウスは「範疇論」と「命題論」以外のアリストテレスの著作をラテン語に翻訳していますが、これらの翻訳は現在は失なわれており、アリストテレスの本格的な受容はアラビア哲学を経由することになります。この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ものであるという立場に加え、その第一章で類、種、種差、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、種差、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「**ポルピュリオスの樹 (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した。ちなみに類と種の関係をポルピ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**」^{*12}の火種にもなっているのです。

*12 「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ようなことではなかったようです (詳細は [21] を参照)。

## 11.2 はじめに

必要なこととして、クリュサオリオス (Chrysarorius)^{*13} より、まずアリストテレス (*Ἀριστοτέλης*, Aristotle) の範疇 (カテゴリー) について学ぶために、類 (*γένος*, genus)^{*14} が何で、種差 (*διαφορά*, difference) が何で、種 (*εἶδος*, species)^{*15} が何で、特有性 (属性, *ἴδιον*, property) と偶有性 (付帯性, *συμβεβήκός*, accident) が何であるかを知ることができます^{*16} - そしてまた定義の論証、それと一般的な (類の種への) 分類や証明に関わる事象にとっても、これらの研究が有用なので - 私は、あなたに要所を押さえて説明するときに、手短に入門的形式をとって、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淵で、他の物事やより広範囲の探求を必要とするものだからです. ここで私はあなたに古の賢人達 - 中でも殊に逍遙学派の人々 (*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.

^{*13} ポルピュリオスの弟子. ローマの元老院にて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ピュリオスに求め、それに応じて書かれた文書がこの「手引」です.

^{*14} バーンズ [26] によると *γένος* は「族」、「種類」や「型」といった意味で、後述の「種」と誤語があてられている *εἶδος* も「型」、「族」、「種類」とほぼ同じ意味なので、これらは混合して記述されることが多い言葉であるとのことです.

^{*15} *εἶδος* は一般的に形相 (form) と誤され、ものの形を意味します. ポ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、これが現在の英語の 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、アリストテレスはもっぱら *εἶδος* を用いています.

^{*16} この類、種、種差、特有性と偶有性の 5 項目はポルピュリオスによるものです. 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、後世ではポルピ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遙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 [26] 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} ポエティウスのイサゴーゲー第二注解 [19] 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、ここで实在を明言しないということはポルピュリオスが正真正銘の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实在ということはあり得ないことです.

### 11.3 類について

どの類 (*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。だから、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ぶのです。ヘーラクレース一族 (*Ἡραλεῖδαι*, 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) に由来す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のは、各人の出生の起源、それは彼の先祖であったり、生まれた場所であったりします。その意味で、我々はオresteス (*Ορέστης*, Orestes) はタンタロス (*Τανταλός*, Tantalos) からの類²⁰、ヒュロス (*Υλλος*, 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、ピンドロス (*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、父親もそうであるような、起源の一種になるのです。この意味付けは理解し易いものでしょう; というのも、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 (*Κέκροψ*, 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; その後で、单一の起源 (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であると言います。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て、先行する事象との疑いな類似性からです; というのも、そのような類はそ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のです。

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 (逍遙学派の) 哲学者たちへの説明規定 (*λόγος*) になります。その説明の下書きをすると、彼等は類が ‘それが何であるか?’ に対する回答で、種で異なる幾つかの事象を述定 (*κατηγορεῖν*)²⁵していると語ることでそれを表現します²⁶; たとえば、動物です。

²⁰ オreste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reste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 ピンド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* という言葉は本来、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」とも言えますが、ここでは「カテゴリー論」[1] の註にしたがって「述定」と訳します。

²⁶ アリストテレスは形而上学の△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

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 ‘これ’ や ‘あれ’) が語られ^{*27}、またあるものは幾つかの事象 - いわゆる類や種や種差や特有性や偶有性 (これらは何かを固有ではなく共通で保持するもの) です。たとえば、動物は類です；人間は種です；理性的であるということは種差です；笑うことができるということは特有性です；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、それら (類) が幾つかの事象を述定するもので、ただ一つの事象だけを述定するものと異なります。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ても、種ではなく数で異なる事象を述定するからです。だから人間は、種であるので、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、類なので、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からです (笑うことができるということは人間だけ、ことに人間の述定だからです) 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、類は種差とも共通の偶有性でも異なりますが、というのも種差と共通の偶有性は、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?」に対する回答なのです。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は種差で、黒色ということが偶有性なのです。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なのです。

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

---

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图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。ポル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图形の類といわれる意味」については明瞭に述べていませんが、最初に述べているもの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ものが対応するでしょう。そして、ポルピ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、ポルピュリオスは「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が、その存在はさておいて「言語的」なものであると主張しているのです。このことによってプラトンとアリストテレスの差異を目立たないものにしているのです ([1] 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} ポルピュリオスは述定を類、種、種差、特有性と偶有性の 5 項目に分けて述べています。また、ポルピ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。

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種差や共通の偶有性からそれらを区分します。この類というものの輪郭を描くということでは、そして、何らの過剰も何らの不足もありません。

## 11.4 種について

我々が種 (*εἶδος*, species) と呼ぶものは、第一に、あらゆるものものの形 (*εἶδος*, form)^{*29}なのです - それはこう言われています：

ことのはじめに彼の姿は王国の要なれ ...^{*30}

我々はまた前述のものあつまりでとある類の下にあるものなどを種と呼びます^{*31} - 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^{*32}.

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であると言いました.），また種とは類の下にあるものであるところで言うのであれば、類はある何かの類であり、種はある何かの種なのだから、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。

^{*29} 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* を区別することなしに用いていますが、アリストテレスはプラトンのイデアの意味で専ら *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**」の意味があるからです。なおポルピュリオスは外観を *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むものであり、さらにはものの本質を含む意味になっています。たとえば *εἶδος* が白いミルクだとすれば、*μορφή* は白く塗ったものです。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するというようにです。

^{*30} 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。

^{*31} 第二の意味として、「種」には類の下にあるもの、つまり、類 - 種という階層があることを述べています。

^{*32} 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）になります。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るのです。また、類は種により、種は類によるという循環的な定義が導入されています。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はありません。

そこで、彼ら(逍遙学派)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するものです - ところが、これでは最も特定的(*ειδικός*, special)なものとただ一つの種のものの表現になり、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*³³.

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)*³⁴には、最も総合的(*γενικός*, general)な事象があれば、他に最も特定的な事象もあるのです；そして最も総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。最も総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；それから最も総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(とはいって、あるものと他のものとの間に関係を持たされています)*³⁵.

私が言っていることは单一の述定の型(範疇、カテゴリー)の場合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*³⁶。その下に物体(*σῶμα*, body)があり、物体の下で生命のある物体があり、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るのです*³⁷.

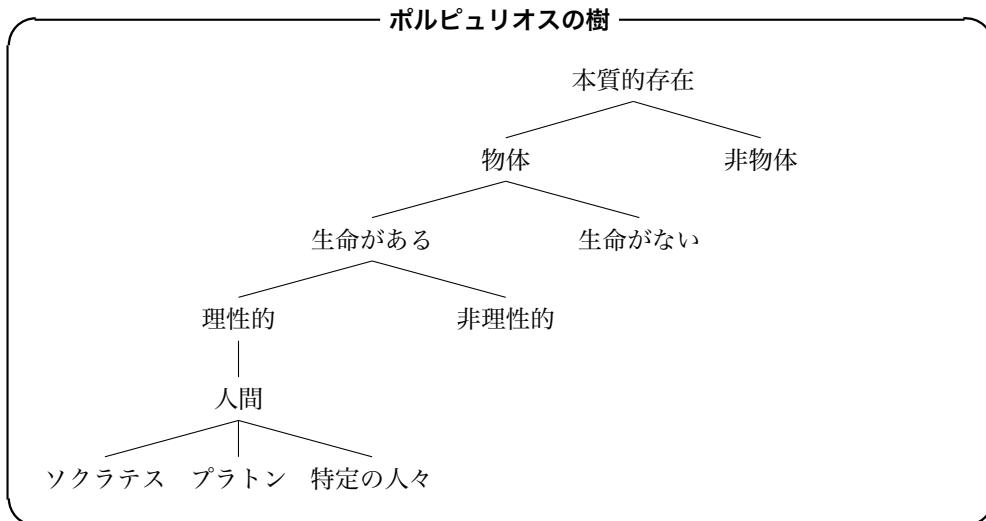
*³³ ‘それが何であるか’という問い合わせに対しては、それはソクラテスという個人、あるいは人間という種の二通りの回答が考えられるのです。

*³⁴ パーンズは‘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。

*³⁵ このように総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るのです。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しょう。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」なのです。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」とい、あるいは存在しないものを存在すると言ふこと」が偽で、「存在するものを存在する」とい、あるいは存在しないものを存在しないと言ふこと」が真([2]11b27)とし、具体的な対象ではなく状態を示すものとして導入しており、フレーゲの論理主義が陥る大きな陥穰(命題に外延が存在するとは限らないこと)を非常に上手く避けているのです！

*³⁶ 本質的存在(実体)の内の一つです。

*³⁷ これを図示したものがいわゆる「ポルピュリオスの樹」です。なお、「ポルピュリオスの樹」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



これらの事象について、本質的存在が最も総合的であり、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、生命のある物体の類です。生命のある物体は物体の一つの種で、動物をその種とする類なのです。また、動物は生命のある物体の一つの種で、理性的動物の類なのです。理性的動物は動物の一つの種で、人間の類なのです。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 - 単なる種なのです。

個体 (*ἄτομος*, individual) *³⁸に先行して存在する全ての事象は単なる一つの種であつ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、最も総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なので) を除いて本当に何物にも分割され得ないという種として、ただの一つの種であり、そして (個体に) 近接する (*προδεξές*, proximate) *³⁹種でしかなく、そして我々が語ったように、最も特定な事象なのです。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、二つの関係が並立することになり、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。その両端は单一の関係を持ちます。というのも総合的な事象はその下にある事象との関係を持つので、それら全てのものの類であれば、そ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

るとのことです [26].

*³⁸ 不可分のものという意味。原子 (atom) の語源です。

*³⁹ *προδεξές* (proximate) は間に中間的な種等が入らないという意味です。

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、最も特定な事象は单一の関係を持ち、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り、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、それらの種なのです。

だから彼(逍遙学派)らは最も綜合的なものをこのように区別します：それは類であつても種ではなく；それから再び；その上に他の上位の類はありません。最も特殊なものは：それが種であつても類ではなく；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するのです。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するのです(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、類や種や下位の類と語られます。

*** アガメムノーン (*Ἀγαμέμνων*, Agamemnon) *⁴⁰はアトレウス (*Ἄτρεύς*, Atreus) の子供という類、それからペロブス (*Πέλοψ*, 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なります。しかし、系図であれば、その大半は起源を遡ると单一の個人 - 要するにゼウス - になるのですが、ところ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は全ての共通の類でないどころか、ある单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ないのです- ちょうどアリストテレスが主張するように*⁴¹。(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 (*ἀρχή*, origin) になります*⁴²。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(アリストテレス)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单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 個あるので、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*⁴³。

*⁴⁰ 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、その祖父はペロブス、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。

*⁴¹ 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 種類に最高類を分類しているのです

*⁴² 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。

*⁴³ 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るとともに、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つて名前

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個数個ありますが、無限個のものではありません；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。それがプラトンが最も綜合的な事象から最も特定的な事象へ降下しようとする人に対してそこで止めて、それから特徴的な種差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忠告した理由なのです；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識がないだろうからです。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、分類したり複数性を通じて進めてゆくことが必要で、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性をまとめることが必要なのです。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に分類します。だから種で共有することによって、多くの人々は一つの人間（という種）になり、そして、特有性によって、その一つで共通の人間（という種）は個々になります - というのも特有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からです），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からです^{*44}。なぜなら、等しいもので等しいものを（嘶くことで馬をするように）、あるいはより広範なものでより狭いものを（動物で人間をするように）述定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；ところで狭いもので広範なものをではありません - あなたは人間が動物であると言つても、動物が人間であるとは言いはしないでしょう^{*45}。

種であると述定されるものは何でも、それらの事象を、必要性により、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。というのも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

---

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ので「同名異義的」になると主張しているのです。

*44 「AはBである」だからといって「BはAである」とは言えません。そして、類は種の上位概念であるために下位概念である種で類を説明することができないと述べているのです。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になります。

*45 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、人間で動物を述定することができないということです。ここでの「広範」、「狭い」は外延の大きさのことです。このようにポルピュリオスはクラスの分析を行っているのです。

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からです。最も綜合的な事象はその下の全て - 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; そして单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であると語られるのです。

ソクラテスは一つの個体であると語られ、そしてこれは白色のもので、この人は魅力的で、ソフロニスクス ( $\Sigma\omegaφoνίσκoυ$ , Sophroniscus) の息子です(ソクラテスは彼(ソフロニスクス)の一人っ子です)。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同じものが決して見られないものの集積物だからです - ソクラテス固有の特徴は決して他の個体で見付けられるものではないでしょう。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)の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 - 同じだと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 - しかし、一つのものの一部であり、それからその他の事象の間では(他の事象のではなく)全体なのです(というのもその一部分においては全体だからです)。

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。

## 11.5 種差について

種差 ( $\deltaιαφopά$ , difference) は広範で、厳密に、最も厳密にそう呼ばれるべきです。というのも、一つの事象が通常、多様性のある事象と異なると語られるのは、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いずれかで、多様な事柄によって区分されるときです - ソクラテスはプラトンと色々な側面で異なり、さらには実のところ彼自身と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、おまけに彼が類似しているというものについて色々な側面で異なります。一つの事象が厳密に多様性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、青い目であるということ、鉤鼻であるとい

うこと、傷の堅くなった瘡蓋といったものさえもです。一つの事象が最も厳密に異なると語られるのは、特定の種差によって区分されるときです- 人間は特定の種差、すなわち理性的であるということで馬と異なるようにです。

一般的に、全ての種差は、それが何かに付与されるときに、その事象を多様なものにします；とはいっても、共通で、特有な種差はそれを他の別物のようにしてしまいますが、最も特有な種差はそれを全くの別物にしてしまいます。というのも種差のあるものは物事を別物みたいにし、さらにあるものはそれらを別物にしてしまうからです。ここでそれらを別物にするものが特徴と呼ばれ、それらを別物のようにするものが単に種差と呼ばれます。そんな訳で理性的であるという種差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種差は、ただ止まっていることと比べて単に別物のようにする程度なのです；だからあるものはそれを別物に、あるものは単に別物のようにするのです。ここでそういうたがいの種差から構成されるもの- が表現されますが、ところであるものを単に別物のようにしてしまう種差のために、多様性だけが構成され、さらにそれが類似しているものの中で変化するのです。

最初から再び始めましょう、種差のあるものは離在可能で、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。離在不可能な種差のあるものはそれら自体の原因であり、またあるものは偶然です- 理性的であるということは人間それ自体が原因ですが、死すべきことも、そして知識を享受できるということもそうです；とはいえ鉤鼻であることや獅子鼻といったことは偶然で、それら自体が原因のものではありません。それら自体が原因となる種差が前もってあるときに、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；偶有的な種差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- が、別物のようにするのです。再度、それら自体が原因となる種差はそれ以上でもそれ以下であることも許容しませんが、一方で偶有的な種差は、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；だから、類も類の種差の双方も類が分類されるものであるため、それが類であるものを多かれ少なかれ述定しないのです。というのも各々の事象の説明規定を完全にする種差があるからです；そして任意の事象が存在するということは、それがひとつで同一のものなので増加も減少も許容されませんが、鉤鼻であるということや獅子鼻であるということ、あるいはある色であるということの双方は増えたり減ったりします。

三つの種差の種^{*46}が判別されていますが、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はまたそれ自体で、それからあるものは偶然のものであり、また種差のあるものはそれ自体で、それらによって我々が類を種に分割するものであり、それからあるものはそれ自体で分類された事項が明記される理由になるのです。たとえば、以下に続く与えられたものの全ては動物それ自体の種差になります - 生命があつて知覚がある、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 - 生命があつて知覚があるという種差は動物という本質的存在を構成し得るものです(なぜなら動物は生命があつて知覚のある本質的存在だからです)、ところで死すべきことと不死であることと理性的であるということと非理性的であるという種差は動物を区分する原因となる種差なのです(というのもそれらを通じて我々は類を種へと分類するからです)。しかし、これらの強く区分する原因となる類の種差が完全なもので種を構成するものであることが判ります。というのも動物は理性的であることと非理性的であることという種差で分類され、それから再び死すべきことと不死であるという種差で分類されます; それから理性的であつて死すべきものであるという種差で人間が構成されることが判り、理性的であつて不死であることから神が、それから非理性的であつて死すべきものから非理性的な動物になるのです。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種差と知覚があるということと知覚がないということの種差は最高位の事象の本質的存在を区分する原因となるものであり、生命があるということと知覚があるということの種差は、互い集められた本質的存在から、動物を生成しますが、その一方で生命があるということと知覚がないということの種差が植物を生成するのです。そして、一つの方法で取られた同じ種差は構成的で、区分される原因となるものであることが判るので、それらは全て特定的と呼ばれます; それから類の分類や定義ですこぶる便利なのがまさにそれです - 離在不可能で偶然そうなる種差でもさほど離在可能なそれでもありません。

それら(種差)を定義するときに、彼等(逍遙学派の哲学者達)はこう言います: 種差はまさにそれによって種がその類を越えるものである。というのも人間は理性的であることと死すべきことで動物を越えているのです - 動物はこれらの事象の双方ではありませんし(するとどこから種は種差を得るのでしょうか?), 真反対の種差全てをそれ(動物)が保有するという訳でもありません(するとその同じものが同時に真反対のものを持つことになってしまうでしょう); むしろ、彼らが主張するように、潜在的にそれはその下の事象の種差全てを保有しますが、顕在的に何等も保有しません。そして、この方法で非存在のものから何者も導出されることがなければ、おまけに同時に同じ事象について真反対のことが見出されることもないでしょう。

^{*46} 離在可能なものの、離在不可能なものの、それと偶有的なものの三種類です。

彼等(逍遙学派の哲学者達)はまたそれ(種差)をこのように定義します; 種差は‘それがどういったたぐいのものか’に対する回答で、種で異なる幾つかの事象を述定するものです。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、‘人間はどのようなたぐいのものか?’に対する回答で語られることであって、‘人間は何か?’に対するものではありません。人間は何かと尋ねられたときに、こう言うのが妥当です: 動物だ; ところで、彼等が‘動物はどんなたぐいのものか?’という質問を付け足すなら、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(*ὑλη*, matter)と形相(*εἶδος*, form)で構成されており、ちょうど銅像が青銅を質料、その姿を形相とするように、質料と形相に少なくとも類似した構成を持つ場合、そのようにまた共通で種としての人間は質料に対して類が、種差が形相として構成され、これら - 理性的であり死すべき存在である動物 - が全体が人間として、ちょうど、それらが銅像のそれのように取られます*47。

彼等(逍遙学派の哲学者達)はまたこういった種差というものの大枠を説明しています: 種差は同じ類の下にある事象を区分するような性質のものです- 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、これらは同じ類の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: 種差はそれによってもののそれぞれの型(type)が異なるものである。なぜなら人間と馬は彼等の類で異なりません - 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ものです。しかし理性的であるが付け加えられると双方が区分されることになります。それから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、双方が区分されることになります。

種差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つたいかなるものも種差ではなく、むしろ、それらの存在に寄与して、その対象であろうとするもの一部になるものだと彼等(逍遙学派の哲学者達)は語っています。というのも船に乗って航海するといった性分は、たとえ人間の特有性であったとしても、人間の種差ではないからです; 我々がある動物に航海する性質があり、他ではそうでないと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的存在を完全なものにするものでもそれの一部でさえもありません - むしろ、本質的存在の素質の一つでしかありません、というのも特徴であると確実に語られるそういった種差としてそれが同じ種類のものではないからです。種差は特徴なのです、だから、種差が種を多様なものにしたり、種差がそれであるべきものに含まれたりするのです。

---

*47 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、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は対立するのです。ここ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するとも言えるのです。

これで種差については十分です。

## 11.6 特有性について

彼等(逍遙学派の学者達)は特有性(*ἰδιον*, 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, たとえその全てでなかつたとしても(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つたとしても(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, ‘单体, 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). というのもの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つて - 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^{*48}; 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.

## 11.7 偶有性について

偶有性(*συμβεβηκός*, accident)^{*49}はそれら(偶有性)の基体(*ὑποκείμενον*, subject)^{*50}を壊すことなしに, 行き来する事象なのです. 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^{*51}. 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することが可能です^{*52}. 彼等(逍遙学派の学者達)はそれら(偶有性)をこう定義します: 偶有性は同じもので保有したり保有しなかつたりすることが可能なものです; あるいは

^{*48} 「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.

^{*49} *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. しかし, 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.

^{*50} 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’」([2]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.

^{*51} 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ことができるものが偶有性なのです.

^{*52} 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なのです.

は: 類でも種差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なのです。

## 11.8 共通の特徴

我々が提案した全ての事象 - 私は、類、種、種差、特有性、偶有性を意味しています - が述定されましたが、我々はそれらに対して前もってある共通で特有の特徴が何であるのかを語ることにしましょう。

それら全てに共通する点は幾つかの事象を述定するということです。ところで類は種と個体を述定し、それから種差もまたそうですが、その一方で種はそれらの下にある個体を述定し、特有性はそれらが特徴になるものの種やその種の下にある個体を、偶有性は種と個体の双方です。たとえば動物は馬や牛といった種を述定し、この馬やこの牛といったことは個体、それから非理性的であるということは馬や牛や特定のものを述定しますが、その一方で人間のような種は特定のものだけを述定し、特有性は人間や特定のものが笑うことができるということのように、カラスの種や特定のものの双方の黒色であれば、離在不可能の偶有性、人間や馬が動いているということなら、離在可能な偶有性です - しかし、第一に個体を、また、第二の説明規定では、その個体を包含する事象をとなるのです^{*53}。

## 11.9 類と種差

類と種差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; 種差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 (という種差) - は非理性的であるという事象を動物という類がするように包含しませんが、人間と神を包含し、それらは種です。

類としてある類を述定するものはまた、その下の種を述定し、そして、種差としてある種差を述定するものはそれ(種差)から構成される種を述定します。というのも動物は類なので、本質的存在と生命のあるものは類としてそれ(動物)を述定します - それからこれらの事象はまた動物の下の全ての種、個体に至るまで述定します; それから理性的であるということは種差なので、理性を用いるということは、それ(理性的であるということ)を種差として述定します - 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

^{*53} 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)」を用いるもので、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。

ではなく、理性的であるということの下にある種をまた述定することになるでしょう⁵⁴.

共通点はまた、もし類か種差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、種差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⁵⁵。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、嘶くのは馬と特定の馬共、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種差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く。)

また、類は種差を潜在的に含みます; 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⁵⁶。

そして類は類の下にある種差に先行しているもので、このことが、類の種差を除去してもその類が除去されない理由なのです。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、種差は類と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、知覚があり生命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なのです。

また、今まで語ってきたように、類は‘それは何であるか?’に対する回答で、種差は‘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)、種差になると幾つかになります(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)⁵⁷。

---

⁵⁴ ‘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’なので‘人間は動物である’と推移律が成立すると述べているのです。

⁵⁵ 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。

⁵⁶ アリストテレスは形而上学の△卷にて「類の諸性質が種差と言われる」と述べており、ここでの「潜在的」は類の諸性質としての種差のあり方を指しています。

⁵⁷ ここでの主張は、まず種に近接する類が一つ存在すること、それに対して種差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るのです。

類は物質に似ていて、種差は形相に似ています。

他の共通性や特有な事象は類や種差に先行しています - が、これらは十分ということにしておきましょう。

## 11.10 類と種

類と種は共に、今まで述べてきたように、幾つか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っています。）

(類と種の) それらの共通性はそれらを述定する事象の前にあるという事実、それと各々が全体の集まり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*58。

また、類は（種よりも）先行して存在していなければなりません、そして、特定の種差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行して存在します；そして、それらは一緒に削除しても削除されず、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*59。

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、種は種自身の種差によって類よりも一層広範です*60。

そして、種が最も総合的なものになる訳でもなく、類が最も特定的なものになることでもないでしょう*61。

---

*58 類は常に種よりも広範囲である。（トビカ [3]121b3-4.

*59 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。

*60 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種差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。

*61 ここでの「総合的」と「特定的」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。

## 11.11 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^{*62}.

類は等しくその種を述定し, それから特有性もまたその中で分有する (*μετέχειν, participate*) ^{*63}ものを述定します - 人と牛は共に動物で, アニユトスとメレトス^{*64}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 - 動物がまず先行して存在しなければならず, それからあとに種差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^{*65}.

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)一つだけではありません^{*66}.

^{*62} 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對して‘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るのです.

^{*63} 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關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идンより). つまり, 参与, 与ることや共有することとおまかに言えるでしょう. アリストテレスでは「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ができるということ」(トピカ [3] 121a11) を「与ること」の定義としています. ここではポルピ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.

^{*64} アニユトス *Ἀνύτος* 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.

^{*65} 主語と述語の關係で言えば, 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關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 になるということなのです.

^{*66} 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るのです.

さらに、もし特有性が除去されたとしてもそれら（特有性）は類と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。

## 11.12 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、それらが幾つか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^{*67}。

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、偶有性とするものがその偶有性よりも先行しているからです。

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^{*68}。

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?’に対する回答なのです。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。

我々は類が他の四つ（種、種差、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四つの事象と異なり、だから、5つの事象があれば、

*67 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。

*68 類は説明規定で語られるもので、程度が語られるものではありませんが、偶有性はその程度を語ることができます。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。

各個が他の四つと異なっているので、その違いの全ては4かけ5で20になるのです。また、それらは統々と数えられることで、第二群は一つの違いで短く、すでに判っているよう、第三群は二つ、第四群は三つ、そして第五群は四つになります；それゆえにその違いは4, 3, 2, 1 - だから10になります。類は種差、種、特有性と偶有性と異なります - だから四つの違いがあります。種差に関して、それらが種とどのように違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二つのです。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、種差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種差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四つの差異、種差では三つ、種では二つ、そして特有性(偶有性との関係で)一つになります：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。

### 11.13 種差と種

種差と種の共通点は、それらが等しく分有するという事実です：特定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種差を共有するのです。

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行し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。

種差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、むしろ種差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、種差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、種差はそれらの種に先行して存在します。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、種差は別の種差が混入しています：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驥馬をかけ合せると驥馬が生れます；ところが馬はというと、単に、驥馬を製造するために驥馬と混ぜ合わせられるという訳ではありません^{*69}。

## 11.14 種差と特有性

種差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なのです。

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のことです。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。

種差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^{*70}。

種差はそれらが種差となるものの事象に続きますが入換えが効きません^{*71}。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ので、だからそれらは入替が効くのです。

## 11.15 種差と偶有性

種差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行してあることで、そして同様に黒色であるということもそうです。

^{*69} 複数の種差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はできないということです。

^{*70} 理性的であるという種差は人間と神の双方で適用できることから判るように種差は複数の種に適用することができます。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、この点で種差と異なっているのです。

^{*71} 主語と述語の入れ替えができないということです。

それら(種差と偶有性)は異なります。なぜなら種差は包含するものの包含されません(理性的であるという種差は人間を含みます)が、ところで偶有性はある点でそれらが幾つか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幾つかの偶有性を受容するものに包含されます。

種差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^{*72}。

逆に種差は混合しませんが、その反対に偶有性は混合することがあります^{*73}。

そのような共通点であり、そしてそのような種差と他のものの固有の特徴があります。種がどのように類と種差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、種差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。

### 11.16 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。)^{*74}

種はそれらの中で分有するものの中で等しく先行してあるもので、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發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のですが、その一方で特有性は潜在的で時折のものです。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

^{*72} 種差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ないです。しかし偶有性は度合を持つので、増減を語ることができます。

^{*73} 類が互いに混合することがないと同様に種差も互いに混合することないと述べています。

^{*74} 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。

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行して存在していることです。

## 11.17 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。

種は偶有性に先行して考えられるのですが、たとえ、それらが離在不可能（その偶有性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、さらにそれらは偶發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つても濃くなつ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、種差や類と異なっているかが語られているからです。

## 11.18 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⁷⁵。

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つたりするのです⁷⁶。

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。

---

*75 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。

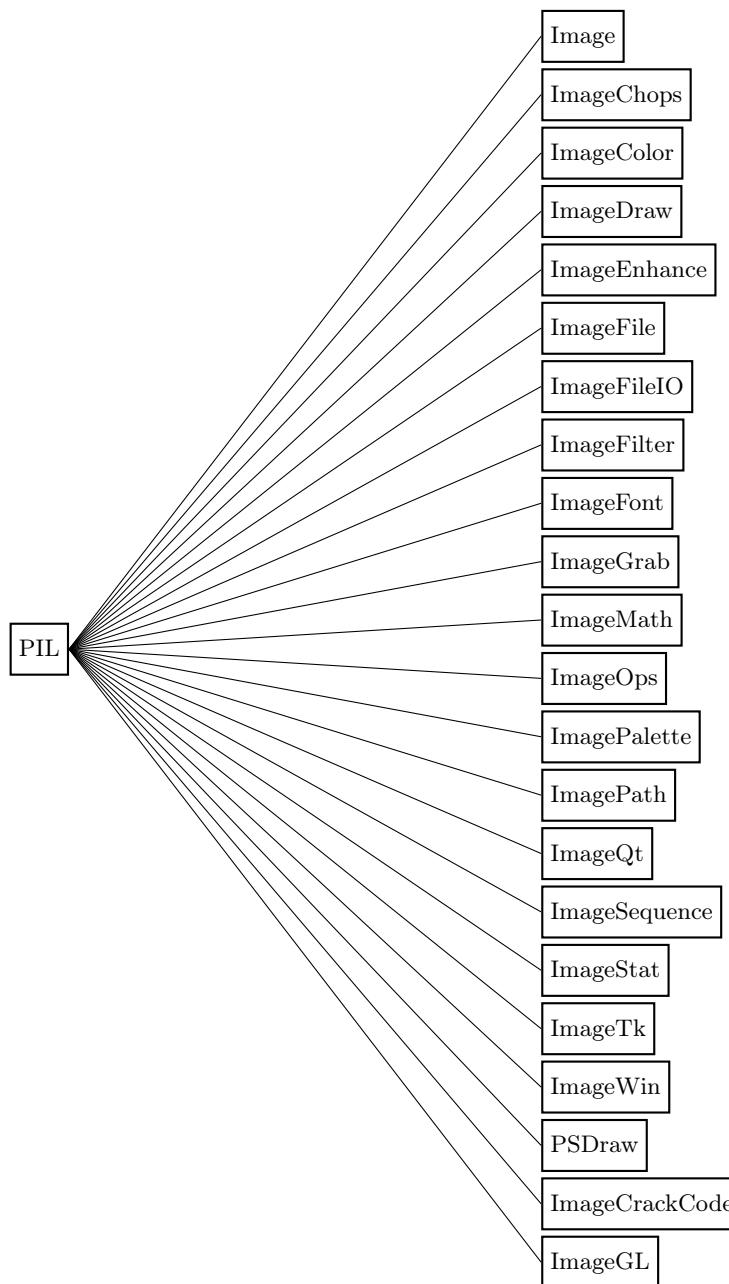
*76 特有性と種は1対1に対応し、だからこそ主語と述語の関係で、主語と述語としての入換が効くのです。ところが、離在不可能な偶有性と種については1対1の対応とならないために主語と述語の関係で入換が効かず、さらに偶有性の性格上、その程度を表現することが可能だということです。

## 第 12 章

# Python のライブラリ

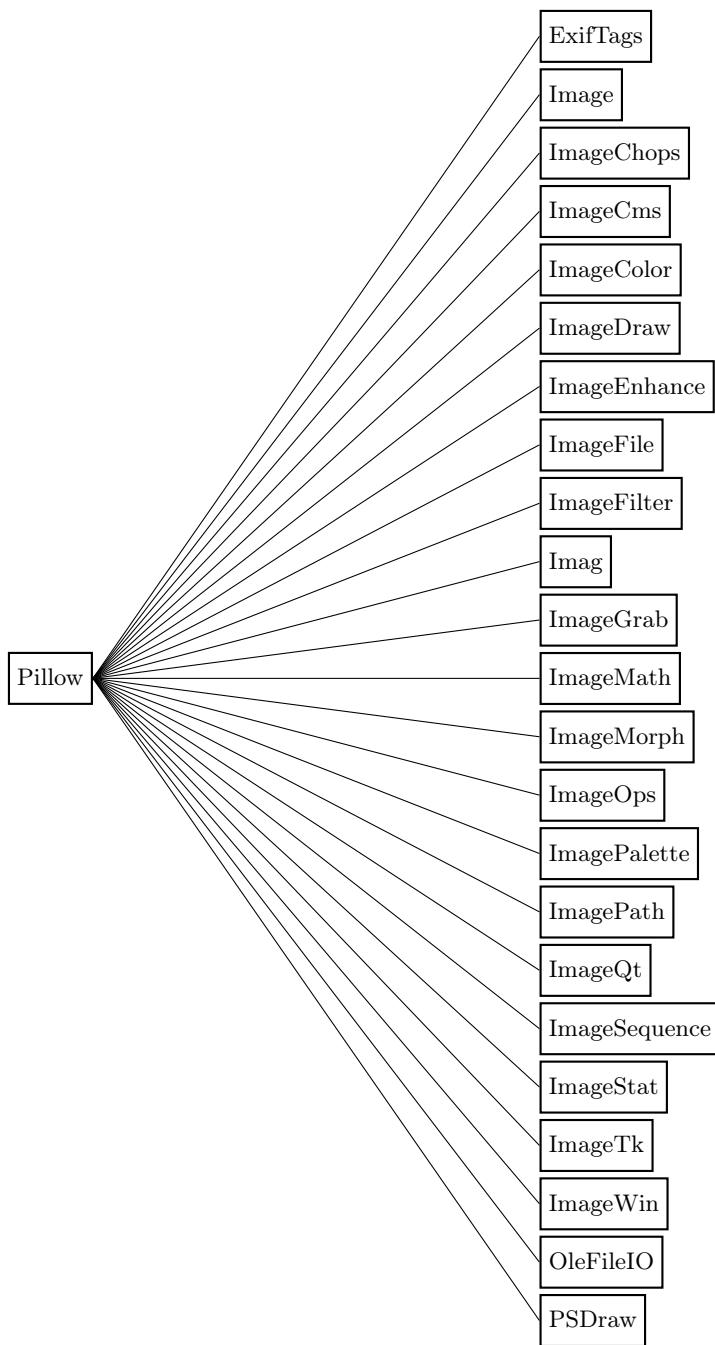
## 12.1 PIL

読み込んだ画像データは Python の instance になり、後述の Matplotlib と異なり NumPy の配列データになりません。ただし、NumPy の函数 array() を用いて NumPy の配列に変換することができます。後述の Pillow が後継で、PIL の開発は終了しています。



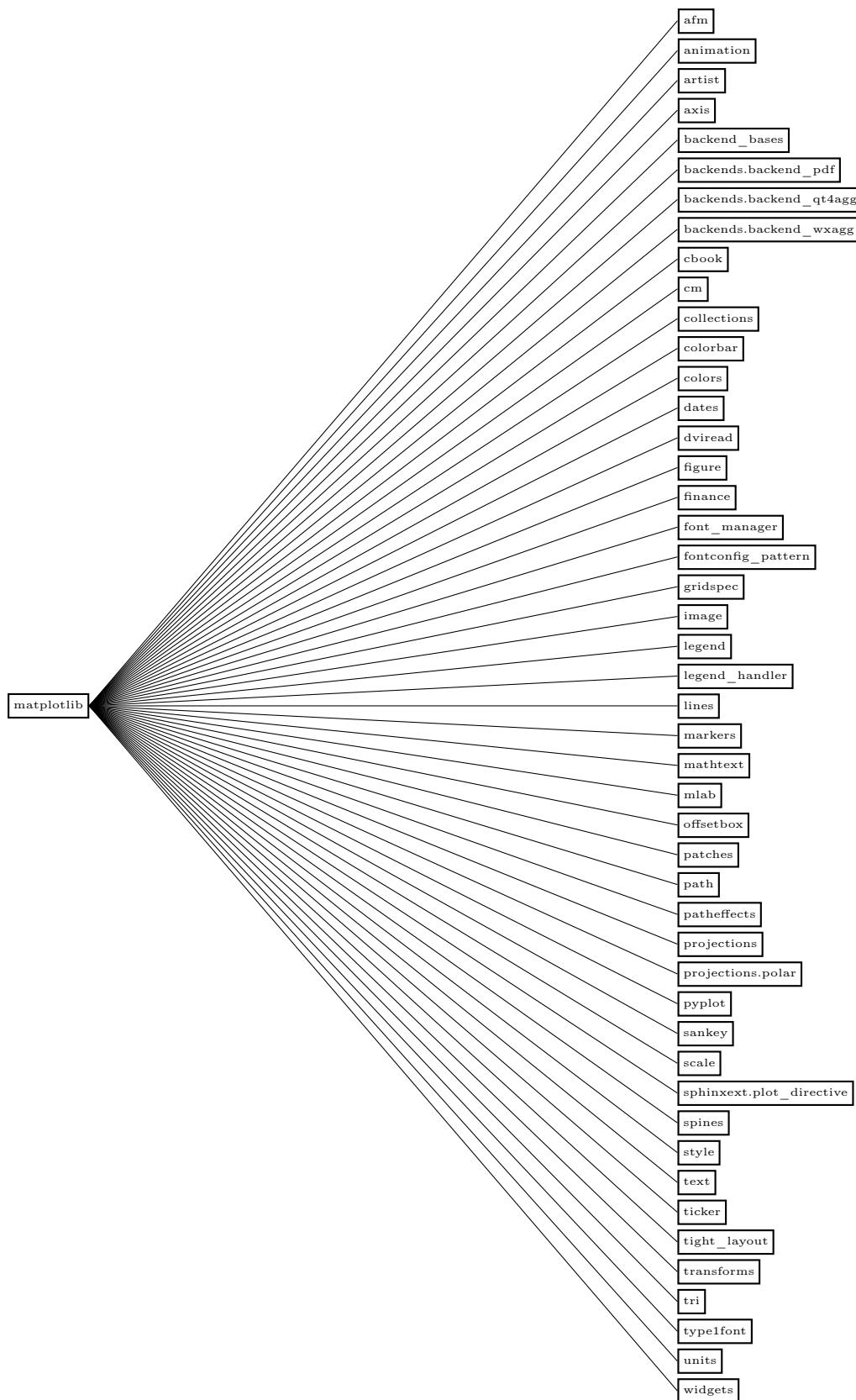
## 12.2 Pillow

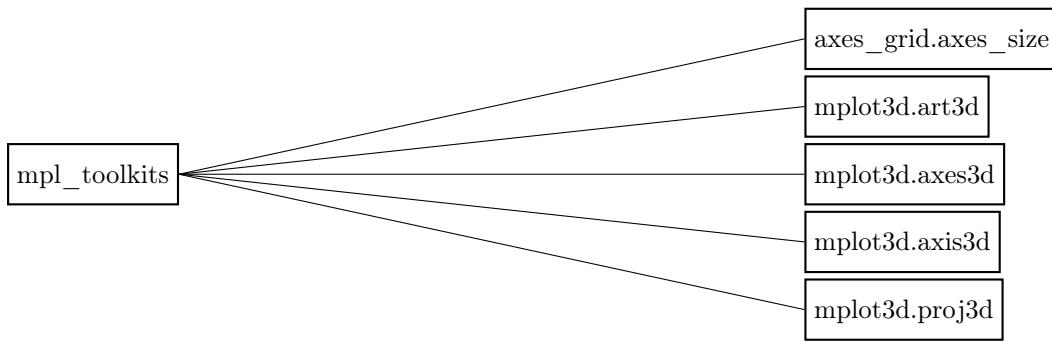
PIL のから分枝したもので、PIL が未対応である Python 3.X にも対応しています。また、Python のパッケージ管理ツールの setupinstall にも PIL と違い対応しています。



## 12.3 Matplotlib

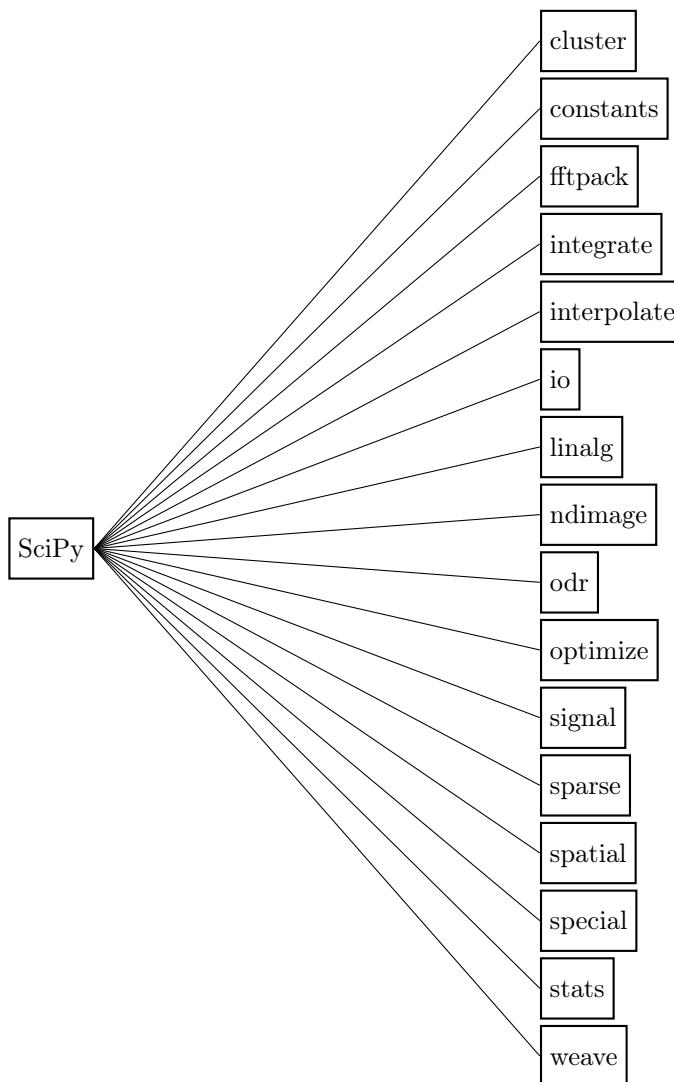
Python の配列ライブラリである NumPy を基に構成されたライブラリで、主に 2D グラフに関する処理が可能です。





## 12.4 SciPy

Python で数値計算を行うためのライブラリで、信号処理、フーリエ変換、数値積分、線形代数、行列処理、最適化や統計処理ライブラリ等を含みます。Python の数値配列処理ライブラリの NumPy はこの SciPy の開発で用いられていた Numeric ライブラリを従来の Python で数値配列の処理で用いられていた Numeric ライブラリとを統合したものです。この経緯もあって、SciPy は NumPy を基底としています。また、NumPy, SciPy, Matplotlib を併用することで、MATLAB と似た操作が可能になります。



## 参考文献

- [1] アリストテレス, アリストテレス全集 1 カテゴリー論・命題論, 岩波書店, 2013.
- [2] アリストテレス, 形而上学(上下), 岩波文庫.
- [3] アリストテレス, (旧)アリストテレス全集 2, トピカ・詭弁論駁論, 岩波書店, 1987.
- [4] 飯田隆, 言語哲学大全 I 論理と言語, 効草書房, 1987.
- [5] 井筒俊彦, イスラーム思想史, 中公文庫, 中央公論社, 1991.
- [6] 今道友信, アリストテレス, 講談社学術文庫, 2004.
- [7] 大畠明(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.
- [8] 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.
- [9] 後藤和茂, BLAS の概要 ([http://jasp.ism.ac.jp/kinou2sg/contents/RTutorial_Goto1211.pdf](http://jasp.ism.ac.jp/kinou2sg/contents/RTutorial_Goto1211.pdf)), 2006.
- [10] 柴田有, グノーシスと古代宇宙論, 効草書房, 1982.
- [11] 清水義夫, 圈論による論理学 高階論理とトポス, 東京大学出版会, 2007.
- [12] 藤野登, 論理学 -伝統的形式論理学-, 内田老鶴園, 2003
- [13] 田中尚夫, 選択公理と数学, 星雲社, 1987
- [14] 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.
- [15] プラトン(著), 藤沢令夫(訳), 国家, 岩波文庫, 岩波書店, 1976.
- [16] フレーゲ, フレーゲ著作集 1 概念記法, 効草書房, 1999.
- [17] フレーゲ, フレーゲ著作集 3 算術の基本法則, 効草書房, 2000.
- [18] 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.
- [19] 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](http://www002.upp.so-net.ne.jp/tetsu/study/t01_boepor.pdf)
- [20] M.Lynne Murphy, Ane. Koskela, 意味論キーティーム辞典, 開拓社, 2015
- [21] 山内志朗, 普遍論争, 平凡社ライブラリー, 2008
- [22] 横田博史, はじめての Maxima, I/O Books, 工学社, 2006.
- [23] 横田博史, はじめての Maxima 改訂  $\alpha$  版 (MathLibre に収録)
- [24] 横田博史, 数値計算・可視化ツール Yorick, I/O Books, 工学社, 2010.

- [25] 吉田光邦, 錬金術 - 仙術と科学の間 -, 中央公論新社, 2014.
- [26] J.Barnes, PORPHYRY INTRODUCTION,Oxford University Press, 2006.
- [27] Birman, Braid groups and mapping class groups, Ann. Math, Princeton University Press, 1963.
- [28] Boethius, Isagoge, <http://www.forumromanum.org/literature/boethius/isag.html>
- [29] G. J. Brose, MATLAB 数値解析, Ohmsha, 1998.
- [30] Goodger, reStructuredText ディレクトイズ, <http://docutils.sphinx-users.jp/docutils/docs/ref/rst/directives.html>
- [31] Mac Lane, The Category theory for working Mathematician, Springer
- [32] Mac Lane, Moerdijk, Sheaves in Geometry and Logic, A First Introduction to Topos Theory, Springer Verlag, 1992.
- [33] Porphyry, Introduction(Iagoge) to the logical Categories of Aristotle, [http://www.ccel.org/cCEL/pearse/morefathers/files/poRphyry_isagogue_01_intro.htm](http://www.ccel.org/cCEL/pearse/morefathers/files/poRphyry_isagogue_01_intro.htm)
- [34] Porphyry, Letter to Marcella, [http://www.tertullian.org/fathers/poRphyry_marcella_02_text.htm](http://www.tertullian.org/fathers/poRphyry_marcella_02_text.htm)
- [35] B.Russell, The Principles of Mathematics,W.W.Norton & Company,Inc.,1996.
- [36] B.Russell & A.N.Whitehead,Principia Mathematica to *56, Cambridge Mathematical Library,Cambridge University Press,1997.
- [37] Diving into Python: <http://www.diveintopython.net/toc/index.html>
- [38] MathWorks 日本: <http://www.mathworks.co.jp/>
- [39] アテナイの学堂 <http://ja.wikipedia.org/wiki/アテナイの学堂>