

統合数学環境 Sageへの招待

- Python で記述された統合数学環境 -

USO863 版

横田博史

平成 27 年 07 月 31 日 (金)

Wolfram *Mathematica*® は Wolfram Research の登録商標です. MATLAB®, および Simulink® は The MathWorks Inc. の登録商標です. WINDOWS® は Microsoft Corporation の登録商標です. POSTSCRIPT® は Adobe Systems Incorporated の登録商標です.

統合数学環境 Sage への招待 ©(2015) 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

Sage 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、Sage 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 Sage 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、Linux を基底に置いたシステムでありながら、仮想環境やクラウド環境へ柔軟に対応しながら機能を拡充してゆくことに顕著に現われています。ともあれ実体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 が齎しているのです。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、フルスクリ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な訳で、この Sage の手法はその面からも非常に興味深いシステムなのです。

この文書では、Sage の解説を行いますが、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 自体のことが主体になります。その意味でもこの文書は Sage を直ちに習得することには向かないでしょう。

なお、この文書はまだ下書き以前の段階で、嘘も間違いも大量に含んでいます！だから USO800 版なのです。そのために、この文書の二次配布はご遠慮願います。とはいって絶対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 27 年 6 月 5 日（金）

狸穴主人 横田博史

目次

| | |
|--|-----|
| 第 1 章 Sage について | 1 |
| 1.1 背景 | 1 |
| 1.2 Sage の使い方 | 12 |
| 1.3 Sage が目指すものは? | 21 |
| 1.4 Sage が包含するアプリケーションとライブラリ | 23 |
| 1.5 一般の Python パッケージ | 30 |
| 1.6 この本の方針 | 31 |
| 第 2 章 オブジェクト指向について | 33 |
| 2.1 Sage と Python の関係 | 33 |
| 2.2 Python はどのような言語か | 33 |
| 2.3 イデア論とオブジェクト指向プログラミング | 34 |
| 2.4 集合論について | 47 |
| 2.5 圈 (Category) | 56 |
| 2.6 トポス (Topos) | 80 |
| 第 3 章 Python について | 83 |
| 3.1 この章の目的 | 83 |
| 3.2 簡素化された構文 | 83 |
| 3.3 Backus-Naur 記法 (BNF) | 100 |
| 3.4 字句解析 | 104 |
| 3.5 データモデル | 111 |
| 3.6 名前空間とスコープ | 134 |
| 3.7 実行モデル | 135 |
| 3.8 名前付けと束縛 | 135 |
| 3.9 例外 | 136 |
| 3.10 Python の式 | 137 |
| 3.11 単純文 | 144 |

| | | |
|---------------|--|-----|
| 3.12 | 複合文 | 148 |
| 第 4 章 | Sage プログラムを書く上での指針 | 153 |
| 4.1 | はじめに | 153 |
| 4.2 | PEP(Python Enhancement Proposal)について | 153 |
| 4.3 | ファイル名やディレクトリ名に関する指針 | 160 |
| 4.4 | ライブラリに関する指針 | 160 |
| 4.5 | 文書文字列の利用について | 161 |
| 4.6 | reStructuredTextについて | 162 |
| 4.7 | Sphinx | 168 |
| 第 5 章 | 数学的对象の表現 | 169 |
| 5.1 | はじめに | 169 |
| 5.2 | 数の構成 | 170 |
| 第 6 章 | SQLite を使った解析 | 171 |
| 6.1 | SQLite 速習 | 171 |
| 6.2 | Sage から SQLite を使う | 172 |
| 第 7 章 | Sage で画像処理 | 177 |
| 7.1 | 画像の読み込み | 177 |
| 第 8 章 | Sage の拡張 | 183 |
| 8.1 | sagemath からのパッケージ入手 | 183 |
| 8.2 | 一般の Python パッケージのインストール | 183 |
| 8.3 | GNU R のパッケージのインストール | 184 |
| 第 9 章 | SageMathCloud | 185 |
| 9.1 | SageMathCloud とは | 185 |
| 9.2 | ファイルのアップロード | 187 |
| 9.3 | 端末, Jupiter, L ^A T _E X の利用について | 189 |
| 第 10 章 | 手引(ポルピュリオス) | 193 |
| 10.1 | 概要 | 193 |
| 10.2 | はじめに | 197 |
| 10.3 | 類について | 198 |
| 10.4 | 種について | 200 |
| 10.5 | 種差について | 205 |

| | | |
|---------------|----------------------|------------|
| 10.6 | 特有性について | 208 |
| 10.7 | 偶有性について | 209 |
| 10.8 | 共通の特徴 | 209 |
| 10.9 | 類と種差 | 210 |
| 10.10 | 類と種 | 211 |
| 10.11 | 類と特有性 | 212 |
| 10.12 | 類と偶有性 | 213 |
| 10.13 | 種差と種 | 214 |
| 10.14 | 種差と特有性 | 214 |
| 10.15 | 種差と偶有性 | 215 |
| 10.16 | 種と特有性 | 215 |
| 10.17 | 種と偶有性 | 216 |
| 10.18 | 特有性と偶有性 | 217 |
| 第 11 章 | Python のライブラリ | 219 |
| 11.1 | PIL | 219 |
| 11.2 | Pillow | 220 |
| 11.3 | Matplotlib | 222 |
| 11.4 | SciPy | 224 |
| 参考文献 | | 227 |
| 索引 | | 229 |

第1章

Sageについて

1.1 背景

1.1.1 車輪の再発明はしない

Sage 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。まず、Sage 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。のために Sage の開発で採用した手法・手段が今迄存在した OSS のシステムが採用した方法と比較して非常にユニークであることに尽きます。実際、Sage の開発者の「**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してゆくのではなく、それどころか既存の優れたソフトウェアを取り込むことで必要とされる機能を実現するという手法なのです。このような開発手法が可能である背景には研究目的のために開発された高機能の OSS のアプリケーションが多数存在していることに加え、それらを動作させるための環境にもさまざまな意味で余裕があるという二つの事実があります。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何らかの研究の成果として一般に公開されたもので、それらは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、非常に限られた用途や利用者を対象とすることが多いために処理言語やデータ構造が独特なものになり易く、その結果、専門家以外、場合によっては専門家にとっても使い難いものが多く、逆に GUI や使い勝手が良かったとしても、今度は特定の専門分野に限定されているのために、入力データの作成に専門的な知識、習慣等が必要とされるために誰にでも使えるというものではないということが現状です。とはいえ、扱っている事項が他の分野に全く使えないという本質的に特殊なものばかりではなく、他の処理に転用の効くものが実際は多くあります。

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、さらに数学的構造等のデータ構造を 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は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 上で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としての Python 言語と対象が Python でどのように表現されているかといったこと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、それを Python でどのように処理すればよいかが理解できていれば、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 Sage のやり方が十分に実用的になった背景として、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事実があります。実際、2015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ね。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ラーやコンパイラ言語を利用する等の工夫や調整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は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 Sage 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、実は近年の商用の 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 年代前半にかけてですが、その当時のいわゆる「西側諸国」では、サッチャリズムで代表される新自由主義の影響を受けた政策によって、国公立の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す。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、販売や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ることになりました。こうして研究機関から出て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、やがて市場も成熟してゆきます。このように市場が成熟してゆくにつれて研究機関を

^{*1} だからといって高速処理への要求が無くなるという訳ではありません。あくまでもコストか、それとも所要時間に耐えられるどうかという選択なのです。

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を中心とした利用者から、より単純化された手続に従った作業を中心に行うオペレーター的な利用者へと比重が変化してゆき、その結果、非専門家でも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、このようなことで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って、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。

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も出てきます。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 の導入段階では計算機の処理能力の問題もあって 2次元 CAD が主流であり、それも製図手段が計算機で置き換えられた程度であり、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では、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、それから解析を行っていました。しかし、この中間段階の紙は絶対的に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み取り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次元が中心だった CAD も計算機の処理能力の向上とともに 3次元 CAD が主流になります。こうなると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み取り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に複雑なものになります。さらに計算機の処理能力も、そのような複雑な形状であっても処理ができる程の能力を持っており、計算機の能力の低さから 2次元モデルに限定する理由が希薄になっているのです。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可能となるのです。

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のでどうやって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、もしも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み取り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、紙ベースであればその図面から座標の読み取りといった処理を行ってモデルの修正を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大本を変更を変更すれば共通するものであれば自動的に更新されるでしょうし、そうでなくと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、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。このようになると最早、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あり、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 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。そのはじめの段階では 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いました。

^{*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。

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 を導入した時点では各機能の GUI に統一感がないどころか、その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って、その結果、従来と異なる結果を出して信頼性を落としてしまう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ていること、言語的にも C に似ている点、オブジェクト指向の言語であり、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。

一方の Python は言語仕様が比較的単純なために学習し易くて生産性が高い言語になっています。また、Python 本体を簡易なものとし、その代りに拡張性を高めており、このことから Python の高い拡張性と対話処理が可能な点を利用すること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この処理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となるアプリケーションやライブラリが現われればそれを用いれば良いのです。つまる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多々ある訳ですが）。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走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に加え、オブジェクト指向言語であることから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。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が現状なのです。

1.1.4 Multi-paradigm 言語

この Python にも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「**multi-paradigm 言語**」と呼ばれる多様なプログラム様式を許容する言語となっています。

たとえば、伝統的なオブジェクト指向の考え方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は使えません。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）が定められますが、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つもので、メソッドは集合内の元に対して許容された「**演算/処理**」に相当することになります。

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、まず、その数式が所属するオブジェクトを定義するクラスを定め、そのオブジェクトを実体化したものとしてインスタンスを生成することになりますが、一般的な函数を定義するのではなく、多項式という類の元に対してのみ因子分解を行うという処理は、多項式という類に含まれる元に対する処理として定義されなければならないことになります。この類の元に対する処理が「メソッド」なので、多項式という類に属さないデータには当然、メソッドが使えないことになります。

ところでメソッドも、式、つまり、オブジェクトがどのクラスに属するものかで動作

が異なることもあります^{*4}. 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はこれ以上は分解できませんが、複素係数環であれば分解できて $(x - i)(x + i)$ になりますね。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ということになります。しかし、クラスの実装方法によっては、その多項式が属する多項式環を生成しなければ、その多項式が定義できないこともあります、下手すれば多項式の展開を行うためだけに煩雑な手順が増えてしまうこともあるのです。

このような処理に慣れてしまえば「あたりまえ」のことになるでしょうが、単純に「整数係数の多項式の因子分解を素早く行いたい」だけであれば、利用者にとっては本筋から外れた作業になるのです。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、大きなものとなるでしょう。こういったことを真面目にやっていく数式処理ソフトに 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」になってしまふ傾向があります。このことは多項式の展開を覚えたての中学生にさえも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障壁(?)が加わることになり兼ねません。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」で呼び出さねばならないということは煩雑なことでしょう。

ところが Python は前述のように多重模範 (multi-paradigm) 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利用することができます。そのためにいきなり多項式を入力して因子分解といったことができるのです。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」と覚えなくとも「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。この点は一般の利用者にとって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
```

^{*4} メソッドのオーバーライドによるものです。

ここで式の因子分解ではメソッドを利用して ‘a.factor()’ で分解したり, フункци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でてはいません。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, Sage が立ち上がった時点で変数 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。

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の開発をする段階になると顕著になります。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しましょう。この場合は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, 実数の四則演算を活用して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方を取り入れている Python を基底に用いている Sage 上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するのです。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, この抽象化を行うことによって, 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となり, プログラムに普遍性を持たせ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, 様々な類似の問題への適用が可能となることが最大の長所であると言えるでしょう。

1.1.5 電池込みだよ

Sage は Python で記述された数式処理のためのライブラリ SymPy をその基底として, 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通常は見えません。また, 必要に応じてこれらの専門ツールだけを表に出でて使えるようになっています, その意味でも, Sage は独自のシステムというより

はむしろ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。

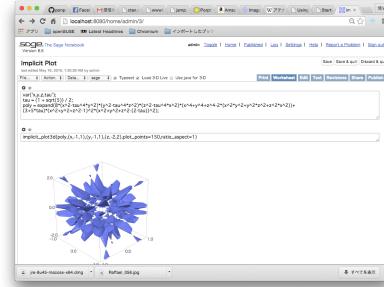


図 1.1 ウエブ ブラウザを用いた Sage のノートブック

このように Sage は開発の時点で使えるものを利用する方針があり、フロントエンドにて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も可能にし、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)」^{*5}を彷彿させるものでもあります。

1.1.6 Sage が使える環境は？

Sage 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 Sage 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 Sage 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、また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 Sage のような非常に複合的なシステムではコンパイルを含めてインストールを行うのも大変な作業となり、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は言い難くなります。そこで Sage の開発者が採用した面白い点は、Sage が必要

^{*5} 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。

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は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 Sage 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らなくなっているのが現状です。

この Sage 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、Sage のコンパイルは比較的重い処理なので、ディスクやメモリの容量や貴方に十分な時間がないのであればバイナリ版の入手を勧めます。そうでなくて暇を持て余しているとか、自炊が趣味といった方であればソースを入手されると良いでしょう。なお、OSX 版には前述のように UNIX 環境に対応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 版を素直に利用する方が良いのではないかと私は思います。その理由は Sage.app を起動すると Finder 上に Sage 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 Sage のノートブック形式や仮想端末上の CUI 形式の Sage、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ます：

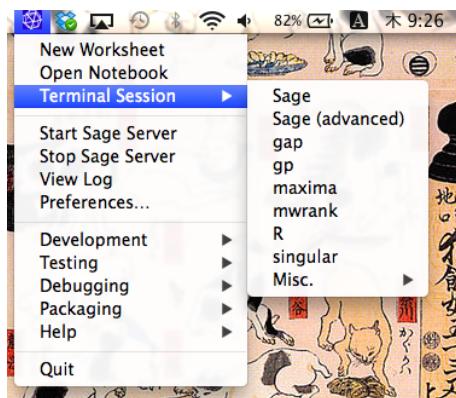


図 1.2 Sage.app のメニュー

と、このように Sage.App 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 Sage を導入すべきなのです。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管理が主要な Linux ディストリビューションのリポジトリと比べて格段に優れている訳ではありません。個人的な感想になりますが、現状は 2000 年頃に Old-World な Mac に Linux のディストリビューションの一つである SuSE Linux をインストールして、その SuSE Linux 上でパッケージ管理を行っていた頃に近い状態に思えます。その当時、私が使っていた Mac は PowerPC であったために Intel 版程のアプリケーションがリポジトリになく、何かと自力でアプリケーションのコンパイルを行っていたからです。そうすると微妙なライブラリ等の整合性の問題が生じ易くなりますが、現時点の OSX で Unix 版のアプリケーションを導入するとそういう問題が出易いように思えます。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 Sage 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からです⁶。

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、いつのこと MathLibre⁷ 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、Sage 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。Sage 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⁸。

1.1.7 Sage の情報

Sage 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。ただし、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

⁶ 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場合、Sage 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 Sage がバージョンアップした場合、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、多少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 Sage 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方向に持ってゆくのが妥当なところでしょう。

⁷ 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。

⁸ とは言え、Sage 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。

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^{*9}. この文書は MathLibre(KNOPPIX/Math) の DVD にも収録されていますが, Sage 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. ただ, この文書は Python について幾らかの知識を要求しています. 実際, Sage 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. 様々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.

1.2 Sage 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 Sage 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. 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. | |
| Type "help()" for help. | |

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

この例では `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 では冪乗の演算子は “`**`” で, 演算子 “`^`” はビット単位の演算子の排他的論理和の XOR に相当しますが, Sage では演算子 “`^`” は冪乗の演算子として扱われていることに注意して下さい. そして ‘`sage:`’ が Sage の入力行であることを示すプロンプトで, ここに続け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 がフロントエンドとなっています. そして, 入力

^{*9} 筆者のサイトから辿って入手することができます.

式の評価は [Enter] キーで行います。だから数式処理関連のさまざまなライブラリが含まれている Python として使えばよいのです。さて、ここでの例でも判るように Sage には Maxima のように入力行の末尾であることを示す記号はありません。Maxima では行末尾に記号 “;” を入れなければなりませんが、Python 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り、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。そこで割当てられた値の確認は上の例のように変数名を入力するか、函数 print() を用いるのも一手でしょう。

さて、Sage 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？Sage で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。Sage 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.
```

ここでは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、およびメソッドの補完が 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 UI として用いる場合でも同様に表示が行われます。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るのです。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 Sage の UI 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空間に含まれる名前を補完したり、候補が複数存在する場合はその候補を列記するものです。ここでやろうとしていることは名前 p1 に束縛された多項式の展開です。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もので、処理をしようとしている式は整数係数の多項式なので ‘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
```

この式の展開ではオブジェクトの情報だけで式の展開が行えるので, ‘p1.expand()’ で展開することができます。ところで, Python は Multi-paradigm 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もできます。つまり、メソッドを通常の函数として用いることができます。だから, ‘expand(p1)’ と記述することもできます。この函数風に処理する場合でも、TAB を使った補完機能は有効です。このように他の Maxima や Mathematica といった数式処理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 Sage 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「**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いませんね。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’ には当然、何も束縛されていない状態であるために未定義とされていることが ‘NameError:’ に続く「name ’y’ が未定義」という文言からも判ります。ここで「NameError」や「name ’y’」という表記ですが、これは Python の**名前空間 (name space)** が関係しています。つまり Python では名前でオブジェクトの参照を行う方法になっており、この名前から構成で構成された集まりのことを**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するものです。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 Sage は Multi-Paradigm 言語だからといって、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 Sage 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、前のようなエラーは出なくなります。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 Sage 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、Sage のリストの生成では使い易

いように拡張されたものとなっており、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 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**と呼ばれる処理がおこなえます。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になります。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することができます。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。Sage では代数方程式を函数 solve() で解くことができます:

```
sage: solve(x-123,x)
[x == 123]
sage: solve(x^2-3*x+1 == 0,x)
[x == -1/2*sqrt(5) + 3/2, x == 1/2*sqrt(5) + 3/2]
sage: var(['y','z'])
(y, z)
sage: solve([2*x - y + z == 0, x^2-y^2+z^2-1 == 0,x^3-z^2+2*y-1 == 0],[x,
y,z])
[[x == (0.0255946656987 - 1.63139339042*I), y == (0.0295897155531 -
2.19244726264*I), z == (-0.0215996158442 + 1.0703395182*I)], [x
== (0.0255946656987 + 1.63139339042*I), y == (0.0295897155531 +
2.19244726264*I), z == (-0.0215996158442 - 1.0703395182*I)], [x
== 0.788032678295, y == 0.667795080117, z == -0.908270133622], [
x == (-0.138360986224 - 0.103194243068*I), y == (0.988075254834
```

```
- 0.994925122194*I), z == (1.26479722728 - 0.788536636057*I)], [
x == (-0.138360986224 + 0.103194243068*I), y == (0.988075254834
+ 0.994925122194*I), z == (1.26479722728 + 0.788536636057*I)]]
```

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 と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^{*10} 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、Python 向けの shell である IPython^{*11}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() を調べる場合は `?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 の機能のために通常の Python のシェルでは使えません。

1.2.2 ノートブック形式のユーザインターフェイス

Sage 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 が用いられています。そのためおおよその操作は IPython Notebook に準じることになります。なお、後述の 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 Python に依存する箇所を外した後継の Jupyter が用いられています^{*12}。

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 Sage を起動させているのであれば

^{*10} 整数係数の多項式の零点となる数です。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。

^{*11} 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。

^{*12} 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 と似た操作になります。

`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 Sage の Notebook が表示されます。貴方が MathLibre 上で Sage を使おうとしているのであれば \sqrt{Math} メニュー や Launcher から Sage を選択すればノートブック形式の Sage が立ち上がり、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、ノートブック形式の Sage が立ち上がります。また、Finder 上には Sage のアイコンメニューが現われるので、そこからノートブック形式で Sage を開いたり、仮想端末から Sage を起動することもできます。

なおノートブック形式で Sage 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、それから `New Worksheet` ボタンを押すとノートブック名を設定する入力ウィンドウが表示され、このウィンドウを閉じると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できます。もちろん、PC だけではなく Android 携帯、iPhone や iPad でも利用できますが、3 次元グラフ表示は Java が使えなければならないので Android や iPhone/iPad での機能は使えません^{*13}。こちらの初回の利用も利用者登録が必要ですが通常の利用では無課金です。この SageMathCloud の利用の詳細については §9 を参照して下さい。

さて、Sage 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公開したり、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。Sage 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 §7 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ます。さて、Matplotlib を利用することにしましたが、ここで Sage 上でライブラリの読み込みを行う必要があります。Sage 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

^{*13} 3 次元グラフでは Java アプリケーションが用いられているためです。なお、この JMol が JavaScript に移植されたので将来は Java が不要になるかもしれません。

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 Sage 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っており、Matplotlib の直下に pyplot と image というモジュールがあります^{*14}。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’ から親の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()’ としていますね。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()’ としなければならないところを ‘matplotlib.image’ を ‘mi’ と読み替えて ‘mi.imread()’ とできるのです。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えなのです。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が、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るのです。そしてこの数値配列の具体的な形（大きさ）はメソッド shape() で調べることができますが、ここでは '(509, 800, 3)' と返却されていますね。このことから本来の画像は 509×800 の大きさの RGB 画像であることが判ります。というのも最初の二つの整数値が画像の縦と横の画素数で、最後の ‘3’ が Red, Green, Blue の RGB に対応する数値配列であることを示しているのです。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

^{*14} この階層はクラスの継承関係に基づくものもあります。

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 1.3 に示しておきましょう：

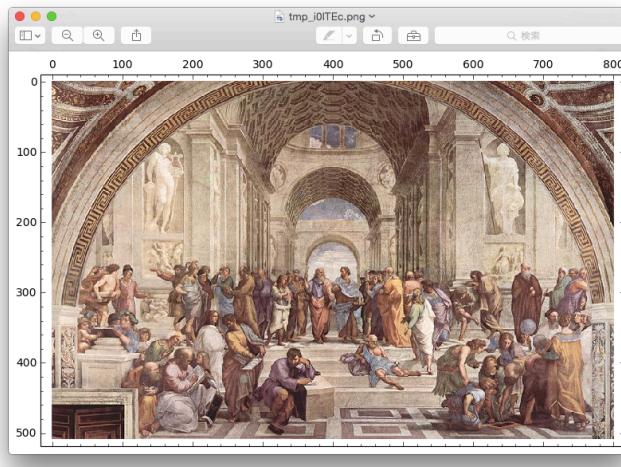


図 1.3 読込画像の表示例

ここまで処理は Sage でなくとも 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 モジュールの `save()` を使って画像を保存してしまえば、あとは Sage 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：

と、このように表示されるのです。さらに、Sage のノートブック形式の UI で処理した結果はワークシートとして保存することができますし、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 Sage で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いに、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 Sage を導入しないという理由は皆無なのです。

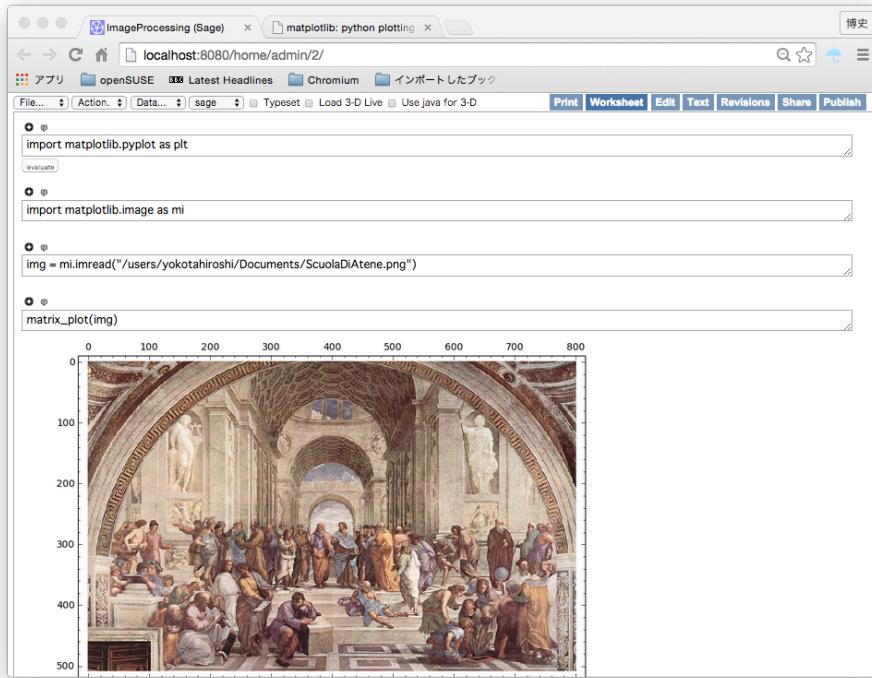


図 1.4 ノートブックへの画像の表示例

1.3 Sage が目指すものは？

まず Sage 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、Sage 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 Sage 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, UbuntuCentOS, 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、同じ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によって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。このことはさまざまなライブラリの参照を行うアプリケーションをコンパイルした方ならしばしば体験されたことだと思いますが、ライブラリ間の依存関係でアプリケーションのコンパイルに失敗したり、ライブラリやディレクトリ構成の些細な変更が原因で一部の機能がまとま

動作しないといったことが生じるのです。現在の Linux ディストリビューションの多くは (MS-Windows 以上にまともな) パッケージ管理が行われていて、そこでシステムやパッケージの自動更新が行われていたりしていますが、こういった点が Sage のように巨大なシステムになればなる程、ライブラリやアプリケーションの依存関係の管理という点では問題が生じ易くなるのです。

そこで Sage の開発者が採用した方法は、Sage のコンパイル、あるいは動作に必要なライブラリや実行ファイルといったものを全部一纏めにして配布するということです。こうしてしまえばライブラリ等の整合性の問題に踏み込まずに済むことになります。これは決定的と言える程の手法ですが、その代わりに利用者は大きな Sage のパッケージを入手しなければなりません。しかし、この制約はブロードバンド環境が一般化した現在では、さほどに困難なことではなくなっているのが実情です。実際、Sage のソースが 700MB と CD-ROM 一枚分になりますが、通常の映画 DVD の ISO イメージが 4.7GB 程度となるネット経由の DVD レンタルを考えると大きな問題ではないでしょう。また、多少時間がかかるって良いのであれば入手に Torrent を使う方法もあるのです¹⁵。

さて、Sage には必要なアプリケーション一切合切を取り込んだパッケージであるということは判りました。では Sage を構成しているパッケージには何があるのでしょうか？これらのパッケージを分析するだけでも、Sage がシステムとしてどのような趣向があるかが判別できるというものです。

まず、Sage は何よりも数式処理システムですが、表は IPython をシェル¹⁶とし、数式自体の基本的な定義は SymPy が受け持っています。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 Sage 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 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処理言語でさえも、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 の拡張は各種のライブラリ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 上で行うためのライブラリがあるので基本的な数式の定義といったことは SymPy に任せています。

それから Sage に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 については、ここ最近、関連書籍も多く出版されている統計処理では標準的なア

¹⁵ 昔話になりますが、1996 年頃はブロードバンドも一般的でなかったために、OLD-World Mac 上で動作する Linux である MkLinux の新しい開発版が公開されるとパッケージをダウンロードした CD-ROM を仲間内で郵送することで OS をインストールしたものです。それを考えると今は夢のような時代なのです。

¹⁶ 利用者とプログラム本体との間を取り持つプログラムのことです。CUI によるユーザインターフェイスと思って良いでしょう。

プリケーションで、Sage に包含されているということは高度な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、Sage に包含された 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？たとえば、Sage に含まれている Maxima にも一応 LAPACK ライブラリが存在しますが、こちらは Common Lisp のコードに LAPACK の函数を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ものではありません。それに対して Sage の数値計算は遙かに本格的です。Sage に付属の線形代数ライブラリには netlib 版の BLAS, LAPACK に加え ATLAS があります。まず、BLAS や LAPACK は線形代数の諸問題を効率的に解くためのライブラリで netlib で公開されていますが、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。それに対して 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^{*17}。そして、これらのライブラリを Python で使うための Python ライブラリとして NumPy もあります。

さらに Sage 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、Sage 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ます。

1.4 Sage が包含するアプリケーションとライブラリ

Sage は様々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 Sage が成り立っているかを知つていれば、Sage を利用するにあたって「**車輪の再発明**」を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、ここでは Sage 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おきます。

^{*17} とは言え、商用の LAPACK(Intel の MKL や AMD の ACML) や GotoBLAS と比べるとやや劣ります。

1.4.1 Sage の中核としての Python

Sage 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 系がありますが Sage は 2.X 系で構築されています。ちなみに 3.X 系では 2.X にある「古典的クラス型」が廃止されといった点や、他にも細かな違いがあつて上位互換性があるとも言えません^{*18}。そしてこの Python という言語は Sage 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 Sage との対話処理を行うためのシェル^{*19}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なります。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 というものがあり、こちらは SageMath Cloud にて GNU R, Julia や Bash を利用するために用意されています。なお、SageMath Cloud 上の Sage のフロントエンドは通常 IPython Notebook になっていますが、UI のデザインは正規の Sage のノートブックよりも洗練されたものになっています。この SageMathCloud については §9 を参照して下さい。

つぎに 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の SymPy が Sage では用いられています。Sage ではこの SymPy を数式処理の骨格として用います。つまり、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 SymPy であらかじめ定義されているので、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用いているのです。なお、この SymPy は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ないライブラリもあります。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である NumPy があります。この NumPy を基に科学技術計算を強化した SciPy では、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。また、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 があります。そして、NumPy を中核とした SciPy や Matplotlib の組合せによって Python で MATLAB 並の数値配列処理と可視化を可能にしています。一般的に数式処理は一般的に数値計算や数値配列の計算で MATLAB のよう

*18 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。

*19 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。

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、Sage では 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数値配列の計算できえ MATLAB と同水準の処理を可能にしているのです。

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。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ので、この PIL から分枝した Pillow が 2.X でも広く用いられています。ただし、MATLAB 上で行うように画像を数値配列として読み込んで処理を行うのであれば上述の Matplotlib を使うことになります。そして、PIL で読み込んだ画像データを NumPy の数値配列に変換することも Matplotlib で行うことができます。

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 を用いなくても Sage 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。このことは後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 Python で同機能のライブラリを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。

その他の 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。**PyCRYPTO** 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るものです。

mpmath は多倍長浮動小数点数演算ライブラリです。**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 Sage ではグラフ描画で用いられています。

Sphinx は文書整形ツールで、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、Sage では文書生成ツールとして用いられます。ちなみに Sage 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います。なお、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 ブラウザを利用した UI の構築で用いられています。この MoinMoin は 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。

1.4.2 Python 以外の処理言語

Sage 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。逆に言えばそういったアプリケーションやライブラリー式が一纏めになっており、Python を介さなくとも単体で使うこともできるのです。ただし、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、Sage に含まれて Sage 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設定が行われた Bash が起動されるので、その環境で C や FORTRAN 等の利用も行えるのです。

1.4.3 数式処理と統計に関連するアプリケーション

Sage 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である Maxima を併用することで数式処理システムとしての Sage 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、汎用の数式処理であ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 Sage 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 が利用できます。まず、PARI/GP は数論向けに優れたライブラリを持っていて、ちなみに数式処理システム RISA/Asir も Sage と同様に数論函数等で PARI ライブラリを利用してしています。そして、GAP は莫大な群のデータを持っています。また、Maxima は 1960 年代に開発が進められた古参の数式処理システムのために近年、応用が進んでいる Gröbner 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。そのために可換環専門の数式処理の Singular

を用います。また、統計については GNU R を包含しています。この GNU R には莫大なパッケージが CRAN^{*20}で公開されていますが、Sage 付属の GNU R にはそ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^{*21}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。この Tachyon には 3 次元分子可視化プログラムの VMD(Visual Molecular Dynamics) が組込まれています。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 Sage 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 Sage 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 1.5 に示しておきますが、メソッド show() によって Sage 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を使って、Sage のノートブックを利用すればノートブックに画像が表示されます：

このように Sage 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 に組まれているのです。このように Python さえ知ってい

^{*20} <http://cran.r-project.org/>

^{*21} <http://www.okada.jp.org/RWiki/>

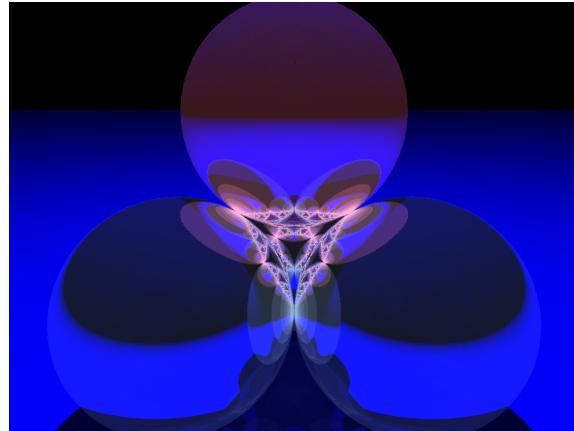


図 1.5 Sage から Tachyon を利用

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 Sage の大きな特徴なのです。

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を扱うことができます。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**^{*22}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)**」^{*23}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 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[8]。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

^{*22} <http://www.netlib.org/blas/>

^{*23} ATLAS と GotoBLAS2 はに BSDL で配布されています。

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 [8]。さて、Sage にはこの中で ATLAS が組込まれています。もし、貴方が Sage 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、乱数生成、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)、**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 との併用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リで、 $\mathbf{GP}(q)$ 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 関連のライブラリとなっています。**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。**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** は整数の素因子分解向けの 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、その利用で他の DB 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るのです。Sage で SQLite を利用するためには sqlite3 ライブラリを用いることになります。この SQLite の利用については §6 を参照して下さい。 **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る命令です。 **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

Sage 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 Sage 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 Sage 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 Sage 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 Sage 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 Sage がどのように使えるかを試すことだけでは物足らないし、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 Sage の利用者を次の四種類に分類しています：

Sage 利用者の分類

| | |
|-----------|-------------------------------------|
| Sage 習熟者 | Python も Sage もよく判っている |
| Sage 知見者 | Python は知っているが Sage を少し齧った程度 |
| SAGE 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、SAGE を使うためには 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、この本では単純に「**SAGE を使ってみたいという人**」を対象にし、Sage 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 Sage 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 Sage 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 Sage 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 Sage 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

2.1 Sage と Python の関係

Sage は既存の数学に関連するアプリケーションを Python を使って繋ぎ合せて創り上げた数学のための統合環境です。そのために Sage を新手の数式処理アプリケーションと考えるよりも、さまざまなアプリケーションやライブラリにアクセスが可能な Python 環境と考える方がより自然です。このことから数学に関するアプリケーションを使う必要があり、Python が自在に使える方にとって Sage を使わないという理由は考えられず、むしろ、是が非でも本格的な数式処理ができる Python 環境と思って徹底的に使い倒すべきシステムなのです。だから Sage を使いこなすためには Python がどのような言語であるかを知っ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 で検索したり、Wikipedia を調べればおよそのことが判ります。たとえば Wikipedia で調べてみると、オランダ人の **Guido van Rossum** が作った OSS (Open Source Software) のプログラム言語で、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**」への言及があります。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は必要最小限に抑えていることと大規模な標準ライブラリがあるといったことが書いてあります。

しかし、Python の特徴はその機能や拡張性、さらには言語的な仕様に留まらない一つの文化的な側面もあります。具体的には Python の開発ではコミュニティの存在が前提となっており、そこでの議論を反映した「**PEP**」と呼ばれる文書を基に開発が進められてゆく点が挙げられます。なお、Sage は Python の上に立脚したプロジェクトであるために Python の影響を受けますが、Sage にはまた Sage の流儀があり、これらの PEP に厳密に

従っている訳ではありません。

そこで、オブジェクト指向プログラミングとはどのようなものであり、Python の言語としての解説と Python や Sage の文化的な側面について解説することにします。そこで手始めにオブジェクト指向プログラミングがどのようなものかを述べることにします。

2.3 イデア論とオブジェクト指向プログラミング

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一般的には言われます。さて、ここでオブジェクト指向プログラミングというものがどのようなものなのかという疑問があるでしょう。ここでは脇道に大きく逸れて説明しましょう。このこと説明するために術学的にプラトン (Πλάτων, Plato) の「イデア論 (Theory of Forms)」が引き合いに出されることが多々あります。

このプラトンのイデア論によると実世界にあり、我々が考察の対象とする「個体」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存在する「イデア (ἰδέα, idea)」が対応し、現世の対象はその対象に対応するイデアの像であるということになります。だから、貴方のそばに居る三毛猫の「みけ」は、「みけ」に対応する「三毛猫のイデア」が「思惟によって知られる世界」、すなわち「イデア界」に存在し、その「みけに対応するイデア」を現世に投影することで得られた像が貴方のそばに居る「みけ」であるという主張なのです。なお、プラトンの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(属性)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「鑄型」のような役割をしていると言えるでしょう。

さて、このことをオブジェクト指向プログラミングの話に戻しましょう。ここで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」であるとして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。

さて、計算機であれば「鑄型」を使ってインスタンスとしてデータを生成すれば良いだけですが、「イデア論」でイデアの実体化の原因が何かと問われると途端にプラトンは不明瞭になります。イデアがある種の鑄型として存在することは認めたとしても、どういった理由から実体化するかと言えば、デミウルゴス (δημιουργός, demiurge) がイデアを模倣して世界を創成したので、この世の物は「模倣物 (εἰκών)」であり、模倣の理由は貧欲

な神エロース (*Ἐρως*) がイデアの美に憧れたからだとか述べていますが、デミウルゴスを持ち出されたところで正直なところ明確なものとは言えません。また、イデアは美や善に関わるもので、逆に醜いものや悪といったものにイデアは存在しないと述べていますが、その美醜や善悪についての基準も明瞭なものではありません。むしろ、非常に宗教的ですらあり、のちのイデア論と宗教との関わりに関連することになります。さて、プラトンは肉体を魂の牢獄として考えていたようですが、これがのちのプロティノス (Plotinos) の新プラトン主義 (Neo-Platonism) の超越的な「一者 (*το εν, to hen*)」と、その一者からの流出による世界の創造という考えに繋がり、これとデミウルゴスによる世界の創造という考えは、そののちのグノーシス主義 (*Γνωσις*) やマニ教やキリスト教に大きな影響を与えることになります。たとえばグノーシス主義のヘルメス文書の一つのポイマンドレース (Poimandres)[9] によると人間は元来、神の子であって美しい神の似姿として創られたとされます。その彼があるとき高次で純粹な天上界からより下位の地上に向うことで星辰の支配を受けることになり、さらには地上にてフュシス (*φύσις*, 自然) 内に写った自分の姿に恋することでフュシスと愛欲に陥り、やがて「**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しまったといいます。この「伝説」^{*1}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^{*2}であるという二面性を持つことへの説明になっていますが、このことからヘレニズム文化圏でのバビロニア等のオリエント諸国の占星術の伝播とその影響力に加え、イデア論を中心とした哲学が神秘化して宗教へと変じてゆく有様が刻印されていると言えるでしょう。面白いことにキリスト教はその逆で、それまでの默示的な宗教からより合理的な宗教へと逆に変化してゆきます。この変化は初期のキリスト教哲学は教父と呼ばれる神学者によるもので、特に若い時分にマニ教徒でもあった教父アウグスティヌス (Augustinus Hippoensis, Augustine of Hippo) を通じて、新プラトン主義が初期のキリスト教の理論付けに用いられています。このときにキリスト教哲学は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) の哲学を導入します。このアリストテレスの哲学から新プラトン主義的な解釈を排されるには、中東への十字軍遠征を契機とするイスラム諸国との交流によってアリストテレスの著作である「形而上学」等が再導入される 12 世紀以降の話になります^{*3}。

*1 おおよそ宗教、あるいは宗教的な代物はその伝説を続々と生成するものです。現在でもカトリックでは列聖といった形で伝説が生成され、共産主義はその英雄を量産するといったあんばいです。

*2 星辰に支配されるからこそ星占いに意味があるのです。

*3 新プラトン主義の哲学の入門のためにアリストテレスの論理学が重視され、その入門書としてのボルピュリオスのエイサゴーゲー (*Εἰσαγωγή*)¹⁰ はボエティウスによってギリシャ語からラテン語に翻訳され、西ヨーロッパに伝播したさほど多くない哲学文献に含まれていたことと、哲学を学ぶ上での最初の入門書として用いられたこともあって、この注釈書は中世スコラ哲学に大きな影響を与えることになります。なお、プラトンのイデア論に対する反論が見られる「形而上学」といったアリストテレスの他の文献はイスラム教諸国を通じて 11 世紀以降に本格的にヨーロッパに入ります。詳細は「カテゴリー論」[1] の解説や「普遍論争」[16] を参照して下さい。

と、ここまで大きく脇道に逸れました。ではオブジェクトとは何なのでしょう？ここではもう一步踏込んで「概念」が何なのかということを次で述べることにしましょう。

2.3.2 概念について

さて、イデアと現物がどのような関係にあるにせよ、まず、イデアは「**思惟にのみによって知覚される**」という性格を持っています。このイデアのように思惟によって知覚されるものに「**概念, concept**」があります。ただこの概念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。むしろ逆の方向性を持つものです。さて、この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「**徵表**」、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で得られます。そうして得られた概念は「**A は B である**」という命題で、複数の主語 (A) の述語 (B) となり得るという性質を持ちます。このように複数の主語の述語になる性質のことを「**普遍**」と言います⁴。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。その一方で、「みけ」や「たま」は個体に強く結びつけられているために、精々、「これがたまです」という風に個体を特定するものであって、上述の意味で複数の主語を取り得るという意味での普遍できません。

それから「**猫**」という括り(あつまり)に対して「三毛猫」、「黒猫」、「白猫」、「虎猫」等の毛並みで分類することもできます。これらは「**猫**」の毛並みについて述べたもので、「**猫**」という概念よりも個々の猫をより詳細に説明することになります。このように「**個体をより詳しく説明することになる概念**」、つまり、「**より個体に近い側の概念**」のことを「**種概念**」、あるいは「**種 (species)**」と呼びます。また、逆に個体から一段離れた側の概念のことを「**類概念**」、あるいは「**類 (genus)**」と呼びます。たとえば「三毛猫は**猫**である」という命題では、「**猫**」が類で「三毛猫」が種になります。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種概念の「三毛猫」で、類概念の「**猫**」はそれよりも大雑把な説明であることが判るでしょう。しかし、大雑把であるが故に「**猫**」という類概念には「白猫」、「黒猫」、「虎猫」といった「三毛猫」以外の種概念も包含されるので「三毛猫」よりも「**猫**」が普遍であることが理解されるでしょう。さて、ここで「虎猫」の「とら」と「三毛猫」の「みけ」が貴方のそばに居たとしましょう。すると「**とらは猫**」で「**みけは猫**」ですが、「**とらは三毛猫**」ではなく、「**みけは虎猫**」でもありません。このように類の「**猫**」の方が種である「**三毛猫**」や「**虎猫**」よりも、より多くの個体の述語になります。このように「**類**」の方が「**種**」よりもより多くの個体の述語となり得るために「**類**」が「**種**」よりも「**より普遍である**」と言え

⁴ 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」の名前を冠したものが多いの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。

るのです。そして、より普遍的な概念のことを「**上位の概念**」と呼び、逆により個体に近い側の概念のことを「**下位の概念**」と呼びます。したがって、類の方が種よりも上位の概念であり、逆に類よりも種の方が下位の概念となります。そして「猫」の例では「猫」が類で「三毛猫」や「虎猫」が種になりますが、「猫」の上位の概念としては天下り的ですが「動物」があります。この動物を類とすれば今度は猫が種になるという訳です。そして「**最下位の概念**」が「**個体**」、「**最上の概念**」のことを「**範疇**」、あるいは「**カテゴリー**」と呼びます。このように概念には階層があり、最下層の概念が具体的なもの、つまり個体を指示することになるのです。

では「**概念**」はどのように表記され得るでしょうか？先程の「猫」の話なら、どのようなものが猫になるのかその特徴を列記する方法、それと「みけ」、「たま」、「とら」等々と個体を列記する方法があるでしょう。このように概念の表現の方法には二通りの表現方法、一つは「**内包**」、もう一つは「**外延**」と呼ばれるものです。最初の「**内包**」は概念が持つ特徴表/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**」等の属性から構成されるでしょう。一方で外延なら単純に「**黒猫**」、「**白猫**」、「**三毛猫**」といった猫の種や、「粟根さんの飼い猫のタマ」のように個体を列記する方法になります。このように内包は概念を表現する述語から、外延は概念に対応する具体的な個体や下位の概念の列記で構成されます。また、あるものを「**定義付ける**」とは「**三毛猫は猫である**」のように類概念や種概念で定義付ける「**類と種差による定義**」である「**実体的定義**」、あるいは「**分析的定義**」と呼ばれる方法と「**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があり、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。

ところで、外延で表現された概念は内包で表現することができますが、逆に内包で表現された概念は外延で表現できるとは限りません。それに加えて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が、この命題は「**ラッセルの逆理**」と呼ばれる非常に有名な逆理で、これをラッセルが分かり易くしたものが次の「**床屋の逆理**」として知られる逆理です：

— 床屋の逆理 —

とある村には床屋が一軒あります。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は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ないのですが、エピメニデス (*Ἐπιμενίδης*, Epimenides) というクレタ人^{*5}が主張したためにややこしくなったもので、これらの逆理の本質は前述の ' $x \notin x$ ' という命題です。この命題では「自分自身を元として持たないもの」と自分自身を定義するために自己を引用するという循環的な定義方法を採用しています。

ラッセル (Russell) の論理主義やカントール (Cantor) の（素朴）集合論^{*6}に批判的であったポアンカレ (Poincaré) は「科学と方法」 ([14]) で幾つかの逆理を分析して循環論法を含む定義に問題があると述べています ([14], p.204)。たとえば「偶数の集合」や「身長 170cm 以下の人集合」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、床屋の逆理のようにそれ自体に言及するという循環論法に訴えなければ定義できない定義方法を「非可述的」と呼び、この非可述的な定義に問題があると述べています。実際、ラッセルの逆理のような命題は、ラッセルの「数学の諸法則」やフレーゲの「算術の基本法則」といった論理主義^{*7}の初期の著作の体系やカントールの素朴集合論のように命題に外延や集合が存在することを前提としている体系では排除ができません。そのためにフレーゲは類に制限を加えることを検討したものの最終的に類に制限を入れることない修正を採用したものの解決に失敗し、ラッセルは無クラスの導入を検討したりしています。そしてラッセルは最終的に「悪循環原理」を導入することで自分自身に言及する非可述的な命題を除外することで対処しています。しかし、この悪循環原理を導入したことでラッセルの逆理の排除ができますが、今度はそのままでは数学的帰納法が使えないという副作用が生じてしまいました。この事態は非常に厄介で、この難点を除外するために「還元公理」と呼ばれる公理を導入すると、今度はその天下り的な性格が問題になるといったありさまでラッセルの試みは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、集合の公理系によって「ラッセルの逆理」自体を排除しています。

さて、内包と外延には「内包外延反比例増減の法則」と呼ばれる関係があります。この関係は内包が増大するに従って外延が減少し、外延が増加すれば内包が減少するという反

^{*5} 紀元前 6 世紀頃のクレタのクノッソスの哲学家だそうです。

^{*6} ここでの素朴集合論とは集合としての公理系を持たない、何らかの命題の外延を集合とみなす立場の集合論です。)

^{*7} 19 世紀末から 20 世紀初頭にかけて、数学を論理学から導出しようとする数学上の哲学で、フレーゲの「算術の基礎」 [13] やラッセルの「Principles of Mathematics」 [27] が初期の論理主義の代表的な著作です。なお、フレーゲの「算術の基礎」の後書きにラッセルの逆理のことが記載されています。

比例の関係のことです。たとえば「猫」という概念に対して「茶、黒、白の三色の毛並みである」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、白の三色の毛並み」の外延から消えてしまいます。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、白の三色の毛並みである」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、それだけ述語付けさ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ことになるために内包が減少するということなのです。

二つの概念の外延を比較したときに、より大きな外延を持つ概念のことを「上位概念」、あるいは「類概念」、逆に外延がより小さな概念を「下位概念」、あるいは「種概念」と呼びます^{*8}。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「毛並み」の違いが種差になります。なお、ここでの「上位」とか「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関係します。実際、より普遍的な側の外延は広くなることから理解できるでしょう。そして最上位の上位概念のことを「範疇 (Category)」、逆に最下位の下位概念を「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 (individual)」を直接指示する概念になるので、個体に最も近い概念になります。

2.3.3 プラトニズム

この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、イデアは個体とは別個に存在するために実在すると言えるでしょうが、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」が定義されているからです。アリストテレス (*Αριστοτέλης*) はカテゴリー論にて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。またこの範疇論への入門書として古代から有名なポルピュリオス (*Πορφύριος*, Porphyry of Tyre) のエイサゴーケ (Isagoge[20][25], *Εἰσαγωγή*, 「手引き」という意味です)^{*9}では、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といったことをの最初の章で述べています。のラテン語訳と彼の第二注釈が西洋の中世スコラ哲学にいわゆる「普遍論争」を引き起すことになります[16]。

^{*8} 類と種の語源 *γένος*, *εἶδος* も共に「形」という意味があり、このことから判るように、これらの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。

^{*9} 英訳はボエティウスのラテン語訳をオーエンが翻訳したもの[25]とバーンズがギリシャ語文献から翻訳したもの[20]があり、前者はイサゴーゲーが範疇論の入門書との立場、後者がアリストテレスの論理学全体への入門書として捉えた訳で詳細な解説があります。

この概念/イデアの存在について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、これらの原理や定理が導出されると考えるかといった議論にも繋がります。ここで事物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」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、所謂、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、プラトンやアリストテレスが用いた古代ギリシア語が属する印欧語族では‘A = B’という命題にて、その主語 A と述語 B の関係として表現する「**繋辞 (copula)**」として主語 A が存在する意味が付随する「**存在動詞**」と呼ばれる動詞が用いられることに関係しています。たとえば日本語の「A は B である」^{*10}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、すなち、存在含意があるのです。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、二項論理学としての特徴を持ちます^{*11}。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」のものや「仮説」に対しては三段論法等の推論が行えないことになってしまいます。しかし、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、自動的に存在含意を充して推論を行う際の障害がなくなることになるのです。

とはいっても、この三段論法による推論は非常にやっかいな性質を持っています。つまり、大前提が明らかに真であると判断できるものであったり、帰納的に求められたものであればまだ良いのですが、上述のような仮説であれば、その仮説から演繹的に導かれた命題を巡って大きな問題が発生するでしょう。

ちなみにイスラム哲学はどうだったかと言えば、アッバース朝初期の公認神学であった「**ムアタズィラ (Mu'tahzilah)**」と呼ばれる超合理主義派は、自らを「正義と神の唯一性の提唱者」と自称していた程で、彼等は三段論法を駆使してともすれば異端的な結論を導出していましたと言われます。このムアタズィラの教義には次の特徴があるそうです ([5], p.54-65. 参照):

^{*10} 「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。

^{*11} 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。

ムアタズィラの教義の特徴

- 予定論を認めない。
- ムハマンドによる執り成しを認めない。
- 神を人格化して表現することを認めない等

最初の「**予定論**」は、あらかじめ神によって人間の運命は完全に定められているとの主張です。この予定論を認めない理由は、正義の神が人間に不正義を行わせるはずがないので、人間の行為は全て人間に帰着するというものです。さらに人間は神と並んで第二の創造主となるという Goethe の Faust に顕著に見られる主張で、さらに善悪は理性に照して判断されるものであるために、全知全能の筈の神ですら理性の規準に従わなければならぬと主張しています。

次の「**執り成し**」は、イスラム教徒であれば最後の審判で預言者ムハマンドが信徒のために罪の軽減を神に執り成すことです。人間が第二の創造主であるとすれば、全ては自分に責任があるので悪行故に地獄の業火で焼かれるのは当然ということになります。

最後の「**神の人格表現の否定**」は、クルアーン(コーラン)で述べられた神の人間的表現を字義通り解釈するのではなく、一種の比喩として捉えるというものです。この結果、神を**知識や理性**と見做すことで、より具体的なものを望む一般信者にとっては非常に迷惑なことでしょう。そして、この考え方の先には「**哲学こそが全て、宗教は一般大衆向けの幼稚な哲学**」という考え方になりますが、これに反対する側の反応は非常に原理主義的なものになります。実際、この徹底した合理主義はガザーリ(al-Ghazali)に非難され、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によって再三、妨げられ、それらの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^{*12}。

2.3.4 イデア論の問題点

ところでプラトンのイデア論によると現世の対象はイデアの像として捉えられるのでイデア自体は個体から離れて存在することになります。たとえば「三毛猫のイデア」は個体の「みけ」の鋳型として「みけ」とは別個に離れて存在し、しかも、現世の「みけ」と異なって「三毛猫のイデア」は永遠不滅であると言う訳です。

ところでイデア論のややこしい点は、その個体の持つ性質からイデアが繰々出てくることです。実際、「三毛猫」の「みけ」には「猫」という類に由来する「猫のイデア」があれ

*12 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皮肉なことです。

ば、当然、「三毛猫」、「白猫」、「黒猫」、「虎猫」等の「毛並みに由来するイデア」もある筈です、さらには「日本猫」、「ペルシャ猫」等の「猫の種類に由来するイデア：もあっておかしくない筈です^{*13}」。さらに「猫」は「動物」で「4本足」なので「動物のイデア」や「四本足のイデア」等々と「三毛猫のみけ」のイデアは一つどころか多数存在することになります。このようにイデアは「多対一」でもあり「一対多」でもあったりと一意に定まるものではなさそうです^{*14}

この有様に対してプラトンの弟子であったアリストテレスさえも、その著作の「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」とプラトンのイデア論を批判しています[2]^{*15}。

ものの鑄型としてイデアとして考えてこの有様ですが、理想的な人間として例えられるソクラテスにしても、最初は赤ん坊で、それから子供、若者、壮年、老年といった過程を迎る訳ですが、すると、それぞれの瞬間にイデアがある筈で、そうすると、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とか逆に複雑になっている訳です。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、個体が生成し、変化し、あるいは運動して、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、世界の生成にしても、デミウルゴスやエロスといったある意思を持った部外者を引っ張り出して何とか創世神話を揃えたり、生物の生殖の理由をができたとしても、何気ない自然現象の説明にはとても無理があるのです。

2.3.5 形相 ($\epsilon\hat{\imath}\delta\circ\varsigma$)

さて、プラトンは「イデア」をイデア ($i\delta\varepsilon\alpha$ と呼んだりエイドス ($\epsilon\hat{\imath}\delta\circ\varsigma$) と呼んだりと、これらの二つの言葉を特に区別していません。ここでイデアはギリシア語の「見ること」に由来し、エイドスは「形」に由来するとのことで、どちらも形に関係します。

ここでプラトンの弟子の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 ($\epsilon\hat{\imath}\delta\circ\varsigma$, eidos)」はプラトンの「イデア ($i\delta\varepsilon\alpha$)」のような「個体から離れた存在」 ($\chi\omega\rho\sigma\tau\alpha$) ではなく、むしろ、現実にある個体はこの「形相 ($\epsilon\hat{\imath}\delta\circ\varsigma$)」と、これといった特性を持たない「質料 ($\varepsilon\lambda\eta$)」との「結合体 ($\sigma\hat{\imath}\nu\omega\lambda\circ\nu$)」として捉え、この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」という設計図のような働きをするものとして捉えています。これを先程の種の話に戻すと、まず、種に木としての形相が内部に存在し、その形相が結合体

^{*13} 日本猫は尻尾が短いといった特徴があるのです。

^{*14} この辺はヒンドゥー教のように表向は多神教でも、実は神々がビシュヌやシバの化身であったりと一神教的な側面も持っていたりする点に似てなくありませんが、どちらにしても一意に定まらない点は厄介です。

^{*15} 形而上学 第一卷九章

としてのもう一方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、細かな点ではかなり怪しいかもしれません、現代の科学でも対象が何であるか、どのような理由でそれがそれ自体であるかを説明しようとするものであり、この流儀はアリストテレスの考察にその源流があることが判ります。だからこそアリストテレスは「万学の祖」と言えるのです。

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。つまり、イデア論に訴えるよりも、より自然な対応付けができるのです。

2.3.6 類, 種, 種差, 特有性, 偶有性

アリストテレスの範疇(カテゴリー)を解説する前に、歴史的な経緯を含めて用語を導入しておきましょう。前述のポルピュリオスのエイサゴーゲーではアリストテレスの論理学を学ぶにあたって類、種、種差、特有性と偶有性が何であるかを知っていることの重要性を説いています。これらのもともとの概念について簡単に触れておきましょう。

まず「類」と「種」は「もののあつまり」です。まず「類」は‘それは何であるか’という問に対するものです。この類は単純なものあつまりと言うよりはむしろ、整理、分類されたもので、類を整理し、分類する役割のものが「種」です。つまり(素朴)集合のことばで言えば種は部分集合に対応し、種の外延は類に包含されるものために類よりも個体に近いと言えます。この類と種の関係は、人類と人種のように動物の分類で現在も見られるものです。そして、両者ともにギリシャ語の語源となる $\gamma\acute{e}nuς$ と $\epsilon\acute{i}\deltaoς$ はともに「形」の意味があります。実際、動物の分類にも「類」や「種」がありますが、この分類にせよ、もともとは「形」を主体とした分類なのです¹⁶。そして「種差」はその類を分類する上の理由、「特有性」はそのものだけに特有の特徴や特性で、そのものの本質を説明するものではありませんが、そのものを指示し得るものです。それに対して「偶有性」はその時点で持っている性質で、その程度を表現することができるもので、その性質を持たない状況も考えることができます。たとえば「黒いプードル犬」の黒は、本当に真っ黒であったり非常に濃い茶色であったりする訳で、さらに黒であることはプードル犬であるために必要な特徴でありません。年を取って灰色になってしまった黒いプードル犬も想像できますね。だから黒いプードル犬にとって黒は偶有性なのです。それに対して種差や特有性は程度がありません。たとえば類を動物とするなら人間はその動物の下の種、

*¹⁶ 最近のDNAの解析によって形ではなくDNAの近さで思わぬ種類の動物に近親性があることが判ったりしていることは興味深いことです。

種差としては「理性的」であるということ、その人間の下にソクラテスやアリストテレスといった個体があるとなり、偶有性は「色が白い」とか「座っている」といった様子や状態を示す表現に対応しするでしょう。ここで「理性的である」という種差は数値化して90だから理性的で50だから理性的でないとは言えません。理性的か理性的でないかどちらしかないので。

そして逍遙学派で「**ものの定義**」とは、類と種や種差を用いて、その「**説明規定**」(λόγος, logos)を与えることです。

2.3.7 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説明すること、すなわち、どのように述語付けられるかをアリストテレスは「範疇（カテゴリー）論」[1]にて説明しています。ここで範疇 (Category) は最上位の概念であって最も普遍的なものであると述べましたが、この哲学用語の「範疇」に対応するギリシャ語のカテゴリアー (*κατηγορία*) は法律用語の「**責を負わせる**」という意味のカタゴレイスタイル (*κατηγορεῖσθαι*) に由来し、アリストテレスが哲学用語として導入した経緯があります。そして、この範疇はものを語るときに、ものごとに述語付けたり関連付けたりすること、すなわち、ものごとを述定することに対応します。

アリストテレスは「カテゴリー論」にて「AはBである」という命題の述語Bを取り得るものを次の10個の範疇に分類しています*17:

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. どうか (性質, 質): 「面白い」, 「文法的」
4. 何に対する (関係): 「二倍」, 「半分」, 「より大きい」, 「より小さい」
5. どこか (場所): 「千代田公園」, 「ペットショップ」
6. 何時か (時間): 「昨日」, 「去年」
7. 置かれている (態勢): 「寝転んでいる」, 「立っている」
8. 持っている (所有): 「靴を履いている」, 「首輪を付けている」
9. 作用する (能動): 「齧る」
10. 作用を受ける (受動): 「齧られる」

ここでの「**本質的存在 (実体, οὐσία)**」は「**第二の本質的存在 (第二実体)**」と呼ばれるものです。第二があれば第一もあり、その「**第一の本質的存在**」は「私」, 「みけ」, 「ソ

*17 ただし、この分類はアリストテレスの他の著作で異なることがあるそうです。

「クラテス」等の個体で主語のみになるもの、つまり、個体により近くて普遍性を持たないものであるのに対し、「第二の本質的存在」は「人間」、「猫」、「[哲学者]」等の主語にも述語にもなり得るものです。そして、類や種になり得るものでもあります。これらの本質的存在はギリシア語ではウーシアー (*oùσία*) と呼ばれ、英語の be 動詞に対応する「存在」を意味する動詞 *εἶναι* を名詞化したものに由来するものです。この日本語の訳語としては「**実体**」が当てられていますが、本来のウーシアーの意味する範囲は広いもので、ここでの訳語は範疇論 [1] の新訳の用語に従っています。

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

この範疇で重要なことは、ある物について「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10種類の範疇の何れかになるということなのです。つまり、我々がこれから処理しようとする対象について、それが何であつてどのようなものであるかを記述するなら上記の10種類の範疇になります。また、そのようにして語られるものについては、類、種、種差、特有性と偶有性によって階層構造が入るのです。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。

2.3.8 オブジェクト指向プログラミングにおけるクラスの表現

こ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。この属性は、「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することになるという訳です。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。

2.3.9 継承

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

この継承という考えは非常に自然な考え方です。実際、ある新しい動物を発見したときに、その動物が何に属するといった系譜が創られます。その動物の調査が進むにつれてさらに細かく分類されてゆくこともあります。この場合、新しく分岐する動物はもとの動物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とあるオブジェクトの実体化として記述したとしても、そのうちにデータの理解が深まるところで、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出来上ることになりかねません。ここで継承関係が一子相伝的な継承であれば、その属性やメソッドが何処から引き継がれたかを探すことが直線的な関係になるので容易です。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、猫の毛並だけを考えているのであれば、「アジア」、「東アジア」、「日本」といったこと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います。これが複数のクラスを継承する関係であれば、属性やメソッドの検索により多くの時間を要する可能性が生じることが理解できるでしょう。さらに、この検索の手間の問題だけではなく、この属性やメソッドの検索順位をどのように定めるかで、新しいクラスの属性やメソッドが反映されなくなる可能性も出てきます。この問題については「C3 MRO」といった手法で改善が図られていますが、最初のクラスの分析が非常に重要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？」と言う問に対しても我々は様々な説明をします。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、これらのことの整理することで概念に到達すると前の節では述べました。そうするとその概念で説明され得るものがあつまりが外延ということになり、それが類や種であったりするのです。さて、ここまで説明をより厳密に隙の無いものにしたいのです。そのため集合論を利用しようというのです。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) 等の公理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ます。これらの公理の組み合わせで Z, ZF や ZFC 等と略記されます。その公理系を基に集合論が構築されています。ところで、この集合論にはその中で用いる言語があります。それが集合論言語と呼ばれるものです。ここではまず集合論の論理式で用いる記号について説明し

ておきましょう:

集合論で用いる記号系

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

ここでは元が集合に属するという意味で用いる記号 “ \in ” と対象の同値性を示す記号 “ $=$ ” の他は論理式の論理和 “ \vee ”, 論理積 “ \wedge ”, 否定 “ \neg ” と含意 “ \supset ”, それと量化詞の記号で「全て」に対応する “ \forall ” と「存在」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 “)”, それと区切記号の “,” 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. 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てゆきます.

2.4.2 集合論の公理系

では, 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 | $\forall x \forall y \exists z (\forall u \in z \equiv (u = x \wedge u = y))$ |
| A3 | 和公理 | $\forall x \exists y \forall z (z \in y \equiv \exists u (z \in u \wedge u \in x))$ |
| A4 | 累集合公理 | $\forall x \exists y \forall z (z \in y \equiv z \subseteq x)$ |
| A5 | 空集合公理 | $\exists x \forall y (y \notin x)$ |
| 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一つ一つ解説しましょう。

■**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す。また、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を一意に定義することができます。

ここで集合の外延は $\{a, b, c, d\}$ のように括弧 $\{ \}$ にその集合の構成元、成分あるいは元を a, b, c, d のように列記することで得られるもので、このとき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、集合 x, y の対集合を $\{x, y\}$ と記述します。特に $\{x, x\}$ は $\{x\}$ と表記して「**1-要素集合**」と呼ばれます。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 の順序に関しては何も述べてはいません。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。ここで成分が 3 以上の場合も順序対を構成することができます。この順序対の構成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 $x \cup y \stackrel{\text{def}}{=} \{x, y\}$ 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.

■幕集合公理 (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「**幕集合**」と呼び, $\mathfrak{P}(x)$ で集合 x の幕集合を表記します.

■空集合公理 (Axiom of empty set) 何等の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, この公理で唯一存在が保証される集合を「**空集合**」と呼んで記号 \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す. つまり v が集合であれば $\emptyset \cup \{v\}$ 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 に, $\emptyset \cup \{\emptyset\}$ 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るというものです. このように空集合公理と無限集合公理を含む公理系においては自然数を構成することが可能であり, 逆に言えば自然数を保持していると言えるのです.

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^{*18}。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 による像も集合となる公理であると言えますが、それと同時に素朴集合論とは異なり、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ものですが、もともとツエルメロが入れていた公理は「分出公理 (Axiom of displacement)」と呼ばれる次の公理です：

分出公理 (Axiom of displacement)

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

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' が直ちに得られます。そして、この分出公理から得られる集合を $\{u \in x : \phi(u)\}$ と表記することにします。

この分出公理によって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。

さて、素朴集合論で問題となった「ラッセルの逆理」をもう一度考えてみましょう。この逆理の本質は外延 $\{x : x \notin x\}$ が素朴集合論の集合から排除できない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はならず、除外することができるのです。

*18 自然数の後者関係についてはフレーゲの「概念記法」[12] ではじめて厳密に述べられています。

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」[11]で紹介されている例を挙げておきましょう：

まず、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導出できません。しかし、置換公理を認めてしまう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るので、集合と集合の元を区別する公理と言えます。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。まず、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。これは自然数の列に必ず最小値が存在することに応し、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^{*19}。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

*19 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、この漸には「オチがない」とも言えます。

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 [11]。

■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」を構築することが可能です。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*20。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。

*20 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。

同様に述語 $\text{Ord}(\alpha)$ を使って、今度は集合 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 にたいして $x < \alpha$, $y < \alpha$ となる x, y に対して $x < y$, $y < x$ か $x = y$ の何れかの大小関係が成立すること、つまり、全順序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 $\{\omega, \mathfrak{P}(\omega), \mathfrak{P}^2(\omega), \dots\}$ も順序集合です。しかし、 ω は自然数を含む順序数の中で最小の順序数になります。

では次に順序数全体 OR を定義しましょう:

順序数全体

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

ところで、この順序数全体 OR は**大き過ぎて**、集合ではなくクラス(類)になります。実際、この OR は推移的であり、また \in に関して全順序となります。ところで OR が集合であれば $\text{OR} \in \text{OR}$ となります。そこでこの OR の後者 $\text{OR} + 1$ を考えることができます。これは $\text{OR} \in \text{OR} + 1$ となります。しかし、OR は順序数の全体なので $\text{OR} + 1 \in \text{OR}$ となって矛盾が生じます。これが「**プラリ=フォルティの逆理**」と呼ばれる逆理ですが、ZFCでは正則性公理からこのような集合の存在が否定されるために OR は集合ではなくクラスになり、素朴集合論上の逆理も「OR は集合ではない」という定理になります。

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、ある順序数の後者とならない 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。

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となる文もあります。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「**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 (óργανον))**」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]^{*21}。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[23]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[23](以後、CMW と略記) の定義に沿って解説しましょう。この CMW[23] 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るのです。

^{*21} その注釈書としてポルピュリオス ($\Pi\sigma\rho\varphi\upsilon\rho\iota\circ\varsigma$, Porphyry of Tyre) のエイサゴーゲー [25] が非常に有名でした。§10 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.

ここで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.

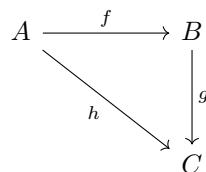
ここでメタグラフ \mathcal{C} の矢は二つの対象の間の関係を与えるものと考えた方が自然で, このときに矢の対象の順序が重要になります. さらに, 矢は伝統的論理学上の「繫辞 (copula)」として考えることもできます. ここで連続は命題「 A は B である」の中の「... は... である」のように主語と述語の関係を表現する機能を持っており, 伝統的論理学ではこの繫辭こそが命題を構成するものと考えられていたとのことです [4]. さて, ここで矢を \xrightarrow{f} と表記することで矢の式 $f : A \rightarrow B$ から $A \xrightarrow{f} B$ 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、ここで「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 座標となり、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 $\{\langle x, y \rangle : x \in A \wedge y \in B \wedge f x = y\}$ として矢を考えることができます。このときに $x \in A$ に対応する対象 B の元を $f(x)$ あるいは $f x$ と表記し、 $y = f x$ のときに $x \mapsto y$ と表記することで $x \in A$ と $y \in B$ が矢 f を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` 型が挙げられるでしょう。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：

$$\begin{array}{ccccc} A & \xrightarrow{f} & B & & \\ & \searrow f & \downarrow 1_b & \swarrow g & \\ & & B & \xrightarrow{g} & C \end{array}$$

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唯一に定まるところで、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対応するので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 Frege 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 $h = g$ となるとき. このときに $f : A \rightarrowtail B$ 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 $A \equiv B$ 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、圏の対象が集合と限らないために勝手がやや異なります。まず、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やメタ圏はそれぞれ**グラフ** や**圏**と呼ばれます。

2.5.5 グラフと圏について

今迄、解説したメタグラフやメタ圏では、その対象や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、 \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たものを考えてみましょう。つまり、対象と矢が構成する類が集合となるものに限定するのです。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 $\text{dom}, \text{cod}: \mathbf{A} \xrightarrow{\text{dom}} \mathbf{O}$
 $f \in \mathbf{A}$ 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 始対象と終対象

次に特殊な対象として「**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。

始対象と終対象は考察する圏によって異なります。たとえば、対象の集合を自然数 \mathbf{N} 、矢を \leq とする圏 (\mathbf{N}, \leq) を考えると、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。そして矢を \geq にした圏 (\mathbf{N}, \geq) では逆に任意の $n \in \mathbf{N}$ に対して $n \geq 0$ を充すことから 0 が終対象になることが判ります。このように 0 と 1 という記号は自然数だけではなく、さまざまな場面で用いられます。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」という枕詞を必ず置くものとし、そうではなく単に $0, 1$ と表記されているときは自然数 $0, 1$ を指名するものとします。

ここで始対象と終対象は反変函手 \circ^P 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に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。

2.5.8 圈の例

ここでは具体的な圈の例を挙げることにしましょう。最初にちょっと特殊な圈を考えます。この圈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圈は「離散圈」と呼ばれます。この離散圈では各対象 $A \in \mathcal{C}$ と対応する恒等矢 $A \xrightarrow{1_A} A$ の他には矢がない圈なので、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。そして離散圈は対象 $A \in \mathcal{C}$ 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」としか言えない圈であることが判ります。これはキュニコス派のアンティステネス (*Αντισθένες*) の主張する「一つの主語は一つの述語あるのみ」^{*22}と普遍を認めない立場に対応します。

その他の重要な圈の例を挙げておきましょう：

- **Set:** 対象が集合で、矢が通常の写像
- **Set_{*}:** 対象が基点付きの集合で、矢が通常の写像
- **Cat:** 対象が圈で、矢が函手
- **Grp:** 対象が群で、矢が準同型写像
- **Ab:** 対象が可換群（アーベル群）で、矢が準同型写像
- **Top:** 対象が位相空間で、矢が連続写像
- **Top_{*}:** 対象が基点付きの位相空間で、矢が連続写像
- **Toph:** 対象が位相空間で、矢が連続写像
- **\mathcal{C}^{op} :** 圈 \mathcal{C} の双対圈

ここで述べている集合、圈と位相空間は、より正確にはそれぞれ「小集合 (small set)」、「小圈 (small category)」、「小位相空間 (small topological space)」と呼ばれます。これは「グロタンディークの宇宙」との関係を表現するものです。このグロタンディークの宇宙は圈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の宇宙に元として包含されるために「小」、この宇宙そのものはグロタンディークの宇宙の元として包含されないために「大」と呼ばれるのです。

また、圈については「局所的に小さい」と「小さい」の二種類があります。圈 \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。

^{*22} 形而上学 [2] 5 卷 29 章, 1024b34

2.5.9 グロタンディークの宇宙について

さきほどの図の例で、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の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「**グロタンディークの宇宙**」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の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。図の場合、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の宇宙にて次の演算が許容されています：

—— グロタンディークの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「**グロタンディークの宇宙**」と呼びます：

—— グロタンディークの宇宙 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 $\supset b \in U$

グロタンディークの宇宙 U 自身は ZFC 公理系の**正則公理**によって U の元として含まれることがありません。さらに、この U が集合になるとも限りません。このように宇宙 U とその元には区分があり、この区分を対象が集合であれば宇宙 U の元を「**小集合**」、対象が図であれば U の元を「**小図**」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**」、図であれば「**大図**」といったあんばいです。

2.5.10 函手

圈 \mathcal{C} と圈 \mathcal{D} が与えられたとき、これらの圈に対して、圈 \mathcal{C} の対象を圈 \mathcal{D} の対象に対応させ、同様に圈 \mathcal{C} の矢を圈 \mathcal{D} の矢に対応させる写像^{*23}を考えられます。そして、より扱い易い性質として、圈 \mathcal{C} の恒等矢 1_A を圈 \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圈 \mathcal{C} から圈 \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。では写した先の圈 \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。たとえば同じ圈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圈から圈への写像のことを「**共変函手**」と呼びます：

共変函手 (covariant functor)

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

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は圈 \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。

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圈から圈への写像を「**反変函手**」と呼びます：

^{*23} 圈では対象や矢のあつまりが集合になるからです

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

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ときに、それが单射、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「充满 (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 の関係を考えられます。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:

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

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

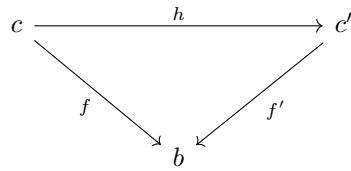
既存の圏と函手を使って新しい圏を構成することができます。ここでは「**コンマ圏 (comma category)**」と呼ばれる圏の構成について述べます。なお、コンマ圏はスライス圏 (slice category) とも呼ばれます。

まず、コンマ圏では対象は圏 \mathcal{C} の矢から構成されます。具体的には圏 \mathcal{C} とその対象 $b \in \mathcal{C}$ に対して $\text{dom } f$ が対象 b となる矢 $b \xrightarrow{f} c$ を対象とし、 $\langle f, c \rangle$ と対で表記します。さて次はコンマ圏の対象 $\langle f, c \rangle$ から $\langle f', c' \rangle$ への矢は何で与えられるでしょうか？これは圏 \mathcal{C} の矢 $b \xrightarrow{h} b'$ で $f' = h \circ f$ を充すもの、つまり、次の可換図式が成立するもので与えられます：

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

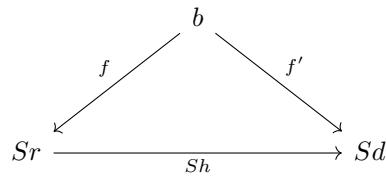
このように構成された対象と矢で構成された圏を「**コンマ圏 (comma category)**」と呼び $(b \downarrow \mathcal{C})$ 、あるいは \mathcal{C}/b と表記します。

もちろん、コンマ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、対 $\langle c, f \rangle$ で表記します。また、矢は先程のコンマ圏の矢と同様で、次の可換図式が成立する矢 h で与えられます：



コンマ圏の双対を $(\mathcal{C} \downarrow b)$ と表記します。

また函手 $S : \mathcal{D} \rightarrow \mathcal{C}$ を導入してコンマ圏 $(b \downarrow S)$ を構成することもできます。この場合、対象は $b \xrightarrow{f} Sr, b \in \mathcal{C}, r \in \mathcal{D}$ で与えられ。 $\langle f, r \rangle$ と表記します。そして、二つの対象 $\langle f, r \rangle, \langle g, d \rangle$ 間の矢 h は $r \xrightarrow{h} d$ で与えられ、次の可換図式が成立することになります：



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^{*24}。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

^{*24} 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。

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：

$$\begin{array}{ccc} \text{みけ} & \xrightarrow{f_1} & \text{三毛猫} \\ & \searrow f_0 & \downarrow f_2 \\ & & \text{猫} \end{array}$$

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。また、 $q : D \rightarrow B$ 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

として包含するということ、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：

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

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なることを意味します。

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：

猫の圏



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：

$$\begin{array}{ccc}
 \text{みけ} & \xrightarrow{u=\epsilon} & \text{三毛猫}_{\text{set}} \\
 & \searrow h=\epsilon & \downarrow Sf=\subset \\
 & & \text{猫}_{\text{set}}
 \end{array}
 \quad
 \begin{array}{ccc}
 & & \text{三毛猫}_{\text{猫の図}} \\
 & & \downarrow f \\
 & & \text{猫}_{\text{猫の図}}
 \end{array}$$

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 [23] 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 **Set** と体 K を係数とするベクトル空間の圏 **Vect** $_K$ を考えます。そして、函手 U をベクトル空間の圏から集合の圏への函手とします。この函手 U は対象については、ベクトル空間を、その演算を忘れることで、単に集合に写すというもので、その矢も単純にベクトル空間の線形写像の対応関係をそのまま集合の写像としての対応関係に置換えた矢とみなすだけです。次に $X \in \mathbf{Set}$ を基底とし、係数体を K とするベクトル空間を V_X と記述すると圏 **Set** の対象 $X, U(V_X)$ 間の矢 $X \xrightarrow{j} U(V_X)$ が定まります。つぎに任意の $W \in \mathbf{Vect}_K$ に対し、 $U(W)$ を考えて X の元を $U(W)$ に対応させることで矢 $X \xrightarrow{f} U(W)$ を構成することができます。それからこの f の対応関係を基に体 K について線形になるように拡張することで新たに **Vect** $_K$ の矢 $V_X \xrightarrow{f'} W$ を構成することができます。以上の結果から次の可換図式が得られます：

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

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

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

対象の積

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

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

圈の積の定義を可換図式として表現することができます：

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

この定義では対象 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^{*25}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。

^{*25} 順序数も集合です

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。

この積の双対は「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:

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

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} では対象の幕も定義することができます:

対象の幕

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

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

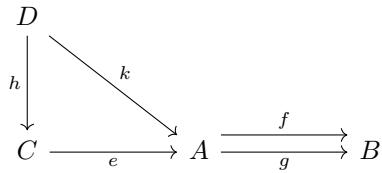
ここで矢 ev のことを「評価 (evaluation)」、矢 \hat{g} のことを矢 g の「転置 (transpose)」と呼びます。さらに対象の幕については $\text{Hom}(C \times A, B) \cong \text{Hom}(C, B^A)$ が成立します。

次に「等化 (イコライザー, equalizer)」について述べましょう:

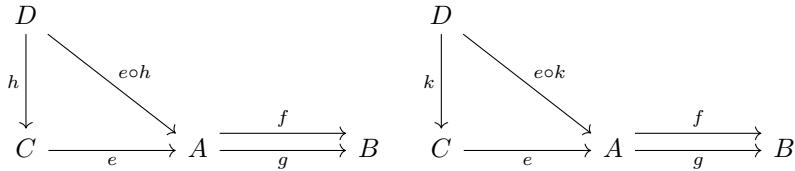
等化 (equalizer)

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。



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:



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。

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

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。

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

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

引き戻しの性質の性質を幾つか挙げておきます：

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

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

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

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

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

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

は引き戻しになりますが、ここで

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

をよくよく考えると次の集合の積と等価が現われます:

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

ここでは集合の圏 **Set** で考えたことですが、このことから引き戻しには対象の積と等価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:

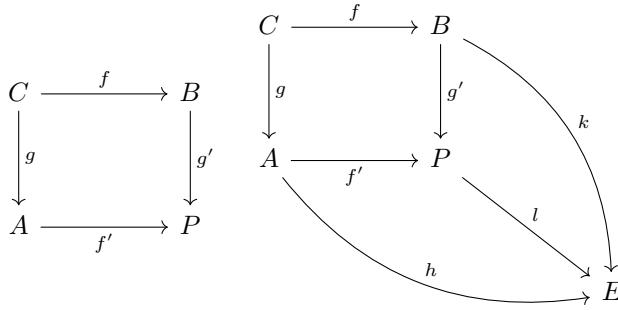
—— 積、等化と引き戻しの関係 ——

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。

「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} とその部分極限 \mathcal{D} が与えられ、圏 \mathcal{D} の対象を D_i 、矢を $D_i \xrightarrow{\delta_{ij}} D_j$ と表記することにします。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} の対象への矢 $X \xrightarrow{\nu_i} D_i, X \xrightarrow{\nu_j} D_j$ 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。

2.6 トポス (Topos)

2.6.1 部分対象分類子

「位相幾何学 (Topology)」の **Topo** が場所や位置を示す言葉ですが、ここで挙げる「トポス (Topos)」はアリストテレスの著作に由来し、そこでは議論のありどころの意味で用いられています。ここでトポスはそれに似た働きをし、判断の枠組を与えるものであると言えるでしょう。

ここでトポスを導入するためには部分対象分類子と呼ばれる対象が必要とされます。この部分対象分類子の雰囲気は、与えられた対象を二つの部分対象に分ける仕組に関連するものと言えます。この見方を代えると議論等で真偽の判断に関わるものになります。

まず、二つの対象 $A, B \in \mathcal{C}$ が与えられたときに対象 A が対象 B の部分対象であるとは A から B への矢で mono になるものが存在するときでした。つまり、矢 $A \xrightarrow{f} B$ が圏 \mathcal{C} に存在するときです。ここで圏 \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ものです:

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。

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

このときに $\Omega \in \mathcal{C}$ を \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「特性写像」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 $a \xrightarrow{!_A} 1$ 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 に写されることになります。このことは χ_f が対象 A とその他の対象を区分する写像として動作し、区分するための特徴付けを行う写像に相当することから写像 χ_f のことを「特性写像」と呼ぶ理由になります*26。

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

ここで 1., 2., 3. を充す圈のことを「デカルト閉圏 (Cartesian Closed Category)」と

*26 機械的学習はこの特性写像を構築するための手続を与えるものと言えます。

呼び、「CCC」と略記します。

トポスの定義

1. 圈 **E** には終対象 1 が存在する.
2. 圈 **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 **E** の任意の対象 A, B に対し、その冪 B^A が存在する.
4. 圈 **E** には部分対象分類子 Ω が存在する.

第3章

Pythonについて

3.1 この章の目的

この章では Python という言語について解説を行います。そこで最初に Python の特徴である簡易化された構文について概要を述べ、その中で Python を使って有理数を定義してみます。これらの簡単な実例のあとで「Python 言語リファレンス」^{*1}を基に Python の言語的な概要を述べることにします。なお、ここでは Sage で用いられている C で記述された Python(**C**Python) の 2.x 系を中心に解説し、必要に応じて Sage の例を追加することにします。

3.2 簡素化された構文

3.2.1 マルチパラダイムプログラミング言語としての Python

Python はオブジェクト指向プログラミング言語 (OOPL) と呼ばれる言語の一つですが、オブジェクト指向プログラミング言語で有名な Java のような使い方に限らず対話処理言語の Basic のように使える言語で、このような言語はマルチパラダイム プログラミング言語と呼ばれます。この系統の言語は複数のプログラミングスタイルに対応しているために目的に応じてプログラミングスタイルを選択することができます。そのお陰で Python は Basic や MATLAB のような対話処理言語に似た気軽な使い方が可能となっているだけでなく、大規模なシステムの開発ではオブジェクト指向プログラミング言語の良さが發揮できる言語になっているのです。ただし、Python 自体は他の言語と比較して際立って処理速度が速い言語ではありません。むしろ、Python 言語を中心として使い勝手を向上させることで全体の効率を上げているのが現状で、さらにそれを可能にするだけの計算機環境が存在するという利点を生かしているのです。

^{*1} Python 言語リファレンス <http://docs.python.jp/2/reference/index.html>

3.2.2 必要最低限の構文

Wikipedia には「核となる構文や文法が必要最低限に抑えられている」とあります。実際、Python は構文が簡素です。実際、その制御文は、条件分岐なら if 文、例外処理なら try 文、反復処理なら for 文と while 文だけです。これが他の言語であれば、条件分岐には case 文や switch 文、反復処理には repeat 文等を持ったりすることがあります。Python ではこのように必要最低限なものに抑えられています。その結果、Perl^{*2} や Mathematica に見られるような「超絶的技巧」な工夫を駆使したプログラミングができない一方で、常識的なプログラミングで妥当な結果が得られるという仕様になるのです。

3.2.3 字下げ

C や Java といった言語と比較して構文については大きな違いがあります。これらの言語ではプログラムの構造を明確にするために括弧 ‘{}’ を用いますが、Python では括弧ではなく「インデント（字下げ）」を用います。このことによって Python のプログラムは二次元的な構造を持つことになり、視覚的にプログラムの構造が示されることになります^{*3}。この字下げについては PEP-8 で空白文字のみの 4 文字単位で行うことが推奨されています。このことを例を交えて解説しましょう。

たとえば、C で if 文は直線的に

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

のように記述しても

```
1 if (x==0){
2     y = 1;
3 else
4 {
5     y = 0;
6 }
```

と記述していても、C では改行や空白文字にはプログラム上の意味がないため、プログラムの意図は別にして、これらの記述にプログラムの意味には違いはなく、双方は C の構文

^{*2} 駱駝形のプログラム例 (camel code): <http://www.perlmonks.org/?node=camel+code> 技巧の方向性が違いますが、4 匹のラクダを ASCII アートで描きます。

^{*3} 仕様として字下げを持つ言語としては FORTRAN77 もあります。ただ FORTRAN77 はカード読取機の制約に由来するもので、プログラムの構造を担うものとの意図はありません。それに対して Python ではプログラム構造の可視化を意図しています。

としても何等の問題がありません。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字下げを行い、その字下げの水準もブロック単位で他の行と合せておく必要があります。そのため Python 向けに書き換えるのであれば

```
1 if x == 0:  
2     y = 1  
3 else:  
4     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’ が同じ水準、つまり同程度の字下げの位置になくてはなりません。のために次の線的な記述:

```
1 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を十分に表現できていない記述:

```
1 if x == 0:  
2     y = 1  
3 else:  
4     y = 0
```

は Python で構文エラーになります。なお、構造が単純な構文に限っては

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

や

```
1 class TEST: pass
```

のように一行で記述することが認められています。より正確には、インデントは文節末尾に記号 “:” が置かれた時点で以降の文節に対してインデント処理を行うことになりますが、文全体で記号 “:” が一つだけであればインデントを行なわずに線形に記述することが許容されます。しかし文中の文脈の末尾に記号 “:” が二箇所以上生じるときはインデント処理が必要になります。

ここで字下げの位置については一段階につき 4 文字の空白文字が推奨されています。このようにプログラミング様式についても規約が PEP で事細かく定められていることが Python の非常に大きな特徴です。

3.2.4 文書文字列 (docstring)

Python ではプログラムの文書性を高める工夫として、プログラム内部に文書を書込みて、それをヘルプとしても利用できるという工夫があります。これは LISP や MATLAB 系の言語でよく見られるもので、このように埋め込まれた文字列のことを「文書文字列 (docstring)」と呼びます。ここでは書文字列の例を幾つか挙げましょう。まず最初に LISP(ここでは SBCL) の例を示しておきます:

```

1 * (defun add2 (x) "2を足すよー" (+ x 2))
2
3 ADD2
4 * (documentation #'add2 'function)
5
6 "2を足すよー"

```

この例では “**” が SBCL のプロンプトで、そのプロンプトに続いて defun 文で函数 add2 の定義を行い、それから函数 add2 に設定した文書文字列を函数 documentation() を用いて表示させています。これと同様のことを Python では次ので行うことができます:

```

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

Help on function add2 in module __main__:

add2(x)
    2を足すよー
>>>

```

ここで ‘>>>’ と ‘verb+...+’ は Python のプロンプトで、この例で函数 add2() を Python 上で直接入力して定義し、それから函数 help() を使って記載した長文書文字列を表示させています。なお、この長文書文字列ではエンコーディングを UNICODE とするために明示的にエンコーディングの指示を先頭の文字 ‘u’ で行っています。このように文書文字列をオンラインヘルプとして用いることができるのです。

最後に MATLAB に似た言語、そのような言語を今後は MATLAB 系言語と簡単に呼びますが、この例として Yorick の例を示しておきましょう。ちなみに MATLAB 系の言

語では文書文字列に相当する註釈行をファイルから検し出すために函数単位でファイルにあらかじめ記述しておかなければ使えません。このようなファイルのことを MATLAB では M-File と呼びます^{*4}。ここでは函数を add2 として、その内容を下記のものとします：

```
1 func add2(x)
2 /* DOCUMENT add2
3 *
4 * 2を足すよ-
5 */
6 {return x+2;};
```

このファイルを ‘add2.i’ と函数名に対応する名前で保存しておきます。それから起動した Yorick からこのファイルを include 函数で読み込んでおけば、あとは必要に応じて help 函数を使うとファイル add2.i に記載された文書文字列が表示できます。以下に実際の Yorick の様子を示しておきます。ここで ‘>’ は Yorick のプロンプトです：

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

Python でも文書文字列の記載はほぼ同様ですが、PEP-257 に基本的な記述に関する規約があります。まず、文書文字列を記載する場所ですが、函数やメソッドを定義する def 文の直後に文書文字列を置くこととなっており、このときの文書文字列は 3 個の二重引用符 ‘“”’ で括られた文字列になります。この文書文字列を閲覧するためには函数 help() を用います。ここでは Python 上で函数 help() を用いて函数 open() を調べた結果を示しておきましょう：

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
```

^{*4} M-File というのも MATLAB ではこのようなファイルの修飾子が “.m” だからです。そのため neko() という MATLAB の函数を定義したければ、‘neko.m’ というファイルに函数一式とその函数内部でのみ用いる函数を記述しなければならないのです。

```
preferred way to open a file. See file.__doc__ for further
information.

lines 1-7/7 (END)
```

なお, IPython を Python のシェルとして用いている場合には函数 help() の他に記号 ‘?’ が使えます. Sage は IPython を UI に用いているので記号 ‘?’ を函数 help() の代りに使えます. ここで, 記号 ‘?’ を使うときには記号 ‘?’ と調べる項目との間に空白文字を入れても入れなくても構いません:

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

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

```
In [2]:
```

函数 help() は文書文字列のみを表示していますが, IPython 上で記号 ‘?’ を用いると函数, メソッドやモジュールに関する情報も同様に得られます.

3.2.5 クラスの定義

オブジェクト指向プログラミング言語で, 「**オブジェクト**」はこれから計算機上で扱うデータを抽象化したもの, すなわち「**概念**」に相当します. そして, 「**クラス**」はそのオブジェクトを定義するもので**概念の内包**や**概念の外延**に相当します. さて, 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**」に相当すると言え, オブジェクトが実体化したものとしてデータを捉えること可能で, このオブジェクトの実体化したもののことを「**インスタンス**」と呼びます. また, クラスについても別のクラスのインスタンスとして考えられることがあります. このときに, あるクラスのインスタンスとなる側のクラスのことを「**クラス オブジェクト**」, 「**クラスのクラス**」になる側のクラスのことを「**メタクラス**」と呼びます. そして, 通常のオブジェクトとして実体化した個体のことを「**インスタンス オブジェクト**」と呼びます. Python ではこれらのオブジェクトが区別されています.

Python でクラスの定義は 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*5:

*5 この定義方法で Python 2.x は古典的クラス, Python 3.x はクラスタイプになるという違いがあります

```
1 class TEST:  
2     pass
```

このクラスは名義的な定義を行うだけのクラスで, ‘pass’は何もしないという文です。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()’ と名前 a への束縛で行ないます。このクラス TEST では対象の振舞いや初期値といったものが一切決まっておらず好き勝手なことができます。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()’ で名前 ‘a’ に束縛するかたちでオブジェクトの実体化, すなわちインスタンス化を行っています。それから ‘a.pet’, ‘a.weighth’ や ‘a.age’ と C で見られる構造体の処理に似たことを行っていますが, ここでの ‘name’ や ‘age’ はクラスの「属性 (attribute)」と呼ばれ, 概念の属性に相当します。このように属性を無宣言でインスタンスに自由に追加することができる点が Python の大きな特徴です。このことは無作法な方法で属性を外部から自由に操れるということを意味します。

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 で見られる構造体に似たものになります:

```
1 class TEST:  
2     x = 1  
3     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:
```

が, ここでの解説で型の細かな違いには踏み込まないために問題がありません。

```
....      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、そのインスタンス オブジェクトを名前 a1 に束縛させ、それからインスタンスの属性を参照しています。ここでクラス TEST で表現されたオブジェクトのインスタンス a1 の属性の参照は ‘⟨インスタンス名⟩.⟨属性名⟩’、インスタンスの属性値の変更は

‘⟨インスタンス名⟩.⟨属性名⟩ = ⟨属性値⟩’ で行えます。このように C の構造体とほぼ同様の使い方になっています。

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。このメソッドはオブジェクトに結び付けられた函数で、オブジェクトに結び付けられているために通常の函数とは異なり、継承関係にない別のクラスのオブジェクトに対してメソッドは使えず、オブジェクトに付属する函数としての性格があります。ただし、このメソッドを函数ではなく、オブジェクトを特徴付ける機能の一つとして考えることもできます。つまり、クラスとはこれから我々が扱う対象が「何であるか」と「どういったものであるか」という問に対する回答であり、このときに「どういったものであるか」という特徴付けるものは何かの値だけではなく何かの機能もあるのです。その機能に相当するものをメソッドとして表現することになります。そして、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となるものや取り得る値が一意でないものもあるでしょう。そして、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であるか、つまり、特有性になる場合と、その値にある範囲があつたり、設定されていないことも考えられる場合、つまり、偶有性になる場合が考えられます。また属性の役割を考えると、そのクラスを特徴付ける種差や特有性のようにクラス単位で共通になる値を格納する属性、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単位で共通になる値を格納する属性のことを「**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のことを「**インスタンス変数**」と呼びます^{*6}。なお、メソッドにも二通りあり、クラス操作に関わる「**クラスメソッド**」とインスタンス操作に関わる「**インスタンスマソッド**」です。ところで Python ではインスタンスマソッドに対応する「**instancemethod**」、クラスメソッドは参照の方向性の違いから「**classmethod**」と「**staticmethod**」の二種類のメソッドがあります。

では先程の TEST クラスのクラス変数に値を定め、それらの和を計算するメソッドを追加したクラスの定義をしてみましょう：

```
1 class TEST:  
2  
3     x = 1  
4     y = 1  
5  
6     def wa(self):  
7         return self.x + self.y
```

クラスのメソッドの定義も Python の函数の定義と構文は全く同じです。ただし、オブジェクト自身を引用する場合は ‘self’ という変数名を用います。ここではメソッド wa() を定義していますが、このメソッドではクラス属性 x と y の和を返す処理を行います。そこでクラス属性の参照を行わなければなりませんが、そのときに上記の方法で ‘self’ をメソッドの引数とし、その ‘self’ のクラス属性の参照をメソッド内部で行います。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  
>>>
```

ここではインスタンス a1 の属性の書換えとインスタンス a1 にてメソッド wa() の実行を行っています。クラス内部の定義で引数が ‘self’ のみのメソッドであれば実行に引数は

^{*6} C++ ではクラス変数とインスタンス変数はその宣言自体が異なります。しかし、Python ではそのような型の宣言がないために使い方で区別する形になります。

不要で ‘a1.wa()’ のように ‘⟨ インスタンス名 ⟩.⟨ メソッド名 ⟩()’ で実行します。

これだけでは構造体みたいな代物が生成できるというだけで、そんなにクラスを用いる有難味を感じることは特にありません。また、属性も構造的な利用方法しかしていません。ここでオブジェクト指向プログラミング言語の有難味の一つは「**継承**」と呼ばれる機能です。つまり、既存のクラスを土台に新しいクラスが構成する機能で、この機能を用いれば大規模なシステムの構築が既存のクラスを自然に再利用することが非常に効率的に行えるのです。そこで自然数を拡張して有理数を構築することで、その御利益を体験してみましょう。なお、整数とその算術演算はすでに定義されているので、それらを上手く利用することになります。

さて、有理数を整数から構成しようと申しましたが、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」ということと「どのようなものであるか」ということを明確にしなければなりません。そのために我々は整数を既に手に入れているので、整数と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になります。そこで整数の対のクラスを定義することにしましょう。この整数対は個々の有理数で一つ定まるもので有理数全体で一定の値ではありません。ということは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るべきです。それと有理数の表示を行う機能を入れておきましょう。つまり、有理数クラスのインスタンス ‘a’ を入力したときに ‘n/m’ のように自然な書式で表示するというものです。実際、‘(n, m)’ といった整数対の書式で返されるよりも明瞭になりますね。それからインスタンス化した時点で個々の有理数が定義された状態にしたいものです。このようにインスタンスの初期化や標準的な表示を定めるメソッドとして、Python には「**特殊メソッド**」があり、これらを用いて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については §3.5.3 で詳細を述べているので詳細はそちらを参照して下さい。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内容表示を行う特殊メソッド `__repr__()` を用います。では有理数のクラス `PairOfInts` を以下で定義しましょう：

```

1 class PairOfInts:
2     def __init__(self, numer, denom):
3         self.numer = numer
4         self.denom = denom
5     def __repr__(self):
6         return '%s/%s' % (str(self.numer), str(self.denom))

```

このクラスでは上述の二つの特殊メソッド（インスタンスの初期化と内容表示のメソッ

ド) が定義されています。これらのメソッドの名前には前後に文字 “_” がありますが、Python ではこの文字を名前の先頭に持ったメソッドや属性は外部から隠蔽されるべき属性であることを意味します。この隠蔽はメソッドの名前が別名で置換えられて本来の名前でアクセスできないという方法で達成されています。また最初に定義されているメソッド `__init__()` はインスタンスの初期化を明示的に行うための特殊メソッドで、ここでのメソッドの引数の `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とされる引数、つまり、インスタンス変数です。ここで二つのインスタンス変数の最初の `numer` が分子、`denom` が分母に対応し、‘`a = PairOfInts(1,2)`’ で対象を生成したときに ‘`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__()` がインスタンスの文字列表示に関わるメソッドで、たとえば Python で ‘`a = 1`’ で代入を行ったあとに ‘`a`’ と入力することで ‘`1`’ が表示されるようになります。つまり、このメソッドはインスタンスが入力されたときにどのような表示を Python が行うべきかを定めるメソッドです。ここでは出力書式を ‘`%s/%s`’ と指定することで ‘`a = PairOfInts(1,2)`’ でインスタンスを生成したときに ‘`a`’ と入力すると ‘`1/2`’ と表示することを指示しています。また、‘`a = PairOfInts(1,2)`’ でインスタンスを生成したときに `numer` や `denom` の値の取り出しあは C の構造体のときと同様にそれぞれ ‘`a.numer`’ と ‘`a.denom`’ で行えます。

このことを実際に試してみましょう。なお、そのまま上の内容を Python (のシェル) に直接入力しても構いませんが、ここではカレントディレクトリ上にファイルを置いて該当ファイルを `import` 文を使って読み込むことにします。このとき単純に `import PairOfInts` で読み込むと、メソッドにモジュール名の ‘`PairOfInts`’ を付けることになって煩わしいために `from PairOfInts import PairOfInts` でファイルの読み込みを行うことでモジュール名を外して使えるようにしています:

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

この例では `PairOfInts` クラスで表現したオブジェクトを名前 `a` のインスタンスとして生成し、単に名前 `a` を入力することでその内容を表示しています。その結果 ‘`1/2`’ という表示を得ていますが、ここでメソッド `__repr__()` が定義されていなければどうなるかを示しておきましょう。そのためメソッド `__repr__()` を持たないクラスの `TEST` を定義して確認しましょう:

```
>>> class TEST:
....     def __init__(self, numer, denom):
....         self.denom=denom
....         self.numer=numer
```

```

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

```

この例では PairOfInts クラスからメソッド `__repr__()` を除いた TEST クラスを直接、Python 上で定義し、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。オブジェクトの識別値はオブジェクトのメモリ上の位置を示す番地であるために、その時点で一意に割り当てられる数値になります。また、メソッド `__repr__()` は組込函数の函数 `repr()` に対応するので、PairOfInts の定義で用いられている文字列を引き渡せばインスタンス `b` の内容が指定した書式の文字列で表示されていますね。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新たな表示が定められます。このように上位のクラスにあるメソッドを新たに定義しなおすことを「**上書き (override)**」と呼びます*7。

ここでクラスの属性名で文字 ‘`_`’ を先頭に持つクラスの属性について解説しておきましょう。文字 ‘`_`’ や文字列 “`__`” を名前の先頭に持つメソッドや変数は、そのままの名前を使って外部から参照ができない仕組みになっています。とはいっても厳密に隠蔽されたものではないので外部から工夫をすれば参照することができます。特に先頭に文字列 “`__`” を持つメソッドや属性は初期化や表示といった外部に影響を与えない内部的な処理やクラス内部のみで参照される属性の名前として用いられます。実際、クラス PairOfInts の二つの属性は初期化と内容の表示に関係するものでしたね。そして文字 ‘`_`’ を一つだけ先頭に持つメソッドや変数はシステムとして隠蔽を行いませんが、プログラムの作者がこれらの属性を非公開にする意図がある属性であることを意味しています。逆に文字 ‘`_`’ が属性名の先頭に表われない属性は公開されることを意図した属性であることを意味します。ここで具体的に文字 ‘`_`’ の働きを確認しておきましょう。そのためにここでは函数 `dir()` を用います：

```

>>> dir()
['__builtins__', '__doc__', '__name__', '__package__']

```

*7 引数の型が異なる場合は上書き (override) ではなく多重定義 (overload) と呼びます。

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

この例で用いている組込函数 `dir()` は Python で展開されているクラスとインスタンスを名前のリストとして表示する函数です。ちなみに組込函数については `'dir(__builtin__)` でその一覧をリストの書式で得ることができます。ここでの例では最初の函数 `dir()` の実行では組込函数のリスト `__builtin__` と名前空間のリスト `__name__`, それと読み込みパッケージのリスト `__packages__` と `__doc__` が表示されています。次にクラスの定義を行います。ここで定義するクラス `test` の属性として ‘`x`’, ‘`_y`’ と ‘`__z`’ の 3 個を `class` 文内で与えています。そして `test` クラスのインスタンス `a1` を生成し、各属性値の確認をインスタンス変数の値として行っています。このときに属性 ‘`a.__z`’ にはアクセスできません。と、このように一見すると属性の隠蔽に成功しているように見えますが、函数 `dir()` でクラス `test` の中を見ると属性 ‘`__z`’ だけに文字列 ‘`_test`’ が先頭に置かれています。つまり、文字列 ‘`__`’ を名前の先頭に持つ属性は文字列 ‘`_< クラス名 >`’ が先頭に置かれた名前に Python 内部で自動的に置換されているのです。だから、本来の ‘`a1.__z`’ では値が見えず、その代りに ‘`a1._test__z`’ で属性 ‘`__z`’ の値が調べられたのです。さらに他の属性と同様に、その属性値の書換も可能なのです。このように Python の「**カプセル化**」による属性の隠蔽の方法は不徹底なものなのです。

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以上のものは他に何も持つ

ていません。また、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はできていません。この定義というものは「有理数とは何なのか」という問への回答なので、この「何であるか」という問に対しても、何よりも、まず最初に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を与えなければなりません。互いを比較するという手段を与えることが必要なのです。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なのです。たとえば $1/2 = 2/4$ なので、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自然数の対として前述の安易な手法では別物になってしまいます。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「自然数の対‘(a, b)’と‘(c, d)’が与えられたときに $ad - bc = 0$ 」を満すとき」と定めることができます。ここで「等しい」ということの判断基準が一つ定まりました。では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)’は等しい自然数なので、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とし、同様に‘(a*c, b*c)’は‘(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なのです。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？整数にある関係では、二つの整数が等しいかどうか、それとどちらかが大きいかということ、つまり、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以外の正整数になるので‘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PairOfIntsクラスを基にRationalNumberクラスを構築しましょう：

```
1 from PairOfInts import PairOfInts
2 class RationalNumber(PairOfInts):
3     def __gcd__(self):
4         __a = self.numer
5         __b = self.denom
6         if __a == 0:
7             if __b != 0:
8                 __b = 1
9         elif __b == 0:
10            __a = 1
11        elif __a > __b:
```

```
12     __d = __a / __b
13     __r = __a -__d * __b
14 else :
15     __c = self.conv()
16 return c.__gdc__()
17
18 def __cmp__(self, other):
19     """
20     有理数の合同性と大小関係を判別するメソッド
21     """
22     return cmp(self.numer * other.denom,
23                self.denom * other.numer)
24
25 def conv(self):
26     """
27     逆元を返すメソッド
28     """
29     tmp = self.numer
30     if tmp == 0:
31         self.numer = 1
32         self.denom = 0
33     else :
34         self.numer = self.denom
35         self.denom = tmp
36
37 def reexpr( self ):
38     """
39     有理数の正規化を行うメソッド
40     """
41     if self.denom < 0:
42         self.denom = - self.denom
43         self.numer = - self.numer
44     if self.numer == 0:
45         self.denom = 1
46     elif self.denom == 0:
```

```

47     self.numer = 1
48 else:
49     tmp = gcd(self.numer, self.denom)
50     self.numer = self.numer / tmp
51     self.denom = self.denom / tmp

```

ここでは既存の `PairOfInts` クラスを利用して新たなクラスとして `RationalNumber` クラスの構築を行っています。このように既存のクラスを利用して新たにクラスの定義を行う場合、その新しいクラスの定義で `class 新しいクラスの名前(既存のクラスの名前)` とすることで既存のクラスを利用した新しいクラスの定義が行えます。この方法による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、このことで、ここから定義される `RationalNumber` クラスが `PairOfInts` クラスを継承するものであることを示しているのです。ここで `RatinalNumber` クラスには正規化を行うため、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の `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の `__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 “`>`”, “`<`” と等価 “`==`” がこのクラスでも使えるようになります。

さて、こうして `RationalNumber` クラスをとりあえず構築しましたが、ここで `PairOfInts` クラスを「**基底クラス (base class)**」、基底クラスを基にして構築した `RationalNumber` クラスのことを「**派生クラス (derived class)**」と呼びます。また、`RatinalNumber` クラスは `PairOfInts` クラスを継承していると言い、この継承関係を親子関係に例えて、基底クラスを「**親クラス**」、派生クラスを「**子クラス**」と呼びます。また、この継承関係を単純に上下関係として捉えるとき、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, sub-class**」とも呼びます。このスーパークラスとサブクラスの関係はポルピュリオスの類と種の関係に似た階層構造を持っているのです（§10 参照）。この点についてはクラスは処理の対象が「何であるか」と「どのようなものであるか」を語るものになるので、そのまま範疇論での歴史的な議論が活用できることになるのです。このように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。そしてより下層のサブクラスは上位のクラスよりもより一層、現実の事物に近いので具象性が増し、逆に上位のクラス程、より抽象的になるのです。

さて、この `RationalNumber` クラスに現時点で足りないものがあります。それは四則演算です。このクラスでは比較を行うことはできますが、整数のような四則演算は未定義のために使えないのです。ではその四則演算を追加しましょう：

```

1 def __add__(self, other):

```

```
2     u"""
3         有理数の和を定義するメソッド
4     """
5     numer = self.numer * other.denom + \
6             self.denom * other.numer
7     denom = self.denom * other.denom
8     c = RationalNumber(numer, denom)
9     c.rexpr()
10    return
11
12    def __sub__(self, other):
13        u"""
14            有理数の差を定義するメソッド
15        """
16        numer = self.numer * other.denom - \
17                self.denom * other.numer
18        denom = self.denom * other.denom
19        c = RationalNumber(numer, denom)
20        c.rexpr()
21        return
22
23    def __mul__(self, other):
24        u"""
25            有理数の積を定義するメソッド
26        """
27        numer = self.numer * other.numer
28        denom = self.denom * other.denom
29        return RationalNumber(numer, denom)
30
31    def __truediv__(self, other):
32        u"""
33            有理数の商を定義するメソッド
34        """
35        numer = self.numer * other.denom
36        denom = self.denom * other.numer
```

37

```
return RationalNumber(numer, denom)
```

この RationalNumber クラスにも “__” で開始して “__” で終わるメソッドが四個あります。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 `__mul__()` が商演算子 “/” に対応するメソッドです。そして、これらのメソッドを定義することでクラスの対象同士の四則演算が再定義されることで上書きされ、RationalNumber クラスに新たな演算子として利用できるようになります。

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。

3.3 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表記を Peter Naur が改良したものが基になっています。その構文規則は単純で次で与えられます：

————— 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, $i \in \{1, \dots, n\}$ を適宜利用することで構成されることを意味しています。また、定義_i, $i \in \{1, \dots, n\}$ を分離する演算子 “|” は論理和 “∨” に相当する演算子で、もし、生成規則が一通りしかなければ右辺は 定義₁ のみとなるために記号 “|” は不要になります。

ここで「**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 “::=” 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、ちょうど変数と同様の働きをする記号になるので「**構文変数**」と呼ばれます。非終端記号があれば終端記号ももちろんあり、この「**終端記号**」は逆に演算子 “::=” 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>” のように括りますが、この表記は非終端記号の区別を明瞭にする以上の働きをするものではなく、「言語リファレンス」でも用いられていないために、ここでも省略することにします。

では、BNF の簡単な実例を示すことにしましょう。まず「**数字**」は ASCII 文字の “0” から “9” です。これを BNF で表記するなら

BNF による数字の定義

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

になります。ここで ASCII 文字の “0” から “9” が終端記号であることに問題はないでしょう。終端記号は、この数字の構成で最下層を構成する記号でなければならないので、“0” から “9” の数字はまさに数字を構成する上での素材になっています。

ここで数字ができたとなれば、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”

この定義は上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「**数字**」であるか、「**0 以外の数字**」と「**自然数**」を並べることで構成されるものであることが第一行で定義されています。この定義では演算子 “::=” の左右に「**自然数**」が現われる帰納的な定義となっていることに注意して下さい。さて、ここで「**数字**」は、第二行目で「**“0”**」または「**0 以外の数字**」で構成されることが定義されています。そして最後の行で「**0 以外の数字**」は “1” から “9” までの数字となることが定義されています。整数の場合は整数が正の自然数と負の自然数の和集合として得られることから上記の自然数の BNF を前提にすると次で定義できます：

BNF による整数の定義

整数 ::= 自然数 | “-” 自然数

これらの定義は自然数や整数の字句的な構成方法を BNF を使って表現したものです。では、BNF はこのようなもの以外には使えないのでしょうか？ 実際、BNF は字句的な定義だけではなく構文の定義もできます。そこで今度は「**命題論理式の定義**」を BNF で書き換えてみましょう。ここで命題論理式は次のように定義されます：

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, そして $A \wedge B$ は命題論理式 A と B の論理積を取る操作, それから $A \vee B$ は同様に論理和を取る操作で, 最後の $A \rightarrow$ 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 ::=  $\top$  |  $\perp$  | 論理記号 |  $\neg$  命題論理式 |
    命題論理式  $\wedge$  命題論理式 | 命題論理式  $\vee$  命題論理式 |
    命題論理式  $\rightarrow$  命題論理式
```

この定義では「命題論理式の定義」の (1) から (4) が BNF の演算子 “ $::=$ ” の右辺に順番に現われています. 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, 言葉として BNF には現われてはいませんが BNF の帰納的な構造で表現されています. このように構文の定義にも BNF は使えるのです.

ところで「Python 言語マニュアル」ではこの BNF に一般的な「**正規表現**」を追加した「**拡張 BNF(EBNF)**」を採用しており, この EBNF を字句解析ならば字句の構成について, その他では構文の定義で用いています. 以下に拡張した部分の要旨を纏めておきます:

Python で用いる 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.
- 順序を持った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.

では、もう少し EBNF の具体例を示しておきましょう。まず、先程の整数を EBNF で書換えましょう：

整数の EBNF

整数 ::= ["-"] 自然数

これは角括弧 “[]” を使った例になります。ここで角括弧 “[]” はオプションの記述に相当すると思ってよいでしょう。また、ここで定義で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 からは整数は自然数、あるいは自然数の頭に記号 “-” を追加したものであるという字句的な定義になっています。こうすることで整数の定義がより単純で明晰なものへと置換えられていますね。

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 name という非終端記号の EBNF を以下のように表現しています：

Python での BNF の実例

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

この例では Python の BNF では文字を二重引用符で括るために文字を “_”, “a”, “z” と表記しています。そして、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 “_” で構成されていることを示しています。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羅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。ここでの表記は正規表現から外れた表記で、通常の正規表現であれば ‘a-z’ となるところですが、そこを ‘a ... z’ としています。このように Python の EBNF では文字列 “...” 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 name に該当しないことが判ります。

3.4 字句解析

3.4.1 トークン (token)

Python はプログラム入力時に構文解析器 (parser) を使ってプログラムの字句解析を行い、プログラムをトークン (token) の列に変換します。ここで「トークン (token)」とはプログラム内で意味を持つコードの最小単位で、自然言語での「語彙素」に相当します。Python のトークンには NEWLINE, INDENT, 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。また、空白文字 (Space, TAB や FF) , には、これらのトークンを区切る作用があります。ここで“NEWLINE”は後述の論理行の区切となるトークンであり、トークン“INDENT”とトークン“DEDENT”は INDENT-DEDENT の対で後述の複合文で用いられ、Python のコードの大きな特徴である字下げに関わります。

3.4.2 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トークン“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へと分割されます。さらに論理行は物理行へと分解されます。

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

■物理行 (physical line): 論理行をプログラム上の記載形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列です。ここで行末端文字は計算機環境によって異なり、UNIX 環境では ASCII 文字の“LF(行送り)”, Windows 環境では ASCII 文字の“CR+LF”, MacOS では ASCII 文字の“CR(復帰)”になります。しかし、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“\n”^{*8}を用いた文字リテラルで表現します。なお、特殊な物理行として次の「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で終えます^{*9}。なお、注釈は論理行を終らせる働きを持ちます。このことから判るように Python の注釈

^{*8} 日本語 Windows で用いられている文字コードである SHIFT_JIS では、記号“\”は勝手に記号“¥”で置換されています。そのために多くの書籍ではこれらの記号を同一視した扱いをしていますが、UTF-8 等の文字コードで全く別の記号であるために、この本では記号“\”を記号“¥”で置換えることを敢てしません。のために日本語 Windows 環境では適宜、記号“\”を記号“¥”で置換えて解釈し直して下さい。

^{*9} 記号“#”は後述のリテラルには含まれていません。

は一つの論理的な区画に対する注釈であるべきことを意味します。そのために後述の行継続を行うときには注意が必要です。

- **エンコード宣言:** 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ます。プログラムの先頭の一行目か二行目に置かれ、正規表現 ‘coding[=:]\s*([-\\w.]+)’ に適合するものでなければなりません。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ドであるかが明示的に指定されます。なお、この宣言がなければプログラムに記載された文字は、エンコードを指定する接頭辞を持たない限り、自動的に ASCII 文字として扱われます。
- **空行 (blank line):** Space, TAB, FF(Form Feed), あるいは注釈だけで構成された論理行です。これらの空行に対しては字句解析でトークン “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として用いられますが、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 (Space あるいは TAB) の個数で字下げの水準が計算され、この字下げの水準によって入力文のグループ化が行われます。この字下げは Python の特徴の一つです。ここで Python 拡張提案 (PEP) に含まれるプログラミング様式を規定する PEP-8 によると、字下げは一段階 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。

■**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で置換することができます。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。ここで継続文字 “\” に続けて注釈を追加することはできません。なぜなら、注釈は論理行を終える働きがあり、その結果、注釈の前後で二つの論理行として分割されることになってしまいます。また、注釈自体も継続文字を使って分割することができません。

なお、改行の例外的な規則として、丸括弧 “()”，角括弧 “[]”，波括弧 “{ }” の内部で改行を行う際に継続行文字 “\” の併用を必要としません。同時に、このこれらの記号で括られたその内部では注釈を続けて記載することもできます。

3.4.3 識別子とキーワード

「**識別子 (identifier)**」は Python で名前に用いられ、その 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 文字の羅列であり、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 文字以外の各言語の文字のように上述の識別子の EBNF の文字に該当しない文字を含む文字の羅列はエンコード宣言の有無と無関係に識別子になり得ません。

Python の識別子には**キーワード**と呼ばれ、予約語として扱われるため通常の識別子として使えないものがあります：

Python のキーワード

| | | | | | | | |
|--------|----------|--------|---------|--------|--------|--------|-------|
| 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 | execr | global | is | pass | try | |

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を Python の「**名前**」として使うことができます。名前で重要な機能は「**名前への束縛**」によって名前とオブジェクトやオブジェクトを実体化したものへの対応付けが行われ、束縛が行われた名前によって「**名前空間**」が構成されることがあります。なお、キーワードは Python の文等で用いられますが、名前として使うことができません。また、オブジェクトやインスタンスの参照では名前を介して行われるため、何も束縛されていない名前を参照しようとしたときに **NameError 例外** が生じます。

3.4.4 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**数値リテラル**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 | 数値リテラル
```

Python のリテラルは全て**変更不能なデータ型**であるため、ある値を持つ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。そのために同じリテラルを値に持つオブジェクトであっても、オブジェクトとして一致するとは限らないことがあります。

文字列リテラル

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**」はプログラム中にエンコード宣言がなく、リテラル自体にも接頭辞がなければ ASCII 文字になります。また、文字リテラルに接頭辞を用いることでエンコードを明示的に行なうことができます。たとえば、文字リテラルの先頭に ‘u’ や ‘U’ といった接頭辞を配置することで後続する文字列リテラルが UNICODE 文字列型として扱われます。ここで注意することとして、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。

Python の文字列リテラルには引用符の個数による型の区別があります。まず、**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、「''' 三毛猫'''」と「"""虎猫""」が長文字列です。なお、長文字列から後述の「**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。そのため文書文字列の中では改行や単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と、その箇所で長文字列が終了することです。

だから

```
1 """
2 そんな訳で、長文字列の中では
3 "こんな使い方"
4 や
5 'こんな使い方'
6 それに改行をこんな風に
7
8
9   """ また、引用符も長文字列を構成する際に用いた
10  引用符でなければ続けて三回使っても構いません"""
11
12
13 といったことを記入しても構わないのです。
14 """
```

と、単引用符や二重引用符、それと行末端文字を含む、上で示す文字列は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場合は 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 ‘\n’ で置き換えることで出力文字の改行が行えます。

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。Python 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。

数値リテラル

数値リテラルの EBNF を以下に示します:

数値リテラルの EBNF —

```
数値リテラル ::= 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 で示すように数値リテラルには整数 (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 種類のリテラルが

あります。ここで整数の表現には整数リテラルと長整数リテラルの二種類があり、ここで整数リテラルが符号付き 32bit 整数、長整数は計算機のメモリに依存するものの任意桁数の整数を表現します。ただし、複素数は実数と虚数の和で生成されるためにそれ自体は数值リテラルには該当しません。

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進表示 | 16進表示 | 2進表示
10進表示 ::= 零以外の数字 数字* | "0"
8進表示 ::= "0" ("o"|"O") 8進数 +
16進表示 ::= "0" ("x" | "X") 16進数 +
2進表示 ::= "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。なお、長整数のリテラルでは末尾に小文字 ‘l’ を置くことを許容していますが、小文字 ‘l’ は数字 ‘1’ と非常に紛らわしいために大文字 ‘L’ を使うことが推奨されています。

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だからです。この点は次に言及することになる複素数も同様です。

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")
```

虚数リテラルは浮動小数点との組合せで構成されたリテラルです。そして、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、Sage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Sage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Sage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
-1.00000000000000
```

最初にSage上の函数type()で‘I’の型を調べていますが、結果の’sage.symbolic.expression.Expression’から‘I’が多項式の仲間の数であることが判ります。この理由ですが、まず、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1変数多項式で最小の字数の多項式に対応させることができます。この純虚数では $x^2 + 1$ がその最小多項式になります。そして、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しまいましょう。こうし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るでしょう。これがSageの‘I’が多項式の仲間となる理由です。ところが、もう一方の‘1J’を函数type()で調べた結果は’sage.rings.complex_number.ComplexNumber’になって先程の‘I’と型が異なっています。この虚数リテラルは先程の‘I’が整数係数の多項式に対応するものと異なり、近似的な数である浮動小数点数に関係する数であ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Sageでは近似的な数として虚数リテラルを用いるのか、代数的数とな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るのです。

3.4.5 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

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

この表の上段が算術演算子、中段が論理演算子、そして、下段が比較の演算子になります。ここで演算子“<>”と演算子“!=”は同じ意味ですが、演算子“<>”は時代遅れの表記であるために演算子“!=”の利用が推奨されています。

なお、Sage では冪乗の演算子として演算子“^”が使えます。ちなみに Python の数式処理ライブラリである SymPy でも Python と同様に冪乗の演算子が“**”だけで、演算子“^”は後述するようにビット単位の演算子で、冪乗と全く異った作用をします。Sage で冪乗の演算子として演算子“^”が導入されている理由として、他の多くの数式処理で冪乗の演算子として演算子“^”が用いられていることもあるでしょう。

次のトーカンは Python の「区切文字 (delimiter)」として用いられています：

Python の区切文字 (delimiter)

| | | | | | |
|----|----|----|-----|-----|-----|
| (|) | [|] | { | } |
| , | : | . | ' | = | ; |
| += | -= | *= | /= | //= | %= |
| &= | = | ^= | >>= | <<= | **= |

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で用いられます。そして、下段の累積代入演算子は区切文字として振舞います。また、单引用符、二重引用符、記号“#”と記号“\”といった ASCII 文字は、他のトーカ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 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。このことは iPython をフロントエンドに用いている Sage でも同様です。

3.5 データモデル

3.5.1 Python のオブジェクトの概要

Python プログラムのデータは全てオブジェクトです。クラスやそれに属する個体もオブジェクトで、オブジェクト間の関係を表現する函数やメソッドもオブジェクトになります。

クラスに基くオブジェクト指向プログラミング言語においてオブジェクトの生成を行う函数やメソッドのことを「**構成子/コンストラクタ (constructor)**」と呼びます。この構成子によるオブジェクトの生成ではオブジェクトの「**メモリの割当 (allocation)**」と属性の設定等の「**初期化 (initialization)**」が同時に行われます。Python ではこの構成子に相当するメソッドとして `__new__()` と `__init__()` の二つのメソッドがあり、前者のメソッド `__new__()` でオブジェクトの生成、後者のメソッド `__init__()` でオブジェクトの初期化を行います。そして、メソッド `__new__` が呼び出されると自動的にメソッド `__init__` も呼び出されますが、メソッド `__init__` が呼び出された時点でオブジェクトは既に生成されています。このようにメソッド二つで構成子の機能を持っており、特にメソッド `__init__` が構成子に最も似た働きをしていると言えますが、C++ の構成子とは異なります。また、クラスの定義で構成子に対応するこれらの二つのメソッドを記入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用いされることになるからです。

次に生成したオブジェクトの消去を行うときに呼出されるメソッドのことを「**消去子/デストラクタ (destructor)**」と呼びます。Python ではメソッド `__del__()` が消去子に該当するメソッドですが、Python でオブジェクトの消去は到着不可能になった時点から、いずれ塵収集で何れ行われるというもので、不要になった時点で即時に削除されてメモリが解放されるという性質のものではありません。このことから Python は C++ のような構成子や消去子を持っていないと言えます。

ここで Python のオブジェクトには「**同一性値 (Identity)**」、「**型 (Type)**」と「**値 (Value)**」に対応する値を持ちます。まず、オブジェクトの同一性値は生成したオブジェクトのメモリ上の番地に対応する整数値、あるいは長整数値として表現され、生成したオブジェクトの固有の値になるために変更することができません。このオブジェクトの同一性は函数 `id()` を使って調べることが可能で、二つのオブジェクトが同一性であるかは演算子 ‘`is`’ を使って調べることができます。

Python のオブジェクトは、そのオブジェクトの値が「**値が変更可能 (mutable)**」なものと「**値が変更不能 (immutable)**」ものの二種類に分類できます。この値が変更可能/不能という性質はオブジェクトの型で決定される性質です。ここでオブジェクトの型は函数 `type()` で調べすることができますが、このオブジェクトの型はインスタンス化された時点で定まるもので、インスタンス化したあとでオブジェクトの型を変更することはできません。

ここで簡単な例を示しておきましょう：

```
>>> a = b = []
>>> b.append(128)
>>> a is b
True
```

```
>>> id(a)
3073670732L
>>> id(b)
3073670732L
>>> c = [128]
>>> id(c)
3073670700L
>>> print a, b
[128] [128]
```

この例では ‘`a = b = []`’ で空リストのオブジェクト `[]` として生成して名前 `a`, `b` に束縛させています。そのために名前 `a` と `b` で参照されるオブジェクトの識別子は一致しています。この生成したオブジェクト `[]` は変更可能なオブジェクトであるリスト型であるために `'b.append(128)'` で名前 `b` で参照されているオブジェクトに ‘128’ が追加されます。ここで、このオブジェクトは名前 `a` からも参照しているために、名前 `a` を評価すると ‘128’ が追加されたオブジェクトの値が得られます。このことは ‘`a is b`’ で調べても `True` が返却されることやオブジェクト固有の同一性値を返却する函数 `id()` で双方が同じ値となることからも判ります。ここで名前 `c` にリスト `[128]` を束縛させたものは同じ値でも同一性値は名前 `a`, `b` のオブジェクトとは異っていますね。

では、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？次に変更不能な型のオブジェクトである数値リテラルを使った例で確認してみましょう：

```
>>> c = d = 128
>>> print id(c), id(d)
148353652 148353652
>>> c = 256
>>> print id(c), id(d)
148356068 148353652
>>> print c, d
256 128
```

この例では最初に ‘`c = d = 128`’ でオブジェクト ‘128’ を名前に束縛させています。この時点で `c` と `d` の識別子の値が一致することが函数 `id()` の結果から判ります。次に名前 `c` に ‘`c = 256`’ によって新たにオブジェクトの束縛を行なうと、その前に束縛したオブジェクト ‘128’ は変更不能なオブジェクトであるためにオブジェクトの値を変更させるのではなく、新しいオブジェクト ‘256’ を名前 `c` に束縛されることになります。ところで、一方の名前 `d` から参照されるオブジェクトには、この束縛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。このことは識別子の値を函数 `id()` による結果を比較することからも判ります。ここで名前 `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’への参照が途切れてしまいます。もしオブジェクト‘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行えなくなります。この状態を「**到達不能の状態 (unreachable)**」と呼びます。Pythonではオブジェクトを明示的に破壊することはできませんが、オブジェクトが到達不能な状態になると、やがて、**塵収集 (garbage-collection)**によってオブジェクトの回収処理が行われます。なお、オブジェクトによってはファイルやウィンドウのような外部リソースへの参照を行うものがあり、これらが不要となっても塵収集が実行される保証がないために外部リソースを解放するメソッド（大抵はメソッドclose()）を用いて明示的に解放する必要があります。

オブジェクトの中には他のオブジェクトへの参照を行うオブジェクトがあります。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。なお、コンテナがオブジェクトとして変更不可能な型だとしても、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よってコンテナの値が変化することがあります。この場合はコンテナが参照しているオブジェクトの名前自体が変更されていないために「**変更不能**」という言葉と矛盾することはありません。要するに箱が同じ箱であれば中身は問わないということです。

3.5.2 Python組込の型

ここではPythonに標準で組込まれている型について述べます。この組込まれている型の幾つかには「**特殊属性**」と呼ばれる型固有の属性があります。これらの属性の詳細については「Python言語リファレンスマニュアル」を参照して下さい。以下にPythonの組込のオブジェクトの型を挙げておきます：

オブジェクトの型

| | | | | |
|----------------|----------|----|--------|-------|
| None | Ellipsis | 列 | 対応付け集合 | クラス |
| NotImplemented | 数 | 集合 | 呼出可能型 | モジュール |

これらの型を持つオブジェクトについて順番に解説をします。

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、その値は真理値‘False’として扱われ、この型を持つオブジェクトは唯一Noneしか存在しません。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’ で両者を比較すると、None 型を持つオブジェクトが一つしか存在し得ないために両者の識別子が一致し、そのため ‘True’ が返却されています。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 と同様の性質を持ちます。実際、圈 $\mathbb{S} \approx \sim$ の矢は通常の写像が対応します。ここで $a, b \in \text{ob } \mathbb{S} \approx \sim$ に対して矢 $f : a \rightarrow b$ が存在したとき、 f を順序対の集合 $\{(x, f(x)) | x \in a\}$ と見做すことができます。ここで $a = \emptyset$ であれば $x \in \emptyset$ となる x が存在しないので $f = \emptyset$ でなければならぬことが判ります。つまり、 \emptyset は圈 **Set** の対象でもあり、矢でもあり、ここでのオブジェクト None と同様の性質を持つことが判りますね。

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も唯一です。組込の名前 NotImplemented で参照され、その値は真理値 ‘True’ として扱われます。

Ellipsis

この型は拡張スライス構文で添字全体を示す省略記号 ‘...’ というリテラルが持つ型です。この型は单一の値しか持たず、逆に、この型を持つオブジェクトも唯一で名前 Ellipsis で参照が行えます。また、名前 Ellipsis で参照される値は真理値 ‘True’ として扱われます：

```

>>> 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])

```

ここでスライス操作と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のことです。ここでの例では最初に numpy パッケージから函数 arange() の読込を行い、それから名前 a で参照される NumPy の一次元配列を生成してメソッド reshape() で 2×8 の配列へと大きさを変換しています。この配列は 2 次元なの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るのです。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

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

という処理を行っているのです。このスライス操作に現われる ‘...’ 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 になります。ただし、Python や MATLAB での記号 ‘:’ は一次元の配列の省略記号ですが、Python の省略記号 ‘...’ は一次元に限定されず、Yorick の「ゴム添字」のように多次元である点で大きく異なります。

数 (numbers.Number)

数値リテラルで生成されたり、算術演算や組込の算術函数から返却されるオブジェクトです。数オブジェクトは変更不能のオブジェクトであり、一度、値が生成されると二度と変更されることはありません。ここで Python 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けられます：

- 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。演算の結果、この整数型の範囲を越えると自動的に長整数型で結果が返却されます。
- 長整数型 (long integer)： 長整数リテラルで生成され、計算機の記憶容量に依存する任意桁数の整数が扱えます。整数型から長整数型の変換は函数 long()、逆に長整数型から整数型への変換は函数 int() で行えますが、この長整数型からの変換で値が整数型の範囲内でなければ長整数型のままで返却されます。
- ブール型 (boolean)^{*10}： 真理値の ‘True’ と ‘False’ で構成されます。算術演算では ‘True’ が整数型の ‘1’、‘False’ が整数型の ‘0’ として扱われます。

^{*10} Bool 型は 19 世紀の数学者 Boole に由来しますが、Bool と何故か最後の “e” 無しで表記されます。

- 実数型 (numbers.Real(float)): 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。ここで **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**。
- 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。複素数 z の実部は $z.\text{real}$, 虚部は $z.\text{imag}$ で取り出せますが、複素数型の性質上、これらは実数型、すなわち、倍精度の浮動小数点数です。そのために近似的数であることに注意が必要です。なお、Sage では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なので浮動小数点数のような近似の数でないことに注意が必要です。

列

有限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ができる集合のことです。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: $x \geq_{\text{order}} y \quad \text{かつ} \quad y \geq_{\text{order}} z \quad \Rightarrow \quad x \geq_{\text{order}} z \quad x, y, z \in S$

ここで Python の列の濃度、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。

列に対しては**スライス操作**と呼ばれる部分集合の生成操作があります。この操作は MATLAB 系の言語でお馴染の操作で、長さ n の列 a に対して $i < j$ を充す二つの正整数 $i, j \in \{0, \dots, n - 1\}$ を添字として $a[i:j]$ で列 a の $i + 1$ 番目から j 番目の成分を持つ部分列を生成します。列の型によっては**拡張スライス操作**と呼ばれる刻幅指定のスライス操作が行えることもあります。たとえば、文字列 '123456789' に対して刻幅 2 で先頭の文字から 8 番目までの文字で構成される部分列を取り出すときに '123456789'[0:8:2] で部分列 '1357' を取り出すことができます。つまり、[0:8:2] によって自然数の列 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

部分列 '1357' になるのです¹¹。この拡張スライス操作で用いられる文字リテラル '...' が Ellipsis 型のオブジェクトへの参照になります。

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：記号 “,” で区切られた任意の Python オブジェクトの列を角括弧 “[]” で括ったものから生成されるオブジェクトの型です。
- ByteArray 型：構成子 `bytearrays()` で生成されるオブジェクトの型です。
- 文字列型：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。
- 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。
- 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 **singleton** と呼ばれます。

集合

列と異なり順序を持たない不变なオブジェクトの有限集合を表現します。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各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：

- Sets 型：変更可能な型で、組込の構成子 `set()` で生成されます。
- Frozen Sets 型：変更不能な型で、組込の構成子 `frozenset()` で生成される型です。ハッシュ可能のために別の集合型の成分になったり、辞書の鍵にすることができます。

対応付け集合 (mapping)

列を一般化したオブジェクトとしての性質を持ちます。ここで列の成分の取り出しへは列 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、対応付け集合については、この添字の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、対応付け集合 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。また、対応付け集合の濃度は列と同様に函数 `len()` で調べることができます。ただし、現時点では Python に組込

¹¹ 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。

まれている対応付け集合は辞書型のみです。

- 辞書型 (dictionary): 変更可能な型です。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

函数として呼出操作が可能なオブジェクトの総称で以下のものがあります：

- ユーザ定義函数型: 函数定義で生成されるオブジェクトです。このオブジェクトの呼出は函数定義で用いた仮引数の列と同じ長さ^{*12}の列を引数にして行われます。そして、函数オブジェクトは任意の属性の設定や取得ができます。なお、函数定義に関する情報は函数の後述の「**コード オブジェクト**」から入手できます。
- ユーザ定義メソッド型: クラスやクラス インスタンス、あるいは None を**一次語**とし、記号“.”で任意の呼出可能なオブジェクトと結合させることで生成されるオブジェクトです。
- 生成函数型: yield 文を用いる函数やメソッドのことを生成函数と呼びます。この類の函数は return 文を用いて都度、値を返却するものではなく、スライス処理によって値の参照が行える反復子 (iterator) オブジェクトを返却します。
- 組込函数型: 組込函数オブジェクトは C の函数のラッパになります。このような函数の例としては函数 len() や函数 math.sin()^{*13}を挙げておきます。
- 組込メソッド型: 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っています。
- クラスタイプ型: 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、このときには「**ファクトリ クラス**」として振舞います。ここでメソッド __new__() の上書き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: 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 オブジェクトとはまた別の型なので注意が必要です。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。

^{*12} 引数の個数を函数のアリティ (arity) と呼びますが、この仮引数の列の長さは函数のアリティと一致します。

^{*13} 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。

- クラス インスタンス型: 古典的クラスのクラスにメソッド `__call__()` があるときには、そのクラスは呼出可能型になります。

モジュール

`import` 文で Python にプログラムを記述したファイル読み込みが行われることで生成されます。なお、この型のオブジェクトに初期化で用いられる後述の「**コード オブジェクト**」が含まれていません。

クラス

Python 2.X には「古典的なクラス」である「クラス オブジェクト」と「新しい様式」である「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、各属性への参照で用いられます^{*14}。古典的クラス型はクラスタイプ型と違い、メタクラスが扱えず、引数無しで `super()` で基底クラスへの属性等の参照が行えないといった性質を持っています。また、Python 2.x では基底クラス ‘`object`’ を何らかの形で継承しなければ古典的クラス型、継承するとクラスタイプ型として定義されます：

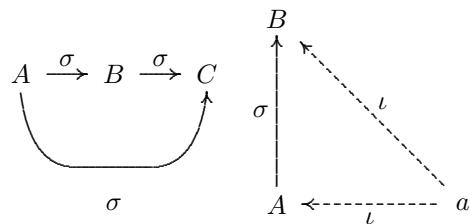
```
>>> class TEST1:
...     pass
...
>>>
>>> class TEST2(object):
...     pass
...
>>>
>>> type(TEST1)
<type 'classobj'>
>>> type(TEST2)
<type 'type'>
>>> a=TEST1()
>>> b=TEST2()
>>> type(a)
<type 'instance'>
>>> type(b)
<class '__main__.TEST2'>
```

この例で示すように Python 2.x では引数無しでクラスの定義を行うと古典的クラスになります。この古典的クラスは関数 `type()` で ‘`classobj`’ と表示されるクラス オブジェクトになり、そのインスタンスは関数 `type()` で ‘`instance`’ と表現されるインスタンス オブ

^{*14} Python 3.X ではクラス オブジェクトが廃止され、クラスタイプ型のみになります。

ジェクトになります。それに対して ‘object’ を継承することを明示的に定義したクラスや クラスタイプ型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に なり、そのインスタンスは古典的クラス型の ‘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。

ちなみに継承する側のクラスのことを「**派生クラス**」と呼び、逆に継承される側のクラ スのことを「**基底クラス**」と呼びます。基底クラスは派生クラスよりもより普遍的なク ラスで、より上位のクラスです。この継承関係を親子関係にたとえると「**基底クラス**」が 「親」に相当するクラスになり、基底クラスを継承する「**派生クラス**」が「子」に相当す るクラスになります。さらに、この継承関係を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は部分集合、subset に対応するので「**サブクラス**」、大本を「**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、 $\alpha \xrightarrow{l} C$ で α が C のインスタンスであることを 表現します。このときに次の図式が可換になります：



まず、左上の図式は 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 $\xrightarrow{\sigma}$ が結合律を充すことを意味します。また、 右上の図式が可換になるということは、クラス A がクラス B のサブクラスのときに A の インスタンス a はクラス B のインスタンスになることを意味します。なお、クラスの定 義では自分自身を用いる循環的定義は許容されません。そのため $A \xrightarrow{\sigma} A$ は現実的には生 じず、このようなクラスの定義を行おうとすればエラー (Name.Error) になります。また、 クラス間の継承の関係はポルピュリオスのエイサゴーゲーにおける類と種の関係と同様で す。つまりクラスは計算機で扱おうとする対象に対して「それが何であるか」と「どのよ うなものであるか」という問に対する回答だからです。

一般的にクラス属性の参照は、クラス名が C で属性が x のときに ‘ $C.x$ ’ で行います。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**」に

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あり、ここでは $\mathcal{L}(C)$ 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 $\bar{L} \stackrel{\text{def}}{=} \text{head}(L), \underline{L} \stackrel{\text{def}}{=} \text{tail}(L)$ と略記します。それから語 $B_1 \dots B_m$ と語 $C_1 \dots C_n$ が与えられたとき、語 $B_1 \dots B_m$ に含まれる各 $B_i (1 \leq i \leq m)$ を語 $C_1 \dots C_n$ から取り除いた語を $C_1 \dots C_n \setminus B_1 \dots B_m$ と表記することにします。たとえば、 $abcd \setminus bd$ 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 ‘ $\dots W(i-1)W_iW(i+1)\dots W(j-1)W_iW(j+1)\dots$ ’ は語 ‘ $\dots W(i-1)W_iW(i+1)\dots W(j-1)W(j+1)\dots$ ’ と検索順序として一致することになります。このように後続の一一致する語を除いたものとの関係を ‘ $\dots W(i-1)W_iW(i+1)\dots W(j-1)W_iW(j+1)\dots \searrow \dots W(i-1)W_iW(i+1)\dots W(j-1)W(j+1)\dots$ ’ と表記します。そして関係 ‘ \searrow ’ によって語 L はより短い語へと置換えることが可能です。そして、この短縮化には下限があるため必ず極限が存在します。この語 L の関係 ‘ \searrow ’ の極限になる語をここでは \underline{L} と表記します。そして作用素 \mathcal{L} は次の性質を持ちます:

———— 作用素 \mathcal{L} の性質 ————

1. $\mathcal{L}(C) = C$
2. $\mathcal{L}(C_0(C_1)) = C_0 + \mathcal{L}(C_1)$
3. $L_1 \searrow L_0$ のとき $\mathcal{L}(L_1) = \mathcal{L}(L_0)$
4. $\mathcal{L}(C(B_1, \dots, B_n)) = C + \mathcal{M}(\mathcal{L}(B_1), \dots, \mathcal{L}(B_n))$

最初の 1. は継承関係を持たないクラス C のときに MRO は C のみであるということを示します。つぎの 2. は $C_0 \xrightarrow{\sigma} C_1$ のとき、つまり、クラス C_0 が C_1 の派生クラスのときに最初にクラス C_0 を検索し、それからクラス C_1 の検索順序に従うということを意味し、つぎの 3. は関係 ‘ \searrow ’ で作用素 $*$ の値は不变であることを示し、そして、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、継承関係を越る MRO を基本に、多重継承のあるクラスにて継承の

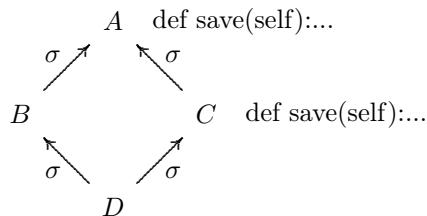
親達を示すタプルをそのまま用いて基底クラスに検索順序を入れるというものです。つまり、クラスの属性で基底クラスを示すタプルの左側から順番に先祖を辿る方法で行われるので、「**深さ優先、左から右の順番規則 (left-to-right depth-first rule)**」と呼ばれるものになります。この操作を表現する作用素 \mathcal{L} は次の性質を持ちます：

———— 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} の値は不変であることを示しています。そして最後の c. は引数の最も左側にある経路を外に出し、その語に含まれるクラス名を他の引数から除去してゆくという処理方法を示しています。もし、引数の語に共通するクラス名がなければ \mathcal{M} は語に対する和になるだけです。

Python の古典的クラス型でメソッド検索で用いられる順序は、MRO です。しかし、この手法は多重継承があるときに有効に動作しない問題があります。このことを次のクラス A, B, C, D の関係が次の菱形状になる図式を使って解説しておきましょう：



この図式は、クラス D はクラス B と C の双方を継承する多重継承の関係にあり、同時にクラス B と C はクラス A の派生クラスであることと、クラス A と C でメソッド `save` が定義されていることを示しています。ここで各クラスが古典的クラス型のときにクラス B やクラス C でメソッド `save` を利用しようとすればクラス A のものがそのまま用いられ、クラス C で上書きされたメソッド `save` はそのままではクラス D で用いられることはありません。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()` が用いられることになって、クラス A の派生クラス C で再定義されたメソッド `save()` が用いられないということになります。つまり、より近い側のメソッドが用いられないという問題が生じることになります。そのためクラスタイプでは「**C3 MRO(Method Resolution Order)** ^{*15}」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 言語に導入されています。この手法は先程の 3. の計算手順の作用素 \mathcal{M} を作用素 \mathcal{M}_{c3} で置換えて、次のようにまとめることができます：

C3 での作用素 \mathcal{L} の性質

1. $\mathcal{L}(C) = C$
2. $\mathcal{L}(C_0(C_1)) = C_0 + \mathcal{L}(C_1)$
3. $L_1 \searrow L_0$ のとき $\mathcal{L}(L_1) = \mathcal{L}(L_0)$
- 4'. $\mathcal{L}(C(B_1, \dots, B_n)) = C + \mathcal{M}_{c3}(\mathcal{L}(B_1), \dots, \mathcal{L}(B_n))$

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：

^{*15} <https://www.python.org/download/releases/2.3/mro/> や PEP-253 を参照

\mathcal{M}_{c3} の計算手順

1. $i = 1$ とする.
2. $h_i = \overline{L_i}$ とする.
3. $j \neq i$ に対して $h_i \notin \underline{L_j}$ のときに $k \in (1, \dots, n)$ に対して $\overline{L_k} = h_i$ であれば, \mathcal{M}_{c3} の引数にある L_k を $\underline{L_k}$ で置換し, $h_i + \mathcal{M}_{c3}(L_1, \dots, L_n)$ を作用素 \mathcal{M}_{c3} の結果として返却する.
4. $h_i \in \underline{L_j} (i \neq j)$ のとき $i \neq n$ ならば $i = i + 1$ として 2. に戻る. もし $i = n$ であればエラーを出力して処理を終える.

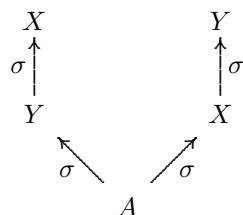
この作用素 \mathcal{M}_{c3} は引数の語に共通するものが何もなければ最初の作用素 \mathcal{L} を拡張したものと同様に語の和として作用します. しかし, 共通する語が現われたときの処理がクラスの階層を合致させる働きになります. このことを先程の菱形状の継承関係で C3 MRO を計算することで確認してみましょう.

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できます.

たとえば次の継承関係を想定しましょう:



この継承関係は $X \xrightarrow{\sigma} Y$ かつ $Y \xrightarrow{\sigma} X$ と, クラスの定義では相互参照的な関係にあるために, いわゆる循環的な定義になっているという問題があり, このような定義は Python では許されません. このような場合でも MRO では AYX が得られますが, 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 の処理は行えず, 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, 最終的に作用素 \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.

下位のクラスの属性に対して新たに値の束縛を行うことで, そのクラスの辞書が更新されますが, その結果, 基底クラスの辞書までも更新されることはありません. しかし, 上位のクラスの属性を変更すると下位のクラスの辞書に書き換えが生じます.

クラスには**特殊属性 (special attribute)** と呼ばれる属性があります. この特殊属性には `__name__` にクラス名, `__module__` にモジュール名, `__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__` にクラスのことを説明するための文書文字列がそれぞれ束縛されています. そして, 各クラスの(公開)属性に何があるかを函数 `dir()` で調べることができます.

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あり, 辞書で実装された名前空間を持っているので最初の属性参照はここから開始します. 属性参照にて辞書内で属性が見当らないもののインスタンスのクラスに該当する属性名があるとき, そのインスタンスが含まれるクラス属性に検索領域が広げられます. このときの検索順序は基底クラスのタプルで, その左側のクラスから右側のクラスへと検索が行われます(MRO). また, 属性の代入や削除によってインスタンスの辞書が更新されますが, それに伴ってクラスの辞書が更新されることはありません.

ファイル

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 インタプリタが内部で用いる型が公開されています:

- コード オブジェクト: 「**バイトコード (bytecode)**」を表現するオブジェクトです。このバイトコードは Python のインタプリタ内部のデータ形式で、Python プログラムはこのバイトコードにコンパイルされたのちに実行されます。なお、バイトコードは修飾子が ‘.pyc’ や ‘.pyo’ のファイルに蓄えられ、実行の都度、コンパイルを行わなくともよくなっています。
- フレーム オブジェクト: 「**実行フレーム**」を表現するオブジェクトです。ここで実行フレームは後述のトレースバック オブジェクトに現われます。
- トレースバック オブジェクト: 「**例外スタックトレース**」を表現するオブジェクトです。まず、トレースバック 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、その時点のトレースバック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: 「**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: 組込の構成子 staticmethod() から生成されるオブジェクトで、他のオブジェクトのラッパーとしての作用があります。
- クラスメソッド オブジェクト: 組込の構成子 classmethod() から生成されるオブジェクトで、静的メソッドと同様に他のオブジェクトのラッパーになってクラスやクラスインスタンスから、そのオブジェクトを取出す方法の代替になります。

3.5.3 特殊メソッド

特殊メソッドと呼ばれる特定の名前のメソッドをクラスに定義することで特殊な構文や特定の演算をクラスに実装することができます。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、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、オブジェクトの比較では「**拡張比較**」と呼ばれる一群のメソッドがあります。これらのメソッドは、これから定義するクラスの二つのインスタンスが等しいかどうか、同一クラスのインスタンス間の大小関係を判断するメソッドで、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 “!=”，また、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 “==” を再定義したからといつても、その否定の “!=” が自動的に定義されないため演算子に “!=” を使う可能性があるならば等号と併せて再定義を行う必要があります。

ここでは特殊メソッドを挙げておきますが、その際にクラス名を便宜的に ‘cls’、オブジェクト名も同様に ‘obj’ と表記します。そして、引数をまとめて ‘args’ と表記します。なお、実用の際にはクラスやオブジェクトの名前で置換し、変数を適宜用いることにします。

インスタンスの生成と削除に関するもの

■`obj.__new__(cls [,args...])` クラス `cls` のインスタンス生成のために呼び出される静的メソッドです。インスタンスの生成を要求されているクラスの名前を第一引数に、残りの引数をオブジェクトの構成子に該当するメソッド `__init__()` に引渡すメソッドで、後述のメソッド `__init__()` と併せて C++ の構成子に似た動作をします。

クラス `cls` を継承するクラスのインスタンスを生成するためのメソッドの上書きでは ‘`super(継承するクラス, cls).__new__(cls[, ...])`’ に適切な引数を指定することで、まずクラスの基底クラス `cls` のメソッド `__new__()` を呼出して、継承クラスのインスタンスとして必要な変更や拡張を加えた上でインスタンスの返却を行います*16。

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 ‘`__init__(self[, ...])`’ によってメソッド `__init__()` が呼出されますが、メソッド `__new__()` がクラス `cls` のインスタンスを返却しないときはメソッド `__init__()` の呼出しが行われません。このメソッド `__new__()` の目的は変更不能の型の下位のクラス (=サブクラス) のインスタンス生成にて補正するためです。

■`obj.__init__(self [, ...])` C++ の「**構成子 (constructor)**」に似た動作を行なうオブジェクトの初期化に関わるメソッドで、インスタンスが生成された時点で呼び出されてインスタンスの初期化を行います。クラスへの引数はこのメソッドの引数になります。一般に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、「**構成子は値を返却してはならない**」という制約があるため、このメソッド `__init__()` も同様に、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` が生じることに注意が必要です。

■`obj.__del__(self)` C++ の「**消去子 (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ます。なお Python のオブジェクトの消去は到達不可能になった時点から、やがて塵収集で不要なオブジェクトとして回収され、C++

*16 Python 3.x では函数 `super()` の引数の省略が可能です。

の消去子のように不要になった時点でオブジェクトを削除する消去子とは異った仕組です。そのために C++ の消去子と同様の働きをする消去子を Python は持たないと言えます。一般的に基底クラスがメソッド `__del__(self)` を持っているとき、派生クラスのメソッド `__del__(self)` にて明示的に基底クラスのメソッド `__del__(self)` を呼出してオブジェクトを消去するようにしなければなりません。

属性値の表示に関するもの

■`obj.__repr__(self)` 組込関数 `repr()` や文字列への変換時に呼出され、オブジェクトを Python のフロントエンド側で表示する「公式の文字列」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、内部表現をより判別し易い文字列として表示することができます。

■`obj.__str__(self)` 組込関数 `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文字列オブジェクトでなければなりません。

インスタンスの比較に関するもの

■`obj.__lt__(self, other)` 拡張比較のメソッドで、比較の演算子 “`<`” に対応します。このメソッドを上書きすることでクラスに属するオブジェクトに適した比較の演算子 “`<`” を導入することができます。

■`obj.__le__(self, other)` 拡張比較のメソッドで、比較の演算子 “`<=`” に対応します。このメソッドを上書きすることでクラスに属するオブジェクトに適した比較の演算子 “`<=`” を導入することができます。

■`obj.__eq__(self, other)` 拡張比較のメソッドで、比較の演算子 “`==`” に対応します。このメソッドを上書きすることでクラスに属するオブジェクトに適した比較の演算子 “`==`” を導入することができます。

■`obj.__ne__(self, other)` 拡張比較のメソッドで、比較の演算子 “`!=`” に対応します。つまり、等しくないことを判断する演算子です。このメソッドを上書きすることでクラスに属するオブジェクトに適した比較の演算子 “`!=`” を導入することができます。

■`obj.__gt__(self, other)` 拡張比較のメソッドで、比較の演算子 “`>`” に対応します。このメソッドを上書きすることでクラスに属するオブジェクトに適した比較の演算子 “`>`” を導入することができます。

■`obj.__ge__(self, other)` 拡張比較のメソッドで、比較の演算子 “`>=`” に対応します。このメソッドを上書きすることでクラスに属するオブジェクトに適した比較の演算子

“>=” を導入することができます。

■`obj.__cmp__(self, other)` 上述の拡張比較のメソッドが定義されていないとき、比較演算を(強引に)行う際に呼出されるメソッドです。インスタンスの比較を行うときにメソッド`__cmp__()`, `__eq__()` やメソッド`__ne__()` のいずれも定義されていなければインスタンスの識別子(整数型)による大小関係の判断が行われます。

■`obj.__unicode__(self)` 組込函数`unicode()`を実現するために呼出されます。`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に変換されます。

整数演算に関連するもの

整数演算の特殊メソッドを以下に示しておきます。

■`obj.__add__(self)` 同一クラスのインスタンスの和“+”を定義します。

■`obj.__sub__(self)` 同一クラスのインスタンスの差“-”を定義します。

■`obj.__mul__(self)` 同一クラスのインスタンスの積“*”を定義します。

■`obj.__truediv__(self)` 同一クラスのインスタンスの商“/”を定義します。

■`obj.__floordiv__(self)` 同一クラスのインスタンスの整数除算“//”を定義します。

■`obj.__mod__(self)` 同一クラスのインスタンスの剰余“mod”を定義します。

■`obj.__pow__(self)` 同一クラスのインスタンスの幕“**”を定義します。

■`obj.__lshift__(self)` 同一クラスのインスタンスで左シフト演算の演算子“<<”を定義します。

■`obj.__rshift__(self)` 同一クラスのインスタンスで右シフト演算の演算子“>>”を定義します。

■`obj.__and__(self)` 同一クラスのインスタンスで and 演算子“&”を定義します。

■`obj.__xor__(self)` 同一クラスのインスタンスで xor 演算子“^”を定義します。

■`obj.__ior__(self)` 同一クラスのインスタンスで or 演算子“|”を定義します。

浮動小数点数演算に関連するもの

演算の結果を浮動小数点数で返す特殊メソッドを以下に示しておきます。

- `obj.__radd__(self)` 同一クラスのインスタンスの和“+”を定義します。
- `obj.__rsub__(self)` 同一クラスのインスタンスの差“-”を定義します。
- `obj.__rmul__(self)` 同一クラスのインスタンスの積“*”を定義します。
- `obj.__rtruediv__(self)` 同一クラスのインスタンスの商“/”を定義します。
- `obj.__rfloordiv__(self)` 同一クラスのインスタンスの整数除算“//”を定義します。
- `obj.__rmod__(self)` 同一クラスのインスタンスの剰余“mod”を定義します。
- `obj.__rpow__(self)` 同一クラスのインスタンスの幕“**”を定義します。
- `obj.__rlshift__(self)` 同一クラスのインスタンスで左シフト演算の演算子“<<”を定義します。
- `obj.__rrshift__(self)` 同一クラスのインスタンスで右シフト演算の演算子“>>”を定義します。
- `obj.__rand__(self)` 同一クラスのインスタンスで and 演算子“&”を定義します。
- `obj.__rxor__(self)` 同一クラスのインスタンスで xor 演算子“^”を定義します。
- `obj.__ror__(self)` 同一クラスのインスタンスで or 演算子“|”を定義します。

単項演算子

- `obj.__neg__(self)` インスタンス a に対して“-a”を定義します。
- `obj.__pos__(self)` インスタンス a に対して“+a”を定義します。
- `obj.__abs__(self)` インスタンス a に対して“|a|”を定義します。

属性値のアクセスに関連するもの

- `obj.__getattr__(self, name)` 属性値の検索において 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)` 属性値への束縛を行おうとした時点で呼出されます。ここで name が属性名、value が属性値になります。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`’としてはなりません。‘`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で利用可能なメソッドで、属性値を返却するメソッドです。なお、メソッド`__getattr__`も実装されているときは`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。

ハッシュ表として

■`obj.__hash__(self)` 組込の函数`hash()`, `set`, `frozensey`, `dict`のようなハッシュを用いたオブジェクトの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__()`を定義しなければなりません。また、比較のメソッド`__cmp__()`と同値性のメソッド`__eq__()`が定義されていても、メソッド`__hash__()`が定義されていなければインスタンスを辞書の鍵として使えません。

■`obj.__nonzero__(self)` 真理値テストや組込演算`bool()`^{*17}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`’, あるいは等価となる整数‘`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__()`が呼出され、その結果が‘`nonzero`’であれば真になります。メソッド`__len__()`と`__nonzero__()`のどちらも実装されていなければ、そのインスタンスの真理値は全て‘`True`’になります。

3.5.4 記述子(descriptor)

「記述子」はクラスタイプ(=`object`や`type`の派生クラス)でのみ利用可能であり、オブジェクトの束縛に関係する属性です。この記述子は「記述子規約(descriptor protocol)」と呼ばれる一連のメソッド`__get__()`, `__set__()`と`__delete__()`を上書きすることで定義されます。ここで記述子はメソッド`__get__()`と`__set__()`の双方が定義されている「データ記述子」とメソッド`__get__()`のみが定義されている「非データ記述子」の二種類に大きく分類されます。また、データ記述子で、そのメソッド`__set__()`の呼出によって`AttributeError`例外が送出されるものを「読み専用データ記述子」と呼びます。

^{*17} 何故か`bool`でない!

記述子の呼出は属性名への参照が基点となります。すなわちオブジェクト a に対して属性名 x の参照を行う ‘a.x’ が基点になります。ここで引数がどのように記述子に結合されるかはオブジェクト a がクラスのインスタンスであるか、あるいはクラスそのものであるかに依存します：

記述子の呼び出し

- 直接呼出：最も単純な呼出操作で ‘x.__get__(a)’ に変換されます。
- インスタンス束縛：クラスタイプのインスタンスに対する束縛で、‘a.x’ が ‘type(a).__dict__['x'].__get__(a,type(a))’ に変換されます。
- クラス束縛：クラスタイプのクラスに対する束縛で ‘a.x’ が ‘a.__dict__['x'].__get__(None, a)’ に変換されます。
- スーパークラス束縛：a が super のインスタンスのときに束縛 super(b,obj).m() を行うと最初に a、次に b に対して obj.__class__.__mro__ を検索し、それから呼出：‘a.__dict__['m'].__get__(obj,obj.__class__)’ で構成子を呼出します。

まず、インスタンス束縛における構成子の呼出の優先順序は定義内容に依存します。そして構成子は上述の 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は該当する属性の参照が行われると構成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ていればデータ構成子、双方が定義されていなければ非データ構成子になりますが、組込函数 property() はデータ構成子として 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成子として定義され、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。

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 owner を介して参照されるときは ‘None’ になります。

■obj.__set__(self, instance, value) オーナークラスのインスタンス instance 上の属性を新たな値 value に束縛する際に呼出されます。

■`obj.__delete__(self, instance)` オーナークラスのインスタンス `instance` 上の属性を削除する際に呼出されます。

記述子は組込函数 `property()` と似た動作になります。 `property` も

3.5.5 コンテナの呼出

3.5.6 クラス生成

クラス `type(新スタイル クラス)` の生成は函数 `type()` を使って構築されます。

- クラスが生成される前にクラス辞書を変更する。
- 他のクラスのインスタンスを返却する。このときはファクトリ函数としての役割を果すことになります。

3.6 名前空間とスコープ

名前空間は、プログラム上で煩雑な命名規則を用いずに名前が不用意に一致するがないように統一的に名前を記述するための手法です。

この名前空間の BNF を以下に示しておきます:

名前空間の BNF

名前空間 ::= 識別子 分離記号 局所名

このように名前空間は分離記号で区切られた文字列としての性格を持ち、識別子を持つことで名前が一致した場合でも、識別子の違いから認識できるように工夫してあります。

名前空間の一つの実例として、ディレクトリ/フォルダの表記方法を挙げておきましょう。まず、ディレクトリの分離記号は計算機の OS 環境で異なりますが、UNIX の分離記号は “/”、DOS/Windows で文字コードが SHIFT_JIS であれば “¥”，それ以外は “\” が用いられています。たとえば UNIX 環境でファイルの指定は ‘/home/yokota/Works/test.txt’ のように行いますが、このときに局所名がファイル名の ‘test.txt’、識別子はファイルへの経路になる ‘/home/yokota/Works’ になります。同様に日本語環境の Windows でファイルを指定するときは ‘¥Users¥yokota¥Documents¥test.txt’ のように行えますが、このときに識別子が ‘¥Users¥yokota¥Documents’ で、その局所名は ‘test.txt’ となります。そして、ディレクトリ/フォルダが異なるファイルは別物ですが、もし、ファイル名だけでディレクトリ/フォルダの情報も含まれていなければ、名前だけで別物かどうかは判別できないことがあります。たとえばディレクトリ A とディレクトリ B にファイル mikeneko

があったときに、これらのファイルは別物ですが、ファイル名 mikeneko だけではディレクトリ A のものなのかディレクトリ B のもののかを判断することができません。このように実体が別物であっても局所的な名前が一致して区別ができなくなることを「**名前の衝突**」と呼びます。名前空間の EBNF で局所名がこの例のファイル名、識別子がファイルへの経路に相当します。

Python ではオブジェクトの参照は**名前**で行われます。この名前とオブジェクトの対応は「**名前への束縛**」で行われ、これらの名前から構成された集合のことを「**名前空間 (name space)**」と呼びます。そして、Python の名前空間はモジュールの階層構造を反映するために分離記号をコロン“.”とする識別子と局所名から構成されます。

また、オブジェクトの参照を行う際に、その参照可能な範囲、すなわち、参照の有効範囲が問題となります。ここで参照される名前の「**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の局所的なスコープの二種類しかありません。

3.7 実行モデル

3.8 名前付けと束縛

名前はオブジェクトの参照で用いられます。名前への束縛によって、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によってオブジェクトへの参照が行われるようになります。

Python で**ブロック (block)** とは、プログラム内の一つの実行単位になる区画です。たとえば、モジュール、クラス、函数はブロックとなります。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ブロックになります。

コードブロックは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式で、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。

スコープは、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式を一例として挙げています：

```
1 class A:  
2     a = 42  
3     b = list(a + i for i in range(10))
```

この例では名前 b に束縛する式が生成式 (for 節を持つ式) で、この式に名前 a への参照が行われています。しかし、この名前 a は関数 list() の関数ブロック内にあるためにスコープ外になります。そのために次の例では名前 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、クラスや関数定義、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。

3.9 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。

例外は `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。

3.10 Python の式

3.10.1 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、リテラルは数値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値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なので、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か、以下の内包表現の何れかになります:

——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]
```

この内包表現は反復処理による元の生成や、if 節によるフィルター処理の組み合わせから、その外延が計算できる仕組となっています。

■生成式: 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。

3.10.2 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 "." 識別子
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：函数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。

3.10.3 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]
```

单項算術演算子には、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 “ \wedge ” を Sage では幂乗の演算子として利用することができます。しかし、この演算子 “ \wedge ” は Python では後述するように排他的論理和 XOR の演算子であって、幂乗

とは別の演算子になるので注意が必要です。

■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 ::= 単項算術演算式 | 乗法的算術演算式 "*" |  
          単項算術演算式 | 乗法的算術演算式 "//" 単項算術  
          演算式 |  
          乗法的算術演算式 "/" 単項算術演算式 |  
          乗法的算術演算式 "%" 単項算術演算式  
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、Sage では演算子 “^” を冪乗の演算子として用いています。この点は Sage 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 となります:

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。

ここでの 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。つまり, C や Fortran の比較の演算式では比較の演算子は厳密に二項演算子であり, 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うことができるのです。つまり, 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 ‘ $a_1 \text{ op}_1 a_2 \text{ and } a_2 \text{ op}_2 a_3 \dots \text{ and } a_{n-1} \text{ op}_n a_n$ ’ として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。ただし, Boolean の True は整数の 1, False は整数の 0 と等価となり, このことを利用した算術演算が可能です。

■**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   ::=  条件式 |  $\lambda$ -式

```

この構文は ‘ $x \text{ if } y \text{ else } z$ ’ 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。

3.11 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** 函数やメソッドで明示的に値の返却を行うために用いる文で、return 文の引数として函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**: 生成函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函数が定義されます:

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: Python ライブドリを読み込むため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)*
```

3.12 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  ::= 単純文 (";" 単純文)*[";"]

```

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。

3.12.1 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。

3.12.2 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
            ["else" ":" 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
        ["else" ":" 一揃いの文]
```

3.12.3 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
```

3.12.4 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]
```

3.12.5 定義に関する複合文

■**函数定義**: フункци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、あるいはサブル래스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

Sage プログラムを書く上での指針

4.1 はじめに

Sage でプログラムを行う上での指針について解説しますが、基本的に Python の流儀に従うことになります。この流儀は **PEP** と呼ばれる一連の文書で定められ、その規定にはプログラムの構造を視覚化するための字下げ、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含まれています。そして、Sage 上で作成したプログラムを Sage と一緒に配布したいのであれば、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 Python の PEP がどのようなものであるかを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 Attachment) と呼ばれる設計書に基いて開発が進められています。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 に取り込まれることになる設計上の決定の文書化を行う上での主要な機構として位置付けられており、テキストファイル形式でバージョン管理されたリポジトリに保管されることになっています。

また 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と機能に関する仕様、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。また、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つことになっています。

^{*1} 原文は <http://www.python.org/dev/peps/pep-0001/>、その翻訳は <http://sphinx-users.jp/articles/pep1.html>

次に 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ついて解説する文書.
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, 意思決定方法等であり, PEP を規定する PEP もプロセス PEP になります.

PEP のワークフローも PEP-1 で定められており, PEP の状態遷移を図 4.1 に示しておきます, なお, その詳細は PEP-1 を参照して下さい:

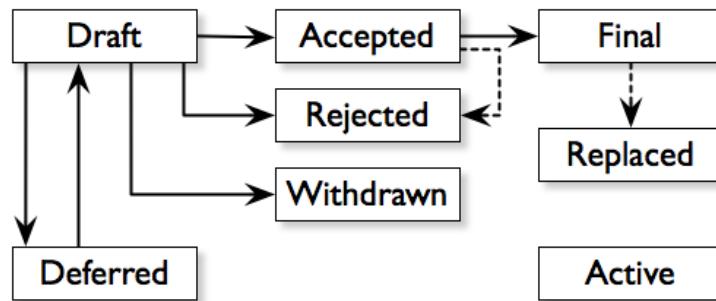


図 4.1 PEP の状態が辿れる経路

4.2.1 The Zen of Python

ここで Python がどのような言語であり, 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 で, その題名を **The Zen of Python** と名付けられた一種の詩で, Python を起動して `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:

^{*2} <http://www.python.jp/Zope/Zope/articles/misc/zen>

美は醜に勝り.(Beautiful is better than ugly.)

直は喻に勝る.(Explicit is better than implicit.)

素は組に勝り.(Simple is better than complex.)

組は混に勝る.(Complex is better than complicated.)

平坦は入籠に勝り.(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。

4.2.2 プログラム作成の流儀

Sage のプログラム作成の流儀は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) の表記に関連する情報 PEP の PEP-257 に従います。

ここではまず Python コードの様式案内という表題の PEP-8 を紹介しましょう。この PEP には「愚かな一貫性は小人物に憑いたおばけである」との見出しを持っており、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、広範囲の Python プログラムに施策を一貫させることにあると明言されています。そして、この文書は言語を Python に限定しなくても非常に有用なものです。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を 4 文字分とします。なお、空白文字とタブの混在を推奨していません。

■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画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 文字列が含まれる場合は “\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きます。

■import 文の書き方：import 文で読み込むパッケージは別々に読み込むべきです。また、パッケージ内部の import 文で相対 import を行わず、必ず絶対パッケージパスを用べきです。このようにすることで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。もちろん、既存の名前を新しいもので置換えるのが目的であれば構いませんが、そのような意図がないのであれば不要

な名前の衝突が生じないように ‘import test’ でそのまま読み込むか、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 ts
```

■**空白文字の利用について:** 余計な空白文字の利用を避けましょう。たとえば無理に演算子や式を合せるために無駄な空白を入れることは避けるべきです。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は空白文字を入れるべきです。とはいっても記号 “=” 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、註釈がない場合以上に問題となります。そのためにソースコードを変更したときは註釈の更新も優先して行うべきです。なお、短い註釈であれば最後のピリオドは省略します。

英語を母国語としない Python プログラマであっ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、英語で註釈を記述すべきです。

■**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、関数、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異った書式で記載されていれば内部の一貫性を優先することを重視し、命名規則に従わなくても良いことになっています。

命名の様式についてですが、以下の代表的な命名方法があります:

- b: 小文字一文字
- B: 大文字一文字
- lowercase: 小文字のみ
- lower_case_with_underscores: 小文字列を ‘_’ で繋げたもの
- UPPERCASE: 大文字のみ
- UPPER_CASE_WITH_UNDERSCORES: 大文字列を ‘_’ で繋げたもの

- CapitalizedWords: ラクダの瘤表記(複数の語を繋げたもので、各語の先頭文字のみを大文字とする)
- mixedCase: 小文字と大文字の混合
- Capitalized_Words_With_Underscore: 複数の語を「_」で繋げたもので、各語の先頭文字のみを大文字とする

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 ライブライアリの公開函数名の先頭には ‘X’ が用いられています。Python の場合はオブジェクト名が接頭辞に相当し、函数名でモジュール名が接頭辞となるためです。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あり、名前空間をむしろ利用すべきです。

なお避けるべき命名として、小文字の ‘l’、大文字の ‘o’、大文字の ‘T’ 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 ‘1’ や ‘0’ といった別の文字と混同し易く、可読性の面で問題があるからです。また、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^{*3}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 “_” で語を区切るものとします。mixedCase は互換性を保つことを目的とした利用のみに限定します。
- 大域変数名：函数と同様の規則で命名します。
- 函数やメソッドの引数：インスタンスマソッドの第1引数を必ず self とし、クラスメソッドの第1引数を必ず cls とします。
- 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しょう。

- 公開属性の先頭に記号 “_” を付けないようにします。

^{*3} 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区別している訳ではありません。

- 公開属性の名前が予約語と衝突するときは名前の最後に記号“_”を付けます.
- 公開データ属性は、属性の名前を公開するのが最善です.
- サブクラス化して使うことを意図したクラスがあり、そのクラスにはサブクラスから使って欲く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:

- ソースコードでは Python の実装の欠点を引き出さないようにすべきです.
- None の比較を行う場合は、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’ とは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. 例外の引数が長い場合、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、ログに保存する場合とコードが何かの後片付けをさせる必要がある場合の二つの場合に限定すると良いでしょう.
- すべての try/except 節にて try 節には最低限のコードを記述すること. バク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、文字列のスライス処理は避けること. フィルタ 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、UNICODE 文字列の可能性もあります.
- 列の処理では空の列が偽であることを利用します.
- Boolean を使った条件分岐では演算子 “==” を用いてはいけません. その分、冗長な処理になってしまいます.

4.2.3 Sage の様式

Sage では PEP をそのまま継承する訳ではありません. この Sage の流儀は <http://www.sagemath.org/doc/developer/conventions.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は空白4文字を用い、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も明確にするために字下げを行うことになっています。ここで、この字下げの文字数は空白文字で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した。たとえば、Sage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Sage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ぬ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

Sageのファイルやディレクトリ名についても指針があります。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とするというものです。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

Sageのライブラリファイルのヘッド部分については次の書式で行うことになります:

```
"""
<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/  
#*****
```

Sage に限らず, Python で文書は非常に重要視されています. そして, Sage 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

Sage では**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 等にあるもので, 註釈としての働きを持ちますが, 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, INPUT:, OUTPUT:, EXAMPLE:, SEE ALSO: といった見出しを用います. それらの記述方法に従います.

最後に Sage ではこの文書文字列は reST といった組版指示 (マークアップ) 言語を利用

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^{*4}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と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文書です。

reST は Python の文書の記述で用いられています。関連する PEP は PEP-256, 257, 258, 287 です。

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 の一部であり、reST を HTML, XML, LaTeX, ODF といった他の組版指示言語を使って記述された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。なお、Sage には Docutils や後述の Sphinx がインストールされているために、これらのアプリケーションを自分で入れる必要はありません。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 Sage の大きな利点です。

4.6.2 reSTでの文の構成

reST の文の構成単位は部、章、節、小節（款）、小々節（目）となっており、それぞれに見出を置くことでこれらの区分を実現することになります。reST で用いられる見出の組版指示は非英数字の 7 ビットアスキー文字を見出として用いる文字列の直下に見出行と同じか、それよりも長い下線として記載することで指示することができます。そして章立ては下

^{*4} “reStructured Text”ではなく一語の “reStructuredText” です。

線として用いられたアスキー文字の順に部、章、節等と自動的に解釈されます。この見出と reST の他の要素の区切としては空行が用いられます。この reST の構成要素を以下に記載しておきます：

reStructeredText の構成要素

- 段落 (Paragraphs)
- 見出
- 行内組版指示 (Inline markup)
- 箇条書と引用 (List and Quotes)
- ソースコード (Source Code)
- ハイパーアリンク (Hyperlinks)
- 章立
- 明示的な組版指示
- 指示 (directive)
- 画像
- 脚注
- 引用
- 置換
- エンコーディング

■**段落 (Paragraphs):** reST 文書で最も基本的な要素になります。段落は一行以上の空行で区切られた文字列の塊です。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インデントの高さは左揃えで全て同じ高さでなければなりません。ここで一例を示しておきます：

- 1 段落はこのような文字の羅列です。
- 2 そして、段落の区切りは空行です。
- 3 だから、ただの改行だけ
- 4 では段落の区切り
- 5 にはなり
- 6 ません。

この例は記述自体は 7 行にわたりますが、間に空行が一つだけしかないので、reST の構文解析器による解釈では次のような段落が二行の文書として解釈されます：

段落の解釈

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: 見出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ビットアスキー文字を見出行と同じか長く挿入します。HTMLと比較すると次のようになります:

HTML と reST の見出の対応例

```
<H1>  ⇔  =====  
<H2>  ⇔  —  
<H3>  ⇔  #####  
<H4>  ⇔  *****  
<H5>  ⇔  ^^^^^  
<H6>  ⇔  ~~~~~
```

HTMLの場合は見出が H1 から H6 までなので, reST でそれ以上を指示しても意味はありません。これは TeX でも同様で実質的に見出で使える ASCII 文字は 6 種類までです。また、ここで用いたアスキー記号と HTML の見出の関係は絶対的なものではなく、reST 文書内部で見出として現われた順番でしかありません。また見出と他の 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として扱われます。以下に具体例として reST 文書の一例とその HTML 変換結果を示しておきましょう:

```
1 EXAMPLS  
2 ^~~~~~  
3  
4 example1  
5 =====  
6  
7 example2  
8 -----  
9  
10 example3  
11 !!!!!!!  
12  
13 example4
```

```
14 ######
15
16 example5
17 #####$
```

この例では HTML との対応とは別順序で別記号も交えて記載しています。これを `rst2html.py` を使って HTML に変換すると図 4.2 のようになります：

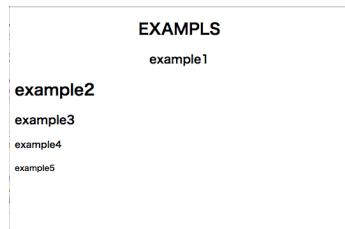


図 4.2 reST の見出 (HTML)

同様に `rst2latex.py` を使って L^AT_EX の文書に変換し、それを解釈した結果も図 4.3 に示しておきます：

このように HTML と L^AT_EX でほぼ同等の結果が得られています。

■行内組版指示 (Inline markup): 文字列を記号で括ることで文字列の強調を実現します。ここでの強調はイタリック、太文字と固定長の三種類です：

行内組版指示の一覧

| | | |
|---------|---|-----------------------|
| イタリック | <code>*italic*</code> | <i>italic</i> |
| 太文字 | <code>**bold**</code> | bold |
| 固定長 | <code>'verbatim'</code> | <code>verbatim</code> |
| ハイパーリンク | <code>'python<www.python.org>'</code> | <u>python</u> |

このようにアスタリスク “*” とバッククオートが用いられていますが、もしも文中に

```
EXAMPLES
example1
```

```
example2
example3
example4
example5
```

図 4.3 reST の見出 (L^AT_EX)

これらの文字が行内組版指示の要素以外で用いられている場合にはバックスラッシュ “\”⁵をそれらの文字の先頭に置くようにします。

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
- テキスト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なればバックスラッシュ “\”を用いると空白文字が表示されません。

■箇条書と引用 (List and Quotations): 通箇条書は先頭に記号 “-”, “*”, “+” 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きの箇条書になります。ただし、番号なしの箇条書と番号付きを続ける場合、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、番号付き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:

- 1 * これは番号なしの箇条書の
- 2 * 一例です。
- 3 1. これが番号付の箇条書の
- 4 2. 一例です。
- 5 #. これが自動番号振りの箇条書の

⁵ 日本語の Windows 環境や Mac OSX ではいわゆる円記号 “¥” を用います。

6 | #. 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、箇条書に含ま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：

```
1 * これは
2 * 箇条書です。
3     * これが子箇条書で、
4     * このように字下げを
5     * 必要とします。
6 * で、親箇条書をこのように
7 * 続けることもできるのです。
```

箇条書には定義を記述する「**定義箇条書**」があります。これは項目の次に改行を入れて字下げを行って項目の定義内容を記述します：

```
1 あれはなにか
2   猿。
3   どのようなものであるか
4     尻尾が短くて小柄だ。
```

これをそのまま HTML に変換すると項目の箇所が強調される点を除くとほぼ同様の文書が得られます。ただし、**LATEX** に変換すると項目の箇所が強調され、その次の行の改行が行われません：

— 定義箇条書の LATEX への出力例 —

あれはなにか 猿。
どのようなものであるか 尻尾が短くて小柄だ。

このように多少の違いはありますが、類似の結果が得られるのです。

■ソースコード (Source Code): リテラルコードブロックは、その前の段落の末尾に特殊な記号 “::” を配置することで開始することができます。そして、実際の開始では空白行をまず挿入し、それからリテラルコードブロックの記述を行い、記述が終われば必要に応じて空白行を挿入してから他の段落を記載します。またリテラルコードブロックは一様の深さの字下げを必要とします。

■ハイパーリンク (Hyperlinks): 外部リンク、内部リンクの二種類があります。なお、内部リンクは Sphix の機能に依存するものです。

■章立: 表題を

■明示的な組版指示:

■指示 (directive): 書式が定められたテキストを包含するための機構の一つです。Sphinx がこの指示 (directive) を多用しています。指示は名前, 引数, オプションとコンテンツから構成されます。

■画像: 画像を用いるためのディレクティブで, ソースファイルからの相対か絶対の何れかで経路を指定します。

■脚注:

■引用:

■置換:

■エンコーディング:

4.7 Sphinx

4.7.1 Sphinx とは

Sphinx は Python 2.6x 以降で標準的な文書生成ツールです。Sphinx 用に拡張された reStructuredText を基に HTML, TeX, PDF, EPub, UNIX の man や Texinfo とブレーンテキストに変換するものです。

4.7.2 使い方

プロジェクトの登録は `sphinx-quickstart` を起動して対話的に設定を行えるようになっています。あとは reST 文書を作成して `make` で文書が生成できます。

第5章

数学的対象の表現

5.1 はじめに

Sage では様々なアプリケーションを利用しますが、一般の利用者にはそのことを感じさせないようにできています。つまり、利用者と各種アプリケーションの間には Python があり、その Python で数学的対象を表現し、あとは必要に応じてアプリケーション固有の入力に変換したデータを引き渡し、アプリケーションからの出力を今度は Sage の対象に変換しているというものです。だから整数 1 を意味する ‘1’ のような整数から意味付けが行われているのです。この対象の Sage に於ける意味付けを見るための函数として函数 type() があります。この函数は引数を一つ取り、その対象がどのような意味を持つ対象であるかを返します。そこで ‘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見てみましょう：

```
sage: type(1)
<type 'sage.rings.integer.Integer'>
sage: type(1.0)
<type 'sage.rings.real_mpfr.RealLiteral'>
```

この結果から、対象 ‘1’ は Sage の整数環に属し、対象 ‘1.0’ は実数環に属する対象であることが判ります。より正確には ‘1’ は sage.rings.integer.Integer で評価されるべき対象で、‘1.0’ は sage.rings.real_mpfr.RealLiteral で評価されるべき対象であるということです。ちなみに iPython で同じことを実行した結果を参考迄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'>
```

この例からも判るように対象 ‘1’ は int 型で対象 ‘1.0’ は float 型であると返しており、Sage の結果とは異っています。このように Sage は Python を基盤にしていますが、統一的な扱いを行うために数学的対象は Python の本来の型を継承して新たな型を構築しているのです。このようななかたちで Python のオブジェクト指向言語としての特性が内部では生かされているのです。

5.2 数の構成

数の類 (Numeric Class) の定義は PEP 3141^{*1}に沿った方法で行われています。この PEP 3141 は抽象基底類 (ABCs: Abstract Base Classes) の導入に関する PEP 3119^{*2} の数に対する適用となります。

^{*1} 原文:<http://www.python.org/dev/peps/pep-3141/>

^{*2} 原文: <http://www.python.org/dev/peps/pep-3119/>

日本語訳：<http://homepage3.nifty.com/text/script/python/pep-3119.html>

第 6 章

SQLite を使った解析

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の一つです。SQLite は C で記述され、非常に軽量で動作も軽快であること、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、サーバーを必要とせず、利用する際に、まえもってさまざまな設定を行わずに手軽に使えること、そのような手軽さに加えて、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 を用いて構成するような大規模な RDB システムを組むことには不向きですが、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、ただちに利用することができるという性質から小規模なシステムに向いているのです。

6.1.2 SQL について

ここで SQL という言語はリレーションナルデータベース管理システム (RDBMS) で、データベースの管理やデータ操作を行うための言語です。SQL という名前は IBM が開発した System R の操作言語 **SEQUEL**(Structured English Query Language) に由来し、その名前が示すように DB に対して問い合わせを行い、それに応じた処理を行うための言語です。

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います。SQLite は 1992 年の規格化である SQL92(SQL2) に準拠しています。

SQL の文法はまずデータ定義言語 (DDL:Data Definition Language)、データ操作言語

^{*1} Atomicity(不可分性), Consistency(整合性), Isolation(隔離性), Durability(持続性):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。

(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:

データ定義言語 (DDL)

| | |
|--------|----------------------------|
| CREATE | DB の対象 (表, インデックス等) の定義で利用 |
| DROP | DB の対象の削除 |
| ALTER | DB の対象の定義変更 |

データ操作言語 (DML)

| | |
|--------|------------------------|
| 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 |

データ制御言語 (DCL)

| | |
|-----------|-------------|
| GRANT | データの挿入 |
| REVOKE | 表の更新 |
| 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 |

基本的に表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り出すといった処理が基本です.

6.2 Sage から SQLite を使う

6.2.1 sqlite3について

ここでは Sage から SQLite を扱う実例を挙げます. SQLite 単体を立ち上げて利用する実例ではありません. Sage は Python 環境であるために, Sage から SQLite を使うというよりは Python から SQLite を扱うというのがより一般的で正しい表記になります. が, ここでは Sage 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^{*2}。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()` で行なうことが判ります。より具体的には、`connect()` が構成子であり、`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> | <code>cursor</code> オブジェクトを生成します。 |

実際の RDB への処理は `Connection` オブジェクトを構成したのちに、`Cursor` オブジェクトを作り、そのオブジェクトに対して行なうことになります。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> | <code>Cursor</code> 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> | <code>SELECT</code> 等による問合の結果を一件取り出します。 |
| <code>fetchmany()</code> | <code>SELECT</code> 等による問合の結果を取り出します。 |
| <code>fetchall()</code> | <code>SELECT</code> 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。 |

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。

^{*2} PEP-248 は DB-API 1.0 と古い版

6.2.2 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Sageに含まれるSQLiteを直接使う方法です。もう一つはSageを起動してSageからSQLiteを利用する方法です。前者の場合は予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Sageメニューから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Cloud SageMathでも同様です。

もう一つの方法はPython向けのDB-API 2.0インターフェイスであるsqlite3ライブラリをSage上で読み込んで、そこから利用する方法です。この場合はまずSageを立ち上げてから、通常のライブラリと同様にsqlite3ライブラリの読み込みを行います。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なければ新規に作成されます。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というテーブルを生成しています。このテーブルにはtext型のname、整数型のageとweightを項目として持っています。それからinsert文でテーブルmycatに値の書き込みを行いますが、このときはメソッドexecute()にSQL文を文字列で与えてしまうと引数の型が全てtext型になってしまうので、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生じます。

このように前処理を行なうこともなく簡単に RDB の表を生成して、それを活用することが可能なので、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

Sage で画像処理

7.1 画像の読込

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 Sage 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で、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 (C) 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 Sage 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、書き込みや表示では、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 Sage 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、Sage の追加パッケージとして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
```

^{*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 のみの対応で setuptools にも対応していませんが、Pillow は 3.x や setuptools にも対応しており、PIL と同様の使い方もできます。

```
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が、それから Image ライブラリの中の関数 open() で画像ファイルの読み込みを行なって Sage のオブジェクトとして取り込んでいます。この際に画像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 Sage 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 ‘open’ が用いられるように設定されています。また Sage 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 形式で画像を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 の、ノートブック形式であれば UI 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関数はありません。そのため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 Sage 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は 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。そのことであって NumPy は SciPy や Matplotlib の基底となっており、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 ライブリのメソッド array() を用いることで, PIL の 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() フィクションを用います. この array フィクションは Python のリスト等のデータを配列に変換するフィクションです. さて, PIL ライブ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の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 で現われなかった ‘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 509×800 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, Sage 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 の配列が緑の強さに対応します. これを Sage 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. そして, この構文を用いた処理のことを「**(拡張) スライス処理**」と呼びます. この表記は Python のリストでは使えません. あくまでも NumPy で定義された配列に対してのみ利

用可能です。

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、MATLAB と同値な表記にするためには $a[i-1:j]$ と表記することになります。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。“()” は函数を構成する要素なので Sage(Python) の配列で間違って使わないように注意して下さい。

さて、ここで画像データ $imat$ は 509×800 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 7.1 が得られます：

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 Sage 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。

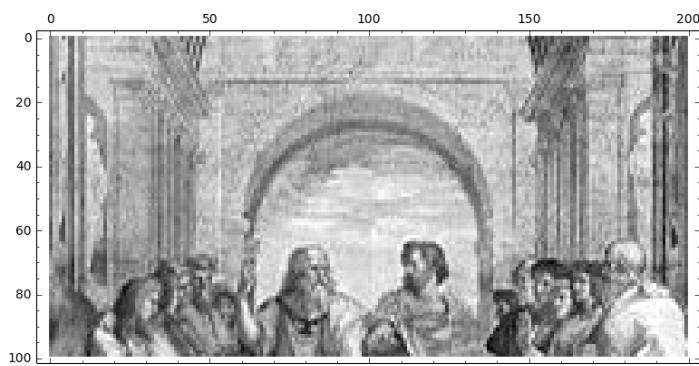


図 7.1 画像の一部切り取り

第8章

Sage の拡張

8.1 sagemath からのパッケージ入手

Sage 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 ライブラリの wxPython や PyQt4 といったライブラリ、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 Sage 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 Sage にインストールすることができます。

8.2 一般の Python パッケージのインストール

Sage は Python 上で構築されたシステムであるため、Sage 側の Python にライブラリやパッケージをインストールすることが可能です。この場合は sagemath.org からのパッケージを入手してインストールする方法と異なり、あらかじめ環境変数の設定を行う必要があります。なぜなら Sage 側の Python やライブラリ一式を利用しなければならないので、環境変数 PATH と LD_LIBRARY_PATH をインストールしてある Sage 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。

8.3 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、Sage 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9章

SageMathCloud

9.1 SageMathCloud とは

SageMath^{*1}はクラウド計算機環境で Sage の実行環境だけではなく、Sage のウェブ ブラウザ上でのノートブック形式の UI を実現させている IPython Notebook の後継の Jupyter を用いて、Sage 以外の IPython, LaTex と仮想端末をウェブ ブラウザ上で実現するものです。



図 9.1 SageMath, Inc

ここで利用可能なウェブ ブラウザについては特に指定はありませんが、3D のグラフ表示で Sage は Java で記述された jmol を用いている関係上、Java の runtime を必要とする程度です。なお、Chrome には「The Sagemath Cloud」というアプリケーションがあり、このアプリケーショ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

^{*1} <https://clouds.sagemath.com>

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、GUI も Sage ノートブック形式の UI よりも洗練されています^{*2}。この SageMathCloud の開発と平行して、従来のオンライン上で Sage を利用できるサービスであった Sagenb^{*3}は 2015 年 5 月にサービスを停止し、sagemath.org の WebPage からのリンクも SageMathCloud のアカウント画面へのリンクに変更されています。

SageMath の利用はあらかじめ利用者登録を行う必要がありますが、この登録と利用は無料です。図 9.2 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：

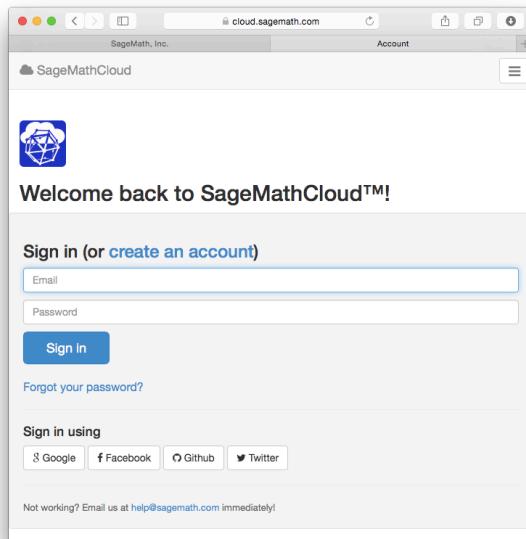


図 9.2 ログイン画面

ここ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 3000MB、メモリは 1GB、

^{*2} GUI については専門家に開発させているとのことです。

^{*3} <http://www.sagenb.org>

CPU は 1CPU, タイムアウトするのは 1 時間の無操作の場合です。なお、処理速度については大雑把ですが、MacBookAir 2013 と同程度とそこそこの本格的な処理が可能です。ウェブベースであることも含め、教育や試算には適していると言えるのではないかでしょうか。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。ここでは SageMathCloud を使うまでの簡単な事例を幾つか紹介することにします。

9.2 ファイルのアップロード

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可能です。この場合は右上の New メニューを押して図 9.3 に示すファイル生成に移ります:

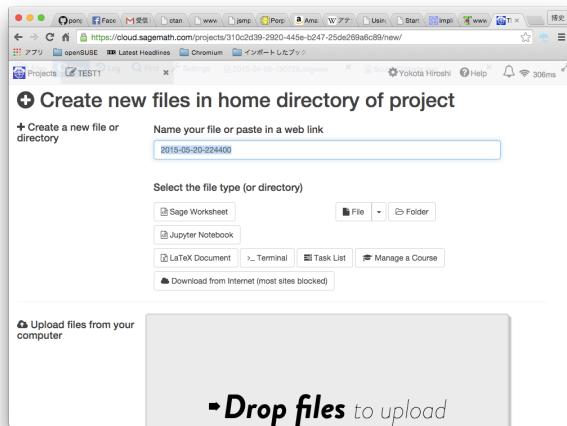


図 9.3 ファイル生成ページ

このページで「Upload files from your computer」とある項目で大きく「**Drop files to upload**」とある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。ここでクリックすればファイルのセレクターが現われるので、そこでアップロードするファイルを選択します。またファイルをドロップしたのであれば、しばらく待つと図 9.4 に示すようにサムネイルがやがて表示されます(大きなファイルだと転送に時間がかかります):

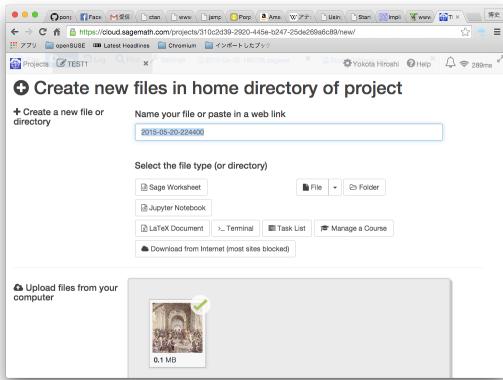


図 9.4 ファイルのドロップ後

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 Files メニューを押すことで表示される図 9.5 に示すページで確認することができます：

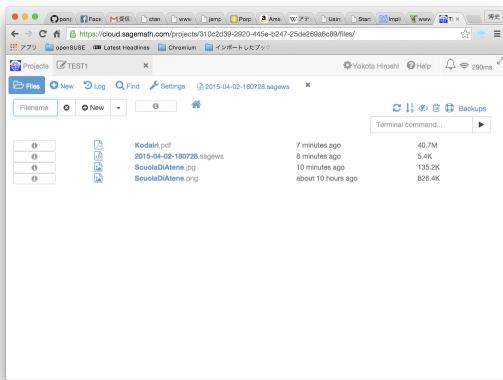


図 9.5 プロジェクト内のファイルの一覧

ファイルはプロジェクト単位に保管され、こうすることで SageMathCloud 上で画像等のアップロードしたファイルの読み込み等が行えるようになります。図 9.6 には先程、アップロードした画像を SageMathCloud 上に読み込んで、関数 `matrix_plot()` で表示した例を示しておきます：

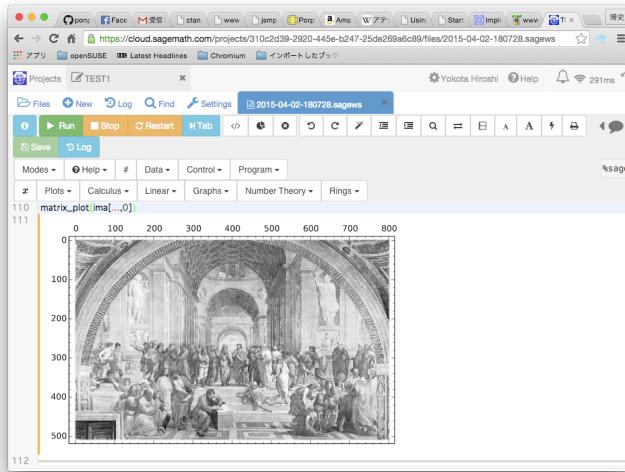


図 9.6 アップロードした画像の表示

9.3 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 Sage をクラウドで実現するだけではなく、さまざまなアプリケーションが起動する環境でもあります。この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 9.3 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 9.7 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：

まず、Sage Worksheet は説明するまでもないでしょう。それから Terminal ですが、こちらを起動するとウェブ ブラウザを通常のテキスト端末として利用することが可能になります。SageMathCloud というサービスは Ubuntu Linux 上で実現していることから、Ubuntu を直接利用することができます。また、Sage 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、Sage を構成するアプリケーションやライブラリを、Sage 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ファイルに残されます。

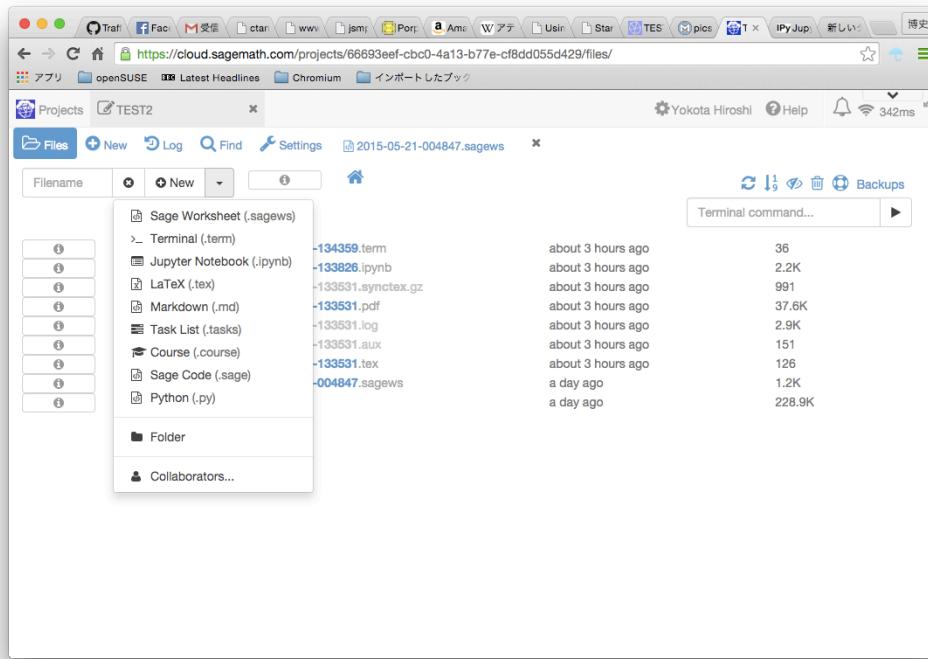


図 9.7 New メニューの一覧

次の Jupyter^{*4}は IPython の言語に依存しない中核を他の言語のシェルとして利用するためのプロジェクトです。したがって、この Jupyter は IPython のように Python のシェルとなるだけではなく、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 等のノートブック形式のシェルとして、ウェブ ブラウザ上でも利用可能となるものです。ちなみに <https://try.jupyter.org/>には 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^{*5}。なお、SageMathCloud では Jupyter Notebook で利用できる kernel として Python2, Python3, GNU R と Julia が用意されており、カーネルの切替はセル単位で行えます。つまり、Notebook にこれらの言語が混在可能なのです。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 を選択すれば良いのです。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 Notebook に

^{*4} <https://jupyter.org>

^{*5} <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。

は記載されていても言語環境が再起動されているため、入力セルに以前の処理結果は引継がれません。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 `%matplotlib inline` をあらかじめ入力しておきます。すると matplotlib に包含される函数で画像の表示を行うと図 9.8 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：

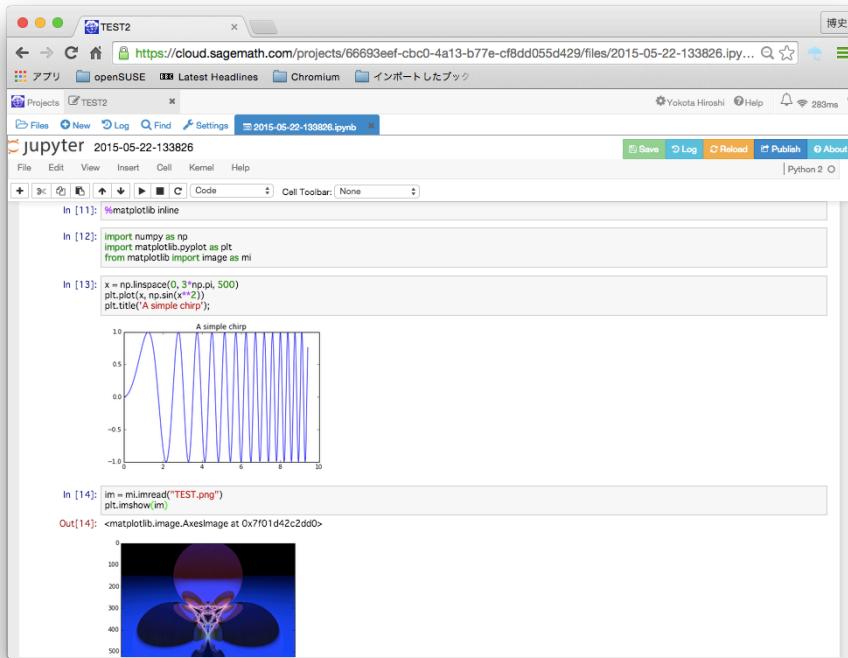
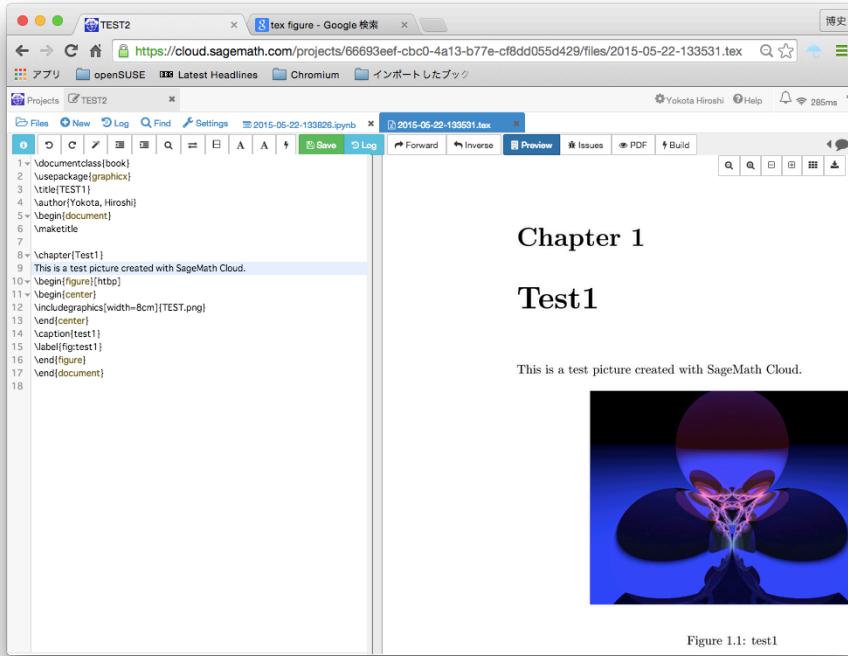


図 9.8 Jupyter+Python

それから L^AT_EX は幅広く利用されている Typeset 環境です。こちらは図 9.9 に示すように、ブラウザ画面を左右に二分し、L^AT_EX ファイルが左半分、生成した PDF が右半分に表示されます。左半分に表示された L^AT_EX ファイルを編集すると早速コンパイルが行われて結果が表示されるというものです。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することが可能となっています：

なお、クラウド上で L^AT_EX のサービスを行うものに Cloud LaTex^{*6}があります。この

^{*6} <https://cloudlatex.io/ja>

図 9.9 SageMathCloud 上の L^AT_EX

Cloud LaTeX は SageMathCloud が提供する L^AT_EX 環境よりも本格的なサービスで、原稿ファイルのドラッグ・ドロップによるアップロードが容易に行えること、日本語、中国語等のマルチバイト言語にも標準で対応しています。それと比べると SageMathCloud 上の LaTeX は日本語のスタイルファイルがあらかじめ準備されておらず、日本語の利用で工夫を必要とします：

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はない、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^{*7}。

^{*7} 「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 言語を中心とした統一という面で注目すべきことです。

第 10 章

手引 (ポルピュリオス)

10.1 概要

この章ではテュロスのポルピュリオス ($\Pi\sigma\rho\varphi\mu\rho\iota\sigma$) 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 $Eἰσαγωγή$ 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。

エイサゴーゲーはラテン語への翻訳「イサゴーゲー (Isagoge)」[21] で西ヨーロッパでは知られています。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、忠実なラテン語への翻訳になっています。また、ビザンツ帝国の領域では要約が知られ、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とのことです。

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) がイサゴーゲーから翻訳したものとバーンズ (Barnes)[20] がギリシャ語の文献から直接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}されていたり、アリストテレスのオルガノンと一緒に併せて廉価で売られていたりします。こちらの章立てはイサゴーゲーと同一で、エイサゴーゲーがアリストテレスの「範疇論」の入門書であるという立場で翻訳されたものです。それに対してバーンズの翻訳はエイサゴーゲーがアリストテレスの「範疇論」だけに限定した入門書ではなく、むしろアリストテレスの論理学 (オルガノン) への入門書として捉えており、それに加えてギリシャ語文献の解釈や歴史的背景を含む詳細な解説、それに加えて原文のギリシャ語と英語の単語の対照もあって、より深く調べる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

^{*1} http://www.ccel.org/ccel/pearse/morefathers/files/porphry_isagogue_02_translation.htm

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。

エイサゴーゲーの著者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少ないものです。まずポルピュリオスはフェニキュアのテュロスで234年に生まれ、シリア語で王という意味でMalcusと名付けられています^{*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(Bασιλεύς)^{*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 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(Λογγίνος)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^{*5}の創始者として知られるプロティノス(Πλωτῖνος, Plotinus)の一門に入り、プロティノスから大きな影響を受けます。そして268年にプロティノスの勧めもあってシチリア島に行き、270年にプロティノスが死んだ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ニアデス(Enneads)の編纂を行っています。それから北アフリカに行ったことやマルセラ(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[26]^{*6}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(新プラトン主義)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たという荒業をやってのけているのです^{*7}。まず、ボ

^{*2} ヘレニズム文明のアラブ世界への伝播ではシリア人のキリスト教徒が文献の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。

^{*4} パーンズの本[20]の表紙が紫色なのもこの洒落でしょう。

^{*5} プラトンのイデア論を継承しつつ、絶対的な一者を想定し、万物はその一者からの流出として捉える「流出論」を特徴とする哲学思想です。

^{*6} マルセラはポル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ないものの、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。

^{*7} その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でしょう。

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が採用されます。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」であることは「秘儀の漏洩を防ぐ」ためと考えられたこと^{*8}、それと新プラトン主義のライバルであるストア派^{*9}と対抗するためにアリストテレスの著作を使って体系化する必要があったと考えられています。このような経緯もあってアリストテレスの著作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、そういった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を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^{*10}。いずれにせよ彼はしばらくローマを留守にしていたことになりますが、ちょうどその頃、ローマでの弟子のクリュサオリオス(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

*8 ある意味で深読みのし過ぎですが。

*9 ヘレニズム哲学の一学派でキティオンのゼノン(Zēnon)が創始者。アカデマイア学派、逍遥学派、エピクロス派と並ぶ四大学派の一つ。五賢帝の一人のマルクス・アウレリウス・アントニヌスも信奉者の一人であったように、古代ローマ共和制末期から帝政期初期にかけて最も有力な哲学学派でした。ストア学派の論理学は断片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。

*10 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。

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くれるか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。

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。その理由ですが、まず、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、この文献は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。これはポルピュリオスの時代も当然そうだったのですが、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作ったのです、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、西ヨーロッパの哲学が発展することになるのです。ところで、こういった著作も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す。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のですが、中世スコラ哲学の歴史的な論争で著名な「**普遍論争**」^{*11}の火種にもなっているのです。

*11 「普遍が存在するか?」という問に関する論争で、存在するという立場は「实在論」、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はなかったようです(詳細は[16]を参照)。

10.2 はじめに

必要なこととして、クリュサオリオス (Chrysarorius)^{*12} より、まずアリストテレス (Αριστοτέλης, Aristotle) の範疇 (カテゴリー) について学ぶために、類 (γένος, genus) *¹³ が何であり、種差 (διαφορά, difference) が何であり、種 (εἶδος, species) *¹⁴ が何であり、特有性 (属性, ιδιον, property) と偶有性 (付帯性, συμβεβηκός, accident) が何であるかを知ることができます^{*15} - 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、手短に入門の形式をとって、古の賢者達^{*16} 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^{*17}。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^{*18}。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 (Περιπατητος, 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。

*¹² ポルピュリオスの弟子で、ローマの元老院にて高位の議員だったようです。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。

*¹³ バーンズ [20] 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。

*¹⁴ εἶδος は一般的に形相 (form) と誤され、ものの形を意味します。ボエティウスはラテン語で種に対応する意味のものを species と形相の意味に対応するものを forma と分けて訳しており、これが現在の英語の species と form に対応しています。ちなみにプラトンはイデアを ἴδεα 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。

*¹⁵ 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。

*¹⁶ 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。

*¹⁷ バーンズ [20] 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

*¹⁸ ボエティウスのイサゴーゲー第二注解 [15] 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)」が生じることとなった歴史的に意味のある一節です。なお、ここで実在を明言しないということはポル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実在ということはあり得ないことです。

10.3 類について

どの類 ($\gammaένος$, genus) も種 ($\varepsilonιδός$, 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 ($Ἡραλείδαι$, ヘーラクレイダイ) という類は、この意味で、ある一つの事象 - 要するにヘーラクレース - との彼等の関係からそう呼ばれ、互いにとにかく関係する多数の人々は、他の類と相互に区別するために、彼 (ヘーラクレース) に由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ース ($Ὀρεστῆς$, Orestes) はタンタロス ($Τανταλός$, Tantalos) からの類^{*19}、ヒュロス ($Ὑλός$, Hyllus) はヘーラクレースからの類と言えます^{*20}; それから再び、ピンダロス ($Πινδάρος$, Pindar) は類としてテーバイ人^{*21}、プラトン ($Πλάτων$, Plato) はアテナイ人^{*22}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)^{*23}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うのです。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い類似性からです; 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 ($κατηγορεῖν$)^{*24}していると語ることでそれを表現します^{*25}; たとえば、動物です。

^{*19} オresteースはミュケーナイの王アガメムノーンの息子、タンタロスはリュディア王で、オresteースの先祖になります。

^{*20} ヒュロスはヘーラクレスの息子です。

^{*21} ピンダロスはテーバイ生まれの詩人です。

^{*22} プラトンは言うまでもなく哲学者のプラトンです。

^{*23} ケクロプス王はアテナイの伝説的な初代の王です。

^{*24} 範疇 (カテゴリー) の由来で、カテゴリーとは要するに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」とも言いますが、ここでは「カテゴリー論」の註に従って「述定」と訳します。

^{*25} アリストテレスは形而上学の△卷「哲学用語辞典」で類の定義として四つの方法を列挙しています: 「同じ形相を持つ事物の連続的な生成の存するもの」、「あるものの事物の存在がそれに由来する所の第一の

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^{*26}、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^{*27}。

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するものから類を区分します; それら (類) が種で異なる事象を語るものであるという事実が種や特有性として述定するものをそれら (類) から区分します; そうして、それら (類) が ‘それが何なのか?’ に対する回答で述定するという事象が、‘それが何なのか?’ ではなくむ

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。ポリピュリオスはアリストテレスの言う「平面がさまざまな平面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)。

*²⁶ 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

*²⁷ ここで述べているように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。

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。この類といふ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。

10.4 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)^{*28}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 ...^{*29}

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とを種と呼びます^{*30} - 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^{*31}。

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こで言うのであれば、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。

そこで、彼ら（逍遥学派）は種を次のように表現します：種は類の下で整理されたものです；それから：‘それは何なのか?’に対する回答で類が述定するものです。それからまたこのように：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

^{*28} 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觀を *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です。

^{*29} 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。

^{*30} 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るのです。

^{*31} 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。

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³².

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)³³には、最も総合的($\gamma\epsilon\nu\nu\kappa\omega\varsigma$, general)な事象があれば、他に最も特定的な事象もあります。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には同時に複数の類や種があります(とはいえ、あるものと他のものとの間に関係を持たされています)³⁴.

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($o\nu\nu\sigma\alpha$, substance)はそれ自体が類になります³⁵。その下に物体($\sigma\nu\mu\alpha$, 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³⁶.

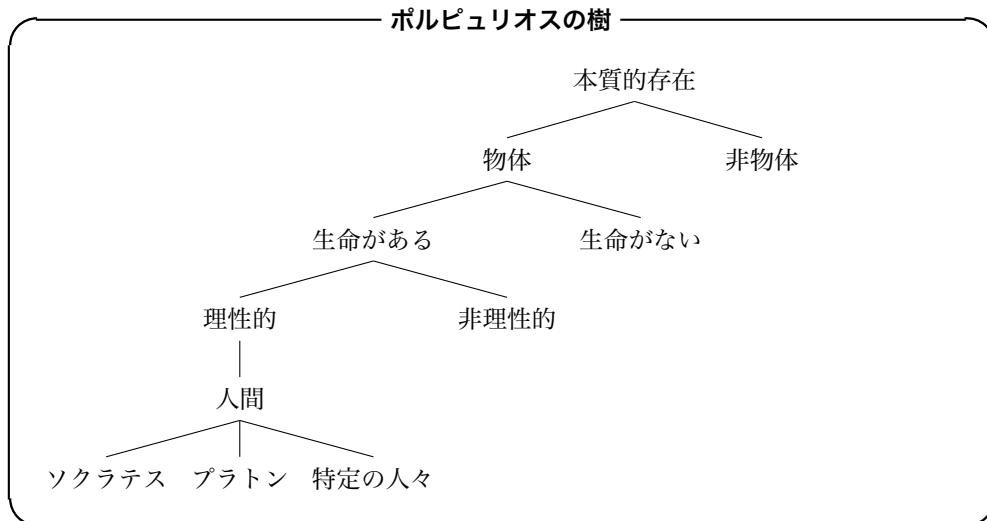
*32 「それが何であるか」という問い合わせに対しては、それはソクラテスという個人、あるいは人間という種の二通りの回答が考えられるのです。

*33 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。

*34 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は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、アリストテレスの場合、真や偽は命題の状態を示すもので、「存在するものを存在しない」と言い、あるいは存在しないものを存在すると言うこと」が偽で、「存在するものを存在すると言い、あるいは存在しないものを存在しないと言うこと」が真([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)を上手く避けているのです！

*35 本質的存在(实体)の内の一つです。

*36 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」等と呼んでいるとのことです[20]。



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 (*άτομος*, individual) *³⁷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) *³⁸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、

*³⁷ 不可分のものという意味。原子 (atom) の語源です。

*³⁸ *προδεξές* (proximate) は間に中間的な種等が入らないという意味です。

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：それは類であっても種ではなく；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：それが種であっても類ではなく；そして：種であるとすれば、我々は再び種に分割できず；それから：‘それは何なのか？」に対する回答で、数で異なる幾つかの事象を述定するのです。その両端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)^{*39} はアトレウス (*Ἀτρεύς*, 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^{*40}。(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^{*41}。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^{*42}。

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るべき

^{*39} 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、タンタロスはとうとゼウスの息子という説があります。

^{*40} 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

^{*41} 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。

^{*42} 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。

ですが - は無数にあります。それがプラトンが最も総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、我々が最も総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^{*43}。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うことがあっても、動物が人間であるとは言ひはしないでしょう^{*44}。

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 - それから最も総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、総合的な事象は類(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。最も総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。

^{*43} 「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。

^{*44} 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。

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 ($\Sigma\omegaφρονίσκου$, 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 - 同じだということが判るでしょう。

このように個体はその種に、そして種はその類に包含されます。というのも類は全体を整理したあつまりで、個体は一部、それから種は全体であったり一部だったりします - 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、それから最も総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。

10.5 種差について

種差 ($\deltaιαφορά$, 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, あるいは単に別物のようにするのです. ここでそういった物事を別物にしてしまう種差のせいで, 類の種差による分類が行われ, それと定義 - 類やこのたぐ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, 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. 異在不可能な種差のあるものはそれ自体の原因であり, 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; とはいっても鉤鼻であることや獅子鼻といったことは偶然で, それ自体が原因のものではありません. それ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, 別物のようになります. 再度, それ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^{*45}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 - 生命があって知覚がある, 理性的であることと非理性的であること, 死すべきことと不死であること - 生命があっ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 (なぜなら動物は生命があっ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

^{*45} 異在可能なものの, 異在不可能なものの三種類です.

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っ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、理性的であって不死であることから神が、それから非理性的であって死すべきもの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。そうして、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らです - 離在不可能で偶然そ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ってしょう）；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。

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（*ὕλη*, matter）と形相（*εἶδος*, form）で構成されてたり、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*⁴⁶。

*⁴⁶ 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。だからブ

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、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的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うたつ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。

これで種差については十分です。

10.6 特有性について

彼等(逍遙学派の哲学者達)は特有性(*ποιον, property*)を四つに分類しています: とある種だけの偶有性であるもの、たとえそれの全てでなかったとしても(医者にかかるている、あるいは幾何学を勉強中の人のように); その種全ての偶有性であるもの、たとえそれだけでなかったとしても(人間が二本足であるように); 種だけ、そして種の全て、そしてあるときに保持するもの(人間が老年では灰色になるように); それから四番目に、「単体、全て、そして常に」で一致するところのもの(人の笑うことができるということのように)。というのも人は常に笑っている訳ではないので、人が笑っていると語られるのは彼がいつ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⁴⁷; もし馬

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。

⁴⁷ 「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。

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。

10.7 偶有性について

偶有性 (*συμβεβηκός*, accident) *⁴⁸はそれら(偶有性)の基体 (*ὑποκείμενον*, subject) *⁴⁹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*⁵⁰。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*⁵¹。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: 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。

10.8 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。

*⁴⁸ *συμβεβηκός* は動詞の *συμβαίνειν* に由来し、本来の意味は予期せずに‘生じる’ことや‘発生する’ことですが、アリストテレスは保持しているという意味でも用いています。このようにたまたま生じさせられたり生じたり、あるいは何かを保持していることで用いられています。

*⁴⁹ 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。

*⁵⁰ 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。

*⁵¹ 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。

10.9 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^{*52}。

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^{*53}。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探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^{*54}。

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、今まで語られてきたように、類は「それは何であるか？」に対する回答で、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。

^{*52} ‘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。

^{*53} 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。

^{*54} 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。

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）^{*55}。

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。

10.10 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^{*56}。

また、類は（種よりも）先行して存在していか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^{*57}。

さらに、類は類の下にある種を含むことから種よりも一層広範で、種は種自体の種差によって類よりも一層広範です^{*58}。

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^{*59}。

^{*55} 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。

^{*56} 類は常に種よりも広範囲である。（トビカ [3] 121b3-4）

^{*57} 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。

^{*58} 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。

^{*59} 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。

10.11 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^{*60}.

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) ^{*61} 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^{*62}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^{*63}.

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^{*64}.

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.

^{*60} 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’に対しても‘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.

^{*61} 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ること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.

^{*62} アニユトス *Anūtos* は政治家, メレトス *Mēlētos* は詩人で共にソクラテスの告発者です.

^{*63} 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.

^{*64} 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.

10.12 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^{*65}。

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^{*66}。

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、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、それらは繰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には、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なのです。特有性が偶有性とどのように異っているかを語ることが残っているでしょう；それらは種、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

*65 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。

*66 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ということです。つまり、人間に対してはどの程度人間であるかを語れませんが、カラスの色の黒さに対しては、その黒色の程度を語ることは可能であるということです。

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。

10.13 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誤ではありません⁶⁷。

10.14 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

⁶⁷ 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。

る関係で語られるのです。といふのも笑うことができるといふことも‘常に’そのような天性があつ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ふ事実です- たとえば、理性的であるといふ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^{*68}。

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^{*69}。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。

10.15 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ふ事実です。

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ふ事実です：二本足であるといふ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ふこともそうです。

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(理性的であるといふ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^{*70}。

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できます^{*71}。

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ふことは、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。

10.16 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ふ事実があります：もしも人間であれば、笑うことができます；もしも笑うことができるのであれば、人間なのです(笑うことが

^{*68} 理性的であるといふ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。

^{*69} 主語と述語の入れ替えができないといふことです。

^{*70} 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のです。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。

^{*71} 類が互いに混合することがないのと同様に種差も互いに混合することがないと述べているのです。

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。)*72

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発的に生じることです。

また、種は常にそれらの基体で実際に先行して存在するのですが、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。

10.17 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がいのものであるか?’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、さらにそれらは偶発的な天性を持つのです。

種で分有するということは偶有性では等しく生じます - 離在不可能のそれなら - 等しくはありません。というのも他の人と比較されている一人のエチオピア人は肌の色の黒さが薄くなても濃くなても構わないのであるからです。

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

*72 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。

差や類と異なっているかが語られているからです。

10.18 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^{*73}。

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^{*74}。

ここで述べた以上のお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。

*⁷³ 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。

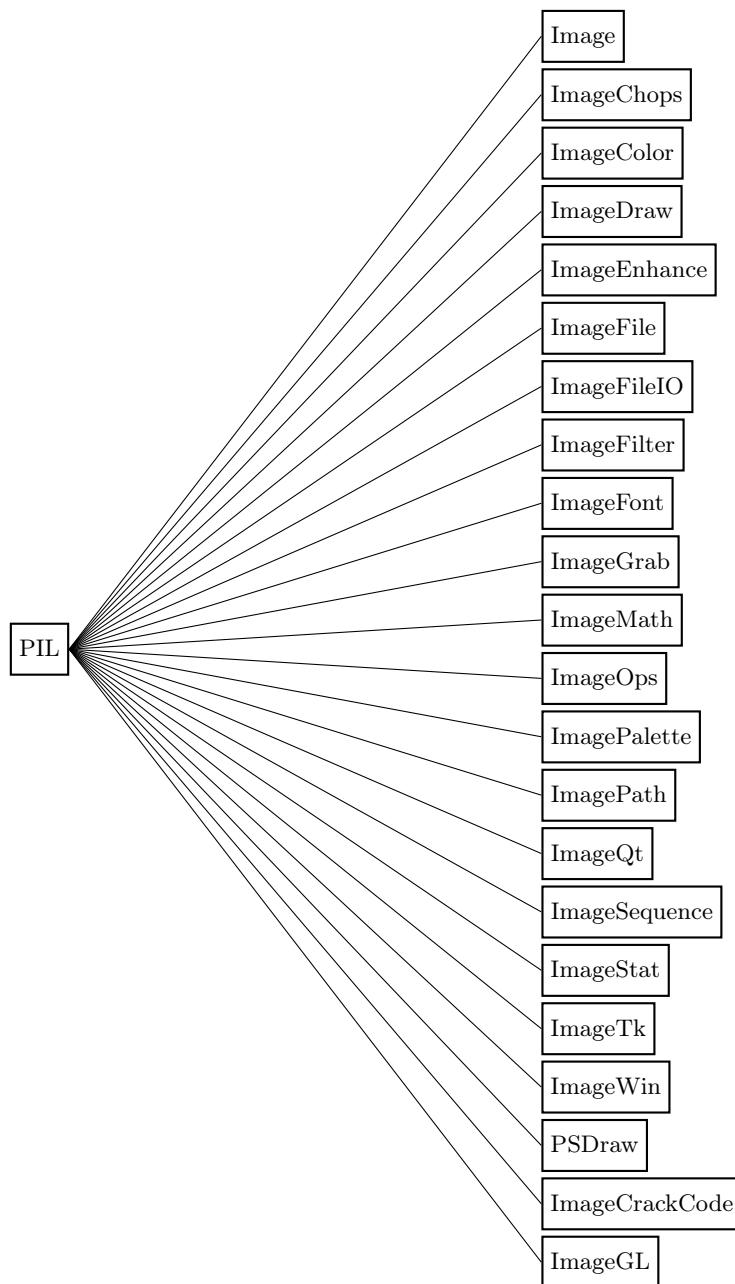
*⁷⁴ 特有性と種は1対1に対応し、だからこそ主語と述語の関係で、主語と述語としての入換が効くのです。ところが、離在不可能な偶有性と種については1対1の対応とならないために主語と述語の関係で入換が効かず、さらに偶有性の性格上、その程度を表現することが可能だということです。

第 11 章

Python のライブラリ

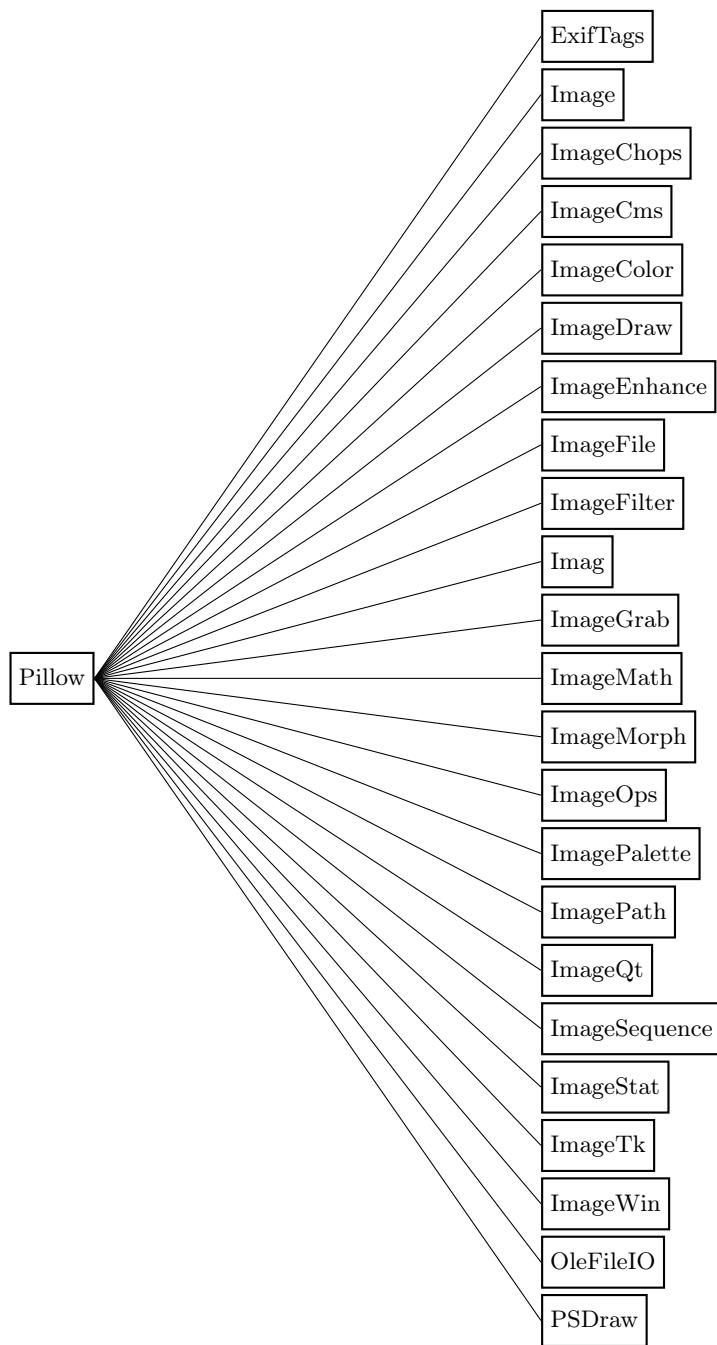
11.1 PIL

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。



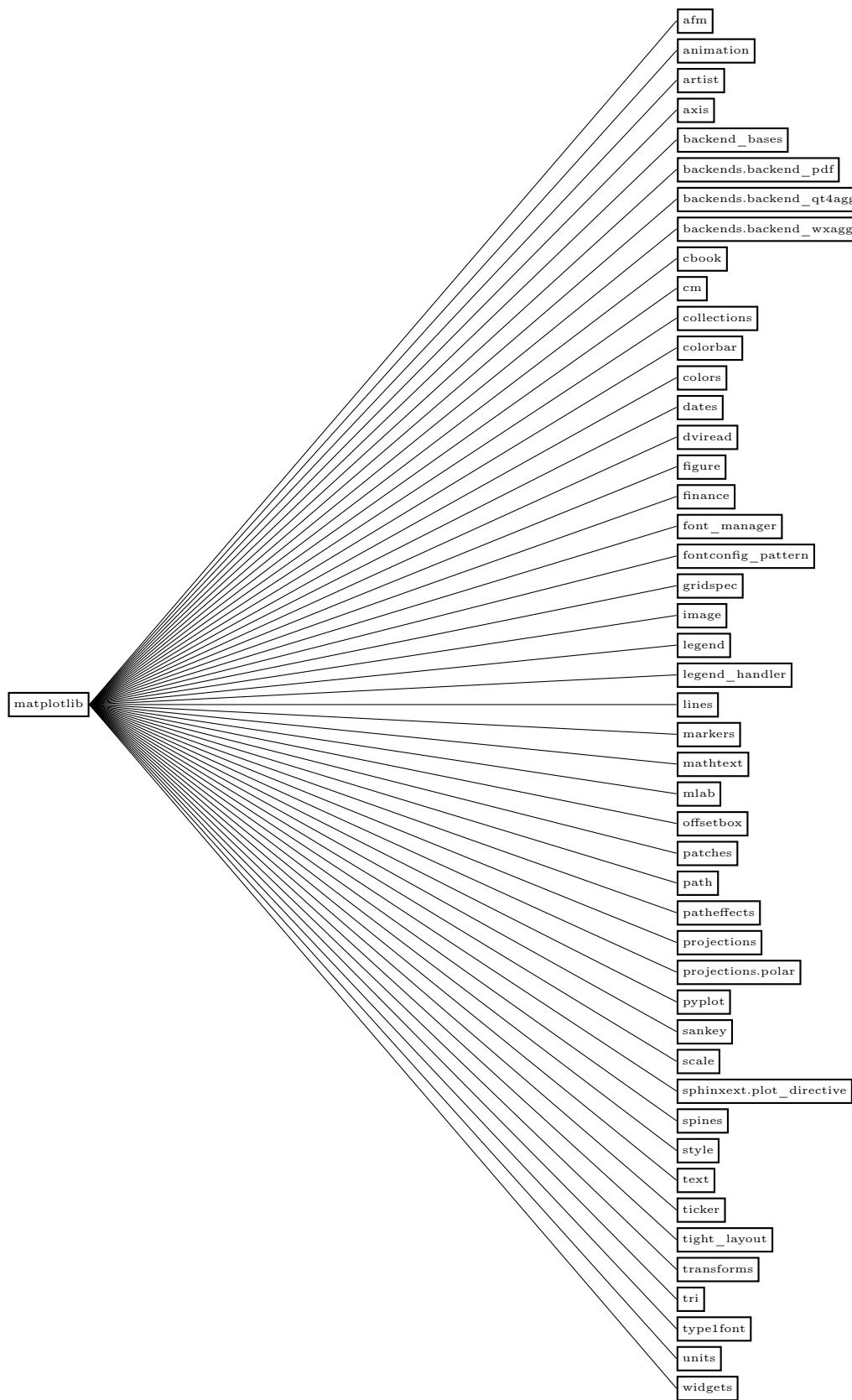
11.2 Pillow

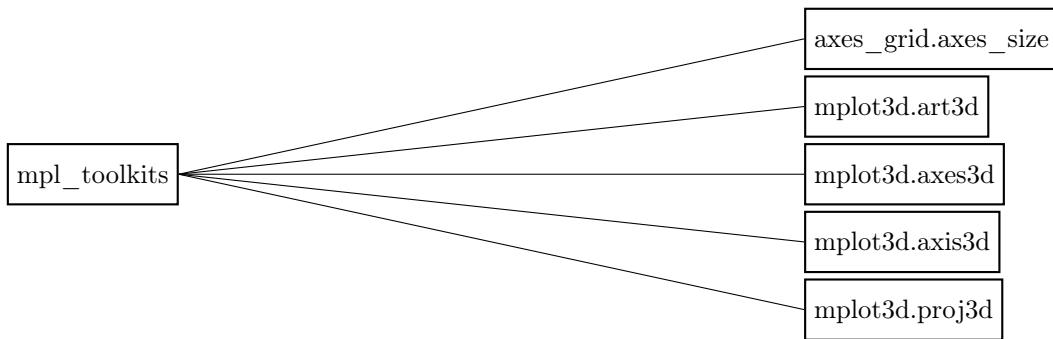
PIL のから分枝したもので、PIL が未対応である Python 3.X にも対応しています。また、Python のパッケージ管理ツールの `setupinstall` にも PIL と違い対応しています。



11.3 Matplotlib

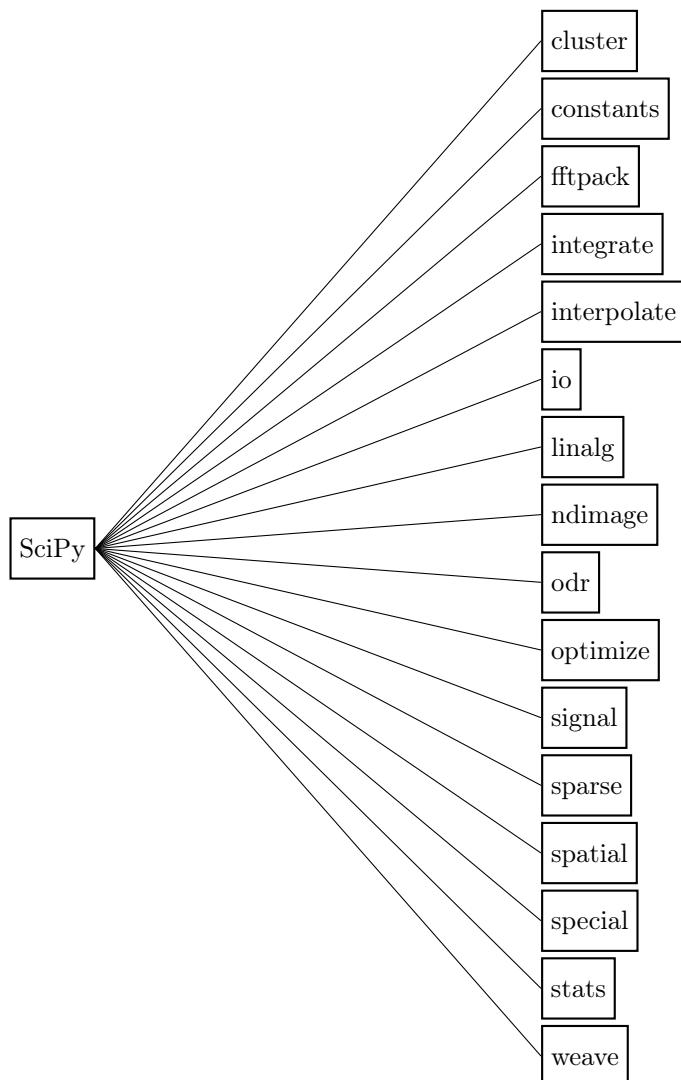
Python の配列ライブラリである NumPy を基に構成されたライブラリで、主に 2D グラフに関する処理が可能です。





11.4 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, 勁草書房, 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] 後藤和茂, BLAS の概要 (http://jasp.ism.ac.jp/kinou2sg/contents/RTutorial_Goto1211.pdf), 2006.
- [9] 柴田有, グノーシスと古代宇宙論, 勁草書房, 1982.
- [10] 藤野登, 論理学 -伝統的形式論理学-, 内田老鶴園, 2003
- [11] 田中尚夫, 選択公理と数学, 星雲社, 1987
- [12] フレーゲ, フレーゲ著作集 1 概念記法, 勁草書房, 1999.
- [13] フレーゲ, フレーゲ著作集 3 算術の基本法則, 勁草書房, 2000.
- [14] 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.
- [15] ポエティウス(著), 永嶋哲也(訳註), ポエティウス「イサゴーゲー第二註解」,
http://www002.upp.so-net.ne.jp/tetsu/study/t01_boepor.pdf
- [16] 山内志朗, 普遍論争, 平凡社ライブラリー, 2008
- [17] 横田博史, はじめての Maxima, I/O Books, 工学社, 2006.
- [18] 横田博史, はじめての Maxima 改訂 α 版 (MathLibre に収録)
- [19] 横田博史, 数値計算・可視化ツール Yorick, I/O Books, 工学社, 2010.
- [20] J.Barnes, PORPHYRY INTRODUCTION, Oxford University Press, 2006.
- [21] Boethius, Isagoge, <http://www.forumromanum.org/literature/boethius/isag.html>
- [22] G. J. Brose, MATLAB 数値解析, Ohmsha, 1998.
- [23] Mac Lane, The Category theory for working Mathematician, Springer
- [24] Mac Lane, Moerdijk, Sheaves in Geometry and Logic, A First Introduction to

- Topos Theory, Springer Verlag, 1992.
- [25] Porphyry, Introduction(Iagoge) to the logical Categories of Aristotle,
http://www.ccel.org/cCEL/pearse/morefathers/files/poRphyry_isagogue_01_intro.htm
- [26] Porphyry, Letter to Marcella, http://www.tertullian.org/fathers/poRphyry_marcella_02_text.htm
- [27] B.Russell, The Principles of Mathematics,W.W.Norton & Company,Inc.,1996.
- [28] B.Russell & A.N.Whitehead,Principia Mathematica to *56, Cambridge Mathematical Library,Cambridge University Press,1997.
- [29] Diving into Python: <http://www.diveintopython.net/toc/index.html>
- [30] MathWorks 日本: <http://www.mathworks.co.jp/>
- [31] アテナイの学堂 <http://ja.wikipedia.org/wiki/アテナイの学堂>

索引

Python

オブジェクト
 コンテナ, container, 114
 同一性値, identity, 112
 値, value, 112
 型, type, 112
 クラス, 120
 クラスインスタンス, 126
 クラスメソッド オブジェクト, 127
 コード オブジェクト, 127
 スライス オブジェクト, 127
 静的メソッド オブジェクト, 127
 トレースバック オブジェクト, 127
 ファイル, 126
 フレーム オブジェクト, 127
 変更可能, mutable, 112
 変更不能 immutable, 112
 モジュール, 120
オブジェクトの型
 ByteArray 型, 118
 Ellipsis, 115
 Frozen Sets, 118
 None, 114
 NotImplemented, 115
 Sets, 118
 UNICODE 文字列型, 118
 数 numbers.Number, 116
 組込函数型, 119
 組込メソッド型, 119
 クラス インスタンス型, 120
 クラスタイプ型, 119
 古典的クラ型ス, 119
 辞書, 119
 実数型, numbers.Real, 117
集合, 118
 整数型, plain integer, 116
 生成函数型, 119
 対応付け集合, 118
 タプル型, tuple, 118
 長整数型, long integer, 116
 文字列型, 118
 ユーザ定義函数型, 119
 ユーザ定義メソッド型, 119
 呼出可能, 119
 リスト型, 118
 列, 117

行
 エンコード宣言, 105
 空行, blank line, 105

字下げ, indentation, 105
注釈, comment, 104
物理行, 104
論理行, 104
スコープ (scope), 135
トークン
 トークン (token), 104
 リテラル, 104, 106
 DEDENT, 104
 INDENT, 104
 NEWLINE, 104
 演算子, 104
 キーワード, 104, 106
 空白文字, 104
 識別子, 104, 105
 字下げ (indentation), 104
名前, 135
 None, 114
 NotImplemented, 115
 参照, 135
 名前, 106
 名前空間, 106, 135
リテラル
 演算子, 111
 虚数リテラル, 108, 109
 区切文字, デリミタ, delimiter, 111
 数値リテラル, 106, 108
 整数リテラル, 108, 109
 单文字列, 107
 長整数リテラル, 108, 109
 長文字列, 107
 浮動小数点数リテラル, 108, 109
 文字, 107
 文字列リテラル, 106, 107
演算子
 is, 112
函数
 id(), 112
 len(), 117, 118
 makefile(), 126
 open(), 126
 os.open(), 126
 os.open(), 126
 type(), 112
構成子
 bytearrays(), 118
 classmethod(), 127
 frozenset(), 118
 set(), 118

staticmethod(), 127
 文
 import, 120
 メソッド
 __abs__(), 131
 __add__(), 130
 __and__(), 130
 __call__(), 120
 __cmp__(), 130
 __del__(), 128
 __delattr__(), 132
 __delete__(), 134
 __eq__(), 129
 __floordiv__(), 130
 __ge__(), 129
 __get__(), 133
 __getattr__(), 131
 __getattribute__(), 132
 __gt__(), 129
 __hash__(), 132
 __init__(), 119, 128
 __le__(), 129
 __lshift__(), 130
 __lt__(), 129
 __mod__(), 130
 __mul__(), 130
 __ne__(), 129
 __neg__(), 131
 __new__(), 119, 128
 __nonzero__(), 132
 __or__(), 130
 __pos__(), 131
 __pow__(), 130
 __radd__(), 131
 __rand__(), 131
 __repr__(), 129
 __rfloordiv__(), 131
 __rlshift__(), 131
 __rmod__(), 131
 __rmul__(), 131
 __ror__(), 131
 __rpow__(), 131
 __rrshift__(), 131
 __rshift__(), 130
 __rsub__(), 131
 __rtruediv__(), 131
 __rxor__(), 131
 __set__(), 133
 __setattr__(), 131
 __str__(), 129
 __sub__(), 130
 __truediv__(), 130
 __unicode__(), 130
 __xor__(), 130
 close(), 114
 PEP
 PEP, 17, 31, 33, 153
 PEP-1, 153, 154
 PEP-8, 84, 105, 156

PEP-20, 154
 PEP-253, 124
 PEP-256, 162
 PEP-257, 87, 156, 161, 162
 PEP-258, 162
 PEP-287, 162
 え
 エスケープシーケンス, 108
 か
 鍵(キー), 119
 鍵値, 119
 拡張スライス構文, 127
 拡張スライス操作, 117
 クラス
 MRO(Method Resolution Order, 122
 C3 MRO(C3 Method Resolution Order,
 124
 基底クラス, 98, 121
 継承, 98
 サブクラス, 98
 スーパークラス, 98
 派生クラス, 98, 121
 こ
 構成子, コンストラクタ, constructor, 112
 塵回収, garbage-collection, 114
 し
 実行フレーム, 127
 消去子, デストラクタ, destructor, 112
 す
 スライス操作, 117
 と
 到達不能の状態, unreachable, 114
 特殊属性, 114, 126
 特殊メソッド, 127
 は
 バイトコード, bytecode, 127
 ふ
 文書文字列, 107
 れ
 例外スタックトレース, 127
 例外
 NameError 例外, 106
 人名
 R
 Rossum, Guid van, 33
 あ
 アウグスティヌス,Augustinus Hippoensis,
 Augustine of Hippo, 35
 アリストテレス,Αριστοτέλης,Aristotle, 35,
 39, 193, 197
 い
 イブン・ルシュド,ibn rušd,Averroes, 195
 え
 エピメニデス,Ἐπιμενίδες,Epimenides, 38
 か
 カントール,Cantor, 38
 き
 キンディー,al-Kindi, 195
 つ

| | |
|--|---|
| <p>ツェルメロ,Zermelo, 47</p> <p>ふ</p> <ul style="list-style-type: none"> プラトン,Πλάτων,Plato, 34, 198 フレーゲ, Frege, 38 フレンケル,Frankel, 47 プロティノス,τεκτγρεκΠλωτίνος,Plotinus, 194 <p>ほ</p> <ul style="list-style-type: none"> ボアンカレ,Poincaré, 38 ボエティウス,Boethius, 193 ボルビュリオス,Πορφύριος,Porphry, 39, 193 <p>ら</p> <ul style="list-style-type: none"> ラッセル, Russel, 38 <p>哲学</p> <p>し</p> <ul style="list-style-type: none"> 新プラトン主義, 194 <p>す</p> <ul style="list-style-type: none"> ストア派, 195 <p>ふ</p> <ul style="list-style-type: none"> 普遍論争, 193, 196 <p>ほ</p> <ul style="list-style-type: none"> ボルビュリオスの樹, 196 | <p>上位の概念, 37</p> <p>属性, 36</p> <p>単独概念, 39</p> <p>徵表, 36</p> <p>内包, 37</p> <p>内包外延反比例増減の法則, 38</p> <p>範疇, 39</p> <p>類, 36</p> <p>類概念, 36, 39</p> <p>カテゴリー, 37</p> <p>き</p> <ul style="list-style-type: none"> 基体,ὑπόκείμενον,subject, 209 逆理 <ul style="list-style-type: none"> Banach-Tarski の逆理, 53 クレタ人の~, 38 床屋の~, 37 ラッセルの~, 37 <p>近接,προδεξές,proximate, 202</p> <p>く</p> <ul style="list-style-type: none"> 偶有性,συμβεβηκός,accident, 197, 199, 209, 213, 215–217 <p>け</p> <ul style="list-style-type: none"> 繫辞,copula, 40 形相,εἶδος,shape, 211 形相 εἶδος,eidos, 42 圜 <ul style="list-style-type: none"> CCC(デカルト閉圜), 82 epi, 60 iso, 60 mono, 60 可換図式, 59 基本トポス, 81 グラフ, 61 結合律, 59 恒等射, 同一矢, 59 始域,domamin, 57 自然変換, 66 射(矢), 57 終域,codomain, 57 双対, 61 双対圜, 62 対象, 57 デカルト閉圜, 81 転置, 74 同一矢, 恒等射, 59 同一矢の公理, 59 特性写像, 81 評価, 74 部分対象分類子 (Object classifier), 81 メタグラフ, 57 メタ圜, 60 矢(射), 57 矢の合成, 58 離散圜, 63 <p>圜論</p> <ul style="list-style-type: none"> コンマ圜, 67 |
|--|---|

| | | |
|--|---|--|
| 始対象, 62 | と | 同名異義的, 203, 212 |
| 終対象, 62 | | 同名同義的, 203 |
| スライス圏, 67 | | 特定的, <i>ειδηκός</i> ,special, 200, 201 |
| こ | | 特有性, <i>τύπος</i> ,property, 197, 199, 208, 212, 214, 215, 217 |
| 個体, <i>άτομος</i> ,individual, 202 | は | 範疇, 37 |
| し | | |
| 実在論, 40 | ふ | 物質, <i>ὕλη</i> ,matter, 211 |
| 種, <i>είδος</i> ,species, 197–200, 210–212, 214–216 | | 物体, <i>σώμα</i> ,body, 201 |
| 集合論 | | 普遍, 36 |
| ∈-モデル, 55 | | プラトニズム, 40 |
| ZFC 公理系, 53 | | 分有する, <i>μετέχειν</i> ,participate, 212 |
| 1-要素集合, 49 | ほ | 本質的存在, <i>οὐσία</i> ,substance, 201 |
| 宇宙, 55 | | |
| 外延性公理, 49 | め | 命題 |
| 共通集合, 51 | | 可述的, 38 |
| 空集合公理, 50 | | 非可述的, 38 |
| クラス,class, 51 | ψ | 唯名論, 40 |
| 後者, 51 | | |
| ∈-構造, 55 | り | 離在可能, 206, 209 |
| 後続, 51 | | 離在不可能, 206, 209 |
| 恒等式, トートロジー, 55 | | |
| 順序対, 49 | る | 類, <i>γένος</i> ,genus, 197–199, 210–213 |
| 正則性公理, 52 | | |
| 選択公理, 53 | | |
| 大小関係, 50 | | |
| 置換公理図式, 51 | | |
| 超限順序数, 50 | | |
| 直積集合, 51 | | |
| 対公理, 49 | | |
| 対集合, 49 | | |
| 分出公理, 51 | | |
| 幂集合公理, 50 | | |
| 無限集合公理, 50 | | |
| モデル, 55 | | |
| 類, 51 | | |
| 和集合, 49 | | |
| 和集合公理, 49 | | |
| 述定の型=範疇, カテゴリー, 201 | | |
| 種差, <i>διαφορά</i> ,difference, 197, 205, 210, 211, 214, 215 | | |
| 述定, 199 | | |
| 順序数 | | |
| 後者, 55 | | |
| 後続, 55 | | |
| 順序数, 54 | | |
| 順序, 117 | | |
| 順序関係, 117 | | |
| 順序集合, 117 | | |
| せ | | |
| 説明規定, <i>λόγος</i> , 198 | | |
| そ | | |
| 綜合的, <i>γενικός</i> ,general, 201 | | |
| 存在含意, 40 | | |
| 存在動詞, 40 | | |