

統合数学環境 Sageへの招待

- Python で記述された統合数学環境 -

USO889 版

横田博史

平成 27 年 04 月 16 日 (木)

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 ©(2014) 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、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。

また、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。その為、この文書の二次配布はご遠慮願います。

平成 26 年 10 月 2 日 (火)

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	11
1.3 Sage が目指すものは?	20
1.4 Sage が包含するアプリケーションとライブラリ	22
1.5 一般の Python パッケージ	27
1.6 この本の方針	27
第 2 章 オブジェクト指向について	29
2.1 Sage と Python の関係	29
2.2 Python はどのような言語か	29
2.3 イデア論とオブジェクト指向プログラミング	30
2.4 集合論について	42
2.5 圈 (Category)	51
2.6 トポス (Topos)	69
第 3 章 Python について	73
3.1 この章の目的	73
3.2 簡素化された構文	73
3.3 Backus-Naur 記法 (BNF)	86
3.4 字句解析	90
3.5 データモデル	98
3.6 名前空間とスコープ	119
3.7 実行モデル	120
3.8 名前付けと束縛	120
3.9 例外	121
3.10 Python の式	122
3.11 単純文	129

3.12	複合文	133
第4章	Sage プログラムを書く上での指針	137
4.1	はじめに	137
4.2	PEP(Python Enhancement Proposal)について	137
4.3	ファイル名やディレクトリ名に関する指針	144
4.4	ライブラリに関する指針	144
4.5	文書文字列の利用について	145
第5章	数学的对象の表現	149
5.1	はじめに	149
5.2	数の構成	150
第6章	SQLite3 を使った解析	151
6.1	SQLite3 速習	151
第7章	Sage で画像処理	155
7.1	画像の読み込み	155
第8章	Sage の拡張	159
8.1	sagemath からのパッケージ入手	159
8.2	一般の Python パッケージのインストール	159
8.3	GNU R のパッケージのインストール	160
第9章	手引(ポリピュリオス)	161
9.1	概要	161
9.2	はじめに	163
9.3	類について	164
9.4	種について	166
9.5	種差について	170
9.6	固有性について	173
9.7	偶有性について	173
9.8	共通の特徴	174
9.9	類と種差	174
9.10	類と種	175
9.11	類と固有性	176
9.12	類と偶有性	176
9.13	種差と種	177

9.14 種差と固有性	178
9.15 種差と偶有性	178
9.16 種と固有性	179
9.17 種と偶有性	179
9.18 固有性と偶有性	180
参考文献	181
索引	183

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。それは *Mathematica* や MATLAB に匹敵する OSS の数学環境を実現するための手法・手段に尽きます。それは開発者の「車輪の再発明はしない」との言葉からも判るように、既存の優れたソフトウェアを取込むことで必要とされる機能を実現するというものです。このような開発が可能な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、動作環境にも余裕があるという 2 つの事実があります。これらの事実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のが現状です。とはいっても、扱っている事項が他の分野に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とどうでしょうか？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 2GB 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。

1.1.2 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、その当時の所謂、「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することになりました。こうして研究機関から商用化されたソフトウェアは企業の製品開発で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。この市場の成熟に合せて研究機関を母体とするベンチャー企業が当初想定していた大学や研究機関に近い研究者や専門家を中心とした利用者から単純化された手続に沿った作業を中心に行うオペレーター的な利用者が増えてゆき、非専門家でも操作が可能なシステムが市場からは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、設計では従来の図面描きから 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た程度で、紙ベースで他の工程に回すものでした。そのことでもあって、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、それを計算機を使って計算していました。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。とはいえ、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、そのためのノウハウも必要だったのですが、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。こうなると部品の接触問題を考えるだけでも曲線から曲面を考えなければならなくなり、もはや物差し、分度器やコンパスを使った座標の読み取り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。こうなってしまうとCADの図面データをそのまま解析ツールに入れてモデル化した方が遥かに効率的になります。それに元々が3次元データなので解析モデルを3次元化する必要もなく、そのまま解析すれば良いのです。それに加えて計算機も3次元モデルの処理に見合うだけの能力を持っているのです。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を行えれば、それまでの間的な作業、たとえば、図面からの座標の読み取を行って幾何的情報を取出すといった作業が不要になり、さらに各工程でモデルを共有することで全体的な作業の流れがより簡素化されて明瞭なものとなります。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^{*1}。そして、製品の開発にしても従来の単発的な処理ではなく、モデルを共有することで総合的な開発を行うようになっています。このことは「モデルベース開発のための複合物理領域モデリング」[6]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^{*2}では Modelica 言語でモデルを記述し、それから代数方程式を生成する手法になっています。また、製品がどのようなものであるべきかを設計する人は解析の専門家であるとは限りません。あくまでも設計を行う上での参考として色々と計算する程度で、解析に重点を置くよりはプロジェクトや文書の作成や管理の方が重要であったりもします。そのために CAE の裾野が広がるにつれて当初の利用者であった解析の専門家向けから、文書作成やプロジェクトの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始めますが、この時点では Tcl/Tk で構築した GUI を従来のアプリケーションに被せる方式がよく用いられてました。

ここで解析ソフトウェアを機能や目的で大きく分けると、モデルの構築を行うプリ、解析を行うソルバ、結果の可視化等の後処理を行うポストの三種類に分類できます。これらは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、まとめて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。そのような冒險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

^{*1} 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 Web 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。

^{*2} 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 言語にまとめて対応している数少ないパッケージです。

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、拡張が容易なより一般的な言語を用いる方が間違いがありません。

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から幅広い分野で使われており、最適化も進んでいるためにパフォーマンスが高い言語の一つになっています。現在、Java はネットワーク越しに Oracle 等の DB の制御を行うアプリケーションや、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の開発でも用いられています。

一方の Python は計算処理もそれほど高速ではありませんが、言語仕様は比較的単純の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、拡張性の高さもその特徴として挙げられるでしょう。さらに 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のような働きを Python にさせることができます。そのために処理速度がさほど高速でなくても、苦手とする部分は既存のアプリケーションやライブラリで処理させて、その結果だけ横取りするようにすれば処理の問題は大きく改善されます。同様に 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問題とならなくなっています。また、前述のように Python の言語仕様は他の言語と比べて簡素であるために、プログラムの学習に費す手間も少なく済み、逆に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、独自の奇妙な処理言語を作成しなくとも Python をシステムの処理言語とすることができるとともに加え、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か 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です。

1.1.3 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動作が異なることもあります。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ります。

このような処理に慣れてしまえば「あたりまえ」のことになるでしょうが、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る数式処理ソフトに Singular があります。このアプリケーションでは「環」と呼ばれる数学上の対象を「世界」として定めることでやっと多項式の処理が行えますが、何も指定しなければ整数の四則演算しかできません。そのために計算を行うために何かと必要なクラスのインスタンスを生成する必要が生じます。このときに前提となる概念の知識があれば利用する上で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」になってしまい傾向があります。このことは多項式の展開を覚えたての中学生にさえ、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に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。そのために、多項式の計算で、「**最初に整数係数の多項**

式環を生成して、それから因子分解メソッド `factor` を使って式の因子分解を行う」と覚えなくても、「多項式を入力して、その式に函数 `factor()` を作用させる」だけで済ますことも可能なのです。この点は一般の利用者にとっては非常に有り難い点ではないでしょうか？

この実例を示しておきましょう：

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

ここで式の因子分解ではメソッドを利用して ‘`a.factor()`’ で分解したり、函数として ‘`factor(a)`’ で分解させています。どちらにしても多項式環は表に出てはいません。とは言え、多項式環が表から見えないだけで実際は厳として存在しているのです。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を取り入れている Python を用いている 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、プログラムの普遍性を持たせることができることが挙げられるでしょう。これらのことはプログラムを「**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。

1.1.4 電池込みだよ

Sage は Python で記述された数式処理のためのパッケージ SymPy を基底として、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が、通常の利用ではそのことを意識する必要はありません。しかし、必要に応じてこれらの専門ツールだけを表に出して使えるようになっています、その意味でも、Sage は**数学上の問題を解くための計算機環境**としての側面を強く持っています。



図 1.1 Web 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、フロントエンドにし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では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。この発想法は Python のいわゆる「電池込みだよ (Battery Included)」^{*3}を彷彿させるものもあります。

1.1.5 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 社の計算機で動作する

^{*3} 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。

OSX もあります。ところで、Linux に限定してもディストリビューションの違い、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、仮想化環境 (VMware や Oracle VirtualBox) を活用しています。この場合は Web 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が含まれるために 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、そして、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、そう言った趣味の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 OS X のターミナルから起動する他の Unix 版に対応するバイナリ版と OSX の通常のアプリケーションとしてインストールできるものの二種類があります。何が何でも OSX を Unix として利用するのでない限り、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。その理由ですが、Sage.app を起動すると Finder 上に Sage のアイコンが現れ、図 1.2 に示すように、そこから選べるメニューには、Sage のノートブック形式やターミナル上の CUI 形式の Sage の起動、さらには Maxima, R, Singular 等のアプリケーションを個別にターミナル上で動かすことができます：

だから OSX で科学技術計算を Python で行なわれている方は、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。しかし、OSX の場合は、FINK, MacPort や Homebrew といったパッケージ管理は、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 をインストールして、その SuSE 上でパッケージ管理を行っている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

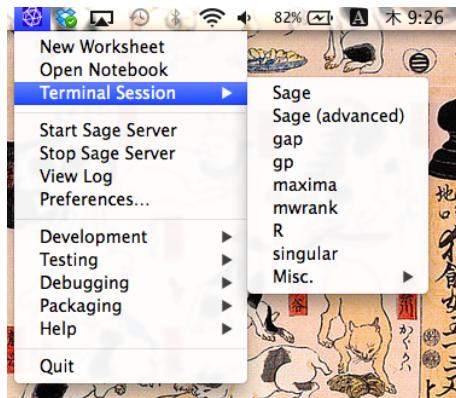


図 1.2 Sage.app のメニュー

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った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^{*4}。

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、いつそのこと MathLibre^{*5}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、さらには日本語の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！ そのようなソフトウェアのカタログとしても使えるでしょう。

1.1.6 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、こちらはまだ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ですので、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。また、日本でも **Sage Days in Japan** というワークショップが開かれることができます。売り物の書籍としては「群論の味わい-置換群で解き明かすルービックキューブと 15 パズル」が現在販売されている程度です。その他には「**Sage for Newbies**」の抄訳の「はじめての Sage」が WEB で公開されています。

^{*4} 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。

^{*5} 以前は「**KNOPPIX/Math**」として開発・配布されていました

す^{*6}. 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 を学習するだけでよく, この Python を使う上で要求される水準は高いものではなく, その上, Python 自体が学習し易いという長所があるのです.

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 が予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:

```
yokota@Zuse:~> sage
```

```
| Sage Version 5.7, Release Date: 2013-02-19
| 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
```

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 では幂乗の演算子は “`**`” 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 “`^`” は幂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あり, 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

^{*6} 筆者のサイトから辿って入手することができます.

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、この割当て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、関数 `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関数 `type()` を用います：

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

この関数 `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ういう処理が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 の構文や関数、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 WEB Browser を UI として用いる場合でも同様に表示が行われます。この機能は関数 `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。ここで ‘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いますが、今回処理す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っていませんね。これは何故でしょうか？そこで、函数 type() で名前 ‘x’ の型を確認しておきましょう:

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

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

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

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**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となっており、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 (name space)**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。つまり、ここでは函数 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を用いた厳密解を返却しますが、代数的に解けない場合は、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、help 函数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 *7 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 の機能のために素の Python 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 `notebook()` と入力することで Web ブラウザが立ち上がり、その Web 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 が立ち上がります。また、OSX 上の Sage.app を利用しているのであれば、普通の OSX アプリケーションと同様に Launchpad から呼び出せば WEB ブラウザを起動してノートブック形式で立ち上がります。また、Finder 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開いたり、仮想端末から開くこともできます。

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 WEB ブラウザに

*7 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。

表示され、それから **New Worksheet** 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 <http://www.sagenb.org/> 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 が使えないからなので Android や iPhone/iPad では使えません^{*8}。なお、メモリ等に制限がありますが、ちょっとした計算で使う用途であれば十分でしょう。こちらの初回の利用も適当なパスワードの設定が必要となります。

さて、Sage のノートブックで式を評価するためには *Mathematica* 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。他の数式処理と比較してやや重いかもしれません、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昔風のキャラクタを用いた数式の表示となります。ただし、**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 PyLab を用いることにします。ライブラリの読み込みは Python では `import` フィルを用います：

```
sage: import Image
sage: skell=Image.open('/home/yokota/kodairi.jpg')
sage: skell.show()
sage: type(skell)
instance
sage:
```

この例では `Image` ライブラリを `import` で Sage に読み込みます。それから `Image` ライブラリに含まれている `open` モジュールを使って ‘/home/yokota/kodairi.jpg’ を読み込んで `show` モジュールで変数 `skell` に束縛された画像オブジェクトを表示させています。この一連の処理は Python でもほぼ同様です。何故、「**ほぼ**」が付く理由は、Sage では PIL(Python Image Library) と呼ばれる画像処理向けのライブラリを読み込んでいるために、`Image` モジュールの読み込みでは `import` を使って ‘`import Image`’ だけで済むからです。通常の Python では、PIL か Pillow^{*9} をインストールしている環境で、‘`from PIL`

^{*8} なお、3 次元グラフの描画で用いられている 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。

^{*9} Pillow は PIL から分岐したライブラリで、PIL と違い、Python 3.x と setuptools に対応しています。

`import Image'` としなければなりません。

ここで Python ではライブラリの読み込みを `import` で行いますが、ここで ‘`import < ライブラリ名 ’` で読み込むと、そのライブラリで定義されるモジュールやオブジェクトの頭に ‘< ライブラリ名 ’ で指示されるライブラリ名を函数名に追加した名前を用いなければなりません。この例では単に ‘`open`’ ではなく、ライブラリ名の ‘`Image`’ を `open` モジュールの頭に追加した ‘`Image.open`’ で函数の呼出を行わなければなりません。そこで ‘`from < ライブラリ名 import*`’ とすればライブラリ名を外して ‘`open`’ のように用いることができるようになりますが、同名のモジュールやオブジェクトが Python 上に存在するときに意図しない上書きが行われるために、それよりも ‘`import < ライブラリ名 as < 別名 ’` で使い易い名前に置き換えることが推奨されています。たとえば、‘`import Image as im`’ でライブラリを読み込んだときに `Image` ライブラリの `open` モジュールを用いたければ ‘`im.open`’ とすることで利用できるのです。それから取り込んだ画像の外部アプリケーションを使った表示は `show` モジュールでできます。図 1.3 に実際の表示画像を示しておきましょう：



図 1.3 読込画像の表示例

この `show` モジュールによる表示では、UNIX 環境の Python にてアプリケーション `xv` を標準で利用する設定が行われているために `xv` がインストールされていない環境では問題が生じます。その場合は `shwo` モジュールの引数として表示アプリケーションを指定することで問題を回避することができます。たとえば、ImageMagick の `display` を画像表示のアプリケーションとして利用したければ `im.show(command='display')` とすればよいのです。

さて、ここで読み込んだ画像データは `type` フェンスで調べると `instance` であることが判ります。つまり、画像データそのものを数値行列として取込んでいる訳ではありません。この状態では画像の処理を数値行列の操作に置き換えて色々と遊ぶことはまだできません。ここで NumPy ライブラリを読み込んで NumPy ライブラリに含まれている函数 `array()` で NumPy の配列に変換することが可能で、こうすると MATLAB 風の処理も行えるの

で色々と遊べることになります。なお、NumPy の関数 array() によって画像は赤 (R), 緑 (G), 青 (B) 単位の 0 から 255 までの範囲の輝度で表現された数値配列に変換されます。

そこで、今度は matplotlib ライブラリに含まれている PyLAB ライブラリを用いて画像データを行列データとして取込む例を示しておきましょう：

```
sage: import pylab as plt
sage: fig1 = plt.imread('/home/yokota/kodairi.png')
sage: type(fig1)
numpy.ndarray
sage: im.save('test.png')
```

ここでは PyLAB ライブラリの読み込みを ‘import pylab as plt’ によって ‘plt’ で置き換えてています。ここでの表記は通常の Python であれば ‘from matplotlib import pylab as plt’ としなければなりませんが、Sage ではあらかじめ ‘import matplotlib’ が実行されているために ‘from matplotlib’ が省略できているのです。また、こうすることで PyLAB ライブラリに含まれる画像ファイルの読み込みを行う imread モジュールの呼出を ‘import pylab’ であれば ‘pylab.imread’ で行うところが ‘plt.imread’ で行えます。また、この手続で読み込まれた画像データを type 関数で調べると numpy.ndarray となっていることが判ります。この ‘numpy’ は Python で MATLAB 風の行列処理を行うための NumPy ライブラリ指しており、そこで定義された配列データであることを示しています。ちなみに imread モジュールによる画像の読み込みでは、256 階調の整数値行列ではなく、0 から 1 までの浮動小数点数値行列が返却されます。

ところで、実は pylab を使わなくても NumPy の配列への変換は可能です：

```
sage: import numpy as np
sage: mat = np.array(im)
sage: mat.shape
(386, 800, 3)
```

この例では最初に NumPy を読み込みますが、このときに NumPy の名前を ‘np’ で置き換え、それからメソッド Image.open() で開いた画像オブジェクト im を引数として NumPy の関数 array() で NumPy の配列に変換しています。このときの行列の大きさは shape() メソッドで調べられ、このことから縦 386、横 800 画素の画像であることが判ります。

ここまで処理は Sage でなくても Python で PIL/Pillow, NumPy や PyLab があればできることです。そこで折角ノートブック形式の UI を使っているので、今度は絵をノートに貼ってしまいましょう。これには特殊な処理は不要です。単純に Image モジュールの 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、大きさを調整してくれる訳ではありません：

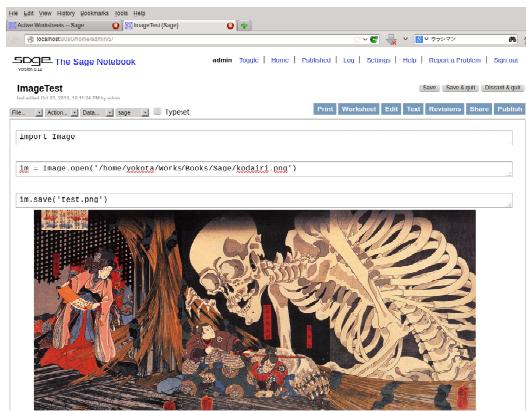


図 1.4 ノートブックへの画像の表示例

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。

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, Ubuntu, CentOS, Gentoo 等々といったさまざまなディストリビューションがあり、それらのディストリビューションの違い^{*10}に加えて、これらが公開された時期やライブラリの更新による差異も無視できません。

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

^{*10} パッケージ管理の仕組、/etc 以下のディレクトリの構成の違い等があります。

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も良いのであれば入手に Torrent を使う方法もあるのです¹¹。

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¹² 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？そのためには 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。この Sage に含まれた GNU R は公開された R 向けに公開されたパッケージを全て含むものではなく、基本的なパッケージを含むものです。この GNU R に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。

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 のコードに単純に変換し

*¹¹ 17 年程前はブロードバンドも一般的でなかったために、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。

*¹² 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。

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 に付属の線形代え数ライブラリには 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^{*13}。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、数学の諸問題に対応可能な環境を構築するという非常に野心的なプロジェクトであるという側面が如実に現われているのです。

1.4 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。

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。そして、Python は Sage 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。まず、利用者との対話処理を行うためのシェル^{*14}として **IPython** を用いています。この IPython を用いることで、利用者の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。それから大規模な配列や行列を扱うための数値計算ライブラリの **NumPy**、この NumPy を基に技術計算を強化した **SciPy**、NumPy を用いてグラフ表示

^{*13} とは言え、商用の LAPACK や GotoBLAS と比べると劣ります。

^{*14} 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。

を行うライブラリの **Matplotlib** があります。これらのライブラリを用いることで一般的な数式処理ソフトが苦手とする数値計算を MATLAB と同程度の処理速度で実行することを可能にしています。そして、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 環境のみに対応していることもあるって PIL から分岐した Pillow が Python では広く用いられています。そして、数式そのものを Python で扱うために Python で初等的な数式処理が行えるライブラリの **SymPy** を導入しています。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なっています。これらのライブラリだけで MATLAB 本体とほぼ同程度の機能を持ち、多項式、初等函数や特殊函数を含む式の微積分を含む数値計算や数式処理が可能なシステムができることがあります。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-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 だけで同様な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、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 ではグラフの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す。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）と呼ばれるマークアップ言語のテキストファイルとして記述されています。この ReST から Python で記述された **Docutils** を用いることで、HTML、XML や LaTeX 等の各書式のファイルへ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 Web Browser 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 処理言語

Sage が利用するアプリケーションやライブラリの構築やアプリケーションの起動のために Common Lisp の ECL, FORTRAN が含まれます。ECL は組込向けで用いられる Common Lisp で Maxima を動作させるために用いられます。なお, Sage のオプションのパッケージとして Common Lisp の CLISP も提供されています。

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 単体だけでも多項式の展開や因子分解といった基本的な処理、初等函数の微分や積分といった計算が可能ですが、この汎用の数式処理である 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は劣ります。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ました。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 等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はいません。しかし、岡田氏の RjpWiki ^{*15}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

^{*15} <http://www.okada.jp.org/RWiki/>

Sage の 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 Tachion を包含しています。Tachion 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。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、現在は様々な方面でも利用されています。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実は非常に多くあります。

1.4.5 数値計算ライブラリ

線形代数のライブラリとして **ATLAS**, **BLAS**, **LAPACK** を包含しています。ここで ATLAS は GotoBLAS 程ではないにせよ CPU 最適化が行なわれている LAPACK ライブラリです。ちなみに Sage をソースファイルからコンパイルするときには、この ATLAS 関連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 は Shift-JIS, EUC, UTF-8 等の文字コード変換で用いられます。

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

を前提としています。 **IML** は整数行列ライブラリで ATLAS/BLAS+GMP と併用が前提となっています。 **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, $\text{GP}(q)$ 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で, **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, 有限群の関手を扱い, **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。 Python から用いる場合, ‘sqlite3.connect’ で DB に接続し, execute モジュールを用いて DB 操作が行えます。 **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 は橍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** は橍円曲線パッケージです。

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。

```
sagenb sagetex-2.2.5.spkg sage_root-4.7.1.spkg sage_scripts-4.7.1.spkg
```

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、予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可能です（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て、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へのアクセスが可能な 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 を遣わないという理由は考えられません。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ことになります。

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たスクリプト言語であり、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も書いてあります。

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 の影響を強く受けますが、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が、この章では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いうことを意味しますが、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であるかを説明するためにプラトン ($\Pi\lambda\alpha\tau\omega\nu$, 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 ($\iota\delta\epsilon\alpha$, 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「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」とも言えるでしょう。

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。このときにオブジェクトがイデアに対応し、計算機で扱う個々のデータそのものはオブジェクトが計算機内部で「実体化したもの」と説明することができます。ちなみにオブジェクトが計算機上のデータとして「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」を「インスタンス (instance)」と呼びますが、「イデアの現世における実体化」も英語では同じ「instantiation」になります。

ところでイデアの実体化の原因とな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「**鋳型**」として存在するとしても、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 ($\deltaημιουργός$, 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 ($\mathcal{E}\rho\varsigma$) がイデアの美に憧れたからだとか述べていますが、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、その基準も明瞭なものではありません。この辺りは後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。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 ($\alpha\lambda\chi\varepsilon\nu\varepsilon$)」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、その後のグノーシス主義 ($\gamma\nu\omega\varsigma$) 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)[7] によると人間は元来、神の子であつ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粋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 ($\phi\psi\sigma\varsigma$, 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であるという二面性を持つことへの説明となっていますが、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^{*2}。

2.3.2 類, 種, 種差, 属性, 偶有

ここで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つて類, 種, 種差, 属性と偶有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。まず

^{*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} 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ポルピュリオスのエイサゴーゲー ($\varepsilon\iota\sigma\alpha\gamma\omega\gamma\varepsilon$) 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然程多くは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る。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」[13] を参照 1 して下さい。

「類」と「種」は「もののあつまり」です。ただし、類は種を包含するもので、さらに‘それは何であるか’という問に対するものです。つまり、「それはBである」といったときの述語のBに相当します。「種」は類の下で分類されたもので、‘それは何であるか’という問に対しては‘それ’に対応するもので、述語の‘何’よりも一段低いものです。そして「種差」はその類を分類する上での理由、「固有性」はそのもの固有のもの、特性であり、それに対して「偶有性」はその時点では持っている性質になります。たとえば類を動物とするなら人間はその動物の下の種、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いうものです。そして、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、より多い、より少ないといった状態の増減も許容するものとなります。

2.3.3 概念について

ところで「思惟にのみによって知覚されるイデア」は「概念, concept」に似ていますが、その方向性が異なっています。ここで「概念」がどのようにして得られるかと言えば、対象を特徴付けるものである「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が、このように複数の主語の述語になる性質のことを「普遍」といいます。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、「この猫がたまです」という風に個体を特定するものであって普遍的なものではありません。

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」以外の種概念も含まれるので「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「最下位の概念」が「個体」であり、逆に「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。

では「概念」はどのように表記され得るでしょうか？この概念の表現の方法には二通りの表現方法、一つは「内包」ともう一つは「外延」があります。ここで「内包」は概念が持つ微表/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。たとえば、「猫」という概念であれば、内包が「動物である」、「4本足で歩く」、「柔らかい肉球を持つ」、「ニャオと鳴く」等の属性から構成されるでしょう。一方で「猫」という概念の外延には「三毛猫」、「虎猫」、「白猫」、「黒猫」、...といった「毛並み」という「属性」で猫を分類したのであれば、その種を列記したものとなるでしょう。また、「粟根さんの飼い猫のタマ」等と個体を指名した文や名前を列記する方法もあるでしょう。このように内包は概念を表現する述語から構成され、外延は概念に対応する具体的な個体、あるいは類や種から構成されます。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^{*3}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) 集合論に批判的であったポアンカレ (Poincaré) は著書「科学と方法」([12]) 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ことを述べています ([12], p.204)。たとえば、「偶数の集合」や「身長 170cm 以下の人の集合」といった集合の定義では「自然数の集合」や「人間の集合」といった集合の概念に触れずに集合がきちんと定義ができます。これらの定義方法を「可述的」と呼びますが、床屋の逆理のように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^{*4}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のラッセルの逆理の排除ができても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。この点は非常に厄介で、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公理系によって「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、集合の公理系からラッセルの逆理を排除しています。

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

^{*3} 紀元前 6 世紀頃のクレタのクノッソスの哲学者だそうです

^{*4} 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」[11] やラッセルの「Principles of Mathematics」[21] 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。

近付く結果、外延を構成する個体が絞られてしまい、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あんばいです。

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。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」となります。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となるので個体に最も近い概念になります。

2.3.4 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。またこのカテゴリー論への入門書として古代から有名なポルピュリオス ($\Pi\sigma\pi\varphi\rho\iota\varsigma$, Porphyry of Tyre) のイサゴーヶ (Isagoge[17][20], Εἰσαγωγή, 「手引き」という意味です)^{*5}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。この一節がボエティウスによるイサゴーヶ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ていわゆる「**普遍論争**」を引き起すことになります[13]。また、この手引きから分類学で見られる「**ポルピュリオスの木 (Arbo Porphyriana)**」が得られています。

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

*5 英訳はボエティウスのラテン語訳を Owen が翻訳したもの [20] と Barnes がギリシャ語文献から翻訳したもの [17] があり、前者はイサゴーヶがカテゴリー論の入門書との立場、それに対して後者は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、詳細な解説もあります。

「**プラトニズム (Platonism)**」あるいはプラトンの「**実在論 (realism)**」と呼びます。それに対して事物の後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、そのうちに発展した論理学、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關係するものです。たとえば日本語の「A は B である」^{*6}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^{*7}。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([4],p.54-65. 参照):

—— 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

^{*6} 「A は B である」という命題に「ある」が何気に含まれていることに、このような用語を作り定着させた人々の何気ない凄さを私は感じます。

^{*7} 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。

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 にも見られる主張で、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である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非常に迷惑なことでしょう。そして、この考え方には「**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。この様子は世俗的な社会改革が宗教的保守派や原理主義によって再三、妨げられ、改革の失敗後には極端な復古が生じるという 19 世紀以降の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^{*8}。

2.3.5 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」の鉄型として「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続々出てくることです。実際、「三毛猫」の「みけ」には「猫」という「類」に由来するイデアとしての「猫のイデア」があれば、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^{*9}。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

^{*8} 印刷術が西欧諸国での宗教改革に関係したのと同様に、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ように見えます。

^{*9} 日本猫は尻尾が短い！

あつたりと一意に定まるものではなさそうです^{*10}

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 でさえも、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]^{*11}.

ものの鋳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辿る訳ですが、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。

2.3.6 形相 (*εἶδος*)

さて、プラトンは「イデア」をイデア (*iδεα* と呼んだりエイドス (*εἶδος*) と呼んだりと、これらの二つの言葉を区別をしていません。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」に由来することです。

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 (*εἶδος*, *eidos*)」はプラトンのイデア (*iδεα*) のような永遠不滅のものですが、「個体から離れた存在」 (*χωριστά*) ではなく、むしろ、形相 (*εἶδος*) と、これといった特性を持たない質料 (*ὕλη*) との「結合体 (*σύνολον*)」として個体を捉え、この形相こそがその個体を個体たらしめる「形相因」、すなわち設計図のような働きをするものとしています。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存在するために、その形相が結合体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

^{*10} 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つ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もありませんが、どちらにしても一意に定まらない点は厄介です。

^{*11} 形而上学 第一卷九章

源流があることが判ります。

さて、この形相と質料を計算機上で考えるとそれなりに面白いことが判ります。まず、質料はそれ自体では何らの特性を持たないものですが、これをビットの列に、それから形相をデータ構造等の意味付けに対応付けることができるでしょう。すると計算機内部のデータは形相と質料の結合として表現されることになります。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。

2.3.7 アリストテレスの範疇 (Category)

アリストテレスの形相はその個体を個体たらしめる原因となります。その個体が何であり、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」という意味のカテゴレイスタイ (*κατηγορεσται*) 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、この範疇はものごとを語るときに、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、ものごとの述定に対応します。

アリストテレスは「カатегорー論」にて「A は B である」という命題の述語 B が取り得るものを次の 10 個の範疇に分類しています^{*12}:

アリストテレスによる範疇

1. まさにそれであるもの (本質的存在、実体): 「人間」, 「猫」
2. どれだけか (量): 「128cm」
3. どのようか (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」

なお、「本質的存在 (実体, *οὐσία*)」とは「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

^{*12} 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。

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 (*οὐσία*) 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、ここでの訳語はカテゴリー論 [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	单一性 数多性 全体性
質	实在性 否定性 制限性
関係	属性と实体性 因果性 (原因と結果) 交互性
様相	可能性 (不可能性) 現實性 (非現實性) 必然性 (偶然性)

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、何であるべきかといったことが値で表現されるのであればその値、機能であれば、それをメソッドとして表現すれば良いのです。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」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？概念に関しては類と種差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、小さな方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。

これ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ことで、そのデータがより細かく分類されることはそう珍しいことではありませんが、これ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。ここで、この細分は上位のクラスにない値やメソッドを追加することで行われます。このことは最初のクラス構築が間違っていない限り、システムの大枠を変更することなしに自然に拡張が行えることを意味します。

ただし、この継承を上手く行うためには、系統立った分析が必要になることは言うまでもありません。この分析を誤れば、継承が自然に行うことのできないシステムが出来上ることになりかねません。また、継承についても一子相伝的な継承であれば、属性やメソッドが何処から引き継がれたか探すのは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ります。それに加えて経済的な側面も考えなくてはなりません。実際に、あまりにも複雑怪異な継承関係は却って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るのです。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」が入るとどうでしょうか？素朴に考えても、猫の毛並み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でてくるのです。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必要とする可能性が生じることが理解できるでしょう。さらに、この検索の手間の問題だけではなく、この属性やメソッドの検索順位をどのように定めるかで、新しいクラスの属性やメソッドが反映されなくなる可能性も出てきます。この問題については「**C3 MRO**」といった手法で改善が図られていますが、最初のクラスの分析が非常に重要であることは言うまでもないでしょう。

2.4 集合論について

2.4.1 集合論言語について

さて、ここでは類(クラス)と集合の違いを明確にするために集合論の公理系について解説することにしましょう。ここでの集合論の公理系にはツェルメロ(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。

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 ”，最後にその他の記号として論理式のグループ化を行う括弧“(”と“)”，それと区切記号の“,”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， $A \equiv B \stackrel{\text{def}}{=} (A \supset B) \wedge (B \supset A)$ で定義します。また，集合の重要な記号に記号“ \cup ”や記号“ \cap ”等がありますが，これらの記号は後述の集合論の公理から順に定めることにします。

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

論理式の形成規則

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

この論理式の形成規則を持つ系を「**集合論言語**」と呼び \mathcal{L} と表記します. 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書き記しましょう:

集合論の公理系

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

この「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」と呼びます。この対集合 $\{x, y\}$ は集合 x と y をその成分として持つということを意味するだけで、集合 x と集合 y の順序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 $x \cup y \stackrel{\text{def}}{=} \cup\{x, y\}$ によって式 $x \cup y$ を定め、この集合 $x \cup y$ を「集合 x, y の和集合」と呼びます。

■**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 に対応し、 $\emptyset \cup \{\emptyset\}$ が自然数の 1 等々と対応するというもので、無限公理で認められた集合の生成規則から繰り返し生成されるというもので、つまり、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になるのです。

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。これらの記号から、ここで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^{*13}。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もあります。

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

分出公理 (Axiom of displacement)

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

*13 自然数の後者関係についてはフレーゲの「概念記法」[10] ではじめて厳密に述べられているようです。

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 $\psi(x) \wedge x = y$ で置換することで分出公理 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、予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」[9] 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 $\text{rng}(f)$ が集合になることは分出公理から導き出せません。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*14。また関係 \in に対する無限降下列が存在しないことは公理 A8':

無限降下列の非存在性

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

とすることができます。この「**無限降下列の非存在性公理 A8'**」と「**正則性公理 A8**」の間には $A8 \supset A8'$ が成立しますが、その逆の $A8' \supset A8$ が成立するためには次の「**選択公理**」が必要になります [9]。

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

バナッハ-タルスキの逆理

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

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

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

*14 底なし加減は落語の「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」とも言えますが。

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^{*15}。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 A9
ZF	:	A1	A2	A3	A4	A5	A6	A7	A8
ZFC	:	A1	A2	A3	A4	A5	A6	A7	A8 A9

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

2.4.3 順序数

ZFC 公理系で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 $x < \alpha$, $y < \alpha$

^{*15} 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。

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となります。

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, このときに $\alpha \in \text{Lim}$ と表記します。極限数の例としては 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。

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は「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が、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 $A \vdash A$ 、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。このような文のことを「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

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、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 (όργανον))**」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] *¹⁶。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[19]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**」に由来し、「**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[19](以後、CMW と略記) の定義に沿って解説しましょう。この CMW[19] 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と考えており、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。

*¹⁶ その注釈書としてポルピュリオス ($\Pi\sigma\rho\varphi\psi\varsigma\iota\omega\varsigma$, Porphyry of Tyre) のイサケゴ [20] が非常に有名で、最も影響力がありました。

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.

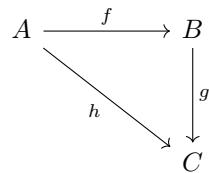
ここで幾つかの記号を導入します. まず, メタグラフ \mathcal{C} の対象のあつまり, すなわち類(クラス)を $\text{Obj}\mathcal{C}$, 同様にメタグラフ \mathcal{C} の矢の類を $\text{Arr}\mathcal{C}$ と表記します. そして, メタグラフ \mathcal{C} の矢の始域になり得る対象から構成される類を \mathcal{C}_0 , 終域になり得る対象で構成される類を \mathcal{C}_1 と表記します. 同様に対象 A を始域, 対象 B を終域とするメタグラフ \mathcal{C} の矢から構成される類を $\text{Hom}_{\mathcal{C}}(A, B), \mathcal{C}(A, B)$, 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「繋辞 (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繋辞こそが命題を構成するものと考えられていたとのことです [3]. 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 $\{(x, y) : x \in A \wedge y \in B \wedge f x = y\}$ として矢を考えることができます。このときに $x \in A$ に対応する対象 B の元を $f(x)$ あるいは $f x$ と表記し、 $y = f x$ のときに $x \mapsto y$ と表記することで $x \in A$ と $y \in B$ が矢 f を仲立とする関係にあることを示します。ここで空集合 \emptyset を始域とするメタグラフの矢 f を考えると、この矢は空集合 \emptyset でなければならないことが判ります。この実例に、後述の Python のオブジェクトの型で None 型があります。実際、この None 型が空集合 \emptyset を始域とする矢と同様の働きを持っています。

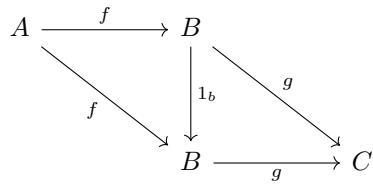
次に「**矢の合成**」と呼ばれる矢の操作について解説しましょう。この矢の合成は写像の合成を抽象化したものです。この合成を考える前に「**合成可能対**」というものを考えます。これは矢の順序対 $\langle f, g \rangle$ であり、この順序対を構成できる矢 f, g は $\text{dom } g = \text{cod } f$ を充す矢でなければなりません。この条件を充す順序対から構成される類を $\text{Arr}\mathcal{C} \times_{\text{Obj}\mathcal{C}} \text{Arr}\mathcal{C}$ と表記し、「**合成可能類**」と呼びます。ここで $\text{Arr}\mathcal{C} \times_{\text{Obj}\mathcal{C}} \text{Arr}\mathcal{C}$ に属する順序対 $\langle g, f \rangle$ で $g \circ f$ 、その始域と終域をそれぞれ $\text{dom } f, \text{cod } g$ になる矢に対応させる操作とします。もちろん、このような操作が可能かどうかはメタグラフでは保証されません。この操作が可能な場合に得られる矢のことを矢 f, g の「**合成**」と呼びます。ここで二つの矢 $A \xrightarrow{f} B$ と $B \xrightarrow{g} C$ の合成 $A \xrightarrow{g \circ f} C$ が矢 $A \xrightarrow{h} C$ と一致するときに、これらの矢の関係を次の図式として表現することができます：



この図式では対象 A から矢に沿って対象 C に向う二つの経路があり、この経路に沿うことと矢の合成が一意に対応します。この図式には矢 f と g を経由する経路から得られる矢 $g \cdot f$ と矢 h を経由する経路の二つが存在し、これら二つの経路が一致することを意味します。この図式のように図式中のある対象を始域と終域とする複数の経路が存在し、それらの経路の何れを通っても矢の合成が一致する図式を「可換図式」と呼びます。

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

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



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対一に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、写像の单射、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 $f : A \rightarrow B$ と表記します.
- **epi** : 任意の矢 $h, g : B \rightarrow C$ に対して $h \circ f = g \circ f$ を充せば $h = g$ となるとき. このときに $f : A \twoheadrightarrow B$ と表記します.
- **iso** : $g \circ f = 1_A$ かつ $f \circ g = 1_B$ を充す矢 $g : b \rightarrow a$ が存在するとき. このとき $A \equiv B$ 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単射、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としてみましょう。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での表記に従って $\text{Obj}\mathcal{C}$ 、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。但し、**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。

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 ${}^{\text{op}}$ 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。この双対によって圈 \mathcal{C} から新しい圈が構築され、この圈のことを圈 \mathcal{C} の「**双対圈**」と呼び、 \mathcal{C}^{op} と表記します。

2.5.6 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^{*17}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 ${}^{\text{op}}$ 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 ${}^{\text{op}}$ では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：

^{*17} 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。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、それが単射、全射、同型となる場合を考えられます。ここで函手が「忠実 (faithfull)」であるとは $\text{Hom}_{\mathcal{C}}(A, B) \xrightarrow{F} \text{Hom}_{\mathcal{D}}(FA, FB)$ が通常の写像として単射になるときで、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 $C' \xrightarrow{f} C$ の始域と終域となる対象 $C, C' \in \mathcal{C}_0$ 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。

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

同様に終対象もその双対圏では始対象になるので 1 と 0 が互いに双対関係にあることが判ります。

2.5.8 圈の演算

圏 \mathcal{C} の対象について積や幂を定めることができます。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。圏 \mathcal{C} の任意の二つの対象に対して積が存在するときに圏 \mathcal{C} のことを「積を有する圏」と呼ぶ。

圏の積の定義を可換図式として表現することができます：

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

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と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、圏 \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。ところで圏 \mathcal{C} の対象

が順序数^{*18}で、矢が \leq のときの $A \times B$ は A と B のどちらかより後者にある対象で与えられるので、対象の積は対象の成分の順序対のようなものになるとは限りません。

この対象の積については次の性質を充します：

対象の積の性質

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

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

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：

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

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：

^{*18} 順序数も集合です

矢の積

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

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

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

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

対象の幕

積を有する圏 \mathcal{C} をの対象 A, B と矢 $C \times A \xrightarrow{g} B$ に対し、以下の図式を可換にする圏 \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。

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

ここで二つの矢 f, g の等化 e は必ずモノになります。実際、 h, k を対象 D から 対象 C の矢で、それぞれが $e \circ h = e \circ k$ を充すときに二つの図式:

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

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

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

余等化 (coequalizer)

二つの矢 $A \xrightarrow[g]{f} B$ に対して矢 $B \xrightarrow{e} C$ が $f \circ e = g \circ e$ を充し、さらに以下に示す

可換図式にて $f \circ k = g \circ k$ であるときに $k = e \circ h$ を充す矢 $B \xrightarrow{h} D$ が一つだけ存在するときに矢 e を矢 f と矢 g の余等化と呼ぶ。

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

余等化については等化の双対であるためにエピとなることが判ります。

2.5.9 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としましょう：

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

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

最後に積、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」の双対として「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：

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

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 $A \amalg B$ となり、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)」を次のように定めます：

—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 $\nu_j = \delta_{ji} \circ \nu_i$ を充すときを言う。

—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 $\mu_i = \mu_j \circ \delta_{ji}$ を充すときを言う。

—— 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。

圏の例

ここでは具体的な圏の例を挙げることにしましょう。最初にちょっと特殊な圏を考えます。この圏は有限個の元を持つ集合 A で矢として恒等矢しか持たないものとします。このように恒等矢しか持たない圏は「離散圏」と呼ばれます。この離散圏では、各対象 $A \in \mathcal{C}$ と対応する恒等矢 $A \xrightarrow{1_A} A$ の他には矢がない圏なので、この矢と繋ぎの「... は... である」を対応させてみましょう。すると $A \xrightarrow{1_A} A$ の意味を「 A は A である」と解釈することができます。そして離散圏は対象 $A \in \mathcal{C}$ を始域とする矢は恒等矢 1_A しか存在しないので、任意の対象 $A \in \mathcal{C}$ に対して「 A は A である」としか言えない圏であることが判ります。これはキュニコス派のアンティステネス (*Antiorthéneç*) の主張する「**一つの主語は一つの述語あるのみ**」^{*19}と普遍を認めない立場に対応します。

その他の重要な圏の例を挙げておきましょう:

- **Set**: 対象が集合で、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

^{*19} 形而上学 [2] 5 卷 29 章, 1024b34

- **Cat**: 対象が圏で、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} : 圏 \mathcal{C} の双対圏

ここで述べている集合、圏と位相空間は、より正確にはそれぞれ「**小集合 (small set)**」、「**小圏 (small category)**」、「**小位相空間 (small topological space)**」と呼ばれます。これは「**グロタンディークの宇宙**」との関係を表現するものです。また、グロタンディークの宇宙は圏の対象全てと後述の対象の演算結果を含む大きな集合で U と表記します。このことから対象はグロタンディーク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、圏については「**局所的に小さい**」と「**小さい**」の二種類があります。圏 \mathcal{C} が「**局所的に小さい**」とは任意の対象 $A, B \in \mathcal{C}$ に対してその矢の集合 $\text{Hom}(A, B)$ が小集合になる場合で、同様に「**小さい**」とは対象全体の集合 \mathcal{C}_0 と矢全体の集合 \mathcal{C}_1 の双方が小集合になる場合です。

グロタンディークの宇宙について

さきほどの圏の例で、対象が「**小集合**」等と小が付くものを示しました。ここでの大小はグロタンディークの宇宙との関係で決まると言明しました。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。圏の場合、対象や矢の類は集合を構成し、それらの集合に対してグロタンディーク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**」、対象が圏であれば U の元を「**小圏**」と宇宙 U の元のことを、その元の型の頭に「**小 (small)**」を付けます。逆に宇宙 U は頭に「**大 (large)**」を付けます。たとえば、対象が集合であれば「**大集合**」、圏であれば「**大圏**」といったあんばいです。

2.6 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、ここで圏 \mathcal{C} を集合の圏 **Set** で説明しておきましょう。このとき、対象 A, B の関係は $A \subset B$ として考えることができます。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*20。

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

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

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

第3章

Pythonについて

3.1 この章の目的

この章では Python について解説を行います。そのために最初に 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}を基にして 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(所謂 **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 のようにも使える言語で、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為、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。その為に 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に対して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 自体は処理言語として他と比較して処理速度が速い言語ではありません。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ると言えます。

^{*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。まず、制御で用いられる文は、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 では必要最低限なものに抑えられています。

3.2.3 字下げ

Python の構文では C や Java といった言語と比較して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）」を用いています。このことによって Python のプログラムは二次元的なプログラムの構造を持つことになり、その結果、Python のプログラムは視覚的にもプログラムの構造が示されていることも意味します。ここで、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。このことを具体的に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

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

のように記述しても

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

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に、プログラマの意図は別にして、これらの記述にプログラムの意味としては違いはありません。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は、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

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

```
y = 0
```

のように if 文内部の文 (ここでは ‘y = 1’ と ‘y = 0’) と if 文を構成する命令 ‘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らず, 次の線的な記述:

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

と字下げの位置がチグハクで構造を十分に表現できていない記述:

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

といった記述は構文エラーになります. とはいえ, ‘if x == 0: y = 1’ のように一行で記述することが認められている構文も一部にあります, 構造が単純なものに限定されます.

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. このような Python の規約が PEP で事細かく定められている点も 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, 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すが, 最初に LISP (ここでは SBCL) の例を示しておきます:

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

ADD2
* (documentation #'add2 'function)

"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 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 documentation 函数を用いて表示させています. これと同様のことを Python では次のように行うことができます:

```
>>> def add2(x):  
...     """  
...     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、この例では add2 フィルを 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。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索するために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。ここでは函数を add2 として、その内容を下記のものとしましょう：

```

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

```

このファイルを add2.i として保存しておきます。それから起動した Yorick からこのファイルを include フィルで読み込んでおけば、あとは必要に応じて help フィルを使うと 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 が出力するプロンプトです：

```

> include,"add2.i"
> help,add2
/* DOCUMENT add2
*
* 2を足すよー
*/
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 による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されており、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。ここで Python から文書文字列を閲覧する場合は函数 help() を用いますが、iPython を Python のシェルとして利用している場合には ‘?’ も利用可能です。Python から函数 help() を用いて函数 open() を調べた結果を示しておきます：

```
1 Help on built-in function open in module __builtin__:  
2  
3     open(...)  
4         open(name[, mode[, buffering]]) -> file object  
5  
6             Open a file using the file() type, returns a file object. This is the  
7             preferred way to open a file. See file.__doc__ for further information.  
8 lines 1-7/7 (END)
```

さらに Sage では iPython を Python のシェルとして用いているために函数 help() の他に記号 ‘?’ が使えます。この記号 ‘?’ 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]:
```

ここで help 函数は文書文字列のみを表示していますが、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のクラスはオブジェクトの定義に用いられるもので、これから取り扱うべきオブジェクトを表現する**概念の内包**や**概念の外延**に相当し、

オブジェクトそのものは「**概念の外延を構成する個体**」に相当すると言えるでしょう。また、「**個体**」はクラスが表現するオブジェクトの実体化でもあるために「**インスタンス**」とも呼びます。ところでメタクラスを考えたときにクラスはあるメタクラスの実体化と見做すこともできます。このクラスそのものをオブジェクトと考えた場合にクラスが表現するオブジェクトと区別する必要があります。そのため、より正確には、クラスが表現するオブジェクトが実体化したインスタンスを「**インスタンス オブジェクト**」、メタクラスの実体化としてのクラスを「**クラス オブジェクト**」と呼びます。Pythonではこれらのオブジェクトは区分されています。

さて、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*2:

```
class TEST:
    pass
```

このクラスは名義的な実体を生成するだけで、'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へのオブジェクトの束縛によって行ないます。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'にオブジェクトを束縛するかたちでオブジェクトの生成を行っています。それから'a.pet'、'a.weigth'や'a.age'で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「**データ属性**」と呼ばれて、C++のデータメンバ、Smalltalkのインスタンス変数に相当するものであり、また概念の属性に相当します。この

*2 Python 2.xでは古典的クラス、Python 3.xではクラスタイプになるという違いがありますが、ここでの解説では型の細かな違いには踏み込まないために問題ありません。

データ属性をインスタンスに自由に追加することができる点が Python の Java 等のオブジェクト指向プログラミング言語との大きな特徴です。

次にもう少しまともで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対応するものです:

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

このクラスの定義では二つのデータ属性 x と y があり、これらには初期値として ‘1’ を設定しています。実際に使ってみましょう:

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

この例ではクラス TEST を定義し、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データ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 のデータ属性の参照は ‘⟨インスタンス名⟩.⟨データ属性名⟩’、インスタンスのデータ属性値の変更は ‘⟨インスタンス名⟩.⟨データ属性名⟩ = ⟨属性値⟩’ で行えます。このように C の構造体と同様の使い方になっていることが判るでしょう。なお、Python のクラスの属性には、データ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す。オブジェクトに結び付けられているために函数とは異なり、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ません。このようにメソッドはオブジェクトに付属する函数であり、そのオブジェクトを特徴付ける機能と考えると良いでしょう。

では先程の TEST クラスにデータ属性同士の和を計算するメソッドを定義してみましょう:

```
class TEST:

    x = 1
    y = 1

    def wa(self):
        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。ここではメソッドの wa を定義していますが、このメソッドではオブジェクトのデータ属性 x と y の和を返す処理を行います。そこでデータ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’ のデータ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 のデータ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’ のみのメソッドのとき、その実行に引数は不要で、‘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ね。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す。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可能となって、非常に効率的に行えるのです。ここでは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。ここで整数とその算術演算は既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。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。

す。そこで自然数の対のクラスを定義することにしましょう。それと算術的演算を取り入れる前に有理数の表示を行う機能を入れておきます。具体的には、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のはいかがでしょうか？Pythonには対象の初期化を行うメソッド‘__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メソッド‘__repr__’があります。Pythonではこれらのメソッドがクラス内で定義されていれば、これらのメソッドを用いて所定の処理を行うことができるのです。有理数のクラスPairOfIntsを以下で定義することにしましょう：

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

このクラスでは上述の二つ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行うためのメソッドで、いわゆる「カプセル化」に対応するものです。こ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です。ここでの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に表示を行う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)’によってインスタンスを生成したときに‘a’と入力することで‘1/2’と表示されることを指示するものです。また、‘a = PairOfInts(1,2)’でインスタンスを生成したときにnumerやdenomの値の取り出しは構造体と同様にそれぞれ‘a.numer’と‘a.denom’で行えます。

では、Pythonで試してみましょう。なお、そのまま上の内容を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読み込むことします。このとき単純にimport PairOfIntsとしたときには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 `__repr__` メソッドが含まれていないときにどうなるかも示しておきましょう。そのため直接、Python に `__repr__` メソッドを持たないクラスの TEST を定義しています：

```
>>> 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 `__repr__` メソッドを除いた `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 `__repr__` メソッド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。この `__repr__` メソッドは組込函数の函数 `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。

ここでクラスの属性名で文字 ‘_’ を先頭を持つ属性について簡単に解説しておきましょう。まず、二つの文字 ‘_’ で構成された文字列 “__” を名前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アクセス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参照することができてしまいます。この先頭に文字列 ‘__’ 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わるものでしたね。そして文字 ‘_’ を

一つだけ先頭に持つメソッドや変数となるとシステムとして隠蔽は行われませんが、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対象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：

```
>>> dir()
[ '__builtins__', '__doc__', '__name__', '__package__']
>>> class test:
...     x = 1
...     _y = 1
...     __z = 1
...
>>> dir()
[ '__builtins__', '__doc__', '__name__', '__package__', 'test']
>>> a1 = test()
>>> dir()
[ '__builtins__', '__doc__', '__name__', '__package__', 'a1', 'test']
>>> a1.x
1
>>> a1._y
1
>>> a1.__z
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: test instance has no attribute '__z'
>>> dir(test)
[ '__doc__', '__module__', '_test__z', '_y', 'x']
>>> a1._test__z
1
>>>
```

この例では組込函数 dir() を使って Python で展開されているクラスとインスタンスを表示させています。このクラス test のデータ属性は 'x', '_y' と '__z' の 3 個を与えています。そしてインスタンス a1 を生成し、各データ属性値の確認を行っていますが、「a.__z」にはアクセスできません。このように一見すると隠蔽に成功しているようですが、ここで函数 dir でクラス test の中を見ると、データ属性 '__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データ属性は文字列 '__(クラス名)' が先頭に置かれた名前に置換されているのです。だから、「a1._test__z」でデータ属性 '__z' の値が調べられたのです。また、他のデータ属性と同様に書換えることも可能なのです。このように Python での「カプセル化」による属性の隠蔽の方法は不徹底なものです。

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何も持っていないません。また、メソッドにも有理数の表記で自然数の対を表示する機能しかありません。有理数の処理で期待される一つの有理数に対しては約分や逆元の計算といった処理、二つの有理数に対して大きい、小さい、等しいといった大小関係、和や積といった算術演算も PairOfInts クラスにはありません。

そこで自然数の単純な対を有理数とみなすためには、まず、二つの自然数の対が与えられたときに、何が等しいかどうかを判別しなければなりません。たとえば、 $1/2 = 2/4$ なので、自然数の対 ‘(1, 2)’ と ‘(2, 4)’ は等しくなければなりません。また大小関係も入れておきたいところです。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、 $ad - bc = 0$ を満すとき」と言い換えることができます。そして、‘(a, b)’ と ‘(-a, -b)’ は等しい自然数なので、分母に相当する denom が常に正となるように置き換え、‘(a*c, b*c)’ を ‘(a, b)’ で置き換えるべきです。つまり、RationalNumber クラスは単なる自然数の対のクラス PairOfInts に正規化を加えたクラスであるべきです。これらのことを見て PairOfInts クラスを基に RationalNumber クラスを構築してみましょう。：

```
from PairOfInts import PairOfInts
class RationalNumber(PairOfInts):
    def __gcd__(self):
        __a = self.numer
        __b = self.denom
        if __a == 0:
            if __b != 0:
                __b = 1
        elif __b == 0:
            __a = 1
        elif __a > __b:
            __d = __a / __b
            __r = __a - __d * __b
        else:
            __c = self.conv()
            return __c.__gdc__()
    def __cmp__(self, other):
        """
        有理数の大小関係を定義するメソッド
        """
        return cmp(self.numer * other.denom,
                  self.denom * other.numer)
    def conv(self):
        """
```

```

逆元を返すメソッド
"""
tmp = self.numer
if tmp == 0:
    self.numer = 1
    self.denom = 0
else:
    self.numer = self.denom
    self.denom = tmp

def rexpr(self):
    """
    有理数の正規化を行うメソッド
    """
    if self.denom < 0:
        self.denom = - self.denom
        self.numer = - self.numer
    if self.numer == 0:
        self.denom = 1
    elif self.denom == 0:
        self.numer = 1
    else:
        tmp = gcd(self.numer, self.denom)
        self.numer = self.numer / tmp
        self.denom = self.denom / tmp

```

ここでは既存のクラスを利用して新たなクラスの構築を行っています。この目的のために Python では `class 新しいクラスの名前 (既存のクラスの名前)` によって新しいクラスの定義が行えます。この方法によるクラスの構築のことを **継承** と呼びます。ここでは RationalNumber クラスは既存の PairOfInts クラスを基にしており、このことはクラスを定義する class 文にて ‘RationalNumber(PairOfInts)’ とすること行えます。

さて、こうして RationalNumber クラスが構築できましたが、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、この関係を集合の包含関係と見做したときに基底クラスを「**スーパークラス, super-class**」、派生クラスを「**サブクラス, sub-class**」とも呼びます。

```

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

```

```

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

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

```

この RationalNumber クラスにも “__” で開始して “__” で終わるメソッドが三個あります。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 `__mul__` が積演算子 “*” に対応するメソッド、そして最後のメソッド `__cmp__` が比較の演算子 “>”, “<” と “==” に対応するメソッドです。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和、積、比較の演算といった処理が可能になるのです。

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、

————— Backus-Naur 記法の構文規則 —————

非終端記号 ::= 定義₁ | 定義₂ | ... | 定義_n

で与えられます。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 “ \vee ” に相当する演算子であり、もしも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**とも呼ばれます。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”的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**」が出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。

整数は正の自然数と負の自然数の和集合として得られるので、上記の自然数の BNF を前提にすれば次で定義できます：

BNFによる整数の定義

整数 ::= 自然数 | “-” 自然数

これらの定義で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。

「Python 言語マニュアル」では 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、ここでの定義では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、‘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Pythonプログラムの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)」はプログラム内で意味を持つコードの最小単位で、自然言語における「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

Pythonのプログラムは字句解析によってトークンの列に変換されますが、この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。そして論理行はさらに物理行に分解されます。

■論理行(logical line): 先頭に「**字下げ/インデント(indentation)**」と呼ばれる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”^{*3}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^{*4}。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 Python の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。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 Python の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 のみとして 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うからです。また、注釈自身も継続文字を使って分割することができません。

^{*3} 日本語 Windows で用いられている文字コードの SHIFT_JIS で“\”は勝手に“¥”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、この本では“\”を“¥”で置換することをしません。日本語 Windows 環境では適宜“\”を“¥”で置換えて解釈し直して下さい。

^{*4} 記号“#”は後述のリテラルには含まれていません。

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、これらの記号で括られたその内部では註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 では名前に用いられるもので、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**”や所謂、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に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**」が構成されることです。なお、キーワードは Python の文等で用いられます BUT、名前として使うことはできません。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に、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, 記号論理学でのリテラルは命題記号 (原子論理式), あるいは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, プログラミングで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を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、さらにリテラル自体にも接頭辞がなければ ASCII 文字となります。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。ただし、接頭辞と後続する文字列との間に空白文字を入れてはなりません。

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。そのために、この文書文字列の中では改行や单引用符や二重引用符を入れることも可能で、このことを利用して例題等を含む長い文字列を記述することができます。このときに注意することは長文字列を構成する際に用いた引用符と同じ引用符を三回連続して配置すると、その箇所で長文字列が終了すると言うことです。

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、单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のです。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

値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進数表示 | 16進数表示 | 2進数表示
10進表示 ::= 零以外の数字 数字* | "0"
8進数表示 ::= "0" ("o"|"O") 8進数 + | "0" | 8進数 +
16進数表示 ::= "0" ("x" | "X") 16進数 +
2進数表示 ::= "0" ("b" | "B") 2進数 +
数字 ::= "0" | 零以外の数字
零以外の数字 ::= "1" ... "9"
8進数 ::= "0" ... "7"
16進数 ::= 数字 | "a" ... "f" | "A" ... "F"
2進数 ::= "0" | "1"

```

この EBNF で示すように Python の整数リテラルには 10 進数の他に 2 進数, 8 進数と 16 進数があります。なお、長整数のリテラルでは末尾に小文字 ‘l’ を置くことも許容していますが、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の結果得られるからです。この点は次に言及す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
-1.000000000000000
```

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 $x^2 + 1$ の零点として得られ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となる数のことを「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は異なり、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、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

以下のトーカンは 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。

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 では幂乗の演算子として演算子 “`^`” が使えます. ちなみに Python で記述された簡易的数式処理の SymPy でも Python と同様に幂乗の演算子が “`**`” だけで, 演算子 “`^`” は後述するようにビット単位の演算子で, 幂乗とは全く異った作用をします. Sage で幂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幂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.

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

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, 他のトークンの一部に用いられて特殊な意味を持ちます. 最後に “`$`” と “`?`” は Python では用いられず, 文字リテラルと注釈以外に現われたときは無条件にエラーになります. ただし, Python のシェルである iPython 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はなりません. 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. ここで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が行えます. 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)**」など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によって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、この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って、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もあり、これらの場合は不要となつ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。

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’ を割り当てていますが、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にて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を取り出すための MATLAB 系の言語でお馴染み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() の読みを行い、それから名前 a で参照される NumPy の一次元配列を生成してメソッド reshape() で 2×8 の配列で置換えます。それから二つの拡張スライス操作 ‘a[0, ...]’ と ‘a[1, ...]’ を行っていますが、これらの処理は行列の一行目と二行目の成分に相当する配列の取り出しを行っています。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。

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)^{*5}：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。
- 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 $z.real$ 、虚部は $z.imag$ で取り出せますが、複素数型の性質上、これらは実数型、すなわち、倍精度の浮動小数点数です。そのために近似の数であることに注意が必要です。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：

^{*5} 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。

順序関係

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

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

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

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⁶。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。

⁶ 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。

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 Python に組込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^{*7}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。

^{*7} 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。

- 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()`^{*8} 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「**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のメソッド `__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。
- 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__()` があるときに限って呼出可能型になります。

モジュール

`import` 文で Python への Python プログラムを記述したファイル読込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は初期化で用いられる後述の「**コード オブジェクト**」は含まれていません。

クラス

Python 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。古典的クラス型はクラスタイプ型と違い、メタクラスが扱えない、`super()` による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ち、Python 3.x で削除されています。また、双方が提供されている Python 2.x では基底クラス ‘`object`’ を何らかの形で継承しないと古典的クラス型、継承することでクラス

^{*8} 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。

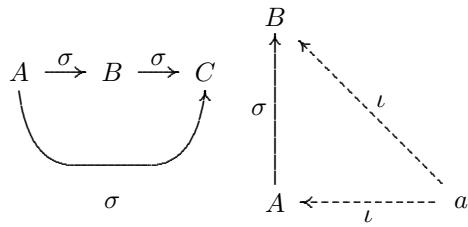
タイプ型として定義されます:

```
>>> 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函数 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函数 type() で 'instance' と表現されるインスタンス オブ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。上の例で 'type(b)' の結果で '__main__.TEST2' とクラス名を含む文字列が返却されていることに注目して下さい。なお、Python 3.x では古典的クラス型は廃止されるために引数を指定しないクラスの定義もクラスタイプ型になります。

ここであるクラスを継承する関係にあるクラスのことを「**派生クラス**」と呼び、逆に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。この基底クラスは派生クラスよりもより普遍的なクラスで、上位のクラスであるとも言えます。この継承関係を親子関係にたとえると「**親**」に相当するクラスになり、それに対して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 $C_1 \xrightarrow{\alpha} C_0$ 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、 $\alpha \dashv^l C$ 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 $A \xrightarrow{\sigma} A$ は現実的には生じませんし、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。ここで継承の関係は類と種差による定義と同様です。つまり、類と種については類に種が含まれされるものの逆は同一でない限りあり得ず、継承関係を行ってゆくに連れて、クラスがインスタンスとして保有するオブジェクトは絞り込まれていくということです。

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 $\mathcal{L}(C)$ と記述します。また、この検索順序を求める処理を「**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： $C_1 \dots C_n$ と表記しすることにします。ここで、リストの先頭にクラス C_0 を追加することは語の先頭に C_0 を追加することに対応し、この追加する操作を $C_0 + C_1 \dots C_n$ と表記します。次に語 $C_0 C_1 \dots C_n$ の先頭 C_0 を取り出す操作を`head`、先頭以外の残りの $C_1 \dots C_n$ を取り出す操作を`tail`と表記し、語 L に対して $\overline{L} \stackrel{\text{def}}{=} \text{head}(L)$, $\underline{L} \stackrel{\text{def}}{=} \text{tail}(L)$ と略記します。それから語 $B_1 \dots B_m$ と語 $C_1 \dots C_n$ が与えられたとき、語 $B_1 \dots B_m$ に含まれる各 $B_i (1 \leq i \leq m)$ を語 $C_1 \dots C_n$ から取り除いた語を $C_1 \dots C_n \setminus B_1 \dots B_m$ と表記することにします。たとえば、 $abcd \setminus bd$

は ac になります。また、検索順序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 ‘ $\cdots W_{(i-1)}W_iW_{(i+1)}\cdots W_{(j-1)}W_iW_{(j+1)}\cdots$ ’ は語 ‘ $\cdots W_{(i-1)}W_iW_{(i+1)}\cdots W_{(j-1)}W_{(j+1)}\cdots$ ’ 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致する語を除いたものとの関係を ‘ $\cdots W_{(i-1)}W_iW_{(i+1)}\cdots W_{(j-1)}W_iW_{(j+1)}\cdots \searrow \cdots W_{(i-1)}W_iW_{(i+1)}\cdots W_{(j-1)}W_{(j+1)}\cdots$ ’ と表記します。そして関係 “ \searrow ” によって語 L はより短い語へと置換えることが可能です。そして、この短縮化には下限があるため必ずその極限が存在します。この語 L の関係 “ \searrow ” の極限になる語をここでは \underline{L} と表記します。そして作用素 \mathcal{L} は次の性質を持ちます：

———— 作用素 \mathcal{L} の性質 ————

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

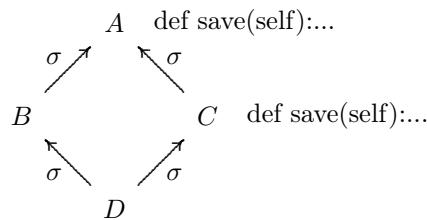
最初の 1. は継承関係を持たないクラス C のときに MRO は C のみであるということを示します。つぎの 2. は $C_0 \xrightarrow{\sigma} C_1$ のとき、つまり、クラス C_0 が C_1 の派生クラスのときに最初にクラス C_0 を検索し、それからクラス C_1 の検索順序に従うということを意味し、つぎの 3. は関係 “ \searrow ” で作用素 $*$ の値は不变であることを示し、最後の 4. が多重継承があるときの処理で、ここで作用素 \mathcal{M} を導入します。この作用素 \mathcal{M} の働きは、継承関係を遡る MRO を基本に、多重継承のあるクラスにて継承の親達を示すタプルをそのまま用いて基底クラスに検索順序を入れるというものです。つまり、クラスの属性で基底クラスを示すタプルの左側から順番に先祖を辿る方法で行われることから「**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. $\mathcal{M}(\cdots, L, \cdots) = \mathcal{M}(\cdots, \underline{L}, \cdots)$
- c. $\mathcal{M}(L_1, L_2, \cdots, L_n) = \underline{\underline{L}_1} + \mathcal{M}(L_2 \setminus L_1, \cdots, L_n \setminus L_1)$

最初の a. は検索順序を示す語 W 一つが引数であれば、関係 “ \searrow ” の極限 \underline{W} を返すという性質です。そして次の b. は関係 “ \searrow ” で作用素 \mathcal{M} の値は不变であることを示しています。そして最後の c. は引数の最も左側にある経路を外に出し、その語に含まれるクラス名を他の引数から除去してゆくという処理方法を示しています。もし、引数の語に共通するクラス名がなければ \mathcal{M} は語に対する和になるだけです。

Python の古典的クラス型でメソッド検索で用いられる順序は、MRO です。しかし、この手法は多重継承があるときに有効に動作しない問題があります。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することにしましょう：



この図式は、クラス D はクラス B と C の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 で用いられることがありません。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 でメソッド `save` が発見された時点で検索が終了し、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り、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問題が生じます。そのためクラスタイプでは「**C3 MRO(Method Resolution Order)**」^{*9} と呼ばれる手法で検索が行われます。ここで「C3」は多重継承における検出順序を提供するアルゴリズムで、最初に Dylan 言語に導入された手法です。この手法は先程の 3. の計算手順の作用素 \mathcal{M} を作用素 \mathcal{M}_{c3} で置換えて、次のようにまとめることができます：

^{*9} <https://www.python.org/download/releases/2.3/mro/> や PEP-0253 を参照

C3 での作用素 \mathcal{L} の性質

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

ここで作用素 \mathcal{M}_{c3} は次の性質を持ちますが、最初の a., b. は作用素 \mathcal{M} の場合と同様です。また c. の計算手順は \mathcal{M} よりも階層を意識した検出方法に代ります：

作用素 \mathcal{M}_{c3} の性質

- a. $\mathcal{M}_{c3}(W) = \underline{W}$
- b. $L_0 \searrow L_1$ のとき $\mathcal{M}_{c3}(\dots, L_0, \dots) = \mathcal{M}_{c3}(\dots, L_1, \dots)$
- c. \mathcal{M}_{c3} の計算は後述の方法で計算される。

この \mathcal{M}_{c3} の計算手順を $\mathcal{M}(L_1, L_2, \dots, L_n)$ が与えられたときにどのように行うかを纏めておきましょう：

 \mathcal{M}_{c3} の計算手順

1. $i = 1$ とする。
2. $h_i = \overline{L_i}$ とする。
3. $j \neq i$ に対して $h_i \notin \underline{L_j}$ のとき、 $k \in (1, \dots, n)$ 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 $\underline{L_k}$ で置換え、 $h_i + \mathcal{M}_{c3}(L_1, \dots, L_n)$ 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。

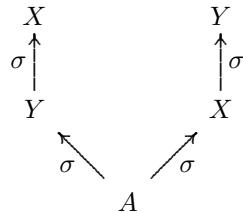
この作用素 \mathcal{M}_{c3} は引数の語に共通するものが何もなければ最初の作用素 \mathcal{L} を拡張したものと同様に語の和として作用します。しかし、共通する語が現われたときにその処理がクラスの階層を合せるような働きになります。このことを先程の菱形状の継承関係で C3 MRO を計算することで確認してみましょう。

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、より新しいクラス *C* のメソッド *save* が用いられ、古典的クラス型の MRO よりも妥当な結果が得られることになります。そして、この C3 MRO の長所は間違った継承関係を検出することが可能です。

たとえば次の継承関係を想定しましょう：



この継承関係は $X \xrightarrow{\sigma} Y$ かつ $Y \xrightarrow{\sigma} X$ と、クラスの定義として循環的な定義になっているという問題があります。ここで MRO では *AYX* が得られますが、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ないため、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**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。属性参照では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にそのインスタンスが含まれ

るクラス属性に検索の領域を広げます。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)** Python の 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ても良いようになっています。
- 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 オブジェクトで現われます。
- 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際に、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。この表記は 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になります。
- 静的メソッド オブジェクト: 組込の構成子 `staticmethod()` で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()` で生成されるオブジェクトです。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、定義したクラスでは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」と呼ばれる一連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といったこと、あるいは大小関係を判断するメソッドで、こ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ってその否定の“!=”が自動的に定義される訳ではないので、“!=”を使う可能性があるならば等号の再定義と併せてこちらの定義も行う必要があります。

3.5.4 特殊メソッドによる基本的な属性変更について

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え、変数を適宜用いることになります。

■`obj.__new__(cls [args...])` クラス cls の新しいインスタンス生成のために呼び出されるメソッドです。メソッド`__new__()`は静的メソッドで、インスタンスを生成するよう求められているクラスの名前を第一引数に取り、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__()`に引渡す働きをするので、後述のメソッド`__init__()`と併せて C++ の構成子に似た働きを行います。

クラス cls で表現されるオブジェクトのインスタンスを生成するときには‘super(現在のクラス, cls).__new__(cls[, ...])’に適切な引数を指定することでクラス cls の基底クラスのメソッド`__new__()`を呼出し、新たに生成されたインスタンスに必要な変更を加えた上で返却を行います。

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、`__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おいて補正を行うためです。

■`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適正に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**」という制約があるので、この `__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)**」に似た動作を行なうメソッドで、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るというもので、C++ 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__()` を持っているとき、その派生クラスのメソッド `__del__()` では明示的に基底クラスのメソッド `__del__()` を呼出して、基底クラスに関連する部位が適正に消去されるようにしなければなりません。

■`obj.__repr__(self)` 組込関数 `repr()` や文字列への変換時に呼出され、オブジェクトをフロントエンド側で表現する「**公式の文字列**」の生成を行います。この性質を用いれば、式や処理結果をより判別し易い文字列として出力することができます。この例はこの章の最初の有理数の定義にもあります。

■`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。

■`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、等しくないことを意味する演算子です。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)による比較が行われます。

■`obj.__hash__(self)` 組込の函数`hash()`, `set`, `frozensey`, `dict`のようなハッシュ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()`^{*10} 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真となります。

■`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

*10 何故か`bool`でない!

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。

3.5.5 特殊メソッドを用いた属性値の変更

■`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は呼び出されません。このメソッドの実装では、再帰的な呼出を防止するために必要な属性全てへの参照で ‘`obj.__getattribute__(self, name)`’ のように基底クラスのメソッドと同じ属性名で呼び出さなければなりません。

3.5.6 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「読み取り専用データ記述子」と呼びます。

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, そのためにインスタンスでメソッドの再定義や上書きが可能となっています. 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

3.5.7 コンテナの呼出

3.5.8 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こと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、識別子を持たせることで名前が一致した場合でも、識別子の違いから識別ができるように工夫しています。

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、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 “\” で記述されます。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となる ‘/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 のものな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、プログラムにて一つの実行単位となる区画です。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、名前が参照可能な範囲のことです。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場合、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 は函数 list() の函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、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ると、その名前の参照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となります。そして、コードブロックで用いられていても、そのブロックで定義や束縛が行われていないとき、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無い場合に NameError 例外がOutput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場合には UnboundLocalError 例外がOutputされます。この UnboundLocalError は NameError のサブクラス (NameError クラスの種の一つ) となっています。なお、del 文で指定された対象は、del 文の目的が対象の束縛の解除であるものの、束縛済みのものと見做されます。

import 文や代入文は、クラスや函数定義、モジュールレベル内で行われます。

global 文で指定された名前がブロック内にあるとき、その名前は名前空間の最上層で束縛された名前の参照を行うことになります。

3.9 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もできます。

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になります。

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 例外となります。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。

Pyython 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り、具体的に内容を列記したものであるか、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) "}"
```

集合表現では、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。‘{}’ は空のタプルとなるからです。

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 Python の式が構成されます。この小節で解説する式が Python の最も基本的な単位となります。

■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、添字を必要とする場合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 とほぼ同様の表記となっていますが、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ります。

さらに重要なことに、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 風に 0 から開始することです。この点についてはプログラムを行う上でも注意が必要です。

■**呼出**: 関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 の関数 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関数を呼出す場合は、通常はライブラリ名の属性として関数を呼出すことになります。たとえば、sin 関数は 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〉.〈関数名〉として通常は呼出が行われます。

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]
```

単項算術演算子には, 一次語そのものと, その一次語を用いて構築した幂乗, それに加えて算術演算子 “+” や “-” に加え, 二進数のビット反転演算子 “~~” を先頭に置いた式があります。ここで幂乗 a^b を Python では ‘ $a**b$ ’ で表現しますが, 他の多くの数式処理で用いられている演算子 “ \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, 幂乗とは別の演算子と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 ::= 单項算術演算式 | 乗法的算術演算式 "**"
                     单項算術演算式 | 乗法的算術演算式 "//" 单項算術
                     演算式 |
                     乗法的算術演算式 "/" 单項算術演算式 |
                     乗法的算術演算式 "%" 单項算術演算式
加法的算術演算式 ::= 乗法的算術演算式 | 加法的算術演算式 "+" 乗法的算
                     術演算式 | 加法的算術演算式 "-" 乗法的算術演算式
```

加法的演算子としては, 演算子 “+” と “-” があります。乗法的演算子には, 積 “*”, 商 “/”, 剰余 “%”, それと小数点以下の切捨を伴う商の演算子 “//” があります。

■ずらし演算式: 整数値を二進数で表現したときに, ビットを左, あるいは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0、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“^”を幂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となっています。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ます。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 | λ-式

```

この構文は ‘x if y else z’ の形式で、y が False または 0 の場合は z、それ以外は x が返却されます。なお、if 節には else 節が必ず含まれていなければなりません。

■λ-式：lambda-式は無名函数を構成する構文です：

条件演算式の構文定義

```

λ-式 ::= "lambda" [パラメータの列]: 一般式
旧 λ-式 ::= "lambda" [パラメータの列]: 旧一般式

```

■式の列：式をカンマ “,” で区切った列はタプルになります。Python Reference Manual ではこれを「式の列」と呼びます：

式の列の構文定義

```
式の列 ::= 式 ( "," 式 )* [","]
```

このことを簡単な例で確認しておきましょう：

```

>>> True, True
(True, True)
>>> True, True,
(True, True)
>>> a=True, True,
>>> type(a)

```

```
<type 'tuple'>
```

函数 type() で確認した結果から判るように「式の列」はタプルになっています。

3.11 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でありながら、何も実行したくないときに用います。

■**del 文**: オブジェクトの削除を行う文です:

del 文の構文定義

```
del 文 ::= "del" 標的列
```

標的列に対する削除では、指定した列の左端の対象で指示されるオブジェクトから右端の対象で指示されるオブジェクトへと再帰的にオブジェクトの削除を行います。

■**print 文**: オブジェクトの値を標準出力に対して出力する文です。

print 文の構文定義

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

■**return 文**: 関数やメソッドで明示的に値の返却を行うために用いる文で、return 文の引数として、関数やメソッドが返却すべき値を記載します：

return 文の構文定義

```
return 文 ::= "return" [式の列]
```

■**yield 文**: 生成関数の定義のときのみに利用されます。このとき、yield 文を用いるだけで生成関数が定義されます：

yield 文の構文定義

```
yield 文 ::= yield 式
```

■**raise 文**: 例外の送出を行う文です：

raise 文の構文定義

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

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

break 文の構文定義

```
break 文 ::= "break"
```

■**continue 文**: break 文と同様に引数を持たない文で、for 文や while 文による反復処理の継続で用います。

continue 文の構文定義

```
continue 文 ::= "continue"
```

■**import 文**: Python ライブラリを読み込むために用いられます：

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

■future 文: 将来の Python のリリースで利用可能になるような構文や意味付けを行うための指示句です：

future 文の構文定義

```

future ::= "from" "__future__" "import" 機能 ["as" 名前]
          ("," 機能 ["as" 名前])*
          | "from" "__future__" "import" "(" 機能 ["as" 名前]
          | ("," 機能 ["as" 名前])* [","] ")"

```

future 文はモジュールの先頭に置かなければなりません。ここで future 文よりも先行して置ける内容に、文書文字列、注釈、空行と他の future 文に限定されます。

■global 文: コードブロック全体で維持される宣言文で、後続の識別子が大域変数として扱うことを指示するものです：

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、プログラム上、global 文に先行して配置されてはなりません。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函数定義、import 文内で global 文で宣言した名前を仮変数として用いてはなりません。

■exec 文: Python コードの動的な実行に関する文です：

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 ::= 單純文 (";" 單純文)*[";"]
```

ここで示すように、複合文では、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に用意された文です：

if 文の構文定義

```
if 文 ::= "if" 式 ":" 一揃いの文
          ( "elf" 式 ":" 一揃いの文 )*
          ["else" ":" 一揃いの文]
```

標準で準備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の場合のみに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 定義に関連する複合文

■函数定義: 函数やメソッドの定義のための文です.

函数定義の構文定義

函数定義 ::= "def" "(" [パラメータ列] ")" ":" 一揃いの文
 函数名 ::= 識別子
 decorated ::= decorators (クラス定義 | 函数定義)
 decorators+ ::= decorator+
 decorator ::= "@" 付点名 "(" [引数の列] [","] ")"
 NEWLINE
 付点名 ::= 識別子 (" ." 識別子)*
 パラメータ列 ::= (パラメータ定義 ".")*
 ("*" 識別子 [, "*" 識別子] | "*" 識別子
 | パラメータ定義 [","])
 副リスト ::= パラメータ (," パラメータ)* [","]
 パラメータ ::= 識別子 | "(" 副リスト ")"

函数定義は Python で実行可能な文ですが、函数本体を実行するものではありません。函数本体は函数が呼び出されたた時点のみで実行されます。なお、函数定義の実行では、局所的な名前空間で函数名に函数オブジェクトの束縛が行われます。

■クラス定義: クラスの定義ための文です.

函数定義の構文定義

クラス定義 ::= "class" クラス名 [継承] ":" 一揃いの文
 継承 ::= "(" [式リスト] ")"
 クラス名 ::= 識別子

このクラス定義文も Python で実行可能な文です。このクラス定義では、最初に継承リストがあればリストの評価を行います。ここで継承リストの各要素の評価結果はクラスオブジェクト、あるいはサブクラス可能なクラス型でなければなりません。それから実行フレーム内部にて局所名前空間と大域名前空間を用いてクラス内の変数への束縛が行われま

す。すると実行フレームは無視されますが、局所名前空間は保持され、それから基底クラスの継承リストを用いてクラスオブジェクトが生成され、局所名前空間を属性値辞書として保存します。それから最後に局所名前空間でクラス名がクラスオブジェクトに束縛されます。

第4章

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。

この 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、それによると、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、

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

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。

さて、PEP-1 によると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 と似ていますが、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 には定められています。その詳細は PEP-1 を参照して下さい。ここでは PEP の状態遷移を図 4.1 に示しておきます：

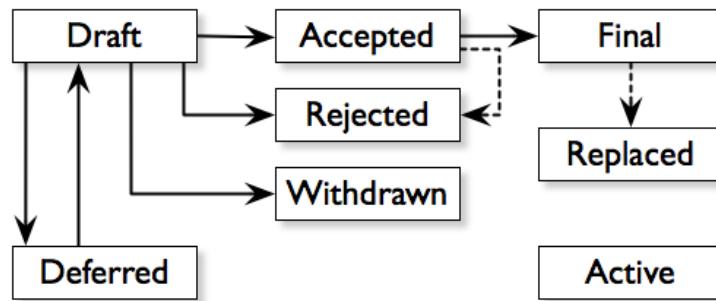


図 4.1 PEP の状態が辿れる経路

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。とにかく面白い詩になっているので下手ながら私が翻訳したものも載せておきますが、PyJUG に日本語訳があるので^{*2}、そちらの訳も参照して下さい：

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

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

素は複に勝る。(Simple is better than complex.)

複は雑に勝る。(Complex is better than complicated.)

平坦は入籠に勝る。(Flat is better than nested.)

疎は密に勝る。(Sparse is better than dense.)

読み易さは重要なり。(Readability counts.)

法を破らんが為の特例は特例に能わず。

(Special cases aren't special enough to break the rules.)

然れど実は理を破る。(Although practicality beats purity.)

エラーは黙認すべからず。(Errors should never pass silently.)

鎮めたるを除きて。(Unless explicitly silenced.)

五里霧中、恣意に囚われること勿れ。

(In the face of ambiguity, refuse the temptation to guess.)

只一無二の大道あらん。

(There should be one— and preferably only one —obvious way to do it.)

然れど、汝、彼の和蘭人にあづんば、先ずは道難し。

(Although that way may not be obvious at first unless you're Dutch.)

成すべきを直ちに成せ。(Now is better than never.)

然れど待つが果報もあり。(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、總員奮励して用うべし！

(Namespaces are one honking great idea – let's do more of those!)

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

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するプロセス PEP の PEP-8 と文書文字列 (docstring) の表記に関する情報 PEP の PEP-257 に従います。

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は推奨しません。

■空行の入れ方：トップレベルの函数とクラス定義の間は 2 行空け、クラス内部でのモジュール定義の間には一行空けます。函数内部でも論理的な区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 以降では 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 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 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。これは名前空間を混乱させる原因になるためです。

悪い例

```
from test import *
```

この場合、モジュール test に含まれる函数や変数には識別子が付かないため、既存の名前との衝突が生じる可能性があります。むしろ、衝突を避けられるように ‘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: 余分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多言語利用者に読まれる可能性があれば、英語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 行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 “””” 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の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るときは内部で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となっています。

まず、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:小文字を_で繋げたもの
- UPPERCASE:大文字のみ
- UPPER_CASE_WITH_UNDERSCORES:大文字を_で繋げたもの

- CapitalizedWords: ラクダの瘤表記
- mixedCase: 小文字と大文字の混合
- Capitalized_Words_With_Underscore: 大文字化と_を繋げたもの

また、短い接頭辞を付けるという Python ではありませんが利用されない様式もあります。たとえば、X11 ライブラリの公開函数名の先頭には ‘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、函数名では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。

ここで避けるべき命名として、小文字の ‘l’, 大文字の ‘o’, 大文字の ‘I’ を一文字の変数名として利用することを挙げています。これらは ‘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: ‘_’ 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、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 ‘_’ 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’ をうしろに付けることとします。
- フункци名: 小文字のみ、あるいは可読性のために ‘_’ で単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: フункциと同様の規則で命名します。
- フункци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 “_” を使って単語を分離したものとします。

■継承のための設計:

- 公開属性の先頭に記号 “_” は付けないないようにします。
- 公開属性の名前が予約語と衝突するときは名前の最後に記号 “_” を付けます。
- 公開データ属性は、属性の名前を公開するのが最善です。
- 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し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。バグの隠蔽が生じることを避けるためです。
- string モジュールではなく文字列メソッドを用いること。文字列メソッドの方が高速で, UNICODE 文字列と同じ API を共有しているためです。
- 接頭辞, 接尾辞を調べるとき, 文字列のスライスの利用は避けること。函数 startswith() や endswith() を用いるべきです。
- オブジェクト同士の型の比較では函数 instance() を用い, 函数 type() 等を使って型を直接比較しないこと。たとえば, オブジェクトが文字列であるかを調べるとき, 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。

4.2.3 Sage での様式

Sage では PEP-8 等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
- 函数名が全て小文字であれば文字 “_” を用いて切り分けます
- クラスや主要な函数名では「**らくだのこぶ記法 (CamelCase)**」を用います

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

は空白文字数で 4 文字が推奨されています。次に、Python では函数名として小文字のみなら ‘set_some_value’ のように文字 “_” を区切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、クラスや主要な函数名では ‘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函数名は「**らくだのこぶ記法**」ではなく「**大文字**」を用いることも許容させています。たとえば、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ないような絶対的な指針にしていません。

4.3 ファイル名やディレクトリ名に関する指針

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单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、多項式環に関連するファイルはディレクトリ ‘rings/polynomial’ に収納されています。

4.4 ライブラリに関する指針

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 では文書は非常に重要視されています. そして, Sage では「**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に, INPUT:や OUTPUT:, EXAMPLE:や 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 や Sphinx といったマークアップを利用する事となっています.

4.5.1 ReST

ReST は ReStructured Text に由来し、名前から予想できるようにマークアップ言語です。

- 段落
- 見出
- インラインマークアップ
- リストと引用
- ソースコード
- ハイパーリンク
- 章立
- 明示的なマークアップ
- ディレクトイブ
- 画像
- 脚注
- 引用
- 置換
- エンコーディング

■**段落:** 一行以上の空行で区切られた文字列の塊です。同じ段落の全ての行のインデントの高さは左揃えで同じ高さでなければなりません。

■**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 “=”, “-” 等の記号を挿入します。HTML と比較すると次のようになります:

<H1>	↔	=====
<H2>	↔	---
<H3>	↔	#####
<H4>	↔	*****
<H5>	↔	^^^^^
<H6>	↔	~

■**インラインマークアップ:**

- 強調 + イタリック: アスタリスク一つで括る
- 強調: アスタリスク二つで括る
- 固定長: バッククオート二つで括る

インラインマークアップの制約として次のものがあります:

- 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。
- インラインマークアップ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ればバックスラッシュ “\” を用いて ‘\’ を用いると空白文字は表示されません。

■リストと引用: 通常のリストは先頭にアスタリスクを置けば番号なしのリスト、数字を置けば数字付きのリストとなります。番号振りを自動で行いたければ記号 “#” を先頭に置きます。リストでもインデントを用いており、あるリストに含まれるリスト項目は同じインデント高さを持っていなければなりません。そして、リストの子リストは親のリストよりも深いインデントとなります。

リストには定義を記述する定義リストもあり、項目の次に改行を入れ、インデントを行つて定義内容を記述します。

■ソースコード:

■ハイパーリンク:

■章立:

■明示的なマークアップ:

■ディレクティブ: 汎用の明示的マークアップで、reST 拡張のための機構の一つです。Sphinx がこのディレクティブを多用しています。ディレクティブは名前、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:

4.5.2 Sphinx

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 type 函数があります。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 を基盤としていますが、統一的な扱いを行うために数学的対象は Python のもともとの型を継承して新たな型を構築しているのです。

ここで 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

SQLite3 を使った解析

6.1 SQLite3 速習

SQLite3 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。PostgreSQL や MySQL のような大規模なシステムを組むことには向きですが、小規模なシステムには軽量さもあって向いています。Sage には SQLite3 が含まれており、この SQLite3 を利用するための事前設定は不要で、ただちに利用することができます。

6.1.1 SQLについて

SQL 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 で言語仕様の標準化が行われています。この 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 (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

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

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

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

6.1.2 SQLite3 の使い方

Sage を立ち上げてから、通常のライブラリと同様に sqlite3 の読みを行います。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)")
<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 の読みを行っています。それからメソッド connect() によって、/home/yokota/ 上のデータベースである 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。最初の "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 型となってしまうので、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。

このように簡単に DB を生成することが可能なので、大量の計算結果を DB に登録して、それらの後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

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}。画像の読み込み、書き込みや表示を行うモジュールは Image に纏められていますので、通常の Python では `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。ただし、Sage では予め PIL の読み込みが実行されているために `import Image` でモジュールの読み込みが行なわれます。

画像の読み込みは通常のファイルのように函数 `open()` でファイルを開いて行います。以後は各種ライブラリのメソッドや函数を使って処理を行います。たとえばディレクトリ ‘/home/yokota’ 上に ‘kodairi.jpg’ という名前の画像ファイルを開いて表示する場合は以下の処理となります:

```
sage: from PIL import Image
sage: im = Image.open('/home/yokota/Athenai.jpg')
sage: im.show()
sage: im.save('Athenai.png')
```

この例に示すように、Image ライブラリの中の函数 `open()` でファイルの読み込みを行い、オブジェクトとして取り込みます。このオブジェクトは名前 `im` に対して束縛されています。なお、ここでの処理では普通の Python でのモジュール読み込みと同様の方式で行っています。それからメソッド `show()` を用いてオブジェクトとして読み込んだ画像の表示を行なっていますが、このメソッド `show()` では外部アプリケーションを用いて画像の表示を行っています。Sage では表示アプリケーションとして `xv` が予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

^{*1} PIL から分枝 (fork) したものに Pillow(<http://python-imaging.github.io/>) があります。基の 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 にも対応しています。

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')`とメソッドshow()のcommandオプションを用いて外部ビューアの名前を必要があればフルパスでPythonの文字列として指定すれば良いのです。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る場合はメソッドsave()でJPEGかPNG形式の画像ファイルとして保存すると、その画像ファイルをノートブックに出力するかたちで表示が行われます。

さて、画像オブジェクトに付随するメソッドの一覧を見たければ、`im.`TABで表示されますが、画像データをRGBの行列データに変換して処理するものはありません。このような処理を行いたければ、matplotlib, SciPyやNumPyといったライブラリが別途必要になります。まず、画像データを数値行列データに変換するためにNumPyが必要です。このNumPyはPythonでMATLAB風の数値行列演算を可能にするための基本的なライブラリです。このライブラリのarrayモジュールを用いることでRGBの配列データに変換することができます。ただこのNumPyは数値行列ライブラリであるために数値行列の可視化は行えません。この可視化では後述のmatplotlibを利用することになります。

```
sage: import numpy as np
sage: imat = np.array(im)
sage: im.size
(800, 386)
sage: imat.shape
(386, 800, 3)
```

ここでは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函数を用います。このarray函数はPythonのリスト等のデータを配列に変換する函数です。さて、本来の画像の大きさはメソッドsize()で調べられます。それに対して配列の大きさはメソッドshape()で調べられます。im.sizeとimat.shapeの結果から、画像配列は画像の縦の画素数が行数、画像の横の画素数が列数になることが判ります。また、この画像配列は‘(386, 800, 3)’であることから‘(386, 800)’の大きさの配列が3個含まれていることが判ります。この第三の添字は画像データの赤(R), 青(B), 緑(G)に対応する配列が存在することを意味しています。

さて、Sageに読み込んだ画像をarrayモジュールで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に数値行列に対する操作で画像の処理が可能となります。そこで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行列が3個含まれていることが函数shape()の結果から判ります。この3個は第三の添字に対応し、第三の添字が0の行列が画像の赤の

強さ(輝度)に対応し、同様に第三の添字が1の行列が画像の青の強さ、そして第三の添字が2の行列が青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行列が得られます。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。

NumPyの配列では、MATLABやYorickに見られる配列処理が可能です。たとえば、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)`と表記しますが、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〉:〈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。

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、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`や`pylab`といったライブラリを利用する必要があります。

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

さらに 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は異なり、予め環境変数の設定を行う必要があります。何故なら Sage 側の Python やライブラリー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 Sage が /usr/local/sage にインストールされ、Bash を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 には `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

手引(ポリピュリオス)

9.1 概要

この章ではテュロスのポルピュリオス ($\Pi\sigma\rho\mu\rho\iota\sigma$) のエイサゴーゲー ($\varepsilon\iota\sigma\alpha\gamma\omega\gamma\eta$) の翻訳を載せます。ポルピュリオスのエイサゴーゲーはアリストテレスの論理学の入門書として書かれたもので、題名の $\varepsilon\iota\sigma\alpha\gamma\omega\gamma\eta$ 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。このエイサゴーゲーのラテン語への翻訳は「最後のローマ人」と呼ばれ、「哲学の慰み」等の著作で知られるボエティウス (Boethius) が行ったものが有名¹ で、その翻訳時の題名のイサゴーゲー (Isagoge) で西ヨーロッパでは知られています。

エイサゴーゲーはラテン語の他にシリア語、アルメニア語やアラビア語にも翻訳されており、哲学の入門書として最初に読むべき本とされていたことから後世に大きな影響を残しています。このエイサゴーゲーの英語への入手し易い翻訳はオーエン (Owen) がイサゴーゲーから翻訳したものと Barnes がギリシャ語の文献から直接翻訳したものの二つが代表的でしょう。オーエンの訳は19世紀半ばの翻訳で、WEB等で公開² されていました。アリストテレスのオルガノンと一緒に併せて廉価で売られていたりします。なお、この英訳は古来からのエイサゴーゲーがアリストテレスの「範疇論」の入門書であるとした立場で翻訳されたものです。それに対して Barnes の翻訳³ はギリシャ語文献から直接翻訳したもので、エイサゴーゲーが「範疇論」の入門書というよりも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、さらに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俄然良いと思われます。そのため、ここでの訳は Barnes の訳を主に参考にしています。

¹ <http://www.forumromanum.org/literature/boethius/isag.html>

² http://www.ccel.org/cCEL/pearse/morefathers/files/poRphyry_isagoge_02_translation.htm

³ https://www.uni-trier.de/fileadmin/fb1/PHI/Strobel_Dokumente/PoRphyry_Barnes.PDF

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アのテュロスで234年に生まれ、シリアル語で王という意味の Malcus と名付けられています。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*)^{*4}と呼ばれていたようです。さらには出身地のテュロスが紫色の染料 (Tyrian Purple) の生産で当時から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」という渾名が付けられています。彼はアテナに留学し、そこでは生き字引、歩く博物館と呼ばれたロンギノス (*Λογγινός*) に修辞学、数学と哲学を学び、それから263年にローマに行って新プラトン主義の創始者として知られるプロティノス (*Πλωτίνος*) の一門に入り、プロティノスから大きな影響を受けます。それから268年には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そこで哲学を教え、301年に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という女性と結婚したことが伝えられていますが、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。また、ポリ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はありませんが弟子は取っていたようです。そして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。またプロティノスから強く影響を受けたことから判るように、ポリピュリオスは新プラトン主義の学者です。このことはアリストテレスの受容に大きな影響を与えることになります。実際、アリストテレスは「形而上学」を見ても判るようにプラトンのイデア論に相当批判的なのですが、ポリピュリオスは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に適合させるという荒業をやっています。こういったこともあって、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羽目になり、12世紀になってイブン・ルシュド (ラテン語名:アベロエス (Averroes), また「注釈者」の名前で西ヨーロッパでは知られています) が「純正アリストテレス」を唱えてそういった夾雜物を排除するまで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の著述の理由については次の伝承が知られています。前述のようにローマでプロティノスと暮らすうちにポリピュリオスは自殺したくなる程の憂鬱に陥ってしまいます。そこでプロティノスの勧めもあって保養のためにシチリア島に行ったと伝えられています。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に行ったとも言われています。何れにせばらくローマを留守にしていたことになりますが、ちょうどそのときにポルピュリオスのローマでの弟子のクリュサオリオス (Chrysaorios, ローマの元老院の議員) がアリストテレスの著作を読むにあたって、ポリピュリオスにローマに戻って指導するか、それができないなら手

^{*4} 東ローマ帝国では皇帝の意味です。

引を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を記述し、これを送付したと伝えられています。

エイサゴーゲ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。その理由の一つに、西ヨーロッパでは前述のボエティウスがエイサゴーゲをギリシャ語からラテン語に翻訳し、更に二つの注釈書を残していますが、このボエティウスの著作がギリシャ哲学の基本的な文献として西ヨーロッパに残され、これらの書籍を基に西ヨーロッパの中世の哲学が発展することになったこと、さらにエイサゴーゲは哲学を学ぶ上で最初に学ぶべき入門書として扱われたことも挙げられるでしょう。たとえば、このエイサゴーゲでは類、種、種差、固有性と偶有性という5つの事象による分類が行われています。この分類はイサゴーゲの第一章に初めて現れており、後世ではこの分類が一般的なものになっています。さらに、ここでの分類からトマス・アクイナス(Thomas Aquinas)はポルピュリオスの木(Arbor Porphyrianae)と呼ばれる系統学で用いられる図を導いています。この著作は西ヨーロッパに限らず、ビザンツ帝国の領域や中東でも権威を持っており、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。

9.2 はじめに

必要なこととして、クリュサオリオス(Chrysarorios)^{*5}よ、まずアリストテレス(*Ἀριστοτέλης*, Aristotle)の範疇(カテゴリー)について学ぶために、類(*γένος*, genus)^{*6}が何であり、種差(*διαφορά*, difference)が何であり、種(*εἶδος*, species)^{*7}が何であり、固有性(属性、*ἴδιον*, 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^{*8} - そしてまた定義の論証、それと一般的な(類の種への)分割や証明に関わる事象にとっても、これらの研究が有用なので - 私は、貴方に要所を押さえて説明するときに、手短で入門の形式をとって、古の賢者達が述べたことを、より深い探究を排除し、そし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。だから、類と種については-

*5 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。

*6 本来、*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、これらはまぜこぜに記述されることの多い言葉とのことです。

*7 *εἶδος*は一般的に形相(form)と誤され、ものの形を意味します。Boethius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 と分けて誤しており、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、アリストテレスは専ら *εἶδος* を用いています。

*8 この類、種、種差、固有性と偶有性の5項目はポルピュリオスによるものです。アリストテレスはトピカ(Tóποι, Topics)では定義、固有性、類、偶有性の4事象を挙げていますが、後世ではポルピュリオスによるこの5事象が用いられています。

それらが存在するのかどうか、それらが実際に素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、それらは実体を持つものなのか、それとも非実体のものなのか、そしてそれらは分離可能なものなのか、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^{*9}。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。ここで私は貴方に古の賢人達 - 中でも殊に逍遥学派の人々 - 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。

9.3 類について

どの類 ($\gammaένος$, genus) も種 ($\εἶδος$, species) も、実のところ、単一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 ($Ἡρολειδαι$, 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 ($Ὀρεστης$, Orestes) はタンタロス ($Τανταλός$, Tantalos) からの類^{*10}、ヒュロス ($Ὑλλος$, Hyllus) はヘーラクレースからの類と言えます^{*11}；それから再び、ピンドロス ($Πινδαρος$, Pindar) は類としてテーバイ人^{*12}、プラトン ($Πλάτων$, Plato) はアテナイ人^{*13}と言えます - とこのように祖國は各人の出生の、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 ($Κεκροποι$, Ceropids, ケクロダイ) はケクロプス王 ($Κέκροψ$, Cecrops)^{*14}、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、単一の起源 (たとえば、ヘーラクレース) に由来する人々の多数、それを区別し、その他から切り分けることで我々はヘーラクレース一族のあつまり全体を類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

^{*9} ポエティウスによるイサゴー第二注解でこの一節を取り上げたことが契機となって中世ヨーロッパで「普遍論争」が生じています。

^{*10} 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。

^{*11} ヒュロスはヘラクレスの息子です。

^{*12} ピンドロスはテーバイ生まれの詩人です。

^{*13} プラトンは言うまでもなく哲学者のプラトンです。

^{*14} ケクロプス王はアテナイの伝説的な初代の王です。

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、さらに多数はその下の全てを包含させられているのです。

さて類というものは三つの方法でそう呼ばれています；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。その説明の縁取りをすると、彼らは‘それが何であるか?’に対する回答として、種で異なる幾つかの事象による述定 (*κατηγορείν*)^{*15} が類であるということでそれを表現します^{*16}；たとえば、動物です。述定する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’）で語られ、またあるものは幾つかの事象 - いわゆる、類や種や種差や固有性や偶有性（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固有性です；そして、白色、黒色、座っているということは偶有性です^{*17}。類は、類が幾つかの事象から述定されたものであるということで、ただ一つの事象だけで述定されたものと異なります。また、類は幾つかの事象で述定されたもの - 種とも異なります。なぜなら種は、たとえ彼らが幾つかの事象で述定されたものであったとしても、種ではなく個数で異なる事象で述定されるものだからです。だから人間は、種なので、ソクラテスやプラトンといった、種ではなく個数で互いに異なる人で述定される一方で、動物になると、類なので、人間や牛や馬といった、個数だけではなく互いに種で異なるもので述定されます^{*18}。また、類は固有性と異なり、それも固有性はただ一つの種 - それを固有性とする種 - それとその種の下にある個体で述定されるからです（笑うことができるということは人間だけ、殊に男の述定だから）、その一方で類は一つの種ではなく幾つかの異なる種で述定されます。また、類は種差とも共通な偶有性で異なりますが、というのも種差と共通の偶有性は、たとえ彼らが種で異なる幾つかの事象で述定されていたとしても、‘それが何なのか?’に対する回答で彼らで述定されるものではなく、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れば、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

*15 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。本来は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」の中に従って述定と訳します。

*16 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續的な生成の存するもの」、「あるものの事物の存在がそれに由来する所の第一の動者」、「平面がさまざまな平面图形の類といわれる意味」と「その物事の‘何であるか’という本質を表すもの」です。ポリビ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、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。

*17 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、固有性と偶有性の5種類に分けて述べています。なお、ポルピュリオスはアリストテレスの範疇（カテゴリー）が10種類あることを後で述べています。

*18 空のものや单一の種だけのものは類として許容されません。類は必ず複数の種で分類されなければならぬものだからです。

であるかと問われれば、我々はそれが黒色のものだと言います - ここで理性的であるということが種差で、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、我々は動物と答えます - そして動物は人間の類なのです。

それゆえに、幾つかの事象で語られる事象がただ一つだけの個体で述定されるものから類を区分します；種で異なる事象で語られる事象は種として、あるいは固有性として述定されるものからそれらを区分します；そうして、「それが何なのか？」に対する回答で、述定された事象が、種差や共通の偶有性からそれらを区分します。それらは‘それが何なのか？」ではなくむしろ‘それがどういったたぐいのものなのか？」、あるいは‘それが何に似ているか？」ということに対する回答で述定されたものなのです。この類というものの輪郭を描くということでは、さて、何らの過剰も何らの不足もありません。

9.4 種について

我々が種 (*εἶδος*, species) と呼ぶのものは、第一に、あらゆるものものの形 (*εἶδος*, form) *¹⁹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 ... *²⁰

我々はまた前述のものあつまりである類の下にあるものを種と呼びます - 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。

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、種で異なる幾つかの事象で述定された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 (*λόγος*) で双方を利用することが重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？」に対する回答で類は述定されたものです。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、個数で異なる幾つかの事象で述定されたものです - ところが、これでは最も特殊 (*ειδικός*, special) なものとただ一つの種のものの表現になり、そうでなければ他のものが最も特殊でないものにも対応することにな

*¹⁹ 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。ところがプラトンはイデア論でイデア *ἰδέα* と併用しており、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。このように本来の意味に「ものの形」の意味があるためにこのような記述になっているのです。なおポルビュリオスは形を *μορφή*(shape), 姿を *σχῆμα*(figure) と区別して記載していますが、種 *εἶδος* はその双方の意味を含んでいます。

*²⁰ 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... と如何にもエウリピデスらしい話らしく、断片が残っているだけです。

るでしょう。

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)^{*21}には、最も総合的($\gamma\epsilon\nu\nu\kappa\omega\varsigma$, general)な事象があれば、他に最も特殊($\epsilon\nu\delta\nu\kappa\omega\varsigma$, special)な事象もあります；そして最も総合的なものと最も特殊なものの間には他の事象もあります。最も総合的なものにはそれよりも上位の事象がありません；最も特殊なもの、この後にはそれよりも下位の種はありません；それから最も総合的なものと最も特殊なものの間には同時に複数の類や種があります(とはいえ、あるものものと他のものとの間に関係を持たされています)^{*22}。

私が言っていることは単体の述定の型(範疇、カテゴリー)の場合であれば明瞭になるでしょう。実体($\o\lambda\sigma\alpha$, substance)はそれ自体が類になります。その下に物体($\sigma\omega\mu\alpha$, 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。これらの事象について、実体($\o\lambda\sigma\alpha$)は最も総合的であり、ただ一つだけの類で、人間は最も特殊で、一つの種に過ぎません。物体は実体の一つの種で、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、動物の類なのです。再び、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($\alpha\tau\omega\mu\omega\varsigma$, individual)^{*23}に先行して存在する全ての事象は单なる一つの種であつて一つの類ではありません。だから、ちょうど実体($\o\lambda\sigma\alpha$, substance)が、いかなる類もその前に存在しないような上位にあり、最も総合的な事象であるようにです。そして人間は、その後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、单なる一つの種であり、そして(個体に)近接する^{*24}種でしかなく、そして我々が述べたように、最も特殊な事象なのです。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、それらの後にい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(それらの種であるとそれらが呼ばれます)，もう一つはそれらの後の事象に対する関係(それらの類であるとそれらが呼ばれます)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、その前にい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、我々が述べたように、それよりも上位

^{*21} Barnes 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。何れにせよアリストテレスの範疇論のことです。

^{*22} ここで述べているように述定には階層があります。この点はラッセルの導入した型理論との関係から見ても面白いものでしょう。

^{*23} 不可分のものという意味。原子(atom)の語源です。

^{*24} $\pi\rho\omega\delta\epsilon\chi\acute{\epsilon}\varsigma$ (proximate) で、間に中間的な種等が入らないという意味です

の類はありません。そして、最も特殊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って、その後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; そして再び; 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、個数で異なる幾つかの事象で述定されているのです。その両端の中間の中間のものを彼等は下位の類や種と呼び、そして彼ら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言われます。

*** アガメムノーン (*Ἀγαμέμνων*, Agamemnon)^{*25}はアトレウス (*Ἄτρεύς*, Atreus) の子供という類、それからペロブス (*Πέλοψ*, Pelops) の子孫という類、そしてタンタロス一族であり、最後にはゼウスのそれとなるようになります。しかし、系図であれば、その大半は起源を遡ると单一の個人 - 要するにゼウス - になるのですが、ところが類や種の場合はそうではありません。というのも、そういった存在するものは全ての共通の類ではないどころか、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^{*26}。(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^{*27}。だからたとえ貴方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道義的ではないと。というのも、その実在が全てに共通する単体の類であったとすれば、全てのものは同名同義的に実体であると言われます。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 (*λόγος*) がありません^{*28}。

最も綜合的な事象は、つまり、10 個です; 最も固有的なものはある個数になって、無限

^{*25} ミノス王アガメムノーンは前述のようにタンタロスの子孫ですが、その父はアトレウス、その祖父はペロブス、そして曾祖父がタンタロス、タンタロスはというとゼウスの息子という説があります。

^{*26} 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対しています。

^{*27} 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、それらを実体、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。

^{*28} 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殊なるもの、「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と述べています。この場合は同名でも意味が違うので「同名異義」なのです。

ではありません; 個体 - こう呼ぶべきで、その事象は最も固有的な事象の後にあります - 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固有的な事象へ降下しようとすることは止めるべきであり、特徴的な種差で中間物を分類しながら、それらを通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、我々が最も固有的な事象に向かって降下するときに、分割したり多数性を通じて進行することが必要で、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多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; 特殊性、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。だから種において共有することで、多くの人々は一つの人間(という種)になります、そして、特殊性によって、その一つで共通の人間(という種)は個々になります - というのも特異性は常に区分するものであり、一方で共通性は集約的であり、統合的だからです。

類と種 - それらの個々は何なのかということ - を表現しているものであって、そして類は一つだとしても種は幾つかになります(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)で述定されますが、種は近接する類でも、上位の事象でも述定されるものではありません - なぜなら、それが入れ替えられないからです^{*29}。等しいものが等しいもので(嘶くことを馬するように)、あるいは大きなものが小さなもので(動物を人間するように)述定されているといった事態でなければならないのです; 大きなものが小さなものではありません - 貴方は人間が動物であるとは言っても、動物が人類であるとは言わないでしょう^{*30}。

種を述定されるものは何でも、そういった事象で、必要性から、その種の類もまた述定されるでしょう - それから最も総合的な事象に到達するまで類が類で述定されるでしょう。ソクラテスが人間であり、人間が動物であり、動物が実体であるということが真であるなら、ソクラテスが動物でも実体でもあることも真になります。そして、より上位の事象はより下位の事象で、種は個体で、類は種と個体の双方で、そして、最も総合的な事象は類(あるいは複数の類で、幾つかの中間的で下位の事象があれば)と種と個体で常に述定されるからです。最も総合的な事象はその下の全て - 類や種や個体 - で言い尽くされます; もっとも固有的な事象の前にある類はすべてのもっとも固有的な事象や個体で言い尽くさられるからです; そして单一の種である事象は全ての個体で言い尽くされ; そして個体はただ一つの特殊性で言い尽くされます。

ソクラテスは個体であると言われ、そしてこれは白くて、この人は魅力的で、ソフロニスクスの息子です。このような事象を個体と呼びます。なぜなら、各自は固有の性質で構成

*29 「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下位概念を説明することができないのです。

*30 主語と述語の入替ができない例です。

され、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まりだからです - ソクラテス固有の特徴は決して他の個人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固有性を意味しています）の固有の特徴は幾つかの事象で同じだということが判るでしょう - あるいはむしろ、すべての特定の人間でも、彼らが人間である限り。

このように個体はその種に、そして種はその類に含まれます。なぜなら、類は全体のあつまりで、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（他の事象ではなく）全体なのです（というのもその一部分では全体だからです）。

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、それから最も固有的なものが何か、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、何が個体になります、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。

9.5 種差について

種差は共通に、固有的に、最も固有的にそう呼ばれるべきです。というのも、一つの事象が共通に多様な事象と異なると言われるのは、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、多様性によって区分されるときです - ソクラテスはプラトンと諸々のことと異なり、さらに実のところ彼自身とも少年のときや成人のとき、そしてそしてあるやり方で活動しているとき、あるいは休んでいるとき、おまけに彼が似ているものについて多様性から異なります。一つの事象が固有的に多様な事象と異なっていると言うのは、それがそれと分離不可能な偶有性で異なるときです - 分離不可能な偶有性は、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、傷から酷い傷跡といったものです。一つの事象が最も固有的に異なると言われるのは、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、つまり理性的であるということで馬と異なるようにです。

総合的に、全ての種差は、何かにそれが付け加えられるときに、事象を多様なものにします；とはいっても、共通で、固有な種差はそれを他の異なったものにしてしまうので、最も固有の種差はそれを全くの別物にしてしまいます。種差についてあるものは物事を別物みたいにしてあるものはそれらを別物にしてしまいます。ここでそれらを別物にするものが特徴と呼ばれ、それらを別物みたいにするものが種差と単に呼ばれます。だから、理性的であるという種差が動物に付け加えられたときに、それを別物にしてしまい、動物の一つ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では、単に止まっていることと比べて単に別物みたいにする程度なのです；だからあるものはそれを別物に、あるものは単に別物みたいにするのです。ここで類の分類で種にしてしまうことと定義 - この集まりの類と種差から生成されるもの - が表現されることとは、こういった物事を別物にしてしまう種差の所為であり、あるものを単に別物みたいにする種差の所為で、多様性だけが構成され

たり、それが似ているということで変わったりします。

最初から再び始めるなら、種差についてあるものは分離可能であり、そしてあるものは分離不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、それとそれらに似た事象、こういったことが分離可能なことです；鉤鼻、あるいは獅子鼻、あるいは理性的、あるいは非理性的といったことは分離不可能なことです。分離不可能な種差については、あるものはそれら自体の原因で持ち、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持っていることですが、死すべきことも、そして知識を享受できるということもそうです；とはいえ鉤鼻であること、あるいは獅子鼻といったことは偶然で、それら自体が原因で持つものではありません。それら自体が原因となる種差があらわれるとき、それらはその実体を説明する上で採用され、そしてそれらはその事象を別物にします；偶然の種差はその実体の説明で言われることも、その事象を別物にすることもありません - そうではなく別物みたいにするのです。再度、それら自体が原因となる種差は過多であることも過少であることも許容しませんが、偶有的な種差は、たとえそれらが分離不可能であっても、増大や減少を受け入れます；だから、分類されるために類も類の種差の双方がそれが類であるというもので多かれ少なかれ述定されるといったことがないのです。だから各々の事象を説明を完全にする種差があるのです；そして任意の事象で、それがひとつで同一のものだから、増加も減少も許されませんが、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ことの双方は増えたり減ったりします。

三つの種差の種が観察され、あるものは分離可能で、あるものは分離不可能であり、また分離不可能のもので、あるものはそれ自体を原因とするもので、またあるものは偶然のものであり、そしてまたそれ自体を原因とする種差についてあるものは、我々が類を種に分割する理由になり、あるものは類を指摘された事象に分割する理由となります。たとえば、以下に示す全ては動物のそれ自体を原因とする種差になります - 生きていて知覚がある、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 -、生きていて知覚があるという種差は動物という実体を構成し得るものです（なぜなら動物は生きていて知覚のある実体だからです）、ところで死すべきことと不死であること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の多様な種差なのです（なぜなら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、これら非常に多様な類の種差は種を全うし、構築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るということと死すべきものであるという種差で人間が構築されることが判り、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、生きていることと生きていな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種差は最高位の事象である実体を分類し得るものであり、生きていて知覚があるという種差は、実体を集め

ることで動物を生成し、それから生きていて知覚がないという種差は植物を生成するのです。そして、ある方法で取られた同じ種差は構成的であることがわかり、それから、ある方法で分類可能なので、それらは全て特徴と呼ばれています；それからそれら - 分離不可能で偶然そうなる種差でもなく、それ程、分離可能でないそれ - は類の分類にも定義の双方で殊に使えます

種差を定義するときに、彼等(逍遙学派の学者達)はこう言います：種差は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 - 動物はこれらのどちらか一方でもありませんし(というのも、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が保持するという訳でもありません(それでもなければ、その同じものが同時に真反対のものを持つことになるでしょう)；むしろ、彼らが主張するように、潜在的にそれの下にある事象の種差全てを保持し、そして実際にそれらの何れでもありません。そして、この方法で存在しないものということから何者も導出されることもなければ、おまけに同時に同じ事象について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)はまた種差というこれらの種類の解説の輪郭をこのように与えます；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で述定されたものです。だから理性的で死すべきものであるということは、人間を述定としたときに、‘人間はどのようなたぐいのものか?’に対する回答であって、‘人間って何であるか?’に対するものではありません。人間って何であるかと尋ねられたとき、こう言うのが妥当です：動物だ；ところで、彼らが‘動物ってどんなたぐいのもの?’という質問を付け足すなら、我々は適切に理性的であって死すべきものであると表現するでしょう。というのも質料(*ὕλη*, matter)や形相(*εἶδος*, form)で構成される対象の場合、あるいは少なくとも質料と形相に類似する構成を持ち、ちょうど銅像が青銅を質料とし、その姿を形相とするように、それと共に、そして人間という特徴は質料に対して類が、形に対して種差で構成され、それから、それら - 理性的であって死すべき存在であって動物である - がその人間というものの全体として、ちょうど、それらが銅像のように取られます。

彼等はまたこれらの種差の集まりを解説しています：種差は同じ類の下にある事象を分離する性質のものです - 理性的であることと非理性的であることは人間と馬を分離し、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：種差は各ものの型の違いである。なぜなら人間と馬は彼らの類では異なりません - 我々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分離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あります。しかし、死すべきものということが追加されると、双方が分離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分離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的一部になる何かだと彼等は言います。だから船に乗って航海するといった性分は、た

とえ人間の固有性であったとしても、人間の種差ではありません；というのも我々がある動物の性質に航海することがあり、他ではそうでないと言えるでしょうが、他からそれらが分離されたとしても、それでも航海するという性質はそれらの実体を全うにするものでもその一部でさえもありません - むしろ、実体の素質の一つでしかありません、というのも特徴であると確実に言えるそういった種差として同じ代物ではないからです。種差は特徴であって、だからそれらが多様な種をつくり、そしてそれらはそうであるべきものに含まれるのです。

これで種差については十分です。

9.6 固有性について

彼等は固有性を四つに分割しています：ある種単体の偶有性が何であるか、たとえそれが全てでなかったとしても（人を診断したり、あるいは計測したりするように）；全ての種の偶有性が何であるか、たとえそれ単体でなかったとしても（人が二本足であるように）；それ単体、そしてそれ全て、そしてあるときに保持している何か（人間が年を取ると灰色になるように）；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（人の笑うことのように）。というのも人は常に笑っている訳ではないので、人が笑っていると言われるのは彼がいつも笑っているということではなく、彼が笑うようなそういった天性 - そしてこれは彼が常に持ち、馬がそうでないのと同様に共通の天性なのです。そして、彼等が厳密な意味で固有性があると言うのは、それらが入替えられるからです^{*31}；もし馬であれば、嘶くでしょうし、もし、嘶くのであれば、馬なのです。

9.7 偶有性について

偶有性は、その基体 (*ὑποκείμενον*, subject) *³²を壊すことなしに、あつたりなかつたりする事象なのです。それらは二つに分解されます：あるものは分離可能なもので、またあるものは分離不可能なものです。眠ることは分離可能な偶有性ですが、カラスとエチオピア人が黒色であることは分離不可能なことです - そ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選んでいるエチオピア人を想像することは可能です。彼等（逍遙学派の学者達）はこう定義します：偶有性は同じものを保持したり保持しなかつたりすることが可能なのです；あるいは：類でも種差でも種でも固有性でもないもので、基体の中に常に内在するものなのです。

*³¹ 「A ならば B」の A, B の入替ができるという意味です。

*³² 基体とは「他の事物は‘それ’の述定とされるが‘それ’自らは決して他のなにもの述定とされない‘それ’（主語そのもの） [2]1028b36.

9.8 共通の特徴

我々が提案した全ての事象 - 私は、類、種、種差、固有性、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共通で固有の特徴が何であるのかを示すことにしましょう。

それら全てに共通する点は幾つかの事象で述定されることです。ところで類は種と個体で述定され、それは種差もそうですが、種はそれらの下にある個体で述定され、固有性はそれらが特徴となるものの種やその種の下にある個体で、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で述定され、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別のもので述定されますが、人間のような種は特別のものだけで述定され、固有性は人間や特別のものが笑うことのように、カラスの種や特別のものの双方の黒色なら、分離不可能の偶有性、人間や馬の動いているということなら、分離可能な偶有性です - つまり一次的な個体の述定ですが、二次的な叙述では、その個体を包含する事象で述定されるのです。

9.9 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；種差はまた種を含みますが、類が包含するものの全てのような訳ではありません - 理性的であることという種差 - は非理性的な項目を動物という類のように包含しませんが、種である人間と神を包含します。

類としてある類で述定されたものはまた、その下の種で述定され、そして、種差としてある種差述定されたものもそれから構成される種で述定されます。というのも動物は類なので、実体 (οὐσία, substance) でありそれから生命のあるものは類としてそれ(動物)で述定されます - それからこれらの事象はまた動物の下の全ての種、個体も含めて述定されます；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)を種差として述定されます - 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でもまた述定されることになるでしょう。

共通性はまた、もし類か種差のどちらか一方が除去されたなら、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、もしも動物がなければ馬や人もなく、だから、理性的であるということがなければ、理性を使う動物はありえないでしょう。

類に対する固有性は、種差や種や固有性や偶有性以上のより多くの事象でそれらが述定されるという事実です。というのも動物は人間や馬や鳥や蛇に関わりますが、四本足は四本の足を持つものだけ、人間は個体だけ、嘶くのは馬と特定の馬共、そして偶有性は似た僅

かな事象だけです。(我々は、類に含まれる実体を全うにするものではなく、類を分類することで種差を取らなければなりません。)

また、類は種差を潜在的に含みます; というのも、動物ではあるものは理性的であり、あるものは非理性的だからです。

また、類は類の下にある種差に先行するものであり、このことが、類がそれらの種差を除去しても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実体が考えられるでしょう - そして、それが動物というもののものです。

また、今まで述べてきたように、類は‘それは何であるか?’に対する回答から、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たものです。

また、各々の種には一つの類があります(たとえば、人間には動物)が、幾つかの種差もあります(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)。

類は物質($\epsilon\lambda\eta$, matter)に似ていて、種差は形($\epsilon\delta\omega\varsigma$, shape)に似ています。

他の共通性や固有性といった事象は類や種差の前にあります - が、これらで十分です。

9.10 類と種

類と種は共に、今まで述べたように、幾つかの事象で述定されています(もしも同じ事象が種と類の双方であるなら、種を種としても類としては取らせません。)。

それらの共通性はそれらで述定された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。

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。

また、類は前方に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前に置かれます; 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。

類は同名同義的にそれらの下にある種で述定されますが、種は類で述定されません。

さらに、類はそれらの下にある種を含むことで種に勝り、種はそれら自体の主催によつて類に一層勝ることになります。

そして、種が最も総合的なものになる訳でもなく、類が最も特殊なものになる訳でもないでしよう。

9.11 類と固有性

類と固有性は共に各々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.

類は等しくその種で述定され, それから固有性もまたその中で共有しているもので述定されます - 人と牛は共に動物で, アニユトスとメレトス^{*33}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種で, 固有性はそれが特徴となるもので述述定されるという共通性もあります. それらは類がはじめにあるということと固有性があとにあるということで異なります - 動物がまずははじめに存在しなければならず, それからあとに種差と固有性で分類されなければならないのです.

類は幾つかの種で述定され, 固有性はそれを固有性とする一つの種で述定されます.

固有性はそれを固有性とするもので互いに述定されますが, 類は何物にも互いに述定され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といったこと等々です.

また, 固有性はそれを固有性とするものの全ての種を保持し, それ単体で, また常にそうです; 類は類であるものの全ての種を持ち, そして常にそうですが - それ単体ではありません.

さらに, もし固有性が取り除かれてもそれらは類も取り除かれないでしょう; というのも, もし類が取り除かれてしまえば, それら類は固有性が所属する種を除去してしまい, それらが固有性となるものが除去されることで, 固有性それ自体もまた除去されてしまうからです.

9.12 類と偶有性

類と偶有性の共通点は, 既に言われているように, それらが幾つかの事象で述定されているという事実です - ここで偶有性は分離可能であったり, 分離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で述定され, カラスやエチオピア人やある非生物の事象も同様だからです.

類は偶有性と異なり, 類はその種にその前にあり, それに対して偶有性はその種の後にあります - というのも, たとえ分離不可能な偶有性を取ったとしても, それを偶有性とするものがその偶有性の前にあるからです.

類の中で共通するものは等しく共通し, 偶有性の中で共通するものはそうではありません.

^{*33} アニユトス *Anūtōs* は政治家, メレトス *Mēlētōs* は詩人で共にソクラテスの告発者です

ん - というのも偶有性の共通は増加や減少を許容しますが、類の中の共通はそうでないからです。

偶有性は第一に個体に存在しますが、しかし、類や種は性質上、個別の実体に先立っています。

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で述定され、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、貴方は黒色だと言うでしょう；ソクラテスがどうななのかと問われるなら、貴方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、固有性、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、それらは続々と数えられることで、第二群は一つの違いで短く、既に判っているようだ、第三群は二つ、第四群は三つ、そして第五群は四つとなります；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、固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話しされたときに話されており、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話されています。だから、どのように種が固有性や偶有性と異なっているかを話す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。固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既にそれらとの関係でこれらの種差について語られています。だから、我々はその他の事象との関係で類についてでは四つの差異、種差では三つ、種では二つ、そして固有性(偶有性との関係で)一つとなります；それらはすべてで10になります、それらの内の四つ - それらは類とその他の関係のもの - を我々は既に説明を終えているのです。

9.13 種差と種

種差と種の共通点は、それらが等しく共有されるという事実です：特定の人間は等しく人間性を共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共通となるもので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。

種差にとって固有なことは、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‘それであるか?’に対する回答で述定されるという事実です。たとえ人間がものの集まりとして取られたとしても、彼は単なるものの集まりではなく、む

しろ種差がその種にして実体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、非常に沢山の動物は四本足で、種で異なりますが、その下にある個体に対してのみに種が適用されます。

また、種差はそれらの種に先立って存在します。というのも、もしも理性的であるということを除去してしまえば、それで人間も除去されますが、だからといって人間が除去されても理性的であるということは削除されません、神が存在するからです。

そして、種差は他の種差で構成されます：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実体を構成します。しかし、種は種で構成されず、別の他の種を生成します。ある馬はある驢馬と一緒に驢馬を生ませるためにやって来ます；ところが馬は単に、驢馬を生成するために驢馬とかけあわせるのではありません。

9.14 種差と固有性

種差と固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前もって存在する事は双方に共通することです。というのも、たとえ二本足のものがバラバラにされたとしても、「常に」ということが、その本質に対する関係で語られるのです、というのも笑うことができるということも「常に」そのような天性があつても常に笑っている訳ではありません。種差に対する固有性はそれらがしばしば幾つかの種で語られるという事実です - たとえば、理性的であるということは人間と神の双方に対して適用します - ところで固有性は一つの種（それが固有性となるものの種）に適用します。種差はそれらが種差となるものの事象に続きますが入れ替えが効きません^{*34}。ところで固有性はそれらが固有性となる事象で互いに述定されるので、だからそれらは入れ替えが効くのです。

9.15 種差と偶有性

種差と偶有性の共通点はそれらが幾つかの事象で語られるという事実です。

分離不可能な偶有性との関係での共通点はそれらが常に、そして全ての場合に対し前もってあるという事実です：二本足は常に全てのカラスに対して前もってあり、そして同様に黒色であることもあります。

それらは異なりますが、なぜなら種差が含もうが含まれなかろうか（理性的であるということは人間を含みます），とにかく偶有性はそれらが幾つかの事象にある限り含み、それからそれらの基体が感知され得るものであって、一つの偶有性ではなく幾つかのものとし

^{*34} 主語と述語の入れ替えができないということです。

て含まれています。

種差は加増可能なものでなく、減少可能なものでもありませんが、それに対して偶有性はそれ以上になったりそれ以下になることを許容します。

逆に種差は混合せず、逆に偶有性は混合するでしょう。

共通性はそのように、そして種差とその他のものの固有の特徴はそのようになります。種がどのように類や種差と異なるかということは、我々が類がどのようにその他のものと異なり、種差とどのようにその他のものと異なるかを我々が語ったときに既に語られています。

9.16 種と固有性

種と固有性は共にそれらが互いに述定されるという事実があります：もしも人間であれば、笑うことができます；もしも笑うことができるのであれば、人間なのです。（笑うことができるということはものの本質として取られるべきであって、笑うということはしばしばそのように言わされてきました。）

種はそれらの中で共有するものの中で等しく前もってあるもので、そして固有性はそれらが固有性となるものの中にあります。

種はまた他の事象で類となれるような種の中で固有性と異なりますが、だいたといつて固有性はその他の事象の固有性となることができません。

また、種は常にそれらの基体で実際に前もってあるのですが、固有性は時にはそのように潜在性があります。というのもソクラテス常に実際にひとりの人間ですが、だからといって彼が常に笑う訳ではありません（たとえ彼が常に笑うことができるというような天性であったとしても）。

そして、もし種差が異なっていると、定義された事象もまた異なります。種の定義は類の下であることで、そして‘それは何であるか?’に対する回答で、個数で異なる幾つかの事象で述定されています；固有性についてはそれに対して単体で、常に、そして全ての場合に対して前もって存在しています。

9.17 種と偶有性

種と偶有性の共通点はそれらが多くの中の事象で述定されるという事実です。その他の共通の特徴は殆どありません。というのも偶有性とそれらが偶有性となるものが互いに遙かに離れているからです。その二つの各々に対する固有性は種が‘それが何であるか?’に対する回答でそれらが種であるもので述定されるという事実ですが、だからといって偶有性は‘それがどのようなたたぐいのものであるか?’あるいは‘それが何に似ているのか?’に対する回答で述定されます。

また、各実体は一つの種や幾つかの分離可能や分離不可能な双方の偶有性で共有するという事実があります。

種は偶有性の前で考えられますが、たとえそれらが分離不可能（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そのような天性であり、そしてそれらは偶発的な天性を持つのです。

種にて共通することは等しく偶有生じます - たとえ分離不可能のそれであっても - 等しくはありません。というのも他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良いからです。

固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、どれだけ固有性が種、種差と類から異なっているかが語られているからです。

9.18 固有性と偶有性

固有性と分離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なのです。というのも笑うことでのきる人間なしに存在しないからです。

固有性が各々の場合と常に存在するように、分離不可能な偶有性もまたそうです。

それらは一つだけの種（笑っていることが人間の中にあるように）で前もって存在していること、ともあれ分離不可能な偶有性でさえも異なっており、たとえば、黒色はエチオピア人だけではなく、カラスや乳牛や黒檀やそういった他のものに対して前もって存在するのです。

また、固有性はそれらが固有性となるもので述定され、それに対して、不可分の偶有性は述定されません。固有性での関与は等しく生じるので、偶有性でも多かれ少なかれとなります。

他の共通で固有の特徴以上にこのように述べたものがあります。しかし、それらの事象を区分して、それらが共通に有するものを指定することには双方で十分です。

参考文献

- [1] アリストテレス, アリストテレス全集 1 カテゴリー論・命題論, 岩波書店, 2013.
- [2] アリストテレス, 形而上学(上下), 岩波文庫.
- [3] 飯田隆, 言語哲学大全 I 論理と言語, 効果書房, 1987.
- [4] 井筒俊彦, イスラーム思想史, 中公文庫, 中央公論社, 1991.
- [5] 今道友信, アリストテレス, 講談社学術文庫, 2004.
- [6] 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.
- [7] 柴田有, グノーシスと古代宇宙論, 効果書房, 1982.
- [8] 藤野登, 論理学-伝統的形式論理学-, 内田老鶴園, 2003
- [9] 田中尚夫, 選択公理と数学, 星雲社, 1987
- [10] フレーゲ, フレーゲ著作集 1 概念記法, 効果書房, 1999.
- [11] フレーゲ, フレーゲ著作集 3 算術の基本法則, 効果書房, 2000.
- [12] 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.
- [13] 山内志朗, 普遍論争, 平凡社ライブラリー, 2008
- [14] 横田博史, はじめての Maxima, I/O Books, 工学社, 2006.
- [15] 横田博史, はじめての Maxima 改訂 α 版 (KNOPPIX/Math に収録)
- [16] 横田博史, 数値計算・可視化ツール Yorick, I/O Books, 工学社, 2010.
- [17] J.Barnes, PORPHYRY INTRODUCTION, Oxford University Press, 2006.
- [18] G. J. Brose, MATLAB 数値解析, Ohmsha, 1998.
- [19] MacLane, The Category theory for working Mathematician, Springer
- [20] Porphyry, Introduction(Iagogic) to the logical Categories of Aristotle,
http://www.ccel.org/ccel/pearce/morefathers/files/po..._isagogue_01_intro.htm
- [21] B.Russell, The Principles of Mathematics, W.W.Norton & Company, Inc., 1996.
- [22] B.Russell & A.N.Whitehead, Principia Mathematica to *56, Cambridge Mathematical Library, Cambridge University Press, 1997.
- [23] Diving into Python: <http://www.diveintopython.net/toc/index.html>

- [24] MathWorks 日本: <http://www.mathworks.co.jp/>
- [25] 相馬の古内裏 <http://ja.wikipedia.org/wiki/歌川国芳>

索引

Python

オブジェクト
 クラスインスタンス, 110
 クラスメソッド オブジェクト, 111
 コード オブジェクト, 111
 コンテナ, container, 99
 スライス オブジェクト, 111
 静的メソッド オブジェクト, 111
 同一性値, identity, 97
 トレースバック オブジェクト, 111
 フレーム オブジェクト, 111
 値, value, 97
 型, type, 97
 クラス, 104
 ファイル, 111
 変更可能, mutable, 97
 変更不能 immutable, 97
 モジュール, 104
 オブジェクトの型
 クラス インスタンス型, 104
 クラスタイプ型, 104
 古典的クラ型ス, 104
 ユーザ定義メソッド型, 103
 ByteArray 型, 102
 Ellipsis, 100
 Frozen Sets, 103
 None, 99
 NotImplemented, 100
 Sets, 103
 UNICODE 文字列型, 102
 数 numbers.Number, 101
 組込函数型, 104
 組込メソッド型, 104
 辞書, 103
 実数型, numbers.Real, 101
 集合, 103
 整数型, plain integer, 101
 生成函数型, 104
 対応付け集合, 103
 タプル型, tuple, 102
 長整数型, long integer, 101
 文字列型, 102
 ユーザ定義函数型, 103
 呼出可能, 103
 リスト型, 102
 列, 101
 行
 エンコード宣言, 89
 空行, blank line, 89

字下げ, indentation, 89
 注釈, comment, 88
 物理行, 88
 論理行, 88
 スコープ (scope), 118
 トークン
 トークン (token), 88
 リテラル, 88, 90
 DEDENT, 88
 INDENT, 88
 NEWLINE, 88
 演算子, 88
 キーワード, 88, 90
 空白文字, 88
 識別子, 88, 89
 字下げ (indentation), 88
 名前, 118
 None, 99
 NotImplemented, 100
 参照, 118
 名前, 90
 名前空間, 90, 118
 リテラル
 演算子, 95
 虚数リテラル, 93, 94
 区切文字, デリミタ, delimiter, 96
 数値リテラル, 90, 93
 整数リテラル, 93, 94
 单文字列, 92
 長整数リテラル, 93, 94
 長文字列, 92
 浮動小数点数リテラル, 93, 94
 文字, 92
 文字列リテラル, 90, 92
 演算子
 is, 97
 関数
 id(), 97
 len(), 102, 103
 makefile(), 111
 open(), 111
 os.open(), 111
 type(), 97
 構成子
 bytearrays(), 102
 classmethod(), 111
 frozenset(), 103
 set(), 103

文 staticmethod(), 111

import, 104

メソッド

- call__(), 104
- cmp__(), 114
- del__(), 113
- delattr__(), 115
- delete__(), 116
- eq__(), 114
- ge__(), 114
- get__(), 116
- getattr__(), 115
- getattribute__(), 115
- gt__(), 114
- hash__(), 114
- init__(), 104, 113
- le__(), 113
- lt__(), 113
- ne__(), 114
- new__(), 104, 112
- nonzero__(), 114
- repr__(), 113
- set__(), 116
- setattr__(), 115
- str__(), 113
- unicode__(), 114

close(), 99

え

エスケープシーケンス, 93

か

鍵(キー), 103

鍵値, 103

拡張スライス操作, 102

クラス

MRO(Method Resolution Order, 106
C3 MRO(C3 Method Resolution Order,
108

基底クラス, 83, 105

継承, 83

サブクラス, 83

スーパークラス, 83

派生クラス, 83

こ

構成子, コンストラクタ, constructor, 96

塵回収, garbage-collection, 98

し

実行フレーム, 111

消去子, デストラクタ, destructor, 97

す

スライス操作, 102

と

到達不能の状態, unreachable, 98

特殊属性, 99, 110

特殊メソッド, 112

は

バイトコード, bytecode, 111

ふ

文書文字列, 92

例外

NameError 例外, 90

人名

R

Rossum, Guid van, 29

あ

アウグスティヌス Augustinus Hippoensis,

Augustine of Hippo, 31

アリストテレス Ἀριστοτέλης Aristotle, 35,

37, 38

アリストテレス Ἀριστοτέλης Aristotle, 31

い

イブン・ルシュド (Ibn Rushd, Averroës),

35

え

エピメニデス, Ἐπιμενίδης Epimenides, 33

か

カントール, Cantor, 33

つ

ツエルメロ, Zermelo, 42

ふ

プラトン Πλάτων Plato, 30

フレーゲ, Frege, 33

フレンケル, Frankel, 42

ほ

ポアンカレ Poincaré, 33

ら

ラッセル, Russel, 33

B

BNF

BNF, 84

EBNF, 84

構文変数, 84

終端記号, 84

非終端記号, 84

い

イデア, 30

お

オブジェクト指向

インスタンス, 30

インスタンス化, 30

オブジェクト, 30

継承, 41

か

概念

外延, 32

下位概念, 34

概念, 31

下位の概念, 32

個体, 34

個体概念, 34

種, 32

種概念, 32, 34

種差, 34

上位概念, 34

上位の概念, 32

属性, 31
 単独概念, 34
 微表, 31
 内包, 32
 内包外延反比例増減の法則, 34
 範疇, 34
 類, 32
 類概念, 32, 34

き

逆理
 Banach-Tarski の逆理, 47
 クレタ人の~, 33
 床屋の~, 33
 ラッセルの~, 33

け

繋辞, copula, 35
 形相 *eidos*, eidos), 38
 圈
 CCC(デカルト閉圏), 70
 epi, 54
 iso, 54
 mono, 54
 可換図式, 53
 基本トポス, 69
 グラフ, 55
 結合律, 53
 恒等射, 同一矢, 53
 始域, domain, 51
 自然変換, 57
 射(矢), 51
 終域, codomain, 51
 双対, 55
 双対圏, 56
 対象, 51
 デカルト閉圏, 69
 転置, 61
 同一矢, 恒等射, 53
 同一矢の公理, 53
 特性写像, 69
 評価, 61
 部分対象分類子 (Object classifier), 69
 メタグラフ, 51
 メタ圏, 54
 矢(射), 51
 矢の合成, 52
 離散圏, 66

圏論

始対象, 58
 終対象, 58

し

集合論
 \in -モデル, 49
 ZFC 公理系, 48
 1-要素集合, 43
 宇宙, 49
 外延性公理, 43
 共通集合, 45

空集合公理, 44
 クラス, class, 45
 後者, 45
 \in -構造, 49
 後続, 45
 恒等式, トートロジー, 49
 恒等式, トートロジー, 49
 順序対, 43
 正則性公理, 46
 選択公理, 47
 大小関係, 45
 置換公理図式, 45
 超限順序数, 45
 直積集合, 45
 対公理, 43
 対集合, 43
 分出公理, 45
 幕集合公理, 44
 無限集合公理, 44
 モデル, 49
 類, 45
 和集合, 44
 和集合公理, 44
 順序数
 後者, 49
 後続, 49
 順序数, 48
 順序, 101
 順序関係, 102
 順序集合, 101

そ

存在含意, 35
 存在動詞, 35

ふ

普遍, 32

め

命題
 可述的, 33
 非可述的, 33