Pythonで学ぶデータ処理の初歩 (2008年1月8日版)

辻野 匠 (TuZino Taqumi)

目 次

第1章　はじめに

ここではプログラム言語 Python を使ってデータ処理をする方法を簡単に提示する.

2006 年九月二日 地質調査船 白嶺丸に於いて起筆.

1.1　なぜプログラムか？

近年は計算機資源が安価になり, 多くの一般の人もパソコンを購入し, 計算機に触れるようになった. それによって, コンピュータユーザは増えたが, 計算機の基礎教育を学んだ人が増えたわけではない. そして, 計算機の利用は高まった結果, 計算機の基礎教育を学んでいない人でも重要なシステムの運用や管理をしなければいけなくなっている. コンピュータビギナーであるかシステムマネジャーであるかに関わらず, 少しコンピュータを使い込むといつか必ず衝き当たる壁がある. **市販や既存のソフトウェアでは, 自分のしたいことが実現できないという壁**である. この時, 人は別れ道に立たされる. **未来永劫, 市販・既存のソフトウェアに, 自分ができることを規定される立場に甘受して暮すのか**, それとも, **計算機技術を学び自分でやりたいことを実現する能力を身につけるのか.** 私は人間の自由と主体性を重んじるので，自分でやりたいことを実現する能力を身につけることを強く勧める.

本書が読者の自由に貢献することができれば幸いである. 次に, 読者の中になぜ Python かという疑問が生じるかもしれない. 私は Python でなければプログラムではない, などというつもりはない. むしろその逆である. プログラムは特定の言語 (Java, C...) でなくとも, プログラム言語であれば使えるし学ぶ対象になる.

1.2　なぜ Python か？

Python 以外にもデータ処理をする方法はある. ざっと思いつくだけでも, (1) Shell Script + Awk, (2) Perl, (3) MS-Excel, (4) C, (5) Java, (6) Fortran, (7) Python とある. それぞれの利点と弱点を考察してみよう. (Ruby, Lisp なども当然，評価すべきであるが私の能力を越えるため割愛する) ちなみに，MS-Excel 以外は全てフリーウェアである.

1.2.1　比較言語学

Shell Script + Awk

Shell Script + Awk は伝統的な Unix 使い向けの方法であり, unix user にとって必須のスキルである. コマンドを組みあわせてデータ処理する技法はデータ処理の基本的な考え方を教えてくれる. プログラム言語といってもよいが, 変則的なワザを組みあわせて, データを処理することが多いので, 最初にデータ処理を学ぶにはおすすめできない (最初に Unix を学ぶにはお勧め/というより必須であるが) . Unix を使うのであれば, そのうち使う. Unix の

シェルの取扱いについては砂原他 (1996, 2001); Arthur and Burns (1997); Kernighan and Pike (1984) 等を，Awk については Aho et al. (1989); Dougherty and Robbins (1997) を参考にされたし．

Perl

Perl は (1) を置換できる機能をもつスクリプト型プログラム言語である．非常の高機能でモジュールも多く, 主要なモジュールは CPAN というサイトに登録されており, 世界中から自由に利用参照できる．しかもユーザー数が多く, 教科書も優れたものが多い．言語は比較的自由に記述でき, 「一つのことをするのに複数の方法がある」ことを目標としており, ある機能を実現するのに, 何通りかの方法で書けるようになっている．弱点としては再利用可能性・可読性の低さがある．再利用可能性の低さはプログラムの自由度の高さと関係している．機能の切り分けができていないモジュールは計算機上では問題がなくても, プログラムの仕様を人間が理解できないため再利用できないものになりがちである．また, Perl では記号 (例 `$_`) に特殊な意味を割当てているので，プログラムが宇宙人語風になりがちである (例 `$_.='@'|'*'&~'!';`)．そのため, しばらく使っていないと忘れてしまう．同じことを何通りでも書けるのもよしわるしで, 如何様にでも書けるため, 気分によってコードの体裁がばらばらになってしまい, 読みにくいコードになる．このような理由で, プログラムを書きはじめの人や, 散発的に改変する人には向かない言語と思っている．とはいえ unix user 必須の技術であることには違いない．入門書としては，深沢 (1999); Schwartz and Christiansen (1997); 斎藤他 (1996) がある．とくに Schwartz のものはラクダ本と呼ばれて親しまれている．ただし本当のラクダ本は Programming Perl(Wall et al., 2000) である．Python との比較には Browm (2002) がある．

MS-Excel

MS-Excel は組織化されていないビジネスユーズで最も広範囲に用いられている表計算ソフトである[1]. MS-Excel は作表の機能は完全に満しているが, データ処理のツールとしては致命的な欠点があるため, Excel はデータ処理の王道にはなりえない (と私は診断している). Excel でデータ処理を修得すると間違った習慣が身につく恐れがある．主な欠点は次の通りである．Excel のワークシートはデータ構造をなんでも二次元の配列 (表) にしてしまい, データの本来もっている論理構造を反映しない形でしか扱えない．たとえば表のタイトルは一番上に掲示される．Excel で云うと一行目の最初の列 A1 にタイトルが記入される．データ構造は一行あけて 3 行目くらいから始まる．しかし最初の行は項目の説明に費やされるので, データ本体は 4 行目からである．論理的になんの関係もないものが，1〜4 行目まで並列に取り扱われてしまうのである．しかも表のタイトルは人によって表の真中くらいの列で D1 くらいに記入されることもある．まったく見て呉れだけで論理的ではない．端的にいってしまえば Excel は計算のために存在するのではなくて，レイアウトを指定するために存在しているのかもしれない (とはいえ，列幅の異なる複数の表を同じシートでは取り扱えないという欠点がある).

人間とは融通が効くものでデータ構造がおかしなものを見せられても, 「ああ, これはこういうつもりなんだけど, 表にはうまく書けないので, このセルに入れたのね」と好意的に解釈してくれることが多い．しかし, 好意的な解釈は「常に」ではない．また, データ構造の

[1] 組織化しているビジネスはたいてい自前のツールをもっている．

おかしなものばかりを見ていて, 構造のおかしさに慣れてしまったり， 構造のおかしなところを修正できないので人間のほうで合せるのが通常になると, その人の論理的なデータ処理能力も衰退する．なんでもかんでも二次元のデータシートに押し込む考え方に欠点がある．次の欠点として，入力系と出力系が整理されていないことがある．人から Excel のファイルをもらった時に，どこに入力して，どこに結果が返ってくるのか，作った人に解説してもらうか，自分でセルを隈なく調べないとよくわからない．多少なりともシステム化された体制をもつ動物は口と尻 (肛門) が決まっていて，どこから入力されて，どこから出力するかは定まっている．アメーバのような単純な動物はそうではなく，どっからでも出し入れができるようになっている．入出力の場所がちゃんと決まっていない Excel はアメーバのように未分化なシステムである．特に，中間的な計算結果がずいぶんと遠くのセルに置いておかれることもあって，セルの中の函数をいちいち追跡していかなければいけない．プログラムは文章と同じで一次元だから，最初から読んでいけばよい．しかしシートは二次元なので順番に，というわけにはいかない．

また, Excel ではインタラクティブ (人が計算機の前に座って, 処理が済むごとに逐一指示を出すこと) でないと処理できない．これは, 計算機に処理の指示書を書いて, 自分は他のこと (家にかえって寝る, ネコと遊ぶ等) をできないことを意味している．計算機の前で, 数分待たされては処理が済み, また次の処理をマウスでクリックして数分待つということを繰り返すことを意味している．これではデータ処理を自動化できない．今回例にあげた他の候補は全て, 処理を自動化できる．VB を使えば自動化はできるが, VB は使いやすくもないし, 設計がきれいでもないし, robust(堅牢) でもない．VB のセキュリティの脆弱性はよく知られたところである．このような理由から Excel はデータ処理の主力兵器としては向かないと云える．どうしても使いたい向きは Hawley and Hawley (2004) を参照されるとよい．

補足 Visual Basic：Visual Basic はプログラム言語であるがお薦めしない．理由については Raymond (2005) を参照されたい．

C と Java

C は unix hacker の mother tougue(母語) といわれている．文法的規則が少なく, かなり自由度の高い言語である．ハードウェアの基本的なところを制御できる強力な言語であり, ユーザーも多い．ただし, 修得は難しく, データ処理用にアレンヂするには「基礎学力」が必要である．その「基礎学力」のレベルが高い．C を理解するには, ハードウェアの原理や計算機の中でどうしてプログラムを走っているのか深い理解をする必要があり, 初学者にそれを求めるのは難しいし, データ処理のユーザーには当面必要のない知識である．Java は Sun が開発した object 指向のプログラム言語で, 一時ブームになり, 多くのユーザーが存在する．多くのプラットフォームでそのままコードが動くのが特徴である．Once write, run everywhere. ただし, 最近では Microsoft が Windows でしか動かない Java の拡張を提案しており, 互換性の高さもいつまで続くのか不安がある．データ処理の実務者としては, 人からもらった Java の code がちゃんと動くのかどうか気になるところである．Java は C ほどベーシックではないが, いろいろなことができるようになっており高機能であるものの, データ処理用に特化していないため, Java でデータ処理をするには, それなりの「基礎学力」が必要である．Java はきちんとした C のコードがかける能力がないと, Perl のように可読性・再利用性に劣るコードを生産してしまいがちである．このように, C や Java は高機能であるため, 逆にデータ処理の実務者の初学には適さないと考える．

C 言語は 1972 年誕生の古い言語で (もっと古いのもあるが)，いろいろなレベルの本が用

意されている．もっとも代表的な参考書は Kernighan and Ritchie (1988) があり，K&R 本と呼ばれている．出版社はこの本が大量に売れることを期待していなかったが 5000 万部以上売れた著名な本である．ただし，この本は初学者には非常に閾が高い．C 言語には非常に多くの参考書があるが，初学者のよい学習書として結城 (1995, 1998) がある．よりコンサイスな入門書には椋田 (1993) がある．C 言語の言語構造のヒントをまとめたものにアンク (2002) がある．Java については結城 (1999a,b) を参照されたい．

Fortran

Fortran は 1980 年代ごろから科学技術計算のために開発された言語で, データ処理を目的としている．ただし, 現在ではユーザーは少ないように思う．また, 言語の設計が古いため, 現在のデータ処理に若干不向きなところがある．たとえば，C 言語のように記述に自由度の高い言語だと, データ処理の内容にあわせて自分でコードを書くので現状にあっていないところも気にならないのだが, Fortran は科学技術計算に特化しているため, 現在の状況にあっていないところを修正するのは面倒になると思われる．量子力学や素粒子物理学，宇宙論では過去の遺産として膨大な Fortran プログラムが存在し，遺産を継承する形で現在も使われつづけている．おそらくこの分野においては将来も継続するものと思われる．言語の詳細は調査中である．教科書には原田 (1986) がある．

Python

Python は perl の高機能をもち, 可読性が高いのを特徴とする．手続き式でも objective でもコーディングできる．コーディングスタイルとしてプログラムの論理構造を反映させたコードの構造にしないといけないので, 自然と論理的で読みやすいコードになる利点がある．Perl が一つのことをするのに複数の方法をできるようにしているのに対して, Python は一つのことをするには一つの方法で実現できるように設計されている．そのため, 記述が気分によってかわったりしない．また, 指示子に記号を使うことを極力避け, 関数やオブジェクトは一見して理解できる名前になっており, あとから (しばらく時間があいて忘れかけた頃に) 読んでも, なにが書いてあるか容易に理解できる．Python のユーザー層には相当の scientist がおり, 科学技術コミュニティおよびモジュールのが充実している．膨大にある C も関数や Fortran のモジュールも簡単に外部関数として参照できるようになっており, 将来ハイレベルのデータ処理をする時にも人の関数・ライブラリを参照できるし, また, 計算速度を求められる重要な演算の部分だけを C で書いて, Python からそれを参照するように使うこともできる．全体のコードの流れを Python で書くことで見易くなり, 律速的な計算を C で書くことで速度もはやくすることができる．このように初学者には利点の多い言語であり, 今回紹介したい．唯一の欠点は, 動作速度が (1)–(7)[Excel 除く] の中で最も遅いことで, 実行速度は Perl の数倍から数十倍遅い (演算内容によっても異なる)．ちなみにもっとも早いのは C であり, C を Perl との間も数倍から数十倍の速度の差がある．なお, Perl では型は非明示であるが, Python では型は明示である．型は非明示のほうがちょっとしたことをやらせるのは簡単だが, 本格的なコードを書くと型はあいまいなので, プログラマーのほうで頭の中で型を把握しておかなければいけないためかえって管理コストが圧迫する (つまりデバッグやレビューが困難になる).

Python の参考書についてはあとでまとめて記述する．

結局のところ...

結局のところ決め方は, 近くにいるプログラム言語を知っていて人柄の印象がよい人 (これ重要) が勧めるものを使うというがもっとも妥当な決め方のようだ. というのは, 一人でやっていてつまずいた時 (かならずつまずく, 最初から全てうまく行く人はいない), 本を読んでもわからず (本を読んでわかるというのは, 自分がなにをわかっていないのはわかっているということ. 初学者はたいてい何をわかっていないもわからないものである), 頼れるものは究極的には近くの人である. 私個人としては, Excel 以外のどの言語であっても (Excel はそもそも言語ではない), 適格なアドバイザーの近くで学ぶなら, どれでもいいと考える. 言語には固執する価値はないが, 教える者の人柄は重要である. 理解というのは究極的には理解する側の問題である. 教える側にできることは限られている. その限られた「できること」は, 必要なことを云うこと, 不要なことを云わないこと, そして, 相手を聞く気にさせることである.

人柄の悪さが何にも勝して効率を低下させる (Weinberg, 1979)

また, 私の上述の評説も他の誰かの言語の評説も, なんの言語も知らない人にが読んでも理解できない. 初学者のための言語の解説も, 言語をある程度知っていて解説が理解できる. しかし, 初学者のための言語の解説は言語を知らない人のために書かれている. 「運転免許を取るためにはクルマを運転しなければいけないが, クルマを運転するには運転免許が必要」という argument のように, これじゃ最初がはじめられない (から先に続かない) ということなりかねないが, 人間は最初がわからないからといってその先もわからないわけではなくて, やっているうちにわかってくるということがある. 最初は難しいことばかりだが, 物事の最初というのは往々にしてそういうものだと思って, 付き合ってあげてほしい.

Python はチュートリアルやレファレンス等のドキュメントが充実している. 実際, チュートリアルはそれらへんの入門書と同じくらいの出来栄えである. (しかし, Python 使いがもっともよく使うのは, Library reference だ. 私がプリントアウトした時で 800 頁にもなった. 今はさらに増えているだろう...)

```
http://www.python.jp/Zope/
http://www.python.jp/Zope/links/python_documents
```

1.3 プログラマーに必要な要件

プログラム言語は言語の一つである. 言語は自然言語と形式言語にわけられる. 自然言語は人間がしゃべる言語である. 形式言語は, 数学と論理学とプログラム言語が含まれる. 自然言語は人間誰でもひとつはしゃべっている. プログラム言語も自然言語と同じ言語であるから, そういう意味では誰しもプログラム言語に対する予備知識はあるといえる. プログラマに必要な要件としてはの第一は自然言語を使えるということである. ただし, 自然言語の一つが使えるからといって形式言語が使えることにはならない. それは自然言語の一つの英語がしゃべれるからといって日本語がしゃべれることにはならないのと同様である. だから, 要件の第二番目以降は, 形式言語と自然言語の違いに起因するものである. 形式言語では厳密な定義があり, 意味が明確な文からなるのに対して, 自然言語は曖昧で, しばしば裏の意味があり, 文意が一意に定まらないことがある. たとえば京都の「ぶぶ漬け食べていってください」=「はよ帰ってください」は有名である. 京都人は, この文は退去の催促という一つの意味しかもたないので明確だと主張するかもしれないが, 京都人を知らない人には日本人

であっても通じない, のみならず誤解を与える．こういう文は明確とはいえない．形式言語は違う．形式言語は「わかる (意味のある文」か「わからない (意味のない文」の二種類しかない．であるから, プログラム言語を上手く使うためには, 言葉を厳密にかつ平明に使うことが要件となる．また, 自然言語では, 文脈や飛躍が許されるが, 形式言語では論理学数学に代表されるように文脈や飛躍は認められない (というか通じない)．計算機は数学や論理学の教師よりも融通が効かない．つまり論理的な思考が要件となる．

しかし, もっとも重要な要件は, 好奇心が強いことである．好奇心の強さがなによりの強い味方となるであろう．

1.4 インストールと初期設定

1.4.1 インストール

Python には，主要な Unix, Linux システム用のバイナリーの他，Windows や Macintosh のバイナリーも既に存在している (下記 url 参照)．もちろんソースファイルも公開されているから，自分で構築 (ビルド) することもできる．MacOS X であれば最初からバイナリーがインストール済みで，Linux ディストリビューションの多くも最初からインストール済みである．NetBSD, FreeBSD などの Berkeley は余分になり得るものを極力排除する設計思想のために，最初からはインストールされていない．しかし，pkgsrc/port システムによりソースファイルから簡便にビルド・インストールできるし，サイトにはバイナリーファイルも用意されている．個人的にはソースから自分が使う仕様を調整してビルドするのが好きであるが，もし，ここまでで何のことを云っているか見当がつかないようであれば，まづは自分のプラットフォーム用のバイナリを入手することを試みるのがよい．下記 URL の日本のサイト (上段) は日本語対応の Python で，Windows と Macintosh のバイナリ，およびソースファイルが置いてある．下段は公式サイトで，各種プラットフォームのバイナリがある．ここや自分の OS のサイトでバイナリーを探してそれでも見付からない時にソースから構築することを検討するべきである．その際は C 言語が話せるか，この種のフリーソフトのインストール経験のある者の助けが必要になるかもしれない．

日本のサイト `http://www.python.jp/Zope/download`
公式サイト `http://www.python.org/download/`

1.4.2 初期設定

3 つの環境変数 `PATH`, `PYTHONPATH`, `PYTHONSTARTUP` を正しくセットする必要がある．

`PATH` は OS の Shell のパスである．簡単に説明すると実行ファイルを置いてある場所である．Python を使用するには `python` が置かれた場所が，`PATH` に含まれている必要がある．`PATH` に関する，これ以上の説明は Arthur and Burns (1997); Kernighan and Pike (1984); 砂原他 (1996) を参照されたい．

`PYTHONPATH` は Python のモジュールのパスである．Python のスクリプトでモジュールが呼び出された時に，この変数のリストから，そのモジュールの位置を特定する．

`PYTHONSTARTUP` は対話的コマンドライン (後述) を起動した時に必ずこの変数に記述されたファイルを実行する．たとえば，歓迎のメッセージを表示するスクリプトを含めておけ

ば，Python を起動する時にその歓迎のメッセージが表示される．また，毎回インポートするモジュールをスクリプトに記述しておくことにより，毎回手入力でモジュールをインポートする手間を省略できる．

C shell 系 (csh, tcsh) では環境変数は次のように設定する．

```
set path = ($path /usr/pkg/bin)
```

あるいは，次のようにする．

```
setenv PATH "${PATH}:/usr/pkg/bin"
```

B shell 系 (sh, bash, ksh) では，

```
PATH=$PATH:/usr/pkg/bin ; export PATH
```

のようにする．

なお，C/B shell 共に，普通に使用しているのなら，PATH の設定はちゃんとなされている筈である．PYTHONPATH もたいていのインストールで適切に設定される．あとから自分でモジュールを追加した時には，その場所を PYTHONPATH に追加しなければならない．

1.5 対話

Python は対話的コマンドラインを実装している．Unix でいれば shell，Windows でいえば DOS 窓のようなものを想像すればよい．端末で，次のように起動する．

```
% python
Python 2.4.1 (#1, Dec 16 2005, 22:46:02)
[GCC 3.3.3 [NetBSD nb3 20040520]] on netbsd2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

最後の>>>がプロンプト (促進 for ユーザーからの入力) である．これに下記上段 print 文を与えると下段の結果を得る．

```
>>> print 'There is a gap between stimulus and response.'
There is a gap between stimulus and response.
```

対話型コマンドラインから抜けるには EOT 文字 (符号) [2]を与えればよい．Unix 系の端末では EOT 符号は Control+D(しばしば C-D/Ctrl-D と表記される) である．Windows では C-Z の場合もある．

[2]EOT は End of Transition の略で「これでメッセージおわり」という意味である．似た言葉に，EOF(End of File: これでファイルはおわり)，EOD(End of Data: これでデータ終わり) があるが，どれも同じ文字符号 (ASCII なら 4) である．

1.6 スクリプト

対話型の使いかたは，簡単にテストでき結果を見ることができるのが取り柄であるが，毎回同じことを入力していては埒があかない．それで，プログラムをファイルに記述し，それを実行することで楽ができる．これを**再利用**という．計算機の利用・プログラムは再利用性を高める方向に進化してきた．

Python のプログラムはコンパイル (計算機が直接理解できる 0/1 の言語 [Binary] に直す作業) をプログラムが呼び出される度に行うスクリプト型の言語である．あらかじめコンパイルしておかなければいけない言語をコンパイル型の言語という (たとえば C 言語)．その都度コンパイルするのは不便に思えるかもしれないが，小規模のプログラムを少し手直しして実行させる，という作業はスクリプト型のほうがやりやすい．

スクリプトは，

```
% python script.py
```

とすれば，スクルプトファイル `script.py` が Python に渡され実行される．Unix 系では更に簡単な方法が用意されていて，

```
% ./script.py
```

とすれば，そのまま実行される．ただし，ファイルが実行可能属性をもっていなればいけない．ファイルに実行可能属性を与へるには，

```
% chmod +x script.py
```

とすればよい．

普通は `PATH` にカレントディレクトリ [今いるところ] は含まれていない (理由は砂原他 (1996) p.30 参照) ので，カレントディレクトリを指す “ `./`” が必要である．また，スクリプトをどの処理系[3]に渡すかをスクリプトの先頭に書いておく必要がある．python の場合，下記のやうになる．

```
#! /usr/pkg/bin/python
```

書式を説明する．普通，行頭の`#`はコメントとして解釈されるが，ファイルの先頭でしかも`!`がつくと特別で，この spell が記述された場合，ファイルの内容がその次に書いてあるファイル (この場合は`/usr/pkg/bin/python`) に渡される．この場合は，NetBSD を想定している．NetBSD では python は`/usr/pkg/bin` にインストールされる (pkg システムを用いた場合)．FreeBSD では`/usr/local/bin` にインストールされる (port システムを用いた場合)．Linux では`/usr/bin` か`/usr/local/bin` であるが，場所も最初からインストールされているかも，ディストリビューションによって異なる．MacOS X(10.2–10.4) の場合は．`/usr/bin` に最初からインストールされている．野良ビルドといって既存のインストールシステムを用いず自分でインストールする場合は特別に設定しなければ`/usr/local/bin` となる．スクリプトは Python がインストールされている全ての計算機で動く (もちろんそのスクリプトが正しいものなら) が，別の OS の計算機にスクリプトを移して使用する場合に，いちいち先頭をその OS の Python にあうように書きなおさなければいけない．これは面倒である．このような Unix では環境の違いによる差異を埋める方法も用意されていて，

[3]Shell，Perl, Python, Ruby といろいろな処理系がある．デフォルトでは sh(B Shell) である

```
#! /usr/bin/env python
```

と記述すると，自動的に PATH’ の中から最初に見付かった Python にスクリプトを渡してくれる．複数の python を入れている時には，思った python に行くとは限らないので注意が必要である．その場合は渡したい python を記述してあげる必要がある．例：

```
#! /optional/bin/python21
```

第2章　手続き型プログラミング

手続き型プログラミングというのは，計算機が行うべき演算の手続きを記述したプログラムで，自然言語で云えば，料理のつくり方を述べた文章が近いだろう．手続き型プログラミングはオブジェクト指向プログラミングの前の世代のプログラミグの流儀であって，当然，オブジェクト指向プログラミングではないが，それは流行遅れとか，時代錯誤ということではない．現在でも立派に通用するやり方である (ただ，手続き型では越えがたい壁を越えるためにオブジェクトは導入されたのである)．Python そのものはオブジェクト指向で設計されているが，手続き型でもプログラミングできるようになっている．プログラムの基礎を修得するために，最初に手続き型プログラミングから述べる．

Python のオブジェクト指向性は随所に発揮されるため手続き型でプログラミングしても，自然にオブジェクトを使うようになっている．これはオブジェクト指向とは何かを実感するにはよい方法に思える．Python を手続き型的に使うことにより,『オブジェクト指向は手続き型プログラムが目指したもの—見通しをよさと再利用性—を追求した結果，論理的帰結の一つとして導出された』ことが理解できるだろう．

2.1　最初のプログラム

プログラムとは何か．計算機がしているのは, 画面でたとえ, どんなにハデなことが起きていようとも整数 (正確には 0 を含めた自然数) の足し算と引き算である．プログラムを用いて整数の足し算と引き算しかできないものを人が理解でき使える形に変え, 人間の用となる知的な道具に変身させる．

2.1.1　最も短かく, かつ有名なプログラム

```
the_most_famous_nd_shortest.py ....................................

#! /usr/bin/env python
print "Hello, World!"
```

これを実行すると, “Hello, World!” というメッセージが端末に出る．これは最も有名で短かいプログラムで, C でも Perl でもどの教科書でも最初のほうに例として出てくるサンプルプログラムである．ここで, print の引数を “Hello, World” から “Goodbye Daring” に変えるとどうなるだろうか？ メッセージが “Goodbye Daring” に変わるのである．実際にやってみよ．また, メッセージを変更するのではなくて, また，メッセージを “Goodbye Daring” にしたプログラムを一から打ち込んでみよ．そして, 手間やタイプミスを比較せよ．さらに, 次のプログラムを導入してみよ．

```
three_times.py ....................................................
```

```
#! /usr/bin/env python
print "Hello, World!"
print "Hello, World!"
print "Hello, World!"
```

これは要するに “Hello, World!” を三回出力するプログラムであるが，これを “Hello, World” から “Goodbye Daring” に変えよ．ウンザリしないだろうか？また一から書きなおすなんて，なにか無駄なことをしている気がしないだろうか．ここで，定数という概念を導入する．数というが，これは，この概念が数学に由来しているので数を称しているが数でなくてもよい (この場合は文字列を示す)．しかし計算機上では全てのものは数字の羅列に他ならないため，数という表現はいみじくも正しい．

three_times2.py .....

```
#! /usr/bin/env python
message = "Hello, World!"
print message
print message
print message
```

この例文では message 定数を “Goodbye Daring” に変更するだけで足る．これをテキストエディターで検索置換で処理する場合と比較せよ．定数を設定する方が，**より問題の本質に近付いた**気がしないだろうか？ この命題に賛成できなくても，上述の動作を相互に比較したらば 次のような命題が導かれるであろう．

前に書いたプログラムの機能を書きかえて別の機能に変えたほうが，
一から書くより楽．

Unix では, 次のような格言がある．

車輪を二回発明してはいけない[1]

しかし, このプログラム, 私には面白くない．なぜか, それはフィルターになっていないからである．つまり, このプログラム (ここでは Hellow, World 型と称す) は入力を必要としない．入力を必要としないプログラムは, 時限爆弾のように決めたことを実行するだけの用途には足るかもしれないが, 時限爆弾とて設定が複数あると Hellow, World 型のプログラムでは対応しきれないところがある．設定ごとに, コードのパラメータを修正するよりは, 引数に設定を入れる (入力！) ようにした方が楽であろう．

それに, データ処理は入力としてデータを入力しそれを処理して出力することである．そこで入力を必要としないプログラムの解説はここで中止し, 入力を必要とするプログラムを先に紹介する．

考えてみると, 人間も外界から自分の感覚器を通じて情報を得 (つまり入力である), その人なりに分析 (つまり，データ処理) し自分の行動に反映させる (出力)．人間もデータ処理をしている．

[1] とはいっても，自分でちょっと考へれば作文できるものはその都度発明すればよい．たとえば，ネット上には本稿で後で紹介するデータの間引きプログラムのやうな簡単なプログラムは滅多に置いていない．それはなぜか？そのような簡単なプログラムは C 言語であれ，awk であれ，perl であれ入門書を読めば誰でも作文できてしまうことだからだ．ブロガーはともかくプログラマーはそのような簡単なプログラムを公開することをよしとしない．せいぜいか本稿のような言語の入門書の例題として紹介しているくらいである．置いておくのが悪いと主張しているのではない，念のため申し添える．

2.1.2 最もプログラムらしいプログラム (私見): フィルタ

フィルターの威力

最も簡単なフィルタ

最も簡単なフィルタは入力をそのまま出力にコピーするフィルターである.

filter.py .....

```
#! /usr/bin/env python
import sys
for line in sys.stdin.readlines():
        print line,
```

filter_while.py .....

```
#! /usr/bin/env python
import sys
while 1:
       line = sys.stdin.readline()
       if line==’’ : break
       print line,
```

使い方

```
filter_while.py < spam.data
```

filter_for.py .....

```
#! /usr/bin/env python
import fileinput
for line in fileinput.readlines():
        print line,
```

使い方

```
% filter_for.py  spam.data
```

2.1.3 書式付文字列 printf 互換の機能

Python の print は書式を指定した表示ができる．これは C 言語の printf 函数と同じ実装である．Python に限らず C 言語で記述されている言語は print(あるいはそれに類する機能) を C 言語の printf を借用しているので，同じ書式で同じ機能が実装されている．printf と同じであるから，完全な仕様は

```
% man printf
% man 3 printf
% man 9 printf
```

で得られる．ここではよく使用する使い方のみ説明する．
書式付 print の構文は

```
%[flag][min_width][.precision]conversion
```

で与へられる．[ ] は optional であるが，% と conversion 変換文字は必須である．変換文字として下記のうちどれかを指定する必要がある．

d	integer
i	integer
o	octal
x	hexadecimal(lower case)
X	hexadecimal(upper case)
c	first character
s	string
f	floating point (123.456)
e	floating point (1.234e-7 for 1.234×10^{-7})
E	E is used for ‘e’
u	unsigned integer
g	f or e in minimum space
G	F or E in minimum space

使用例を示す．

```
>>> spam=2951
>>> print "Decimal: %d, Octal: %o, Hexadeci %x" %(spam,spam,spam)
Decimal: 2951, Octal: 5607, Hexadeci b87
>>> geologist="hammer"
>>> print "Geologist is an earth scientist with %s." %(geologist)
Geologist is an earth scientist with hammer.
```

%d，%o，%x により整数の基数を十進数，8 進数，16 進数で表示できる．x を大文字にすると，abcdef が大文字になる．文字列は%s で表示する．次は浮動小数点の例である．

```
>>> avogad_f=602000000000000000000000
>>> avogad_e=6.02e+23
>>> print "Avogadro Number is \n %f\n or\n %e" %(avogad_e,avogad_f)
Avogadro Number is
 601999999999999995805696.000000
 or
 6.020000e+23
>>> print "or\n %E" %(avogad_f)
or
 6.020000E+23
```

アボガドロ数として与へた値と返ってくる値が微妙に異なることに注意されたい．これは 2 進数で十進数の浮動小数点を書き示す際の誤差である (詳しくは型の項目で説明する)．

また，複数の引数を与へる場合には (*var1* ,*var2*) で括らなければいけない．これはタプルにすることを示す．また，文中に\n とあるのは改行記号である．このような記号には下記のようなものがある．

\e	escape	0x1b
\a	bell	0x07
\b	back-space	0x08
\f	form-feed (≃ new page)	0x0c
\n	new line (改行)	0x0a
\r	carriage return (復帰)	0x0d
\t	tab (horizontal)	0x09
\v	vertial tab	0x0b

ちなみに，%と表示するには%%とする．

```
>>> print "%%" % (4)
%
```

ところで，最初の print 文で整数の spam を様々なフォーマットで整形している．ところが，計算機の習慣として 8 進数は 0 をつけて 04 のように表示する．同じく，16 進数は 0x または 0X とつけて 0x4 のように表示するのが習慣である．これをサポートするのは flag である．flag はそれ以外にも左揃えや符合つけ，スペースで桁揃えや 0 で開き桁を埋めるなど様々な機能がある．一覧は下記のとおり．

-	left adjustment
+	always be signed
␣	spaced for blank
0	fill 0 character padding
#	print in an alternative form
	o(octal): string 0 prepended
	x(hexadecimal): string 0x prepended
	X(Hexadecimal): string 0X prepended
	e E f g G: always contain a decimal point

␣はスペースを意味する (␣と入力するのでない)．それでは，使ってみやう．

```
>>> print "Decimal: %#d, Octal: %#o, Hexadeci %#x" %(spam,spam,spam)
Decimal: 2951, Octal: 05607, Hexadeci 0xb87
>>>                             # 8 進数，16 進数に 0,0x を付ける
>>> print "%+d" %(spam)         # 常に符合をつける．
+2951
>>> print "%#f" %(spam)         # 常に小数点をつける．
2951.000000
```

他のパラメータは幅が設定されないと意味がないので．あとで説明する．

min_width(minium width) で表示する幅を明示する．幅を越える場合は，幅を広げて表示する．幅より狭いところは␣で埋める．*を指定した時は，幅は引数 (int) の値を使用する．

```
>>> print "%08d % 8d %-8d" % (spam,spam,spam)
00002951     2951 2951
```

0 は桁数の残りを 0 て埋める．␣はスペースで埋める．-は幅を左詰めで表示する．
precision(精度) は conversion の種類によって挙動が異なる．

d i o x X	表示桁数 (余りスペースは 0 で埋める)
e E f g G	小数点以下の表示桁数

使ってみやう．

```
>>> print "%8.3f" % 1234.5678
1234.568
>>> print "%8.3e" % 1234.5678
1.235e+03
```

2.2 よく使う術語

2.2.1 コメント

コメントとは何か．コメントはコードの中に含まれる人間が読むための文章のことである．プログラムが複雑になるにつれ，コードの役割や機能的位置づけが人間には難解になる．プログラムの文脈的明瞭さを補うためにコメントは記述される．

コメントは何で示されるか．Python のコメントは，行頭の#で示される．#以降，その行の改行まではコメントと見倣される．

よいコメントとは何か？ よい見本がダイレクトに「よさ」を伝えるように，悪い見本も逆説的に「よさ」とは何か語る．次のようなコメントは悪いコメントである．

```
dmin = min / 60     # min を 60 で割る
```

なぜこのコメントが悪いコメントか？ それは，そんなことは読まなくてもわかるからである．

見てわかることを書いてはいけない．

更に次のコメントは最悪である．

```
dmin = min / 60     # min を 20 で割る
```

なぜか．コードでは 60 で割っているのに，コメントでは 20 で割るとなっている．これではどちらが正しいのかわからない (たいていはコードが正しいのだが)．コードを見ればわかることをわざわざ書いた場合，往々にしてこういう事態になりやすい．最初にコメントした時と状況が変って，それを反映するためにコードを訂正したのだが，コメントの訂正は忘れがちである．次のようなコメントがよいコメントである．

```
dmin = min / 60     # 分 (min) を時 (dmin) に変換．
```

この例では，なぜ 60 で割らなければいけないのか，よくわからない．それをコメントで補うのである．

2.2.2 モジュール

モジュールは，函数 (群) を再利用可能にするために，命令群をまとめて外部函数化したものである．モジュールはブラックボックスとなるので，中でどうなっているか気にせず，インプットとアウトプットにだけ注意を払えばよいので，注意力の節約になる．

```
import sys
```

これにより sys モジュールが組み込まれる．
利用するには

```
sys.exit()
```

のようにする．exit には () がついていることに注目．この () は函数であることを示す．たとえ函数に引数がなくても () は必要である．
函数ではなくて定数である例を示す．

```
import math
```

math モジュールは数学でよく使う函数やら定数をモジュール化したものである．この中には π の値も含まれている．

```
print math.pi
```

とすると，π の値が得られる．

```
print math.sin(math.pi/2)
```

`math.sin` は函数なので括弧「()」が必要で，かつ括弧の中は radian で表記しなければいけない (各モジュールの詳細については Python の Library Reference を参照されたい)．
モジュールによって配列が得られることもある．次に重要な例を示す

```
print_argv.py ..........................................................

#! /usr/pkg/bin/python
import sys
print sys.argv
```

これに適当な引数を与へて実行してみる．

```
% ./print_argv.py a b c d e f        # 実行形式としてスクリプトを実行
['./print_argv.py', 'a', 'b', 'c', 'd', 'e', 'f']
% python print_argv.py a b c d e f  # Pythonの引数としてスクリプトを実行
['print_argv.py', 'a', 'b', 'c', 'd', 'e', 'f']
```

このモジュールのインスタンスは引数を配列をして格納する．配列の最初には，スクリプトのファイル名が格納されていることに注意されたい．なお，配列の番号は最初が 0 から始まる (後述) ので，`sys.argv[0]` がスクリプト名になる．最初に引数 (この場合では `a`) は `sys.argv[1]` となる．このモジュールによりスクリプトの引数をスクリプトに伝えることができる．
モジュールはオブジェクトの一つである．

2.2.3 インスタンス

インスタンス (instance) とは事例やある命題の論拠となる事実のことを一般には意味する．計算機内では，メモリ内の空間を占有 (occupy) するデータの実体のことである．通常は変数により参照される．クラスとオブジェクトの関連で使われる場合はオブジェクトと同義語である．

クラスとはオブジェクトの種類で，属性とメソッドが定義される．

互いに関連のあるデータ要素とそれらに実行できる操作を集めて 1 つの単位として扱えれるようにしたもの．実世界のなにかを抽出してあり，計算機上で操作できるようにしたもの．

2.3 データとは何か

私の理解では，データ (data) とは，そこから推論 (inference) を得ることができる証拠 (evidence) である．

2.4 データの型

データの型を決定する方法には動的型付け (dynamic) と静的型付け (static) がある．Python は動的な型付けをする．静的型付けでは最初に型を定義する．定義と異なる型に対して演算がなされると TypeError でプログラムは停止する．中には実行前に，プログラムを検査して定義と異なる型が代入される可能性がある場合，検査段階で停止する言語もある[2]．動的な型付けでは代入したときに型が決定される．異なる型のものを代入しても，その時点で型が変更される．但し，異なる型のものを演算 (足し算や結合) することはできない．どちらの型に合せるべきか非明示であるからである (perl のようにどちらに合せるか決めている言語もある)．たとえば，

```
q = 7
```

としてから

```
q = 'seven'
```

とするのは valid である．前者では型が整数として定義されるが，後者になると整数という型は破棄され，文字列という型に変更される．しかし，

```
q1 = 7
q2 = 'seven'
```

としてから

```
q3 = q1 + q2
```

ということはできない．整数と文字列の足し算 (あるいは結合？) は定義されていないためである．

[2] このような厳密な型の作法を強い型付け (Strong Typing) と呼ぶことがある．C 言語は弱い型付けで，C 言語の人は，それが普通ともっているので Strong Typing は強い打鍵のことだと思っている．Python は強い型付けである

2.4.1 基本的な型

文字列 (string)

文字列は文字 (文字の型については後述する) の羅列 (シーケンス) である．別に言語の体裁をしている必要はない．文字列はシングルクウォート'something' で括られるかダブルクウォート "something"で括られるものがある．また複数行にわたる (つまり改行を含む) 文字列はトリプルのダブルクウォートで括ることができる．

```
o = 'L"oben Brau, magischen Kr"afte'
     # 「'」でクウォートしている中には「"」が使える．
q = "Never attempt to write something you don't care about"
     # 「"」でクウォートしている中には「'」が使える．
r = 'Never attempt to write something you don\'t care about'
     # バックスラッシュ「\」で 「'」をエスケープする．

quote = """Sloppy authors rarely get published, ---
but perfect authors never,
all you have to do is produce a manuscript
which says something worthwhile,
says it well and is not sloppy
‘my best work at this time' """
```

この方法では改行の他，タブ (インデント) などの制御文字も含むことができる．たとえば文字列中で改行がはいっているところは print 文で改行として表示される．文字列中に別の場所に改行コードを入れるには，

```
quote = """Sloppy authors rarely get \
 published,\n but perfect authors never."""
```

のように改行を\n として入力すればよい．と同時に get の行の終わりの改行は\を用いて無効にしてあげる必要がある (これをエスケープと呼ぶ)．そうしないと，get と published の間に改行が入り，published, と but の間にも改行がはいってしまう．

では，改行を示すシンボルとしての\n を\n として表示したい時はどうすればよいのだろうか．

```
quote = r"""Sloppy authors rarely get \
 published,\n but perfect authors never."""
```

とすると，

```
>>> print quote
Sloppy authors rarely get \
 published,\n but perfect authors never.
```

のように\や\n はそのまま表示される．これを raw 文字列と云ふ．

文字列演算子には連結 (結合) の+と反復の∗がある．

```
>>> s = "abra"
>>> t = "karabra"
>>> print s+t
abrakarabra
>>> print s*2
abraabara
>>> print 2*s
abraabara
```

文字列の定義により文字列は文字のシーケンスであるから，シーケンスとして処理できる．シーケンスとしての処理は，配列 (後述) の操作と同じである．

```
sq = "abcdefg"
```

という文字列が与えられたとする．q に添字 [n] を付けることによって，シーケンス上での位置を特定することができる．これによりシーケンスから部分的に要素 (つまり文字または部分文字列) を取り出すことができる．シーケンスから部分シーケンスを抽出することをスライスと呼ぶ．

```
>>> print sq[1]
b
```

この場合，添字はオフセットを示す．sq[1] は一番目の要素だから a だと思ってはいけない．一番目の要素は計算機では 0 番目である．[1] は二番目である．[-1] は後から一番目を示す．こういう表記をオフセット表記とふ．:を用いて，ある位置からある位置までを指定することもできる．

```
>>> print sq[-1]
g
>>> print sq[0:3]
abc
```

後者の 0 は最初を意味し，3 は三つめのオフセットを指定している．:で区切られているので，三番目の文字の手前までを指す．オフセットについては，下記を参照されたい．

オフセット	0	1	2	3	4	5	6	7
シーケンス	| a	| b	| c	| d	| e	| f	| g	|
逆オフセット	-7	-6	-5	-4	-3	-2	-1	

ちなみに

```
>>> sq[:0]
'abcdefg'
>>> print sq[:0]
abcdefg
>>> print sq[1:]
bcdefg              # 最初の一文字だけ除いた部分文字列
>>> print sq[:-1]
abcdef              # 末尾の一文字だけ除いた部分文字列
```

である．

自然言語 (つまり人間の通常用いる言語) では数字の最初 (数学で言う自然数) は 1 からはじまるが，計算機上では数字の最初は 0 からはじまる．これは奇異に見えるかもしれないが，人間が 1 から数えるのは，「ない」という状態が存在することを発見するより先に数字を発明したから (詳しくは吉田 (1955)) で，人間の歴史に制約されたモノの考え方である．また，自然言語でも British English では，日本語の 1 階を grand floor と言い，2 階を first floor, 3 階を second floor のように計算機と同じ数え方をするものがある．単純に始めだけの違いでしかないが，一から始めるか，零から始めるか，大いなる問題だ (足し算系では 1 ずれるだけで大した違いにはならないが，掛け算系ではシリアスが違いがある．1 は N 倍すれば N になるが 0 は N 倍しても 0 である)．

最後に注意すべきことは，スライスで文字列の一部を切り取れるが，それに代入することはできないことである．たとえば

```
>>> sq[2]
'c'
>>> sq[2]="g"  # Greek の第三文字は Γ であってローマ字でいう g だから (冗談).

Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object does not support item assignment
>>> sq[2]
'c'
```

のように，Python の文字列は部分文字列の変更を受けつけない．

文字 (character)

文字と文字列は異なる．文字列は文字のシーケンスである．C 言語では文字という型があって，それは 0–255 までの整数という意味であるが，Python には文字という型はない．Python では 1 文字は 1 文字からなる文字列である．Python で文字列といっているものは C 言語では文字配列である．

整数 (integer)

計算機上の整数は数学の整数と異なって上限がある．上限は普通，`MAXINT` という定数名をもち，32bit 計算機では $2^{(32-1)}$ の数 (約 20 億) が識別できるから，上限は 20 億である．上限を把握しておくことは重要である．上限を越える値が入力 (overflow) された時に，そのプログラムがどのような挙動をするか，これが問題なのである．Python では overflow flag が設けてあり，overflow が生じたら直ちにエラーで停止するが，そうではないプログラムでは overflow した後も，そのままプログラムが続行することもある．これはセキュリティホールになる以外にも，エラーがでないので，おかしいことに気がつかないまま計算結果を信用してしまう危険がある．

Python で定義される整数には `MAXINT` の上限がある整数 (C 言語の Long に相当) と，上限のない整数がある．前者は 56789，後者は 99999999999999999L のように記す．普通に算用数字で記述すれば 10 進法と解釈される．8 進数で示すためには, 015, 0132 のように先頭

に 0 をつければよい．ちなみに，015, 0132 はそれぞれ 10 進数の 13, 90 である，ASCII ではそれぞれ改行 (CR) と大文字の Z である．16 進数で表記するためには先頭に 0x をつければよい．改行の 015 は 0xd，Z は 0x5a となる．

0–127 までの ASCII の範囲内 (7bit) では `ord()` と `chr()` という函数が用意されている．`chr()` は，引数の数字 (10 進数であれ，8 進数であれ，16 進数であれ) を ASCII のコード番号を解釈してそれに対応する文字を返す．`ord()` は引数の文字 (一文字に限る) に対応する ASCII のコード番号を返す．基数を変換する函数も用意されていて，16 進数に変換する函数が `hex()`，8 進数が `oct()` である．Python では数は 10 進数で表記されるので，たとえ 16 進数で入力しても，表示される時は 10 進数である．`hex()` を用いて明示的に 16 進数に変換した数を 10 進数に戻すには，`int()`(整数にする函数) を用いる．`int()` は二番目の引数に基数を指定できるので，`int('0x5a',16)` のように基数を与えて変換する．

```
>>> ord('Z')
90
>>> chr(90)
'Z'
>>> chr(13)
'\r'
>>> chr(015)
'\r'
>>> chr(0xd)
'\r'
>>> x = 0xd
>>> print x
13                  # 10 進数で表記される．
>>> x
13                  # 中身も 10 進数に変っている．
>>> x = hex(x)      # 16 進数に変換
>>> print x
0xd
>>> x
'0xd'               # string の型で入っていることがわかる．
>>> x[1:2]          # だから，このように
'x'                 # スライスできる．
>>> int(x,16)       # 10 進数に再変換
13
>>> int(0xd,16)     # 値を直接入力すると... 型が違うのでエラーになる．
Traceback (most recent call last):
 File "<stdin>", line 1, in ?
TypeError: int() can't convert non-string with explicit base
>>> int('0xd',16)  # だから，string として quote してあげる必要がある．
13
```

算術演算子として「 `+ - * / % **`」が用いられる．それぞれの使い方は，中学校の数学で習ったとおりである．ただし,「`*`」は掛け算である．ASCII に「×」という記表はないた

めである.「/」は割り算，ASCII では「÷」という記号はないし,「$\frac{a}{b}$」という表記は困難であるため.「%」は剰余といって，整数の割り算において割った余りを指す．たとえば $\frac{5}{3}$ は 1 あまり 2 である．これを商を 1，剰余を 2 と云ふ.「5%3」の戻り値は 2 である.「**」は累乗である．x**2 は x の 2 乗を意味する．これらは，他のプログラム言語でも用いられている表記法である．Python の方針として演算子にしても函数にしても，わかりにくい記表を避けている．

Python ではビット演算子も使える．

```
>>> x = 1           # 1 = (0001)
>>> x << 2          # 2 bit 左にシフトする．
4                   # 4 = (0100)
>>> x | 2           # (0001) と (0010) の論理和
3                   # (0011) = 3
>>> x & 1           # (0001) と (0001) の論理積
1
>>> x & 3           # (0001) と (0011) の論理積
1
```

ビット演算の詳細については高林他 (2006); Warren (2006) を参照されたい．

浮動小数点 (floating point)

浮動小数点は数学でいう実数に相当するが，計算機上の浮動小数点数は実数とは大きく様相が異なる．数学では 3^6 のように仮数と指数に分解して数を表現する．浮動小数点では底は常に 10 である．内部ではもちろん底は 2 である．ここから来る齟齬は数学的 (あるいは計算機科学的) にはなかなか面白いが，特に注意が必要になることだけ述べる．同じ値となる数を比較しても等しくならない事態である．たとえば，4.0000... になるべき数が計算方法，あるいは計算の順番によって，4.00000...0001 とか，3.99999999.... となることがある．この二つを比較しても等しくないという結果になる．浮動小数点を比較する時は，ある許容範囲の閾値を設けて,「まる」めてから比較するか，あるいは両者の差をとって，その差が許容範囲以下であれば等しいと見做す，のように判別すべきである．

複素数 (complex)

物理を取り扱う向きにはうれしい複素数が定義されている．複素数は (2+4j) のように実数部と虚数部にわけてガウス表記をする．ただ数学と異なるのは，虚数単位を i ではなく j で示すことである．ラジカルな数学者のように $\sqrt{-1}$ のようには示さない (示せない)．物理では虚数単位を j で表記することは稀ではない．たとえば，電磁気学では i に電流の意味を与えることが通用していたため，虚数単位に i を避けて j としていた．計算機では i は整数の一時変数にしていることが多いので避けたのもかもしれない (数学でもそれは同じだが数学で i を使いづつけ，プログラム言語でそうではない理由は未解決)．

真偽値 (Boolean) true or false

真偽値は論理学でいう真理値である．真理値というのは，ある命題が正しい時は 1 であり，正しくない時は 0 になる値のことである．命題とはたとえば下記のような文のことで，

命題：アリストテレスはアレキサンダー大王の家庭教師であった．

真が偽かを可能性として決定しうる文のことである．上記の命題は歴史を紐解けば真偽を決定できる．この時の真偽を値にしたものが真理値である．真理値は命題の正しらしさの比率 (割合) と思えばよい．ただし，普通，命題は正しいか正しくないか 1 か 0 の二通りの離散値しかない (もし，0.3 とか 0.5 とか 0 から 1 の間の複数の値をとることができるのであれば，それは真理値というよりも確率である．真理値も確率も定義の根本は同じである.「それが真実である割合」である)．真理値は，真理値を論理学と数学に導入した Bool の功績を賛えて，Boolean とも呼ばれる．

Boolean の演算子は全て spell-out する．

```
(x=1) and (y=0)
(x=1) or  (y=0)
x==1
x!=1            # これは x<>1 とも表記できる．
not x
```

演算の順番

2+3*5+1 のような式は，先に掛け算の 3*5 が実行されてから残りの足し算が実行される．中学校で習ったことと思うが，プログラム言語でも同じで，演算子には，それぞれ強さが定義されていて強い順に演算される．乗法 ({*, /}) のほうが加法 ({+, -}) より強い．累乗 (**) は乗法よりも強い．演算の順番を変えたいときは (...) を用いればよい．2+3*(5+1) とすれば 5+1 が先に演算される．(...) を用いる時の注意は 2 つある．一つは (...) が対応していない (閉じていない括弧や閉じ括弧だけある場合) とエラーになるということ．もう一つは，括弧が多すぎる文章はそれがたとえ正しく対応していても人間には読みにくい．プログラムはコンパイラーだけが読むものではない．人間が書き，人間が読み，人間が訂正するものである．

2.4.2 コレクション

コレクションは基本的な型からなる変数の集合 (コレクション) である．

配列 (vector) とリスト

リストも配列も，項目が一続きになったものである．リストと配列の違いは，リストはサイズを後から変更できるが，配列はできないこと， 逆に，配列はインデックスがついているが，リストはインデックスがついていない．配列なら，A 配列の 3 番という風に，配列の中の項目を数字で呼び出せるが，リストはそうはいかない．ところが，Python のリストはインデックス化されていて，配列とリストの長所が統合された恰好になっている．そこで，リストと配列を区別せず，配列と呼ぶ．Python では配列を作成したり，配列の項目を参照

するには，[ ] を使用する．文字列の項目で少し説明したが，文字列は配列の一種なので，配列のインデックスのつきかたは文字列と同じく，先頭が 0 のオフセット表記である (ただし，配列はあとから要素の一部を変更できるが，文字列は一部だけの変更はできない)．

```
>>> DNA = []            # 空配列を生成
>>> DNA=["a","b","c"]   # 値を代入.
>>> print DNA
['a', 'b', 'c']
>>> print DNA[0]
a
>>> DNA[-1]             # print 文でない時，クオートされて出力される.
't'                     # つまり，データの入力された時の恰好になる.
```

配列の内容 (値) を変更するには新しく代入すればよい．また，配列に新しく要素を付加にするには，append() 演算子 (正確にはメソッド) を用いる．

```
>>> DNA[1]="g"
>>> DNA
['a', 'g', 'c']
>>> DNA.append("t")
>>> DNA
['a', 'g', 'c', 't']
```

配列は入れ子にすることもできる．

```
>>> Science =["math","phys","chim","biol","geol"]
>>> print Science
['math', 'phys', 'chim', 'biol', 'geol']
>>> Geology = ["petro","strati","volcan","geogra"]
>>> Science[4] = Geology
>>> print Science
['math', 'phys', 'chim', 'biol', ['petro', 'strati', 'volcan', 'geogra']]
```

配列の中に配列が格納されているのが [ ] の位置からわかる．入れ子になった配列の項目を参照するには [ ] を二回 (あるいは必要なだけ) 並べる．この方法をもちいて，行列 (や，もっと複雑な集合) を表すことができる．

```
>>> print Science[4][1]
Stratigra
>>> Eigen = [[1,0],[0,1]]   # 二元の正方行列の単位行列
```

配列の削除は del を用いる．結合は文字列と同じで+である．*を用いて同じ値をもつ項目が複数回繰り返す配列を作成できる．最後に，len() 函数により配列の長さを調べることができる．配列を読み込み逐次処理するまえに，配列の長さが知りたい時に重宝する．

```
>>> del Science[1]
>>> print Science
```

```
['math', 'chim', 'biol', ['Petrol', 'Stratigra', 'Volcanol', 'Geogra']]
>>> eigenvector = [1]*3
>>> print eigenvector
[1, 1, 1]
>>> len(eigenvector)
3
>>> len(Science)
4
>> len(Science[3])
4
```

タプル

タプルは変更のできないリストである．タプルに対して変更しようとすると，エラー (TypeError) によりプログラムが終了することで，タプルが破壊されることを防ぎ，プログラムに誤りがあることを，ユーザーに伝える．表記は，( ) で定義し，[ ] で参照する．

```
>>> Colour = ("Red","Grn","Blu")
>>> Colour[2]
'Blu'
```

ディクショナリ

他の言語ではハッシュということが多い．ディクショナリという名前からわかるように参照表のことである．たとえば，

```
netbsd.org    204.152.190.12
freebsd.org   69.147.83.40
openbsd.org   199.185.137.3
gnu.org       199.232.41.10
```

これは，DNA(Domein Name Server) の IP アドレスとドメインネームの対照表の見本であり，ディクショナリとしての機能を果す．ハッシュの作成 (初期化) と，代入，追加代入は下記のとおりである．

```
>>> DNS ={}                                          # 初期化
>>> print DNS
{}
>>> DNS ={"netbsd.org":"204.152.190.12",\
... "freebsd.org":"69.147.83.40",\
... "openbsd.org":"199.185.137.3"}                   # 一括代入
>>> print DNS
{'openbsd.org':'199.185.137.3','freebsd.org':'69.147.83.40',\#継続行
'netbsd.org':'204.152.190.12'}
>>> DNS["gnu.org"]="199.232.41.10"                   # 追加代入
```

```
>>> print DNS
{'openbsd.org':'199.185.137.3','freebsd.org':'69.147.83.40',\#継続行
'gnu.org':'199.232.41.10','netbsd.org':'204.152.190.12'}
```

肝心の，ハッシュを参照するには [ ] の中に自分の探したいものを入れるだけである．

```
>>> print DNS["netbsd.org"]
204.152.190.12
```

クラス (class)

クラスは Python で唯一定義できる型である．いいかえると Python におけるユーザー定義型は全てクラスである．たとえば，書誌データ型というのをここで定義する．書誌データというのは，下記のようなものである．

```
@book{kandr98,
   author="Brian Kernighan and Dennis Ritchie",
   title="The {C} programming langrage",
   edition=2,
   year=1988,
   publisher="Prentice Hall",
   address="New Jersey",
   pages=274,
}
```

これは BibTeX というアプリケーションのフォーマットで記述したある本 (K&R と称される C 言語の著名な本 (Kernighan and Ritchie, 1988)) の書誌データである．これを Python でのクラスとして再定義する．

```
class Book:
   def __init__(self,author,title,edition,year,publisher,pages)
      self.author = author
      self.title = title
      self.edition = edition
      self.year = year
      self.publisher = publisher
      self.address = address
      self.pages = pages
}
```

まづ，インデントで系統的に字下げされていることに注意されたい．インデントは伊達ではなく，Python は段落の区切りをインデントで理解する．そのため，インデントが揃っていないと文法的に間違った文とされる．さて，ここで用意したのは，データを格納する容れ物であってデータそのものではない．クラスというのは，型のことである (最初にもそう述べたが)．書誌データが入力されたものをインスタンス (実体) という．クラスは，もっと抽象的でどんなデータセットが必要か定義したものであり，もっと具体的にデータが入ったものがインスタンスである．クラスの中のそれぞれはオブジェクトといわれる．クラスの中のオブジェクトにアクセスするにはドット演算子「.」を利用する．

2.5 制御構文

型と演算がわかれば，次は制御構文だ．プログラムには繰り返しや，条件分岐やループなとの制御する構文が必要である．そうでないとプログラムは一直線の道しか走れないことになる．

2.5.1 if

if 文は条件分岐を制御する構文である．if で条件を提示し，条件に適う時の処理をブロックインデントされた範囲に記述する．else で，条件と違った時の挙動をブロックインデントで記述する．たとえば，

```
#! /usr/bin/env python
i = raw_input()
if i % 2 == 0:
        print "even number"
else:
        print "odd number"
```

というプログラムがあったとする．raw_input() 函数で，ユーザーから入力を促し，i に代入する．i を 2 で徐したものの剰余を求め，それが 0 なら偶数 (even), そうでないなら奇数 (odd) という判断をしている．==は比較演算子で，等しければ Boolean 1 を返し，そうでなければ 0 を返す (代入演算子=とは違うことに注目されたい)．一見うまくいっているように見えるが，このプログラムには落し穴があって，i が整数以外のものの場合でも else に落ちて,「奇数である」と云われてしまう．おかしな挙動をすることはエラーよりも始末がわるい．これはいろいろ修正方法があるが，たとえば下記のように修正できる．

```
#! /usr/bin/env python
i = raw_input()
if i % 2 == 0:
        print "even number"
if i % 2 == 1:
        print "odd number"
```

この場合，i が整数以外のものだと，なにもしない．なにもしないのもさみしいので,「整数ではありませんよ」というメッセージを出す文を付加する．

```
#! /usr/bin/env python
i = raw_input()
if i % 2 == 0:
        print "even number"
if i % 2 == 1:
        print "odd number"
else:
        print "not integer"
```

しかし，この文章は間違った動作をする．i が偶数の時，最初の `if i % 2 == 0:`で，“even number” と print される．ここまではよい．しかし，次の `if i % 2 ==1:`でも条件判断され，当然ながら偽となるので，else に落ちてしまい，“not integer” と print されてしまう．それを回避するためには `if` 文をネスト (入れ子) にすることができる．

```
#! /usr/bin/env python
i = raw_input()
if i % 2 == 0:
        print "even number"
else:
        if i % 2 == 1:
                print "odd number"
        else:
                print "not integer"
```

この文章では，２回目の `if` にかかるのは，一回目の `if` が偽の時に限るので，偶数がもう一度２回目の `if` にかかって，`else: print "not integer"`に陥いることを避けられる．しかし，条件分岐をネストすることの最大の問題点は，条件を重ねるごとにインデントが深くなり，可読性が落ちることである．

Python にはネストによるインデントの累重を避けるために，`elif`(=else if の略) という条件分岐構造が準備されている．これは，前件の条件に対して偽の時に，新たな条件を提示して，真偽を問うものである．

```
#! /usr/bin/env python
i = raw_input()
if i % 2 == 0:
        print "even number"
elif i % 2 == 1:
        print "odd number"
else:
        print "not integer"
```

これで随分とすっきりとした．条件構文の `if`, `elif`, `else` の三者ともインデントが同じ位置にある (全部インデントなし) ことに注目されたい．

複数の条件式は Bool 式を用いて結合できる．たとえば，

```
if i % 2 == 0:
        if i % 3 == 0 :
                print "6 の倍数です"
```

というコードがあったとしよう．これも複雑にネストしているが，２つの条件式を `and` 演算子で結合して，

```
if (i % 2 == 0) and (i % 3 == 0):
        print "6 の倍数です"
```

のように簡潔に記述できる．もちろん `or` 演算子も使える，たとえば，下記のようになる．

```
if (uname = "BSD") or (uname = "Unix"):
        print "Unix 使いなら自分で出来るでせう！"
else:
        print "本ソフトは対応しておりません．"
```

ここで uname は OS の名前を指すユーザー定義の変数とする．

or 演算子で注意しなければいけないのは，計算機上の or 演算子は数学の OR と異なり条件式を最後まで読まない．条件を判断できるところまで読む「短絡評価」という処理を行っている．これは Python だけに限らず，ほとんどのプログラム言語の実装である．たとえば，上の例なら，BSD 使いだと判明した時点で,「Unix 使いなら...」となる．

2.5.2 for

for ループは，インデックスとなる変数に逐次リストの要素を代入していく制御構文である．書くと難しいが，たとえば，前にこんなプログラムがあった．

```
#! /usr/bin/env python
message = "Hello, World!"
print message
print message
print message
```

これは for ループを使うと

```
#! /usr/bin/env python
message = "Hello, World!"
for i in range(1,4):
      print message
```

となる．range() 函数は，最初の引数から，[次の引数-1] までの数の列 (リスト) を返す．この例では，[1,2,3] である．このリストの要素が前から順番に i に代入され，インデントされた文において実行される．Python では制御構文のブロックはインデントで範囲づけられることを思い出してほしい．また，for の文末に:が必要なことも忘れないでほしい．これは次のインデントされたブロックが for の動作部であることを示す徴である．ブロックが複数の文ではなく，一つの文からなる時は，同じ行に記述してもよい (当然，インデントはなし)．

```
#! /usr/bin/env python
message = "Hello, World!"
for i in range(1,4): print message
```

この例では，全然インデックスの i を使用していないのでさみしいが，動作部には普通，i に関係づけられる処理を記述する．たとえば，ある数列の挙動を見たいとしよう．次の例は Fibonacci 数列と呼ばれる数列である．

```
#! /usr/bin/env python
old=0
new=1
for i in range(1,30):
        print "%2d %6d" % (i,old)
        old,new = new, new+old
```

この例では，何番目かと，i 番目の時の数列の値が示される．なお，桁数を揃えるために print 文と printf() の使い方をしている．

2.5.3 while

for ループは大変便利なのだが，欠点が一つある．それは，最初に代入する候補を全て列挙 (リストアップ！) してあげなければいけないことだ．for ループに入る前に，予めリストが決定すればよいが，いつループが終了するか，やってみないとわからないものもある．たとえば，ファイルから一行づつ読みとって処理するループの場合，いつ最後の行になるかは，ファイルを最後まで読みとらなければわからない．そのような時に重宝するのが while ループである．最初の文例は while ループでも処理できて，下記のようになる．

```
#! env python
message = "Hello, World!"
i = 1                       # i を定義・初期化する．
while i <= 3 :
      print message
      i = i + 1             # i をインクリメントすることを忘れずに
                            # (さもなくば無限ループ)
```

for 文と同様に,「:」とブロックインデントが必要なことに注意してほしい．

for 文と同じく，Fibonacci 数列を演算させてみる．次の while 文では，継続条件を値が 99999 以下としている．数列の値が 99999 の間は演算を続行し，99999 を越えると while 文が終了する．

```
#! /usr/bin/env python
old=0
new=1
i=1
while old < 99999 :
        print "%2d %6d" % (i,old)
        old,new = new, new+old
        i = i + 1
```

これは次のような書き方もできる．

```
#! /usr/bin/env python
old=0
new=1
```

```
i=1
while 1 :
        if old > 99999: break
        print "%2d %6d" % (i,old)
        old,new = new, new+old
        i = i + 1
```

while 1 は常に正しいから，これは無限ループとなる．それを break 文で中断させるのである．中断条件を if で指定している．

2.5.4 ブロックインデント

Python の特徴の一つがブロックインデント (インデントを用いてブロックを示す) である．他の言語ではブロックを示すのに，BEGIN...END や\begin{...}...\end{...} あるいは単に ... などのようにブロックのサインを置いている．サイン式のブロック表記法では，レイアウトに関係なく作文できる自由度がある反面，読みにくいコードも簡単に書けてしまう．ブロックインデントは，レイアウトを強制する点で，プログラマーの自由度を奪っているが，しかし，そのレイアウトはプログラムの論理構造を反映しているため，見通しのよい作文を可能にする利点がある．たとえば，たくさんの構文をネスト (入れ子) にすることが稀にある．たとえば，次のように．

```
for i in ....:
        print message
        ....
        if text == "...":
                ....
                while j => 255:
                        ....
                        print ....
                        if something :
                                ....
                                kakuno unzarisimasu
                                        if demo madamada tuduku...
                                                korede owari to omoikiya
                if totyude tobikyu sitarisite
print "mou unzari"
```

これは可読性に劣る．だから，Python はダメだということになるかもしれないが，ダメなのはコードの方である．ネストしすぎて画面からはみ出るか九十九折りになるようなコードは構造が間違っているから，最初から考えなおしたほうがよい，と Linus 氏も云っている．Python はフォーマットを強制することにより，複雑 (・怪奇) な文章を書くことを抑制している．なお Python のインデントについて幅の制約はない．たいていタブ一文字か，スペース 3〜6 文字が多いようである．原理的にはスペース一文字でも十分である．

2.6 函数

函数は，一連の手続きをセットにしておいて，あとから自由に呼び出して結果を得るための方法である．函数には引数を渡すことができ，外部からは数学の函数と同じように扱うことができる．

$$f(x)$$

x が引数である．引数の値をいろいろ変えることができ，それにより得られる結果 $f(x)$ も異なる．使い方はこれまでにモジュールの使い方のところで説明した．

どうして函数をいうものが必要なのか．それは第一に車輪を二回発明しないため，第二に，人間の思考の整理のためである．第一の理由は，同じことを何回も行う場合，その都度，それを記載するのはバカげている．手続きをまとめておいて，その都度それを呼び出したのほうが紙面 (行数？) の節約にもなるし，あとから別のプログラムで，その函数を呼び出すことができれば，別のプログラムで同じことを再開発するムダを節約できる．第二の理由は，作業のほとんどは—プログラムに限らず—いくつかのステップに分解できる．ステップに分解できるからといって必ずしも分解する必要はないが，ステップに分解した方が，その作業に対する理解が早まるし深まる．なぜなら，全部を一つして一度に捉えるのは大変だが，分割可能なステップに分解して，それぞれを理解してあとで総合するほうが理解がたやすいためである．また，作業全部を一枚岩として捉えるよりも，ステップに分割したほうが作業の隅々まで深く理解できる．

2.6.1 函数の例 Fibonaccian

例として Fibonacci 数列を函数として扱えるようにしたものをこれから示す．最初は，引数を越えない値を出力する函数である．

```
#! /usr/bin/env python
import sys
def fibonacci(n):
        """caliculate a Fibonaccian series upto argument"""
        old=0
        new=1
        i=1
        while old < int(n) :
                print "%3d %6d" % (i,old)
                old,new = new, new+old
                i = i + 1
arg=sys.argv[1]
fibonacci(arg)
print "Fibonaccian series upto",arg
```

次に，引数番目までの Fibonacci 数列を出力する函数である．

```
#! /usr/bin/env python
import sys
```

```
def fibonaccian(m):
        """caliculate a Fibonaccian series upto argument"""
        old=0
        new=1
        i=1
        while i <= int(m):
                print "%3d %6d" % (i,old)
                old,new = new, new+old
                i = i + 1
arg=sys.argv[1]
fibonaccian(arg)
print "Fibonaccian series for",arg
```

最初の def により函数の定義を与へる．まづ，函数の名前を定義し，() の中に，temporary な引数の名前を記す必要がある．引数は函数のブロックの中だけでしか通用しないもので，ブロックの外では未定義の変数となる．また，ブロックの最初には函数の説明文を文字列として与えることができる．これをドキュメント文字列と呼ぶ．

ドキュメント文字列は別に書かなくともよいが，二点から記述することをお勧めしたい．第一に，腕の立つプログラマーは—どの言語でも—その函数とは何かを函数の定義の近辺でコメントする．これは readability を向上させるの貢献している．Python のドキュメント文字列はこのよい実装となっており，よいプログラミングの習慣づけの点からお勧めしたい．第二に，ドキュメンテト文字列を抽出加工して，マニュアルや技術文書を作成するツールがある．これは Knuth・有澤 (1994) が文芸的プログラミングと提唱した考え方で，別途マニュアルを作成する手間をはぶけるメリットがある．プログラムが完成してからマニュアルを書くとマニュアルの内容がプログラムから遊離しがちである．また，マニュアルを書くために再度プログラムを読みなおすよりは，プログラムを書いている時にマニュアルを記述するほうが作業効率が高い．対話的にドキュメント文字列を出力されるには，

```
>>> def documant():
...       """This function is only for documentation string,
...       which is also called docstring."""
...       pass
>>> document.__doc__
This function is only for documentation string,
which is also called docstring.
```

のように，__doc__とする．このやり方はオブジェクト指向の設計によるものなので，説明は後述する．pass はなにもしないコマンドである．函数の中身が空だと都合がわるいので置いてある．

上記の Fibonaccian 数例の例では，函数を実行すると，print 文で数列を出力する．函数には戻り値という概念がある．

$$y = f(x)$$

の y が戻り値である．この場合の戻り値は None である．Python では戻り値が定義されていない場合は None になる．戻り値の指定には，return 文を用いる．上述の Fibonnaccian 数列を出力する函数でも数列そのものを戻り値にすることができる．

```
#! /usr/bin/env python
import sys
def fibonaccians(m):
        """caliculate a Fibonaccian series upto argument"""
        old=0
        new=1
        i=1
        fibo=[]
        while i <= int(m):
                fibo.append(old)
                old,new = new, new+old
                i = i + 1
        return fibo
arg=sys.argv[1]

i=1
for fibonac in fibonaccians(arg):
        print "%3d %6d" % (i,fibonac)
        i = i + 1
```

return 文で注意したいことは，return の後に何を書いても，コードは実行されない．このようなコードを dead code と呼ぶ．

2.6.2 函数のパワー 再帰函数

函数は手続きのまとまりを分離し，再利用可能な形態に整理するだけでも十分有用であるが，函数のパワーはそれだけではない．函数を再帰的 (自分で自分自身を呼び出すこと) により，通常ではとても長いあるいは記述不能な処理を実現することができる．再帰的 (recursive) 定義に似た論理学術語に循環的 (circular) 定義があるが，真実の循環はまったく役に立たない．

愚かとは，愚かものに対する形容詞である．

これでは愚かとはなにかについての情報はなにも得られない．

適切に判断します．なにが適切かは，適切に判断します．

これも同じで，なにが適切か，ちっともわからない．

業績がないから無能である．なぜなら無能でなければ業績がある筈だからである．

これは循環論法と呼ばれる論理的錯誤のひとつで，前提の正しさを証明するために，結論をもちだしている．これでは論証したことにはならない．
再帰的定義は，

定義のなかに，定義されるものを参照している定義

のことである．

the definition which contains a reference to the thing being defined.

再帰は循環を含むこともあるが，**役に立つ再帰的定義**はこれとは全く違って，定義により，定義前より問題が解決の方向に多少なりとも進んでいる．たとえば僅かな前進であっても，それを繰り返すことにより解決に至る場合が数学的問題にはよくある．例として，階乗 (連乗積，factorial) を示す．階乗は

$$n! = n(n-1)(n-2)...2 \cdot 1$$

のような数のことで，たとえば

$$7! = 7{\cdot}6{\cdot}5{\cdot}4{\cdot}3{\cdot}2{\cdot}1 = 5040$$

これを漸化式で書くと，

$$n! = n(n-1)!$$

となる．階乗の定義に階乗を使っている．だから，これは再帰的定義である．これを Python の函数の再帰的定義を利用して解いてみよう．最初に函数をモジュールの形で定義する．ファイル名を `factorial.py` とする．

`factorial.py` .....

```
def factorial(n):
        if n==1: return 1
        else:
               return n*factorial(n-1)
```

これを使用するには下記のようにする．

```
>>> import factorial
>>> print factorial(1)
1
>>> print factorial(2)
2
>>> print factorial(3)
6
>>> print factorial(4)
24
>>> print factorial(5)
120
>>> print factorial(6)
720
>>> print factorial(7)
5040
```

最初の `import factorial` でロードする．拡張子の `.py` は不要であることに注意されたい．コアの部分から説明する．`else` で `n` の時，`factorial(n-1)` を呼び出している．つまり，再度，この函数が呼び出されるわけである，ただし，`n-1` として呼び出されるので，少

しは問題が単純化した．`factorial(n-1)` は，`factorial(n-2)` を呼び出すわけである．こうして連鎖的に函数が呼び出され，いつか `n==1` の時になる．その時は最初の `if` 式で `n==1` に落ち，再帰的呼び出しが終了する．

2.6.3 Leap of Faith

現在のプログラムは複雑であるから，全てに目を通してそれぞれの動作が正しいことを確認していては日が暮れてしまう．自分が書いたプログラムであってもそうである．自分が書いたスクリプトは短かいから全て理解できるなどと思ってはならない．たとえば `print` がどうやって動いているか，諸君は知っているか?，たとえば，諸君はハードディスクの円板からどうやって情報を取り出しているか自分で作れる程度に知っているか？ CPU がどうやって演算をしているか諸君は知っているか？それを知らないで理解していると思ってはいけない．理解するというのは大変なことなのである．それは大変だから，**ある程度の確認作業が済んだらできているものと見做す**わけである．これを Leap of Faith と呼ぶ．Leap of Faith は哲学・究理学の概念であるから最初に科学理論 (特に物理法則) で説明する．たとえば万有引力の法則を全てのケースについてテストすることはできない．あっちのリンゴとこっちのリンゴは同じリンゴと云えば同じリンゴであるが，同時に違うリンゴでもある．それは読者諸子と私は同じ人間であるが，同時に違う人間であるのと同じである．あっちのリンゴで成り立つからといってこっちのリンゴで成り立つ保証はない．しかし，いちいちテストしていては時間が足りないし，学問が進歩しない．生きている時間は限られている．永遠に未来が続くのは若者だけだ．全てを検証する時間はない．そこで，あっちのリンゴもこっちのリンゴも同じものを見做すのである．

計算機も同じである．たとえば，上の Fibonacci 函数をいくつかのケースでテストしたら，その函数が「できた」ものと見做すわけである．これは科学理論と似ている．これを Leap of Faith と呼ぶ．これは少し奇妙である．Fibonacci 函数はとりあえずできたかもしれないが，まだ Fibonacci 函数を用いる本体のプログラムは完成していない．本体が完成していないのに,「できた」というのはどういうことだろう？それこそが Leap of Faith と呼ばれる理由である．

ここで，テストや確認作業をどうするかが重要である．自分に都合のよい引数だけ与へていては，テストにならない．しかし，実際にはありえない引数を与へて函数が動かないというのは過剰テストである．

しかし，たとえテストにパスしても，誰が御墨付を与へようとも，もし，プログラムが変な動作をしたならば，それは反証されたということである．もし，その原因を知りたければ，コードやそこから生成されたバイナリー，バイナリーが実行されるところと，どこまでも掘り下げていって追求する必要がある．たとえば，Python のスクリプトが変なところで，segmantation fault で落ちたとする．スクリプトだけ見ていてもどうしても原因がつきとめられない時は，Python の C 言語で記述されたコード自身を見なければいけない．そして，C 言語がどうやって演算しているのか，バイナリーにはどういう命令がコードされているか追求するのである．もちろん，それは大変手間がかかる上に，技術も要するので，必要に応じてである．学問的追求も大事だが，費用対効果や，十全の法則 (プログラムはその作業に対して十全に機能すれば完璧でなくてもよい．) を忘れてはいけない．

この例は科学理論で云へば，万有引力の法則の予言する結果を微妙に結果が異なる場合，例外とか特殊事情・観測機器の誤差でそうなったと無視することもできるが，それが深刻なものや無視できないものである場合，とことん掘り下げて追求する必要がある．前者はこれ

は費用対効果や十全の法則に従った方針で Leap of Faith である．後者はその Leap of Faith が破綻した後の方針で，学究的態度といえる．それにより，たとえば，重力により空間が歪むため，単純に万有引力は成り立たないということが明らかになったりする．

2.7 例文詳解

2.7.1 データの間引き

棒取りしているようなデータストリームでは，しばしばデータを間引く必要が生じる．たとえば，一秒に 1 回サンプリングされたデータがあるが，実際に解析に使用するのは，10 秒に 1 回でいいということがある[3]．データは一回につき一行づつ増えていくとしよう．これを 10 回に 1 回だけ抽出したいが，最初と最後のデータも抽出したい．その要求を満すスクリプトは下記のとおりである．

skip.py ....

```
#! /usr/bin/python
import fileinput
interval = 10

for line in fileinput.input():
   shot = fileinput.filelineno()
   if shot == 1 :
      print line,
   elif shot % interval  == 0:
      print line,
print line,
```

3 行目に interval 変数を設定して，skip する間隔を後から変えやすくしている．たとえば，間隔を 100 にしたければ interval に 100 を代入すればよい．中途半端な 256 という数字でも大丈夫だ．ただし，間引くかどうかの判定に割切れるかどうかを使っているので，interval は整数でなければいけない．

次に for ループで fileinput モジュールの input() メソッド (函数) を用いて line 変数に一行づつ行の内容を代入している．shot は行数を示す変数である．fileinput モジュールには filelineno() という函数が行数を示すので，それを代入している．スクリプトの 7 行目では，1 行目かどうかをテストし，1 行目ならは，行のラインをプリントする命令を与えている.line，と最後にコンマがつくのは，line 変数は改行を含めた行の内容が格納されているためである．print 文は改行を付加してプリントするので，print line とすると二重に改行がプリントされることになる．そこでコンマを置いて print 文の改行を抑制しているのである．

次に，インデックス番号が 100 増えるごとに間引くスクリプトを書いてみやう．ただし，データにはクセがある．単純なデータなら上のスクリプトの interval を 100 にすればいいが，下記のようなデータだとさふはいかない．

[3]では，なぜ 10 秒に 1 回サンプリングすることにしないのか，とふ疑問があるかもしれないが，最大限取れるデータは取りたいのが科学者の質であるし，他の測定との同期や兼ね合いで，解析に必要な精度以上細かく観測する必要があることがある．

GPS data .....

```
134.7313 33.155 -25
134.7256 33.168 35
134.7187 33.1833 125
134.7085 33.2163 275
134.7015 33.2512 425
134.6957 33.2788 545
```

この例は 1982 年に取得したある位置情報を GMT で用いる xy のデータにしたものである．位置データは電波測量 (ロラン) で取得された古い記録なので，データには抜けがある．しかも，インデックス番号は最初が 1 ではなく，負の数になっている．また，インデックスの値がとびとびになっていて，100 行送っても，インデックス番号は 100 ではない．どうすればいいか？ まず最初にデータを補完して，一行につきインデックスが 1 増えるデータに変換する．そして 100 行おきに間引くことになる．かなり複雑になる．今回はデータの補完だけすることにして，どうすればよいか考へてもらいたい (回答例は後の項目にある)．

2.7.2 データのフォーマットの変換

データの再フォーマットはしばしば必要になる．たとえば，Tiff 形式の画像ファイルをホームページに掲載するために Jpeg 形式に変換する，というのはよくある作業である．

もっとも簡単な変換

ここでは，もっとも簡単なテキストデータのフォーマットの変換を紹介する．もっとも簡単なフォーマットの変換は列の入れ換えである．第 1 列に入っているデータを 3 列目に，2 列目を 5 列目に... のように，列の順番を入れ換える作業である．次に簡単な変換の一つとして，列の区切りを変更する変換を紹介する．たとえば，タブ区切りのデータを固定長のスペース区切りに変更することである．また，列に操作を加へて新しい列を作成することも簡単な変換と云える．たとえば，度分で表示された緯度経度情報を度だけで示す変換や，その逆，あるいは華氏温度の摂氏温度への変換などである．

例として，度 (degree) で示された xy データを，固定長のデータに変換するスクリプトを示す．

```
#! /usr/bin/env python
import string
import sys
import fileinput
linename = sys.argv[1]
linename = string.replace(linename,".xy","")
basename = linename[0:4]
name = basename+"-"+linename
for line in fileinput.input():
    line=line.strip().split()
    shot = int(line[2])
```

```
lon = float(line[0])
londd = int(lon)
lonm = (lon-londd)*60
lonmm = int(lonm)
lons = (lonm - lonmm)*60
lonss = int(lons)
lonsss = int((lons-lonss)*100)
lat = float(line[1])
latdd = int(lat)
latm = (lat-latdd)*60
latmm = int(latm)
lats = (latm - latmm)*60
latss = int(lats)
latsss = int((lats-latss)*100)
print " %-17s %6d%02d%02d%02d.%02dN%03d%02d%02d.%02dE" %(
     name,shot,
     latdd,latmm,latss,latsss,
     londd,lonmm,lonss,lonsss)
```

以下，未稿

華氏 (Fahrenheit) は水の凝固点 (氷点) を 32 °F とし，沸点を 212 °F とし，間を 180 分割した温度である[4]．換算式は以下のようになる．

$$F = \frac{9}{5}C + 32$$

新しい情報をデータに含める場合

やや，ややこしい変換にファイル名をデータセットに付加する変換や，別に用意したリストから条件を決めてどれか選んでデータセットに付加する変換がある．前者の例として，複数の測線の xy データをダイジェストして 1 ファイルにしたい場合がある．後者の例として，住所録から住所から郵便番号をリストから選択して，データセットに付加する作業がある．

前者の例．

```
#! /usr/bin/env python
#  per 10 shots discount
import fileinput
import sys, re
for line in fileinput.input():
        line=line.strip().split()
        shot =fileinput.filelineno()
        lon = float(line[0])
        lat = float(line[1])
```

[4] もともとの Fahrenheit の発想は人間の標準体温を 100 度，当時人工的に作り出せる最低温度 (氷に食塩を掛けてエントロピー増加により熱を奪ったもの-17.8°C) を零度とする温度系であったが，精度の問題で微妙にずれてしまった．現在の華氏では華氏 100 度は摂氏 40 度である．摂氏 40 度は標準体温とは云へない．

```
            name = sys.argv[1]
            name = re.sub(r'\.xy','',name)
            printform=str(name)+","+str(shot)+","+str(lat)+","+str(lon)
            if  shot == 1 :
                    print printform
            if float(shot) /10 - (shot/10) == 0:
                    print printform
    #       print printform
```

後者の例.

ディクショナリ (ハッシュ) を使う．住所を引数とすれば，郵便番号が戻り値となるようなディクショナリ (文字通りディクショナリ) を作ればよい.

未稿

更に複雑な変換

Multi-narrow beam echosounder(MBES) のデータストリームは ASCII であるが，これまで述べたデータ構造と比べて格段に複雑な構造をしている.

未稿

Binary ファイルの変換

MBES のデータ構造は ASCII の体裁をしているものの，一般的な Binary データの構造に酷似している．MBES で学んだことを生かして Binary file の加工に挑戦する.

未稿

2.7.3 時間のフォーマットの取扱い

時間はやっかいだ[5]．時間は 10 進法ではない．まず，60 進法がある．60 秒が 1 分，60 分が 1 時間．しかし，24 時間で 1 日だから，これは 24 進法だ．だいたい 30 日で 1ヶ月だから，ここは 30 進法と云えなくもないが，28 日の月もあるし，31 日の月もある．定数ではない．では，日から直接年を出そう．この考え型を通年日といい，一年の最初からの通算日で，日を表現する．一年は 365 日だから，365 進法といっていいだろうか．これくらいならまだなんとかなりそうである．しかし，そうは問屋が卸さない．閏年といって，一日余分がある年が四年に 1 回ある．しかも，100 年に 1 回 (100 の倍数の年) は閏年である筈の年が閏年でなくなる... かつ，その閏年であるはづだが 100 年に 1 回の閏年ではなくなる年の 400 年に 1 回 (400 の倍数の年) は，閏年になる．これを計算機で処理させるには... 頭いたくなってきた[6].

ここで登場するのが time() 函数である．これは Unix の C の標準ライブラリの一つで，上記のややこしい計算や表示形式の変更などを司ってくれる．Python では，datetime モジュー

[5]旧暦 (太陰暦) と新暦 (太陽暦) の違い．ユリウス歴とグレゴリウス歴の換算など，時間はとてもやっかいだ．これらは本稿では取り扱わない.

[6]これを簡単に決めるには，まづ，その年が 400 で割り切れるかどうか見る．割り切れる (2000 年など) なら，閏年である．割り切れない場合，100 の倍数であれば閏年ではない．100 の倍数でなくて，4 の倍数であれば，閏年である.

ルというものがあって，それが綜合的に時間に関する操作 (表示形式の変換を含む) を受けもっている．しかし，ここでは自分たちが使う時間の形式変換を自分たちの手で実装しよう[7]．

最初に，計算機はどのように時間を処理しているか理解する必要がある．Unix の time() 函数は内部では，あるエポックからの通算秒で処理している．

```
1972/12/24/20:45:12
```

を全秒に変換するプログラムと，逆に与えられた全秒表記を上記のような表記に変換するプログラムがあると，いろいろと便利だ．

未稿

2.7.4

この例は，ある positioning データを加工して，GMT の track ファイルに変換するプログラムである．まづ，track file は

```
2006 6967 12-Sept
2006/09/12/09:22:08 41.85026603 142.807218358 NaN NaN NaN 1
2006/09/12/09:22:14 41.850148225 142.807194888 NaN NaN NaN 2
2006/09/12/09:22:20 41.850030615 142.80716821 NaN NaN NaN 3
2006/09/12/09:22:26 41.8499192517 142.807139283 NaN NaN NaN 4
2006/09/12/09:22:32 41.8498005933 142.807110375 NaN NaN NaN 5
2006/09/12/09:22:38 41.8496752567 142.80709425 NaN NaN NaN 6
2006/09/12/09:22:44 41.8495616883 142.807060533 NaN NaN NaN 7
```

のようになっている．一行目がヘッダーに充当されており，航海年，データの行数 (ファイルの行数ではないことに注意)，トラック名 (この例では英国式の月日で示している) がスペース区切で記入されている．2 行目以降がデータ領域で，最初のブロックは年月日時分秒，緯度，経度，重力 (この例ではデータがない [NaN] が，mgal で記述)，磁力 (この例では NaN だが，mT で記述)，水深 (この例では NaN だが m で記述)，インデックス番号 (これは GMT の定義にはない．音波探査独特の仕様) が含まれている．

一方，今回の positioning データは米国 Geometrics 社の CNT という探鉱器で収録される GPS データである．そのフォーマットは

```
1, Tuesday, September 12, 2006 at \
09:22:08, 514105, 150660, $GPGGA,092208.00,\
4151.0159618,N,14248.4331015,E,2,08,1.1,-1.78,M,32.40,M,4.0,0678*6A
2, Tuesday, September 12, 2006 at \
09:22:14, 514105, 150660, $GPGGA,092214.00,\
4151.0088935,N,14248.4316933,E,2,09,1.0,-1.29,M,32.40,M,4.0,0678*66
3, Tuesday, September 12, 2006 at \
```

[7]使える道具があるのだから，わざわざ作らず，既存のツールを使うのは常道ではあるが，今のご時世，たいていのツールは揃っているのだから，既存ツールの再利用の論理を押し進めていくと，たいていのものは自分で作らなくてもよくて，人のを使いまわせばよいことになる．しかし，いつかは誰も作ったことのないものが必要になる．その時に，全て既存のツールを使っていた者は，いきなり高い壁の前に立たされる．

```
09:22:20, 514105, 150660, $GPGGA,092220.00,\
4151.0018369,N,14248.4300926,E,2,09,1.0,-2.05,M,32.40,M,2.8,0678*6B
4, Tuesday, September 12, 2006 at \
09:22:26, 514105, 150659, $GPGGA,092226.00,\
4150.9951551,N,14248.4283570,E,2,09,1.0,-1.69,M,32.40,M,4.0,0678*6E
5, Tuesday, September 12, 2006 at \
09:22:32, 514105, 150659, $GPGGA,092232.00,\
4150.9880356,N,14248.4266225,E,2,09,1.0,-1.64,M,32.40,M,4.0,0678*66
```

となっている．ただし，行が長いため継続行を\で示している．これは非常に単純で，ショット番号 (インデックス番号) やアメリカ式の日時を並べ，GPS のシリアルセンテンスが張り付いているだけである (この例では GPGGA 形式だが受信機によって入力されるセンテンスの形式は異なる)．

これを track file に変換するスクリプトは下記のとおりである．

```
#! /usr/local/bin/python
import fileinput
import re

year     = 1973
month    = ""
day      = ""
daymonth = ""
lines    = 0
hms      = ""

monthc2n ={
    'January':'01',
    'February':'02',
    'March':'03',
    'April':'04',
    'May':'05',
    'June':'06',
    'July':'07',
    'August':'08',
    'September':'09',
    'October':'10',
    'November':'11',
    'December':'12'
    }

for line in fileinput.input():
    if lines == 1:
        line=line.strip().replace(","," ").split()
```

```
        year=str(line[4])
        month=str(line[2])
        day=str(line[3])
#       daymonth=day+"-"+month[0:4]
        daymonth=monthc2n[month]+day
    lines = lines + 1
print year,lines,daymonth

for line in fileinput.input():
    line = line.strip().replace(","," ").split()
        line[13] = float(line[13])
        line[11] = float(line[11])
        lon = int(line[13]/100)+(line[13]-100*int(line[13]/100))/60
        lat = int(line[11]/100)+(line[11]-100*int(line[11]/100))/60
        #shot = line[0]
        shot = fileinput.filelineno()
        year = str(line[4])
        month = monthc2n[line[2]]
        day = str(line[3])
        hms = str(line[6])
        print year+"/"+month+"/"+day+"/"+hms,lat,lon,"NaN NaN NaN",shot
```

注意しなければいけないことは，入力される GPS がセンテンスが異なれば，スクリプトも改造しなければいけないことである．

情況とは常に変化し得るものである．メンテナンスできなればソフトは動かないも同然である (Weinberg, 1979).

2.7.5 行構造をもつデータの補間

飛び飛びにしかないデータを繋いで連続的なデータセットにしたい時はよくある．たとえば，不定期に飛び飛びなデータから一定間隔のデータを作成する場合には，一旦，連続的なデータにしてから一定間隔ごとに間引く方法が考へられる．今回は，飛び飛びしかないデータを補完する方法を紹介する．例として前述した GPS データをとりあげる．

GPS data .....

```
134.7313 33.155 -25
134.7256 33.168 35
134.7187 33.1833 125
134.7085 33.2163 275
134.7015 33.2512 425
134.6957 33.2788 545
```

これをインデックス番号が 1 づつ増えるようなデータに変換するためのスクリプトは下記のとおりである．

```
#! /usr/pkg/bin/python
import fileinput
counter = 0
for linea in fileinput.input():
    linea = linea.strip().split()
    if fileinput.filelineno() == 1:
        lineb = linea
        prln = linea
        if int(prln[2]) > 0:
            print prln[0],prln[1],prln[2]
    if lineb != linea :
        counter = 0
        diffshot = int(linea[2]) - int(lineb[2])
        diffshot = int(abs(diffshot) -1)
        diffx = float(linea[0]) - float(lineb[0])
        diffy = float(linea[1]) - float(lineb[1])
        while counter < diffshot:
            counter = counter + 1
            linec[0] = float(lineb[0]) + diffx * counter / diffshot
            linec[1] = float(lineb[1]) + diffy * counter / diffshot
            linec[2] = int(lineb[2]) + counter
            prln = linec
            print prln[0],prln[1],prln[2]
            prln = linea
            if int(prln[2]) > 0:
                print prln[0],prln[1],prln[2]
            lineb = linea
```

このスクリプトでは，インデックスが 0 より大きくないと出力しないことに読者はお気づきだろうか？

2.7.6 英文 readability の判定プログラム

動機・目的

英語の読みやすさは，我々のような非英語母語者だけでなく，英語話者にとっても重要である．英語は柔軟な言語であるがためにかえって奇妙な文章が容易に生み出されてしまう．英語の読みやすさ (readability) を定量化するプログラムがあれば，よい作文に貢献するに違いない．

背景調査

readability にはいくつかの index(指標) がある．たとえば，

文に対する単語の数 (つまり，文の長さ) 長い文は読みにくいが，短かい文は読みやすい (ただし，あまり短文ばかりだと‘ぶっきらぼう’あるいは幼稚な印象を与へたり，コンテキストが断片化して思考の流れを防げる弊害がある)．英語では屡，wps(words per sentence), ASL(average sentence length) で測られる．一般的には A4, 10.5pt 程度の原稿で 3 行以上になる文は長文とされる．単語の長さによって 3 行になる単語数は異なるが，だいたい 30–40 語を目安として，長文かそうでないかと判定することが多い．

段落を構成する文の数 (つまり，段落の長さ) 概念としては文の長さと同じ．延々と続く段落は読みにくい．ドイツ語で 3 ページずっと同じ段落の文章を読んだことがあるが，大変読みにくかった．英語では近代英語まではそういうスタイルもあったが，現代英語では許容されない (ドイツ語では許されるらしい)．一般的には段落は 3–5 文程度で構成されるのが望ましいとされる．10 文を越えると長すぎと診断されることが多い．

シラブル (音節) の数 (量比) 日本語では文の長さを文字数で勘定する．伝統的には原稿用紙 400 字何枚という風に原稿の長さを文字数で勘定してきた．しかし，英語では文字数よりも単語数で文の長さを規定する．単語数で規定する場合には長い単語も短かい単語も同じ一単語として勘定してしまうために単語数が少ないけれど，各単語が長いために読みにくい文というのがありえる．また，アルファベットは不完全な表音文字で一字一音ではないために文字数で単語の長さを勘定すると音韻としての単語の長さと齟齬が生じる (たとえば t は 1 字，th は 2 字だが，音韻としては [t] と [θ] とどちらも一音)．そのため，文・単語の長さを正確に定量化するためには音節の数を基準にする必要がある．英語では, ASW(average syllables per words) で示す．

抽象語の数 (量比) 抽象語は理解しにくい．抽象語とは見たり触ったりできないもの全てを指す．ただし，言語そのものが抽象性をもっているので，なにが抽象語かを巡って，しばしば線引問題になる．実際，抽象語の定義は分野依存性が強いので，分野ごとに抽象語のリストを用意する必要がある．抽象語の量比をパーセントで示したものは，Cohen’s Cloudiness Count とも呼ばれる (Mannuals, 1984)．抽象語は煙に巻くので cloudiness である．科学論文では抽象語は 3–6%に留めることが望ましい．

受動態の数 (量比) 受動態の文章は理解しにくい．

否定語の数 (量比) 否定語の多い文章は理解しにくい．二重否定，三重否定などは，科学および非形式論理学の文章においては可能な限り避けなければいけない．

二次統計量・指数 これは上記一次統計量から算術的に導かれる指数で，各要素を綜合的に (ある重みづけをつけて=恣意的に) 判断して，readability を人間にわかりやすく表現したものである．代表的なものに Flesch Grade Level/Flesch Score(Flesch, 1976) や fog index(Gunning, 1973) と呼ばれる指数がある．たとえば，Flesch Score では，科学論文は 40–50 が適当で，70 以上だと童話なみの読みやすさとされる．人の論文の要旨をいくつか代入してみたが，だいたい 20–30 が多い．それでもそんなに難しく感じないので 40–50 は努力目標だろう．また，Flesch Grade Level では文章の可読性を米国の学校の学年で表現する．Level に 6 を足せば年齢になる．一般的な文章では 8 が適当とされる．科学論文は 12 程度が望ましいとあるが，大学院生程度=18 までは許されるであらう．いろいろ文章を代入してみると，Level が 30 を越えるものも結構多い．

視点の一貫性 視点がコロコロ変わる文章が読みにくい．

実装

与へられたテキスト (ASCII) から，上記の index を計算して，readability を報告するプログラムを作成した．ただし，視点の一貫性を判定するプログラムは，統語論解析が必要で，この本の手にあまるので取りあつかわない．syllable の認定も，英語音韻学の知識が必要だが，これは統語論よりは簡単なので，本稿で取りあつかう．二次統計量は一次統計量から自動的に算定される．

TEX のソースファイルは `dvi2tty` プログラムを用いて，ASCII に変換する．この手法は TEX 固有の命令，特に数式が多いと，判定プログラムがうまく機能しないことが予想されるが，数式が多い文章は概して読みやすい (数式の意味がわかるならば，の話)．一方，TEX のソースから，TEX 命令を適当な ASCII に置きかえる函数を容易するのは面倒であるし，それは dvi2tty のすべき仕事と考えて，現時点では TEX のソースには `dvi2tty` filtering で対応する[8]．

私は，次に述べるプログラムを用いて，論文の readability を各 subsection ごとに判定し，あまりよいスコアでないところは訂正してから投稿している．

スクリプト

行数が多すぎるため，またまた推敲を経ていないスパゲッティコードなので未掲載．
未稿

2.7.7 再帰

未稿

```
      1
     1 1
    1 2 1
   1 3 3 1
  1 4 6 4 1
```

$$X(n)[i] = X(n-1)[i] + X(n-1)[i-1]$$

[8] TEX のコードを取り除くスクリプトも凝らなければたいした作文ではないので本文を書き足すことがあれば，取りあげたい．

第3章　オブジェクティブ

未稿

オブジェクトのパワー

例として下記のようなクラスを定義する.

```
class Uisge:
        """ This is a class of whiskies."""
        def __init__(self,
                Name, Distill, Land, Age="NaN",
                Price="NaN",Type="Blended", Note="NaN"):
                self.Name = Name
                self.Distill = Distill
                self.Age = Age
                self.Land = Land
                self.Price = Price
                self.Type = Type
                self.Note = Note
        def printHTML(self):
                print "<dt>"+self.Name+"</dt>"+\
                "<dd>"+\
                self.Distill+\
                "; Age: "+self.Age+"; "+self.Type+"; "+self.Land,\
                "</dd>"
```

これはウヰスキーのデータベースの定義であるが，データを定義しているのは前半だけで，後半はテータの定義ではない．データの出力方法 (函数) の定義である.

これに,

```
claymore=Uisge(
        Name="Claymore",
        Distill="Claymore",
        Land="Scots",
        )

cragganmore12=Uisge(
        Name="Cragganmore",
        Distill="Cragganmore",
        Land="Scots",
        Type="Single Malt",
```

```
        Age="12",
        )
royal_salute=Uisge(
        Name="Royal Satute",
        Distill="Chivas Bros.",
        Land="Scots",
        )
```

のようなデータを与へる．ここまでを uisge.py として，一つのファイルにしておく．そして，下記のように，uisge を import してやり，クラス uisge のインスタンス claymore に対して，クラスで定義された printHTML メソッド (函数) を演算子で結合してあげると，

```
>>> import uisge
>>> uisge.claymore.printHTML()
<dt>Claymore</dt><dd>Claymore; Age: NaN; Blended; Scots </dd>
```

となる．これは，データを HTML として表現するためにフォーマットしなおした結果である．それぞれのインスタンスに対して順番にメソッドを実行してやれば，全てのデータの HTML 用フォーマットが完成する．

このようにオブジェクトを使用することにより，データとメソッドを別々ではなく，統一した形式で管理できるので，たいへん楽である．この例でオブジェクトを使用しない場合は，データベースとそれを加工する函数を別途作成しなければいけない．この例は単純だから有難味が乏しいが，クラスの中にサブクラスをつくったり，クラスをまとめてひとつのスーパークラスにする (サブクラスと同じことだが) 時にクラスの真価が発揮される．データに対応したメソッドがデータと一緒についていってくれるのか，それとも別々に管理しなければいけないのか，で管理コストは大きく異なる．

3.0.8 継承

第4章　おわりに — 比較言語学の勧め

同じ言語を使っていると，その言語特有の世界観があたかも正しいもののように自分の身についてしまいがちである．しかし，そのような考えは間違っている．

もし，比較言語学の対象に新たな言語に挑戦するというのであれば，Java や C++ よりも Lisp や Prolog のような C 言語由来ではない言語をお勧めする．その故は，C 言語由来の言語は，どれも同じような文法構造をしているので，あまり言語による世界観の違いが浮き彫りになることはないためである．それでも，世界観の違いはあって，たとえば perl と python は同じく C 言語由来の兄弟関係にある言語 (perl が兄貴分) だが，コンテキストについての世界観がまるで逆なところ，自由で messy な構文と strict で統一的な構文と，いろいろ世界観の違いが楽しめる．

しかし，しばしば，perl vs. python vs. ruby vs. PHP—のように，どれが優れた言語か論争 (意味のない論争をしばしば flame と呼ぶ) が勃発する．このような言語論争は往々にして belief の世界つまり宗教戦争になりがちである．一例として，goto 文批判をあげる．Goto 文は制御構文の一つで，ループ状構造から抜ける際に行き先を明示して (to)，そこに行って (go) もらう命令である．Goto 文は，しばしばコードの流れを乱し制御をあらぬところに飛すので，コードをこんがらがったスパゲッティにしがちである．そこで「goto 文はよくない，goto 文を使わないでおきませう」という運動が盛んになった．東大の Professer Goto が,「私はいつもどこでもみんなに批判されるんですよ (笑)」と Don Knuth が書いていた Knuth・有澤 (1994)．ここに goto 文批判が belief の世界に属することが垣間見れる．なぜかというと，science, logic and argument の世界では，ものごとを述べるのに理由付け reasoning が必要である．goto 文批判は初期・提言者の意図とは違って，いつでもどこでも goto 文を見れば無条件で「時代遅れだ，コードをわかっていない」と批判し，その goto 文がほんとうに必要なものかどうかは吟味しない．ここには reasoning はない (権威による reasoning はあるかもしれないが) ので，scietific argument ではなく，belief の世界に属すると云える．言語論争はしばしば上記の goto 文批判の様相を呈す．それは，よい/わるいと好き/嫌いと，ある目的に適している/適していないの論点を混同しているためである．

関連図書

Aho, A. V., B. W. Kernighan, and P. J. Weinberger (1989) *The AWK Programing Language*: Addison-Wesley. 邦訳 プログラミング言語ＡＷＫ足立高徳 凸版印刷 (後 CMC 出版が版 権を引継ぐ) 1989(2001), ISBN:4901280406.

アンク (2002) 『C の絵本—C 言語が好きになる 9 つの扉』，翔泳社，191 頁．ISBN: 4798101036.

Arthur, Lowell Jay and Ted Burns (1997) *Unix Shell Programming*, New York: John Willey and Sons, 4th edition, pp.518. ISBN: 0471168947, 邦訳: UNIX シェルプログラミグ, 伊藤・千吉良・西尾・宮下 (訳), オーム社, 1993, ISBN=4274077705.

Bentley, Jon Louis (1991) *More Programming Pearls*: Addion-Wesley(Pearson Education). 邦訳 プログラマのうちあけ話続・プログラム設計の着想．野下浩平・ 古郡延治 近代科学社 ISBN: 4764901773, 256p. 1991.

Bentley, Jon (2000) *Programming Pearls*, Boston: Addion-Wesley(Pearson Education), 2nd edition, pp.239. ISBN: 0201657880, 1st ed in 1986, 邦訳 珠玉のプログラミング本質を見抜いたアルゴリズムとデータ構造 小林 健一郎 (訳) ピアソンエデュケーション, 2000, ISBN: 4894712369, 308p.

Bowney, Allen, Jeffrey Elkner, and Chris Meyers (2002) *How to think like a Computer Scientist—Learning with Python*, Wellesley, Massachusetts: Green Tea Press, pp.278. ISBN: 0971677506, on-line document at http://www.thinkpyton.com/.

Browm, Martin C. (2002) *Perl to Python Migration*: Person Education. Perl ユーザーのための Python 移行ガイド，細谷 昭訳 2002, ピアソンエデュケーション.

Dougherty, Dale and A. Robbins (1997) *sed & awk programming*: O'Reilly, 2nd edition. ISBN: 1565922255, 邦訳 sed & awk プログラミング (改訂版)，福崎俊博 (訳)， 1997, ISBN:4900900583.

Eckel, Bruce (2001) *Thinking in Python—Design patterns and problem-solving techniques*: in editing. on-line document.

Flesch, R. (1976) *The Art of Readable Writing*, New York: Hungry Minds.

深沢千尋 (1999) 『すぐわかる Perl Software Technology 16』，技術評論社．ISBN: 4774108170.

Gancarz, Mike (1995) *The Unix Philosophy*: Digital Press, pp.151. ISBN: 1555581234, UNIX という考え方–その設計思想と哲学, 芳尾 桂 (訳), オーム社, ISBN=4274064069, 2001, 148p.

Gould, Alan・松葉素子 (訳) (2001) 『Python で学ぶプログラム作法』，ピアソン・エデュケーション．ISBN: 4894714019，原題: Learn to program using Python: A tutorial for hobbyist, self-strarters, and all who want to learn the art of computer programming.

Gunning, R. (1973) *The Technique of Clear Writing*, New York: McGraw-Hill, revised edition.

原田賢一 (1986) 『Fortran77 プログラミング』，サイエンス社．ISBN: 47819046110.

Hawley, David and Raina Hawley (2004) *Excel Hacks*: O'Reilly, pp.304. ISBN: 059600625X, 邦訳 羽山 博・日向 あおい，320p, ISBN: 4873112052, 2004.

Kernighan, Brian W. and Rob Pike (1984) *The Unix programming envirionment*: Prentice-Hall, Inc. 邦訳 Unix プログラミング環境, 石田晴久 (訳), ASCII, 1985, ISBN: 4871483517.

Kernighan, Brian W. and Rob Pike (1999) *The Practice of Programming–Simplicity, Clarity, and Generality*: Addison-Wesley, pp.267. 福崎 俊博訳 プログラミング作法 2000 ASCII ISBN:4756136494.

Kernighan, Brian W. and P. J. Plauger (1974) *The Elements of Programming Style*: Bell Telephone Laboratories, 2nd edition. 木村 泉訳プログラミング書法 共立出版 ISBN: 4320020855.

Kernighan, Brian and Dennis Ritchie (1988) *The C programming langrage*, New Jersey: Prentice Hall, 2nd edition, pp.274. 邦訳プログラミング言語C 第二版 ANSI 規格準拠 共立出版.

Kernighan, Brian W.・P. J.Plauger・木村泉 (訳) (1981) 『ソフトウェア作法』，共立出版．ISBN: 4320021428，原題 Software Tools, Bell Telephone Laboratories, 1976,.

Knuth, Donald E.・有澤誠 (訳) (1994)『文芸的プログラミング』，ASCII．ISBN: 4756101909，原題: Literate Programming 1991.

Lutz, Mark (1996) *Programming Python*: O'Reilly, pp.881. 新版および邦訳あり.

Lutz, Mark and Davia Ascher (1998) *Learning Python*: O'Reilly. 初めての Python 紀太 章訳 2000 オライリー.

Mannuals, Creating Technical (1984) *Cohen, G. and Cunningham, D. H.*, New York: McGraw-Hill.

Mertz, David (2003) *Text processing in Python*, Boston: Addison-Wesley. ISBN: 0321112547.

椋田 實 (1993) 『はじめての C [ANSI C 対応] Software Technology 2』，技術評論社，第 3 版，343 頁．ISBN: 4874085466.

Pilgrim, Mark (2004) *Dive into Python*: on-line document, pp.322. at http://diveintopython.org/.

Raymond, Eris S. (2003) *The Art of UNIX Programming*: Addison-Wesley.

Raymond, Eric S. (2005) “How To Become A Hacker (Revision 1.31)”, *url= http: //www.catb.org/ ˜esr/faqs/ hacker-howto.html*, p. 0. 邦訳： url= http://cruel.org/freeware/hacker.html.

Schwartz, Randal L. and Tom Christiansen (1997) *Learning Perl (2nd ed)*: O’Reilly. 初めての Perl 第二版 近藤嘉雪訳 (第三版もあり) 1998 オライリー.

砂原秀樹・石井秀治・植原啓介・林 周志 (1996) 『プロフェッショナルシェルプログラミング』, ASCII.

砂原秀樹・石井秀治・植原啓介・林 周志 (2001) 『プロフェッショナル BSD (改訂版)』, ASCII.

Sussman, Julie (1998) *Structure and interpretation of computer programs, instructor’s manual to accompany*, Cambridge: MIT Press, 2nd edition. ISBN: 0262692201.

高林 哲・鵜飼文敏・佐藤祐介・浜地慎一郎・首藤一幸 (2006) 『Binary Hacks ハッカー秘伝のテクニック 100 選』, O’Reilly, 412 頁. ISBN: 4873112885.

Wall, Larry, Tom Christiansen, and Jon Orwant (2000) *Programming Perl*: O’Reilly, 3rd edition, pp.1067. ISBN: 0596000278, 邦訳プログラミング Perl Vol.1,2 近藤嘉雪訳, ISBN: 4873110963,4873110971.

Warren, Henry S. (2006) *Hackers’ Delight*: Addison-Wesley Pub., pp.368. ISBN: 0201914654, 邦訳 ハッカーのたのしみ本物のプログラマはいかにして問題を解くか. 滝沢 徹他訳. SliB Access. ISBN: 443404683.

山本和彦 (2000) 『リスト遊び—Emacs で学ぶ Lisp の世界』, ASCII, 122 頁. ISBN: 4756134424.

矢沢久雄・日経ソフトウェア (監修) (2001) 『プログラムはなぜ動くのか』, 日経 BP 社, 294 頁. ISBN: 4822281019, How program works.

吉田洋一 (1955) 『零の発見』, 岩波新書.

結城 浩 (1995) 『ANSI 対応 C 言語プログラミングレッスン文法編』, ソフトバンク, 第 2 版. ISBN: 4890527559.

結城 浩 (1998) 『ANSI 対応 C 言語プログラミングレッスン入門編』, ソフトバンク, 第 2 版. ISBN: 4797307579.

結城 浩 (1999a) 『Java 言語プログラミングレッスン (上)Java 言語を始めよう』, ソフトバンク. ISBN: 4797308036.

結城 浩 (1999b) 『Java 言語プログラミングレッスン (下) オブジェクト指向を始めよう』, ソフトバンク. ISBN: 4797310103.

斎藤 靖・小山裕司・前田 薫・布施有人 (1996) 『新 Perl の国へようこそ Perl5 対応版』, Computer Tody ライブラリ 34, サイエンス社, 348 頁. ISBN: 4781907954.