分子生物第３講座/構造生物学研究センター　　2010 年 2 年生コンピュータ実習
文責　成田哲博

Python をつかったプログラム入門

内容

はじめに

目標

生命理学において必要とされる最低限のプログラム作成ができるようになること。本実習においては、習うより慣れろで、美しくなくてもとにかく動くプログラムを気軽に作成できるようにすることに重点を置きます。本実習をクリアすれば、以下のことができるようになるはずです。

- Python を電卓として使える（第 1 回）。
- Python でひととおりのプログラムが書けるようになる(第 3-7 回)。
- アミノ酸や遺伝子シークエンスを解析、操作できる（第 8 回)。
- 蛋白質構造データ(pdb ファイル)を解析、操作できる（第 9 回)。
- 画像ファイルを解析、操作できる（第 10 回)。

全体計画

第一回（10/7） Mac と unix の基本、python を便利な電卓として使えるようにする。
第二回（10/14）turtle モジュールで絵を描けるようにする。
第三回 （10/21）Unix, Emacs の基本的な使い方を学ぶ。
第四回 （10/28）python のプログラム基礎 I。条件分岐、while によるループ
　　簡単なプログラムが書けるようにする。
課題提出（締め切り:第四回終了から二週間以内）
第五回 （11/4）python のプログラム基礎 II。For によるループ、break と continue
　　プログラムの基本がひととおりわかるようにする。
第六回 （11/11）ファイル入出力
　　簡単なデータ解析ができる。
　　課題提出（締め切り:第六回終了から二週間以内）
第七回 （11/18）関数とメソッド
　　大規模なプログラムを書く基礎がわかる。
　　同じプログラムコードを何度も書かなくて良くなる。
課題提出（締め切り:第七回終了から二週間以内）
第八回 （11/25）ハッシュとアミノ酸シークエンス
　　アミノ酸シークエンス解析の基礎がわかる。
第九回 （12/2）pdb file と蛋白質構造
　　蛋白質構造データフォーマットである pdb ファイルを解析、操作できるようになる。
第十回 (12/9) 画像ファイルの扱い
　　様々なフォーマットの画像データを自由に操れるようになる。
課題提出（締め切り:未定。１月半ばを予定)
括弧内の日付は諸般の事情により変更する可能性があります。

課題提出

実習は全 10 回。実習 3-4 回につき 1 回、計 3 回課題を提出してもらいます。課題提出は基本的にはメールで行い、プログラムはそのままファイルとして添付、そのほかはワードファイルで作成して添付してください。提出先は、narita.akihiro@f.mbox.nagoya-u.ac.jp です。

以下の注意を守ってください。守られない場合は減点の対象になります。
１　メールの件名は以下のフォーマットに従ってください。
コンピュータ実習:学籍番号:名前:課題 n-m:提出日(月/日)

n は提出レポートに入っている最初の回、m は最後の回。
たとえば例として、以下の件名の場合、
コンピュータ実習: 51597675:本田一郎:課題 4-6:11/1
第 4 回―6 回の課題で学籍番号 51597675、名前は本田一郎、11/1 の提出という意味になります。
件名がこのフォーマットに従っていない場合、提出を見落とす場合があるので注意してください。
2　図はワードのファイルの中に貼って提出し、別ファイルにしないでください。
3　必ず、添付のワードファイルのファイル名の最後に拡張子.doc が付いているのを確認してください。
4　課題にプログラム作成が含まれている場合は、以下のようにしてください。
プログラムをそのままファイルとしてメールに添付してください。これはプログラムの動作確認をするためです。
レポートにもプログラムコードを書き、作成したプログラムには、解説を必ず入れてください。
5　書いたプログラムの使用例を必ずレポートに入れてください。
提出してもらったレポートには、必ず一週間以内に返事を出します。もし返事がこない場合は、その旨をメールか口頭で伝えてください。

採点基準

採点は以下の観点から行います。
1　締め切りが守られているか？提出が遅れた場合、遅れに応じて最大 30 %減点します。
2　課題はちゃんとできているか？
3　自分が書いたプログラムや結果を自分の言葉で説明できているか？
4　プログラムの内容や、使用例に独自の工夫が見られた場合は加点もあります。
また、途中入院等やむをえない事情で欠席した場合は、補習等考慮しますので、言ってください。

実習場所、時間

木曜日 3,4 限
情報メディア教育センターラボ　主センター　B 教室
（工学部 7 号館 4 階、下地図の 35 番）

テキストとデータ

本テキストは、構造生物学研究センターのホームページ
http://str.bio.nagoya-u.ac.jp:8080/Plone　の中の、”講義資料”のページからダウンロードできます。講義に必要ないくつかのデータもここからダウンロードできます。
また、誤植や間違いがあった場合は遠慮無く指摘してもらえるとありがたいです。いままで誰も気づいていない間違いを見つけた人は、その重大さに応じて加点があります。

注意事項

本テキストの用例やプログラム例はなるべく自分の手で入力して動作を確認してください。

また、課題のプログラムは、人のものを参考にするのはかまいませんが、ファイルのコピーやコピーアンドペーストをしてはいけません。必ず自分の手で入力してください。もし、まったく同じプログラムが提出された場合は、双方を減点の対象にする可能性があります。人のプログラムを参考にする場合でも必ず自分なりの変更を加えてください。

本実習の内容は、すでにプログラミングの経験がある人には退屈かもしれません。そのような人は自分で先に進んでもらってもかまいませんが、TA が少ないので、できれば周りの人に教えてもらえるとありがたいです。

第一回 Mac と unix の基本、python の起動

今回の目的:

1 MacOSX の基本的な使い方を学ぶ。
2 python を電卓として使えるようにする。

１　プログラミング

コンピュータは言うまでもなく現代の文明に欠かせないものであり、当然ながら研究者も手足のように動かせなくてはならない。現在のコンピュータは、まだ自分で思考ができないので、何かさせようと思えば、１から１０まで手順を指定しなくてはならない。その指定書がプログラムであり、それを作成することをプログラミングと呼ぶ。皆さんが使っているワープロソフトやインターネットブラウザもすべて誰かが作ったプログラムである。

生物学の研究者は、それほど大きなプログラムを作る必要はめったにない。また、いままでだれかがやったことがある実験や解析であれば人が作ったプログラムを使うことも良いだろう。しかし、人が作ったプログラムで、皆さんが実際に行いたい実験の解析ができるとは限らない。プログラムは、作った人がその時点で想定した範囲内でだけ働くものであるから、皆さんが最先端の研究をすればするほど、そのために用いることができるプログラムが存在する可能性は小さくなっていく。そのときに、自分で使うためだけの小さなプログラムを自分で書ければ、研究の幅を狭めることが無くなるのである。

プログラムを書くためのプログラム言語には無数の種類があり、おすすめの言語は人によって千差万別である。本実習では、自然科学系のライブラリが特に豊富であること、書いたプログラムがそのまま動くインタープリターであること、初心者が学びやすいと言われていること、書いたプログラムが Mac, windows, linux など多くのシステムで動くこと、それでいて非常に大規模なプログラムが実際に python で書かれていることなどから、python と呼ばれるプログラム言語を用いることにした。しかし、本質的なところはどのプログラム言語を使ってもあまり変わらない。本実習が終わるころには、他の言語でかかれたプログラムも、理解不能の文字列ではなく、意味のある「言語」として認識できるようになっているだろう。

２　コンピュータと OS (オペレーティングシステム)

1980 年代から普及してきた個人用のコンピュータ(パーソナルコンピュータ、パソコン)は、急速にその性能を伸ばし、10 年くらい前まで自然科学計算で主流だった、高価なワークステーションを使うことは、現在ほとんど無くなった。また、スーパーコンピュータさえも、現在の主流はパソコンと同等なハードウェアを無数につないだ並列型である。従って、これから皆さんが研究で使うコンピュータはほぼ 100 %パソコンの部類であろう。

現在のパソコンはすべて OS 越しに操作される。本来コンピュータは二進数しか理解しない。コンピュータが直接理解できる言語を機械語と呼び、実際２０年くらい前はこの機械語を直接打ち込んでパソコンを動かすこともそれなりに行われていた。しかし、普通の人間が機械語を話すのは無理であるし、機械語はハードウェアごとに異なる。

そのため、通常はコンピュータのハードウェアを抽象化し、ハードウェアによらない統一したインターフェースを提供する OS と呼ばれるシステムを通じて、ユーザーやユーザーが作ったプログラムはコンピュータにアクセスする。たとえばメモリの量や CPU のタイプが違っていても、ユーザーやプログラムから見える OS の動作は、(もちろんハードウェアに応じた演算能力の範囲内で) 変化はない。現在パソコン上で主に使われている OS は windows 系と Unix 系の二系統にわけられる。

２－１　Windows 系:

Microsoft 社が開発している商用の OS。現在のパソコン OS の主流を占める。多くの商用ソフトが販売されており、利用者が多い分インターネット上で公開されている無料、有料ソフトの数も多い。また、20 年以上前の MS-DOS（windows の前のマイクロソフト社の OS。UNIX のようなコマンド入力式であった）用のソフトもほとんどが動き、古いソフトを長く使うことができる。

一方で、メジャーなソフトウェア開発環境は有料であり、気楽にプログラムを組める環境ではない。また、Unix 系では無いため、Unix 系が持っている開発用資産(ライブラリや言語など)をインストールする際にかなり癖があって、Unix 系とまったく同じにはならない。

２－２　Unix 系

Unix は、1970 年代最初にミニコン(家庭用小型冷蔵庫くらいのコンピュータ、当時としてはミニであった)上で開発されて以来、ミニコンやワークステーション等小型コンピュータの標準的な OS であり続けている。当初からマルチタスクマルチユーザーの先進的な機構を備えていた。様々な系統に分裂しているが、基本的なコマンド群は同じであり、また Unix 系であれば相互のソフトウェアの移植も容易である。そのため、膨大なプログラム開発用資産を継承している。Apple 社のマッキントッシュの OS である MacOS はヴァージョン 10 から Unix の一流派である BSD 系の OS に変わった。現在パソコン上で主に使われているのは、この MacOSX と Linux および、BSD ファミリーの三つである。

２－２－１　Linux

Linux は、リーヌス・トーバルズが 1991 年に開発した Unix 互換 OS。当時フリー(無料で誰でも改変可能)な OS は知的財産権に関する訴訟を起こされていた BSD 系しかなかったため、1 から Unix 互換 OS を書き直すことで訴訟リスクの無いフリーOS を開発するのが目的の一つであった。gcc（Gnu C Compiler, C 言語の開発環境の中核)をはじめとする無数のプログラム開発環境も開発、移植され、現在はスーパーコンピュータから携帯電話まで使われている Unix のメインストリームとなっている。PC/AT 互換機(Windows がインストールできるパソコン)にも容易にインストールでき、自然科学系のプログラム開発の標準 OS の一つである。OS の中でも最も基本的な部分をカーネルと呼ぶが、Linux は厳密に言えば OS 全体でなく、このカーネルを指す言葉である。(ただ、一般には OS 全体を指す言葉としても使われている。) Linux のカーネルや、一緒に使われる多くのプログラムは GPL(GNU General Public License)と呼ばれるライセンスで提供されている。誰でもプログラムの中を見ることができ、無料で手に入れることができ、改変できるが、改変したものやそれを内蔵したものを配布したり販売したりする場合は、GPL に従わなければならないと決められている。筆者は現在 Linux の一種である OpenSUSE11 をメインに使って、データ解析を行っている。

２－２－２　BSD ファミリー

もう一つの Unix 互換フリーOS。Linux より歴史が古い。訴訟を起こされたことにより、Linux より普及が遅れた。FreeBSD, NetBSD, OpenBSD などがあり、それぞれ特徴がある。たとえば NetBSD は移植性が高く、多種の CPU で動作し、組み込み系で活躍している。OpenBSD はセキュリティの強さに定評があり、標準設定でインストールしたときのセキュリティホールが 10 年以上にわたって二つしか見つかっていないことを売りにしている（Windows や Mac のセキュリティアップデートの数と比べてみよ）。FreeBSD は BSD 系の安定感を保ちながら、新しい技術を積極的に取り込み、ユーザーフレンドリーな環境を構築している。BSD のライセンスは GPL よりも縛りが緩く、商用製品に組み込んで使うこともできる。

２－２－３　MacOSX

アップル社の MacOS は、MacOSX から、BSD ベースの Unix 互換システムになった。Unix 互換であるため、いままで Linux や BSD 等で開発してきたソフトウェアがわずかな変更で移植できる。そのため、さまざまな自然科学計算用のソフトウェアが移植されてきている。また、ユーザーインターフェースが通常のフリーな Unix 互換システムより洗練されているのに加えて、アップル社やほかのソフトウェアメーカーによる有料ソフトも多く使うことができる。Microsoft office や adobe photoshop などはフリーUnix では動かないが、MacOSX では使うことができる。今回は、この MacOSX を使ってプログラミングを学ぼう。ただ、MacOSX は、フリーな Unix ではないため、アップル社の都合により仕様が大きく変わることがある。とくに MacOSX の X window system は

OS のバージョンアップ時に仕様が頻繁に変わり、トラブルが多い。また、Windows や linux, BSD ファミリーに比べて、古いハードウェアのサポート打ち切りがだいぶ早い、ハードウェアの選択の幅が狭いなどの問題点もある。

3　MacOSX の使いかた

では、実際に MacOSX に触れてみよう。以下のようなログイン画面に、名古屋大学 ID とパスワードを入力するとログインができる。

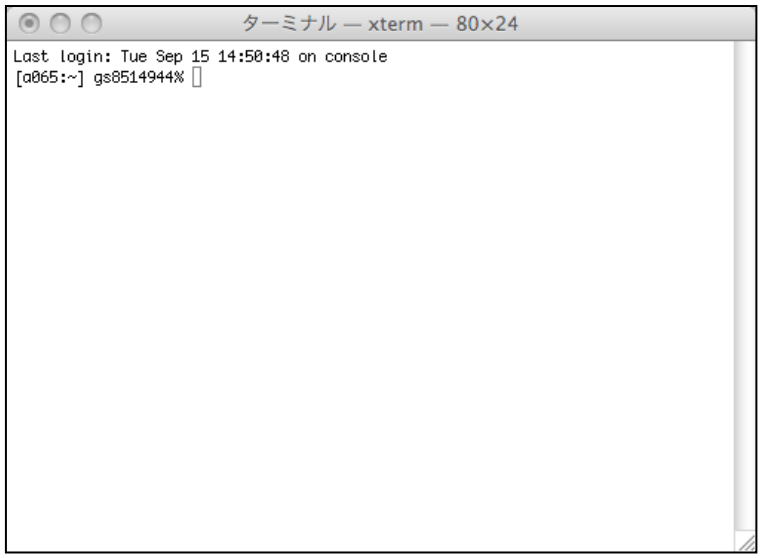
ログインすると下図のようになる。下の Dock と呼ばれる領域に、コンパスの絵があり、そこにマウスを持って行くと Safari と表示される。これは、MacOSX 標準のインターネットブラウザである。Dock の中にあるコンパスの絵をクリックして、Safari を起動してみよう。

情報メディア教育システムのページが表示されるはずである。”1.利用法”の枠の中の一番上にある”基本的な使い方”をクリック、開いたページの中のリストの一番上、”Mac の基本”をクリックすると MacOSX の基本的な使い方に関する説明が表示される。この説明を一通り読んで理解しよう。本実習はこの”Mac の基本”を理解しているものとして進めることとする。ただし、すでに MacOSX をある程度使いこなせている人は、読まなくても良い。

4　X-window と xterm

コンピュータに何かをさせようと思えば、人間が何かを操作しなくてはならない。この操作をコンピュータのハードウェアに伝達する手段をユーザーインターフェース（UI）と呼ぶ。Windows や MacOSX においては、通常はアイコンをクリックすることによって操作を行う。このような絵を媒介にした操作法は Graphical User Interface (GUI)と呼ぶ。このやりかたは直感

的ではあるが、大きな演算能力を必要としたり、アイコンを探すのが面倒だったり、同じことを繰り返すのが面倒だったりする欠点も存在する。



一方、UNIX の基本の使いかたはこれと異なり、キーボードでコマンドをうちこんでコンピュータにコマンドを実行させる(Character User Interface, CUI)。実際に体験してみよう。Dock の中にある X11 のアイコン(白地に X のマーク、前ページ図青矢印) をクリックする。画面内に右図のようなウィンドウが一つ立ち上がるはずである。これが xterm と呼ばれ、Unix にコマンドを打ち込むためのウィンドウである。ターミナルのウィンドウの中には、入力カーソルの前に以下の文字列がある。

[端末名:現在のディレクトリ] ユーザー名%

ディレクトリについては後述する。筆者の場合、端末 a048 からログインすると、

[a048:~] gs8514944%

のようになる。この文字列は、UNIX がユーザからの命令を待っている状態を表し、「コマンドプロンプト」とよばれる。コマンドを実行するには、このプロンプトに続いてコマンドをタイプし、リターンキーを押せば良い。このウィンドウの中で、試しに、date と入力してリターンキーを押してみよう。(**UNIX のコマンドは英数字で打ち込むので、キーボードの「英数」キーを押してマックを「英数字入力モード」にしておく必要がある**)。次のように実行結果が表示されたのち、再びコマンドプロンプトが表示されコマンド待ちの状態になる。現在の時刻が表示されるはずである。

```
[a048:~] gs8514944% date
Wed Aug  4 14:05:57     2010
[a048:~] gs8514944%
```

UNIX では、コマンドを表すアルファベットの大文字と小文字は別の文字として扱われる。つまり、小文字で入力すべき文字を間違えて大文字で入力すると、意図したコマンドが実行されない。試しに DATE とタイプしてリターンキーを押してみよう。下のように、"Command not found." と文句をいわれる。

```
[a048:~] gs8514944% DATE
DATE: Command not found.
[a048:~] gs8514944%
```

また、別の xterm を開きたければ、すでに起動しているターミナルをクリックして、アップルキー(キーボード左下のほうにあるリンゴマークが描かれたキー)と N を同時に押せば良い。プログラミング中は、通常 3 つ以上のターミナルを開いて作業する。

以後、Unix のプロンプトは%で表し、ユーザーが入力したコマンドを網掛けで表示することにする。

５　MacOSX のコピー、ペースト

ここで、コンピュータを使うときに特にお世話になるコピーペーストの方法を紹介しておこう。MacOSX 標準の方法と xterm における方法は若干異なるので、まず標準の方法を述べる。左ボタンでテキスト等を選択し、アップルキー+c (アップルキーと c を同時に押すことをこのように表記する) を押すと、その内容がクリップボードにコピーされる。クリップボードとは、コ

ピーしたい内容を一時的に保存しておくメモリ上の領域のことを言う。この内容を別の場所に貼り付けたい(ペースト)ときは、アップルキー+v を押せば良い。

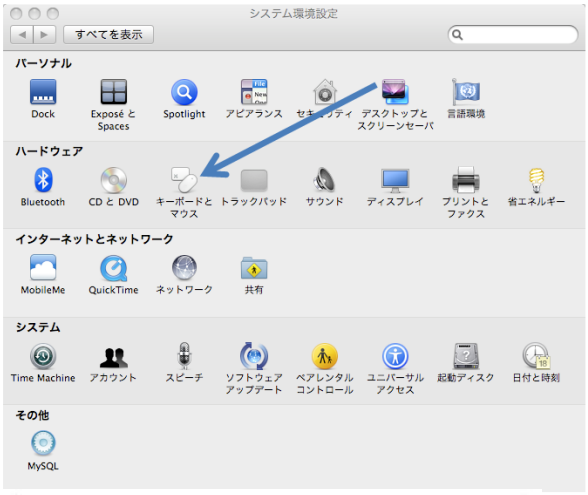
6　xterm のコピー、ペースト

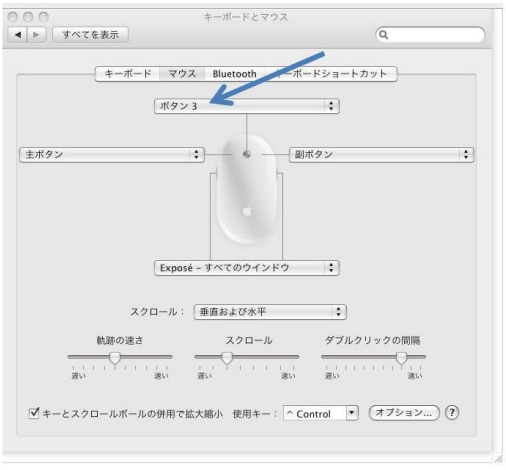
6-1 xterm のコピー、ペースト準備

UNIX 標準のユーザーインターフェースは X window system と呼ばれるもので、xterm はこの X window system の中のアプリケーションである。xterm は MacOSX 上ではそのままでは動かないので、X window system を MacOSX 上で動かし、その上で xterm を起動している。実は、先ほどクリックした X11 のアイコンは、xterm の起動アイコンではなく、X window system の起動アイコンである。X window system は MacOSX とは独立したユーザーインターフェースを持っている。xterm はこの X window system のユーザーインターフェースに従うため、コピーペーストの方法も MacOSX 標準とは異なる。

xterm のコピーペーストはごく簡単である。**コピーは左ボタンで文字列を選択するだけである。ペーストは中ボタンを押せばよい。**しかし、MacOSX の初期状態では、この中ボタンは dashboard という別のアプリケーションにわりあてられている。xterm でペーストをするためには、これを解除する必要がある。まず、画面左上のリンゴマークをクリック。メニューからシステム環境設定を選択する。

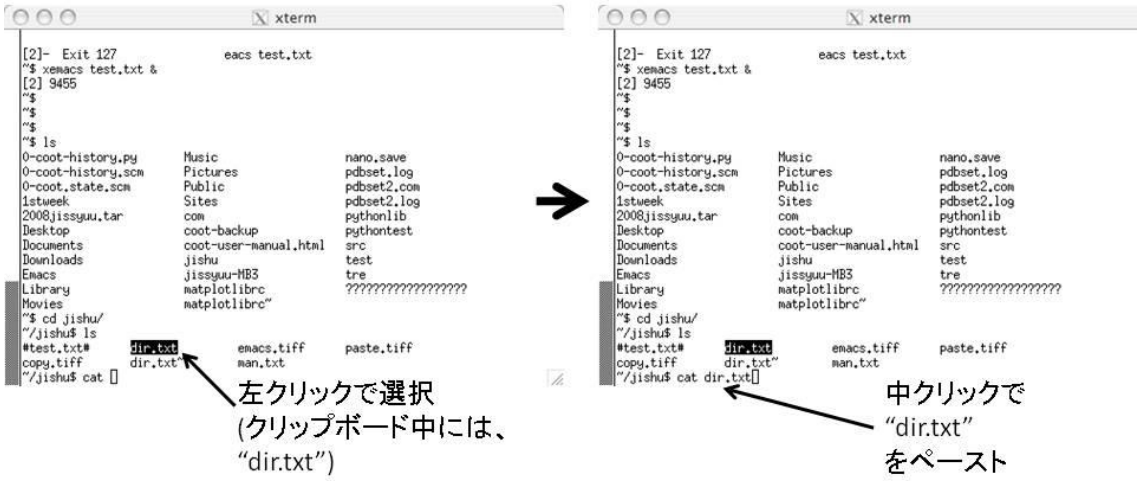
システム環境設定のウィンドウの中のキーボードとマウスをクリックする(右上図矢印)。すると、キーボードとマウスの設定画面がでてくるのでこの画面のマウスタブをクリック。すると右中図のようなマウスの絵がでてくる。青矢印で示した中ボタンの設定は、現在 Dashboard になっている。これをクリックし、右下図のようにボタン３に変更する。このウィンドウを閉じて、設定は終了である。





6-2 X window のコピー、ペースト

xtermなどのX windowアプリケーションにおいては、アップルキー+vでペーストはできない。そのかわり、さきほど述べた通り、中ボタン(ボタン３)を押すだけでペーストができる。これは、X window以外のMacOSX標準アプリケーション(safariやwordなど)からコピーした内容を、xtermにペーストする場合でも同様である。実習用コンピュータにおいてはマウス中央の小さなボタンが中ボタンで、その左側が左ボタン、右側が右ボタンの役割をする。実際の例を下に示す。



また、X windowアプリケーションにおいては、左ボタンでテキストを選択すると、それだけでその内容はクリップボードにコピーされる。この内容は、X windowアプリケーションにペーストするときは中ボタンを押せばペーストできる。しかし、X windowアプリケーションからそれ以外のアプリケーションにペーストしたい場合は、標準通りにアップルキー+cでコピーし、アップルキー+vでペーストする。また、MacOSX標準アプリケーションからX window アプリケーションにコピーペーストしたい場合は、アップルキー+cでコピーし、中ボタンでペーストする。まとめると、

X window アプリケーション間
　　左ボタンでテキスト選択のみでコピー、中ボタンでペースト
MacOSX標準アプリケーションからコピー、X window アプリケーションでペースト
　　アップルキー+cでコピー、中ボタンでペースト
X window アプリケーションからコピー、MacOSX標準アプリケーションにペースト
　　アップルキー+c でコピー、アップルキー+v でペースト
MacOSX 標準アプリケーション間
　　アップルキー+c でコピー、アップルキー+v でペースト

7　python 対話モードの起動

では、いよいよ本実習の要である python に触れてみよう。まず、プログラムを書き始める前に、python を対話モードで使ってみることにする。対話モードとは、unix のシェルのようにコマンドを一回入力するごとにすぐ答えを返してくるモードである。ちなみに通常のモードではプログラムファイルを読み込んで、一気にプログラム全体を処理する。python 対話モードの起動は非常に簡単である。ターミナルから

```
% python
```

と入力すればよい。

```
>>>
```

というプロンプトがでたら、起動成功。>>>は python 対話モードのプロンプトである。

8　四則演算

そのまま式を入力すれば答えが返ってくる。
足し算

```
>>> 1+1
2
```

引き算

```
>>> 3.5-5.5
-2.0
```

かけ算
```
>>> 6*3.2
19.200000000000003
```

ここで、6*3.2 が 19.2 ちょうどにならないのは、次でのべる数値計算の誤差のため。

この誤差は、数値がコンピュータ内部では１０進数ではなく、２進数で表現されているために起こる。２進数と１０進数の相互変換のときに誤差が生じる。これについては次節で詳しく述べる。通常はこの誤差はごくわずかのため無視しても良いが、プログラムや計算内容によっては問題が起こる場合もある。また、計算式の前に print を足すとこの誤差はたいてい隠蔽される。(print は人間が見やすいように自動的にフォーマットをととのえてくれる)

```
>>> print 6*3.2
19.2
```

割り算
小数を含む割り算
```
>>> 5/3.0
1.6666666666666667
>>> 5/2.0
2.5
```

整数同士の割り算
```
>>> 5/2
2
>>> 5/-2
-3
```
整数同士の計算の場合、答えも整数(切り下げ)になる。これを避けて小数の答えを得たい場合は、小数を含む割り算で見たように、
```
>>>5/2.0
```
などとすれば良い。

割り算の余り
```
>>> 5 % 2
1    # 5 を 2 で割った余り。
>>> 5%2.2
0.59999999999999964 # 5 を 2.2 で割ったあまり。
>>> print 5%2.2
0.6
```

累乗
```
>>> 2**10   #2 の 10 乗を計算
1024
```

複素数計算
虚数は実部+虚部 j のように書く。たとえば
```
>>> 1+2j
(1+2j)
```
複素数同士で計算ができる。
```
>>> (1+2j)*(1-2j)
(5+0j)
```

```
>>> (1+2j)-(1-1j) #虚部が 1 の場合、j では無く、1j のように書く。
3j
```

ここでひとつ python 特有の注意点がある。行の最初に入れる空白をインデントと呼ぶが、python ではこのインデントに特別な意味をもたせている。**もし必要のないインデントが入ると即エラーになる。**たとえば、

```
>>>  (1+2j)-(1-2j)

  File "<pyshell#1>", line 1
    (1+2j)-(1-2j)
    ^
IndentationError: unexpected indent
```

のように、(の前にひとつ空白を入れるだけで、IndentationError になる。インデントの正しい使い方は第三回で python プログラムを書き始めるときに示す。

９　コンピュータ内部の数値表現

ここで、すこし寄り道になるが、コンピュータ内部の数値表現について勉強してみよう。現在の全てコンピュータは 0 と 1 しか認識しない。実際のコンピュータにおいてはある種のコンデンサーが無数に並ぶことでメモリを構成しており、そのコンデンサーが電荷を蓄えていれば 1，いなければ 0 と認識されている。このコンデンサー列の電荷の有無の並びが、コンピュータが認識する全てであり、数値も文字列もプログラムも全てこの形式で保存されている。この中で、数値は 0 と 1 だけから表現できる二進法によって表記される。通常の十進数において数値 N は、

$$a_n a_{n-1} a_{n-2} \cdots a_0 . a_{-1} a_{-2} \cdots a_m$$

のように表記される。ここで、a_i は 0 から 9 の整数であり、

$$N = \sum_{i=m}^{i \le n} a_i 10^i$$

である。例えば、123.456 は、a_2=1, a_1=2, a_0=3, a_{-1}=4, a_{-2}=5, a_{-3}=6　に相当する。同様に二進数で数値を表記するには、

$$N = \sum_{i=m}^{i \le n} a_i 2^i$$

となる。ここで、a_i=0 または 1 である。たとえば 12.75 は、

$$1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2}$$

と表記できるので、二進数表記では 1100.11 となる。コンピュータ内部で数値計算をする場合、十進数の数値をこの二進数表記に直してから計算を行い、また表示する際に十進数表記にもどす。しかし、十進数の有限桁の小数が、二進数では有限桁で表示できないことがある。ここで問題が生じる。例えば、十進数で 0.1 は、二進数表記では、0.00011001100110011…となり、無限桁の循環小数になる。コンピュータで扱える桁は有限であるから、変換の際に誤差が生じる。0.6 や 0.2 も二進法では同様に循環小数になる。その結果、

```
>>> 6*3.2
19.200000000000003
```

や、

```
>>> 5%2.2
0.59999999999999964
```

のように、最後の桁で計算が狂うのである。問題になることはほとんど無いが、この計算の狂いは python に限らずコンピュータを扱うさいには常に存在すると考えて良い。

１０　変数

電卓として使っていると、計算した結果を覚えていて欲しいと思うことは多いだろう。このとき、変数に代入することで結果をとっておくことができる。
変数に数値を覚えさせるには、以下のようにする。

```
>>> a=20
```

数値を表示するには、変数名の前に print を入れれば良い。

```
>>> print a
20
```

対話モードでは、ただ変数名を入力してもその内容を表示する。

```
>>> a
20
```

とする。ここで a は変数名。変数名には以下のルールがあり、それを守りさえすれば何でも良い。

1. 先頭は英文字または、下線にし、その後は英文字、下線、数字を用いる。 下線以外の記号は使わない。
   例: _spam, spam, Spam_1 は Ok。1spam, spam$, @spam などはだめ。
2. 大文字と小文字は区別される。
   例: spam と Spam, sPam はすべて別の変数
3. プログラミングに必要な以下の予約語は使えない

予約語一覧
'and', 'as', 'assert', 'break', 'class', 'continue', 'def', 'del', 'elif', 'else', 'except', 'exec', 'finally', 'for', 'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'not', 'or', 'pass', 'print', 'raise', 'return', 'try', 'while', 'with', 'yield'
しかし、前後になにか文字をつけたり、大文字にかえれば大丈夫
例: in はだめだが、in1, In は Ok。

また、変数名はなるべく意味のある単語をつけたほうがいい。変数は数値と同じようにそのまま計算に用いることができる。変数をつかった計算の例。

```
>>> hight=20
>>> width=5*9
>>> area=hight*width
>>> area
900

>>> complex1=19.5+2.3j
>>> complex2=-19.5+2.3j
>>> complex1*complex2
(-385.54000000000002+0j)
>>> complex1/complex2
(-0.97255797063858496-0.23266068371634591j)
```

また、一つ前の計算の結果は変数”_”に保存される

```
>>> _
(-0.97255797063858496-0.23266068371634591j)
>>> -1*_
(0.97255797063858496+0.23266068371634591j)
```

１１　モジュールと import

python では標準状態では三角関数、対数などの多くの数学関数が使えない。そのかわり、python には機能を拡張するモジュールという仕組みがある。たとえば、数学関数は外部モジュール math によって定義されている。外部モジュールを呼び出す方法は主に二つある。

ひとつめは、
```
>>> import math
```
と入力する方法。この方法の場合、モジュールに含まれている関数は、モジュール名.関数名の形でよび出す。たとえば、
```
>>> math.sqrt(2)
1.4142135623730951
```
とすると、2 の平方根が計算できる。このとき、math.sqrt は、モジュール math のなかの関数 sqrt という意味である。一度 import math を行えば、python を終了するまで math の中のすべての関数を使うことができる。
ほかの例も挙げる
```
>>> math.exp(3)  #e^3
20.085536923187668
>>> math.log(10)  #自然対数
2.3025850929940459
>>> math.pi  #円周率
3.1415926535897931
>>> math.e 自然対数の底
2.7182818284590451
```
モジュール math で呼び出せる関数のリストは、
http://www.python.jp/doc/nightly/lib/module-math.html
を見れば良い。

もうひとつは、
```
>>> from math import *
```
このようにしてモジュールを呼び出した場合、すべての関数は関数名そのままで呼び出せる。
```
>>> pi
3.1415926535897931
>>> e
2.7182818284590451
>>> cos(pi)
-1.0
```
この方式は便利だが、モジュールを多数呼び出したり、変数を多く使う場合は、関数名や変数名が互いに干渉する可能性が大きくなり、推奨できない。たとえば e という変数をプログラムに使っていると、自然対数の底の値が上書きされてしまう。
```
>>> e=3
>>> e
3
```
となり、**e はもはや自然対数の底ではない！**

モジュール math では複素数の関数はサポートされない。複素関数を使いたい場合はモジュール cmath を用いる。cmath で使える関数は、
http://www.python.jp/doc/nightly/lib/module-cmath.html
を参照のこと。

```
>>> import math
>>> import cmath
>>> complex=1j*math.pi
>>> cmath.exp(complex)
(-1+1.2246063538223773e-016j)  #e^πi =-1
```

１２　文字列

python では、数字だけでなく、文字列も変数として扱える。文字列はシングルクォート（‘）ま

たはダブルクォート（“）で囲む。
```
>>> a='Hello world.'
>>> a
'Hello world'
>>> b="It is a fine day."
>>> b
'It is a fine day.'
```
文字列の中にシングルまたはダブルクォートで囲った文を入れたい場合は、文字列の中で使わないほうのクォートを使う。
```
>>> c='"Yes." he said.'
>>> c
'"Yes." he said.'
>>> d="'Yes.' he said"
>>> d
"'Yes.' he said"
```
文字列の中に一つだけシングルクォートを入れたい場合は、\記号（バックスラッシュ、ターミナルや、emacs 内で¥を押すとバックスラッシュになる）をクォートの前につける。
```
>>> e="doesn\'t"
>>> e
"doesn't"
```

文字列は足し合わせることができる。
```
>>> c=a+" "+b
>>> c
'Hello world. It is a fine day.'
```

１３　変数の型

python の変数は最初に代入されたときに自動的に生成される。
python の変数にはいくつもの型があるが、型もそのときに自動的に決定される。
いままでできたのは、
整数型(int)　1, 100, -120 など
実数型(float)　0.1, 2.0, -3.4 など
複素数型(complex)　1+1j, -2j など
文字列型(str)　‘Hello world.’ など
これらは最初に代入されたときに決定される。変数型は自動的に決定されるが、変数型によって演算の答えが異なることは多い。
たとえば、
```
>>> a=3  # a:整数型
>>> i=2  # i:整数型
>>> a/i  # 整数同士の割り算
1

>>> f=2.0  # f は実数型
>>> a/f
1.5

>>> s="2"  s は文字列型
>>> a/s
Traceback (most recent call last):
  File "<pyshell#208>", line 1, in <module>
    a/s
TypeError: unsupported operand type(s) for /: 'int' and 'str'
```

int 型と str 型の間の演算は”/”でサポートされていないというエラー。

これをさけるために、変数型を変換する方法が用意されている。
整数型への変換: int(変数名)
```
>>> a=3
>>> f=2.0  #f は実数型
>>> fint=int(f)  # f を整数型にしたものを fint に代入
>>> a/fint
1   #整数同士の割り算になる。
>>> s="2"
>>> sint=int(s)  #文字列を整数へ
>>> a/sint
1
```

実数型への変換: float(変数名)
```
>>> i=2  # i:整数型
>>> ifloat=float(i)  #整数から実数へ
>>> a/ifloat
1.5
>>> s="2"  s は文字列型
>>> sfloat=float(s)  #文字列から実数へ
>>> a/sfloat
1.5
```

文字列型への変換 str(変数名)
```
>>> i=2
>>> istr=str(i)
>>> istr
'2'
>>> f=2.0
>>> fstr=str(f)
>>> fstr
'2.0'
>>> istr+fstr  #‘2’と’2.0’の文字列を足して、
'22.0'          #結果は’22.0’
>>> fstr+istr  #‘2.0’+’2’
'2.02'
```

7 ヒストリ機能
Python の対話モードでは、ヒストリ機能が備わっている。上矢印を押すと前に入力したコマンドが順番に現れる。いきすぎた場合は下矢印を押すと最近入力したコマンドに戻っていく。試してみよう。

課題１　cmath モジュール

cmath モジュールを用いて、

$$e^{0.5\pi i}$$

$$\left| e^{0.5\pi i} \right|$$

log(i)

を計算せよ。レポートには、結果とその結果を再現するのに必要なすべてのコマンドや情報を入れよ。ただし、絶対値を計算する関数は abs を用いよ。たとえば、
```
>>> abs(-1)
1
```
となる。関数 abs は複素数に対してもそのまま用いることができる。

課題２　整数型と実数型

a=14, b=3 と代入し、a/b をそのまま計算した場合答えはいくつか。小数の答えを得るにはどのようにコマンドを入力すれば良いか。

課題３　文字列

変数 c に”abc”, d に”hij” を代入し、c+d を計算せよ。実際に入力したすべてのコマンドと結果をレポートせよ。

１４　リスト

多くの値をひとまとめに扱う方法として、リストがある。リストはコンマ(,)で区切られた値の列を角括弧で囲んだものである。リストの要素をすべて同じ型にする必要はない。そのため、非常に柔軟で多様なデータ構造をこのリストによって実現することができる。

'abc'(文字列), 'def'(文字列), 2 (整数), 1+1j (複素数)の 4 つの値をリストにまとめると、

```
>>> a=['abc','def',-2,1+1j]
>>> a
['abc', 'def', -2, (1+1j)]
```

リストの要素(アイテム)は、番号(インデックス)で呼び出すことができる。一番最初のアイテムのインデックスが 0 であり、順番に番号が与えられる。リスト名のあとに角括弧で番号を囲うことで各アイテムを呼び出せる。

```
>>> a[0]
'abc'
>>> a[1]
'def'
>>> a[2]
-2
>>> a[3]
(1+1j)

>>> a[4]
Traceback (most recent call last):
  File "<pyshell#257>", line 1, in <module>
    a[4]
IndexError: list index out of range
```

a[4]は定義されていないので、エラーになる。

```
>>> a[0]+a[1]
'abcdef'
>>> a[2]*a[3]
(-2-2j)
```

各アイテムに代入もできる。

```
>>> a[1]="-2"
>>> a
['abc', '-2', -2, (1+1j)]
```

また、最後のアイテムは、インデックス-1 として呼び出すこともできる。最後から二番目は-2 である。

```
>>> a[-1]
(1+1j)
>>> a[-2]
-2
>>> a[-3]
'-2'
```

```
>>> a[-4]
'abc'

>>> a[-5]
Traceback (most recent call last):
  File "<pyshell#256>", line 1, in <module>
    a[-5]
IndexError: list index out of range
```
a[-5]は定義されていないので、エラーになる。

リストのために有用なメソッドと関数を挙げる。関数とメソッドについては今後の実習で詳しく勉強するが、
関数名(ひとつまたはカンマで区切られた二つ以上の引数)
の形になっているのが関数である。引数とは関数やメソッドに渡されるデータのことを言う。たとえば、
str(i)
は str という関数であり、i はその引数である。

一方で、
変数名.メソッド名(ひとつまたはカンマで区切られた二つ以上の引数)
のような形になっているのが、メソッドである。メソッドは変数型ごとに定義されており、変数名が持つ変数型によって使用できるメソッドが異なる。たとえば、後で述べるリストで使用できるメソッド append は、a がリストだとして、
a.append(x)
のように使う。x が引数である。もし a がリストではなくたとえば実数や文字列であれば、エラーになる。

１４－１リストのための有用なメソッド１

リスト名.append(x)
リストの最後にデータ x を追加する。
リスト名.extend(L)
リストの最後に追加リスト L を追加する。
リスト名.insert(i,x)
指定した位置にデータ x を挿入する。i はインデックスで、i で指定されたインデックスの前にデータが挿入される。

例：
```
>>> a=['abc','def',-2,1+1j]
>>> a
['abc', 'def', -2, (1+1j)]
>>> a.append(2)
>>> a
['abc', 'def', -2, (1+1j), 2]  #最後に”2”が足される
>>> b=[-1,-2,-3]
>>> a.extend(b)
>>> a
['abc', 'def', -2, (1+1j), 2, -1, -2, -3]  #リスト b が足される
>>> a.insert(0,"ghi")
>>> a
['ghi', 'abc', 'def', -2, (1+1j), 2, -1, -2, -3]  #先頭に’ghi’が挿入される。
>>> a.insert(2,-3.0)
>>> a
['ghi', 'abc', -3.0, 'def', -2, (1+1j), 2, -1, -2, -3]  #index 2 のデータ(‘def’)の前に-3.0 が挿入される。
```

注意：**存在しないリストに対する操作はできない。**Python においては、変数は最初に代入したときに生成される。たとえばリスト a1 に最初の要素を入れようとして、
```
>>> a1.append(1)
```
とすると、a1 に一度も代入がなされていなければエラーになる。このようなときは、
```
>>> a1=[]
>>> a1.append(1)
```
のように、まず a1 に空のリスト[]を代入すれば良い。

１４－２ リストのための有用なメソッド２

リスト名.remove(x)
値が x である最初のデータを消去する。
リスト名.pop([i])
インデックス i のデータを返し、そのデータを消去する。i は省略可。ここで[]は省略可を示す表現であり、実際に入力するときに[]が必要なわけではない。この表現はすべての python のドキュメントにおいて用いられる。i を省略した場合、pop(0)と同じになる。
リスト名.index(x)
値が x である最初のアイテムのインデックスを返す。
リスト名.sort()
リストの中身をソートする。文字列であれば順番は数字-大文字-小文字の順になる。

例
```
>>> a
['ghi', 'abc', -3.0, 'def', -2, (1+1j), 2, -1, -2, -3]
>>> a.remove(2)
>>> a
['ghi', 'abc', -3.0, 'def', -2, (1+1j), -1, -2, -3]    #(1+1j)の後の 2 がなくなった。

>>> b=a.pop(2)  #インデックス 2 のデータ(-3.0)を b に代入し、リストから削除
>>> b
-3.0
>>> a
['ghi', 'abc', 'def', -2, (1+1j), -1, -2, -3]
>>> a.index(-1)  #値-1 を持つインデックスを表示
5
>>> b=['xab','2dc','Xab','etw']
>>> b.sort()
>>> b
['2dc', 'Xab', 'etw', 'xab']
```

リストのための有用な関数
len(リスト名)
リストの長さを返す
range([start],stop,[step])
start から stop の手前までの数列をリストとして返す。step は一つあたりの増分。popp と同様に省略可能な引数は[]で囲んでいる。実際に入力するときに[]が必要なわけではない。start が省略された場合は start=0 となり、step が省略された場合は step=1 となる。**range 関数は整数しか扱えない。**

例:
```
>>> a
['ghi', 'abc', 'def', -2, (1+1j), -1, -2, -3]
>>> len(a)
8  #a のアイテムの数
```

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9] #0 から 9 までのリスト
>>> range(-2,8)
[-2, -1, 0, 1, 2, 3, 4, 5, 6, 7] #-2 から 7 までのリスト
>>> range(7,-4,-2)
[7, 5, 3, 1, -1, -3] #7 から-3 まで、-2 ずつ変化する数列
```

課題４：リストと range

0 から 9 までと 40 から 49 までの数列,51 を含んだ以下のようなリスト
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 51]
を作れ。range と extend, append を用いること。実際に入力したコマンドと結果をレポートせよ。

１５　文字列のインデックス表現

文字列もリストと同様のインデックス表現が可能である。この場合、一文字ごとにインデックスが割り当てられ、一文字目がインデックス 0 である。また、インデックス-1 は最後の文字を示す。

```
>>> a='Hello world.'
>>> a
'Hello world'
>>> a[0]
'H'
>>> a[1]
'e'
>>> a[5]
' '
>>> a[8]
'r'
>>> a[-1]
'.'
>>> a[-2]
'd'
>>> a[-3]
'l'
```

ただし、リストと違って、要素に代入はできない。

```
>>> a[1]='b'
Traceback (most recent call last):
  File "<pyshell#333>", line 1, in <module>
    a[1]='b'
TypeError: 'str' object does not support item assignment
```

のように、エラーになる。

１６　スライス

文字列やリストの特定のインデックス範囲の要素をまとめて取り出すことができる。これをスライスと呼ぶ。使用法は

```
>>> a='Hello world.'
>>> a[1:5]
'ello'
```

この場合は index 1 の e から、index4 の o まで取り出している。

文字列名[n:m]　で、index n から index m のひとつ手前まで取り出す。
文字列名[:m] で先頭から index m の一つ手前まで取り出す。
文字列名[n:]で index n から最後まで取り出す。
たとえば、

```
>>> a[3:8]
'lo wo'
>>> a[3:-1]  #index -1 は最後の文字
'lo world'
>>> a[5:]
' world.'
>>> a[:5]
'Hello'
```

リストでもまったく同じようにできる。

```
>>> b=['abc','def',-2,1+1j]
>>> b[0:4]
['abc', 'def', -2, (1+1j)]
>>> b[1:-1]
['def', -2]
>>> b[:2]
['abc', 'def']
>>> b[2:]
[-2, (1+1j)]
```

リストの場合は代入もできる。

```
>>> b[1:3]=[3,2]
>>> b
['abc', 3, 2, (1+1j)]
```

課題５：スライス

a に It is a fine day.を代入せよ。
文字列 a から fine だけをスライスを使って抜き出して変数 b に代入せよ。
実際に入力したコマンドと結果をレポートせよ。

１７　リストのリスト

リストの要素には変数型による制限は一切無い。ということは、リストの要素がリストであっても良い。

```
>>> a=[]
>>> a.append([2,3,4])  #a にリスト[2,3,4]を追加
>>> b=['abc','cde','efg']
>>> a.append(b) #a にリスト b を追加
>>> a.append(2) #a に 2 を追加
>>> a
[[2, 3, 4], ['abc', 'cde', 'efg'], 2]   #最初の二つの要素がリストである。
>>> a[0]  #普通にインデックスで呼び出すことができる。
[2, 3, 4]
>>> a[1]
['abc', 'cde', 'efg']
>>> a[0][0]    #a[0]がリストなので、a[0][0]とすると a[0]の第一要素を呼び出せる。
2
>>> a[0][1]
3
```

```
>>> a[1][2]
'efg'
>>> a[0:2]   #スライスも使える。
[[2, 3, 4], ['abc', 'cde', 'efg']]
```

さらに多重リストを作ることもできる。

```
>>> b=[a,1,a]
>>> b[2]
[[2, 3, 4], ['abc', 'cde', 'efg'], 2]
>>> b[2][1]
['abc', 'cde', 'efg']
>>> b[2][1][0]
'abc'
>>> b[2][1][0:2]
['abc', 'cde']
```

参考サイト
http://www.python.jp/doc/release/tut/ python のチュートリアルがまとまっている。

１８　応用編: モジュールの短縮名

モジュールの名前が長い場合、関数を呼び出すたびに長い名前を入力するのは面倒である。それを避けるために、

```
>>> import math as mh
```

のようにモジュール名に短縮名を割り当てることもできる。
使用例として、

```
>>> import math as mh
>>> mh.sqrt(2)
1.4142135623730951  #2 の平方根
>>> mh.exp(3)  #e^3
20.085536923187668
```

など。

第二回 turtle モジュール

今回の目的

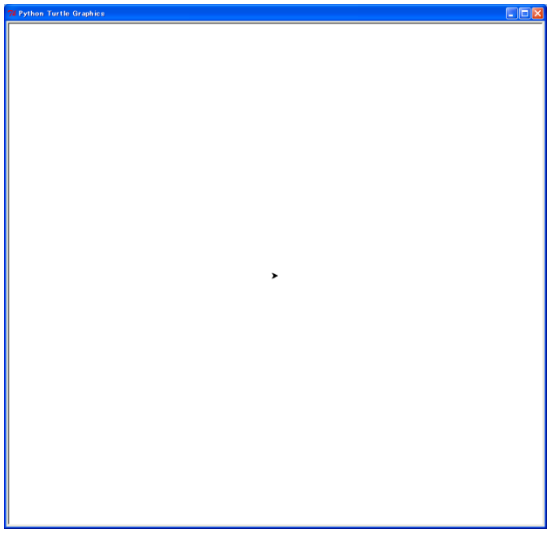
Turtle モジュールで絵を描こう。今後この turtle モジュールを用いてプログラムの基本を学んでいく。また、画面に表示された図の保存方法も今回学習する。

1　Turtle モジュールをまずは使ってみよう

Python の turtle モジュールは、turtle（亀)に指示を出すことで画面上に絵を描くためのモジュールである。まずは python を起動して試してみよう。

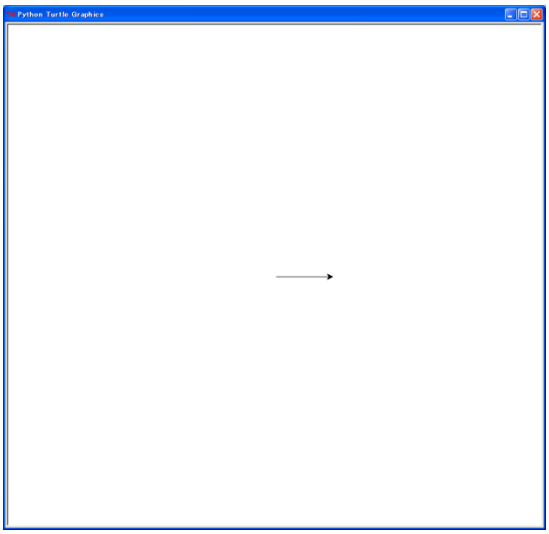
```
# python
>>> import turtle    # turtle module の import
>>> kame=turtle.Turtle() # turtle の生成。この場合 kame が新しい turtle の名前。
```

ここまで入力すると右のようなウィンドウが現れるはずである。
中央の矢印が、turtle の現在の位置を表している。この turtle を動かしてみよう。

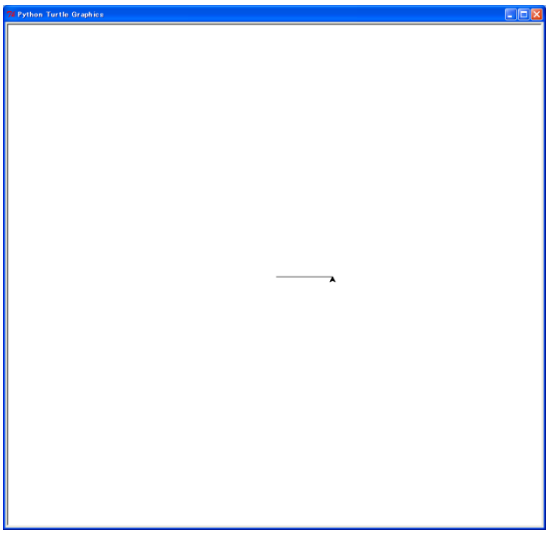


まず、turtle に前に進むように指示する。

```
>>> kame.forward(100)   #kame に 100 前に進むように指示(下図)
```

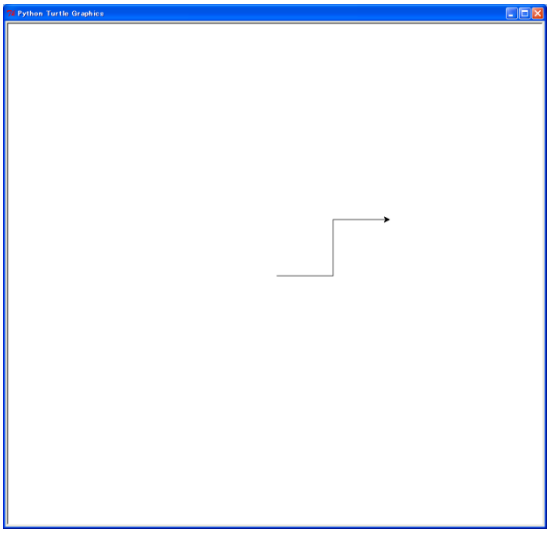



```
>>> kame.left(90) #kame に 90 度左を向くように指示(右図)。
```




```
>>> kame.forward(100)
>>> kame.right(90) #kame に 90 度右を向くように指示
>>> kame.forward(100)
```

(次ページ)

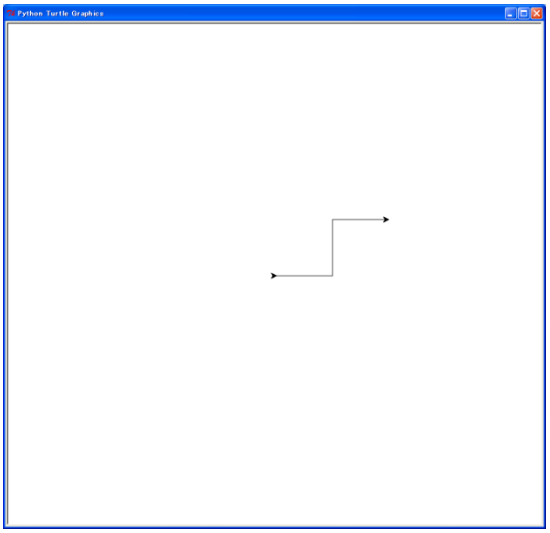


２　turtle モジュールの詳細

さて、少し詳しく見てみよう。最初の import turtle は、前回もやったモジュールの読み込みである。今回は turtle モジュールを読み込む。次の kame=turtle.Turtle()は、絵を描くための turtle を kame と言う名前で生成するという意味である。Turtle は複数生成することができ、そのそれぞれに名前を付ける。たとえば、

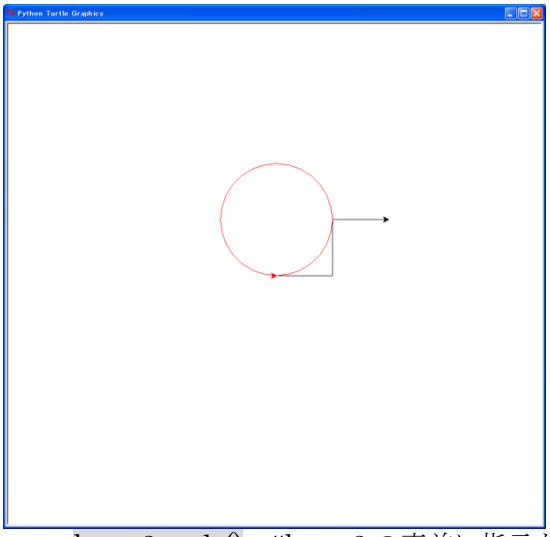
```
>>> kame2=turtle.Turtle()
```

とするともう一つの turtle がウィンドウの中央に生成する。中央に新たに現れた矢印が kame2 である。

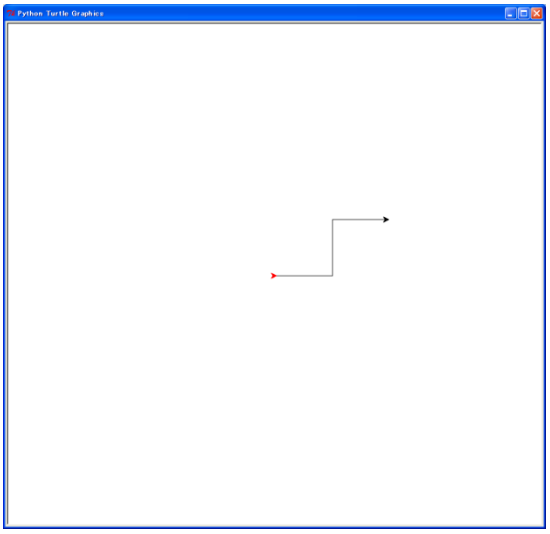


それぞれの turtle に指示をするには、turtle 名.指示内容という形をとる。たとえば、kame.forward(100)は、kame に 100 前に進めという意味であるし、kame2.left(90)といえば、kame2 に 90 度左を向けという指示である。指示内容には多くの種類がある。すでにでてきたのは、forward, left, right であるが、ほかにも例と共に見てみよう。

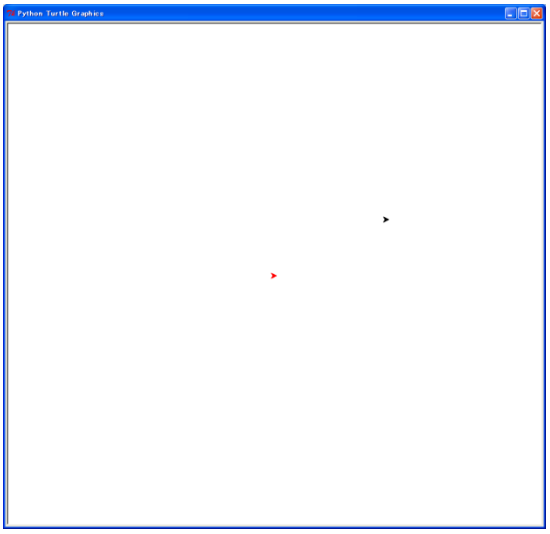
```
>>> kame2.color(1,0,0)   #kame2 の色を赤に変える。色については次節で詳しく述べる。
>>> kame2.circle(100)   #kame2 に半径 100 の円を描けと指示
```



>>> kame2.undo()　#kame2 の直前に指示を取り消す。

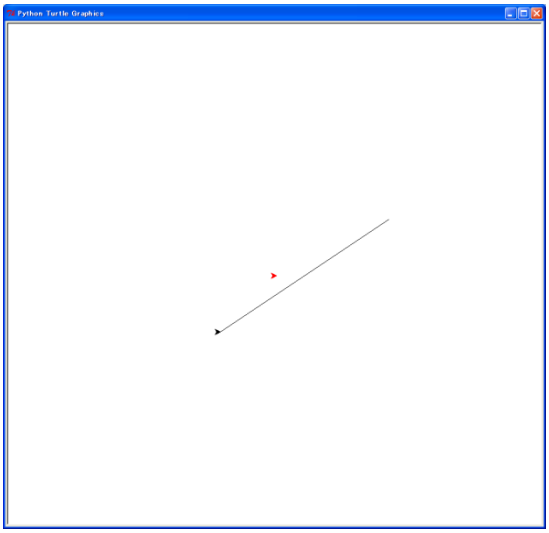


>>> kame.clear()　#kame が描いた線を全て clear する。



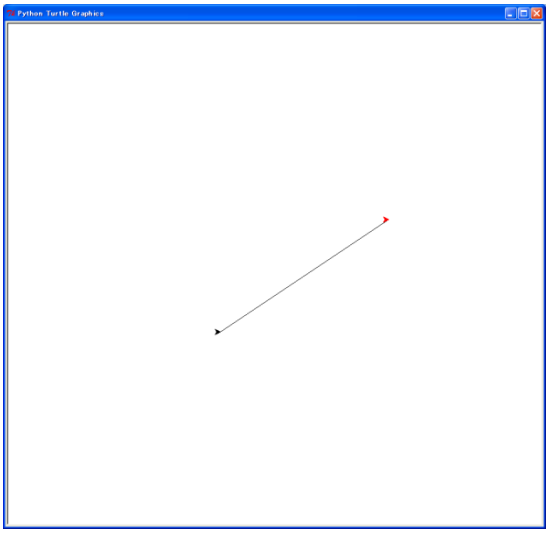
>>> kame.position() kame の現在いる場所を(x 座標, y 座標)の形式で表示する。
(200.00,100.00)

>>> kame.goto(-100,-100) kame に(x 座標,y 座標) = (-100,-100)に移動するように指示

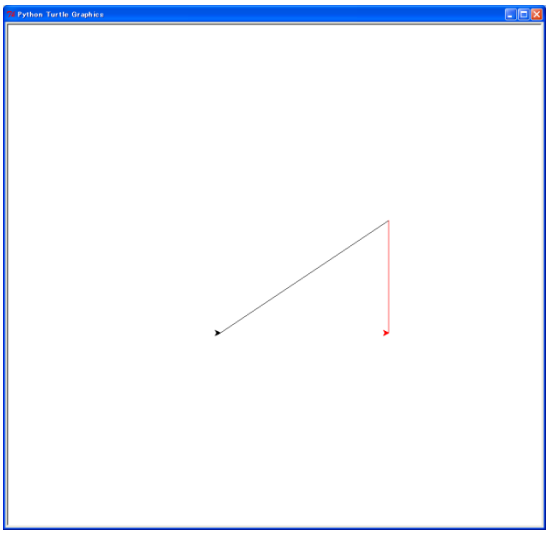


kame は goto で移動するさい、軌跡を残して移動したが、軌跡を残さず turtle だけを動かしたい場合もある。この場合は up()を用いる。up()を行うと、down()するまで turtle の移動の軌跡を残さない。これは、紙の上でペンを使って絵を描くことを考えるとわかりやすい。ペンを紙から離せば、次に紙にペンをおろすまでいくらペンを動かしても紙には何も残らない。これが up()に相当する。紙にペンを下ろすのが down()である。

```
>>> kame2.up()   #ペンを離す
>>> kame2.goto(200,100)
>>> kame2.circle(200)
```



```
>>> kame2.down()   #ペンをおろす
>>> kame2.goto(200,-100)
```



これでひととおりの使い方は学べたが、ほかにもいくつかの機能がある。turtle の他の機能については、必要であれば
http://www.python.jp/doc/2.5/lib/module-turtle.html
を参照せよ。

3　RGB 法による色指定

さて、ここで少し turtle から離れて、コンピュータにおける色の指定法のひとつである RGB 法について説明しよう。turtle の色を color を用いて指定する際、この RGB 法を用いているし、python 以外の多くのプログラムにおいて、色はこの RGB 法によって指定される。RGB は red, green, blue の頭文字である。この三つの色は光の三原色とよばれ、この三色の合成により、人間が認識できるどんな色でも作り出せる。

もともと可視光は 380-750 nm 程度の波長を持つ電磁場のことであり、連続したスペクトルを持っているので、物理的な意味では三原色というものは存在しえない。しかし、人間の網膜には、色を感じる視細胞が、赤、緑、青に対応する三種類しか無い。人間はこの三種類の視細胞に与えられる刺激の比率によって色を感じる。そのため、必然的に人間が認識できる色は全て赤、緑、青の三つから構築できることになる。多くのほ乳類は視細胞が二種類しかなく、またほ乳類以外の脊椎動物には、4 種類の視細胞を持つ生物が多い。そのような生物にとっては光の三原色ではなく、二原色だったり、四原色だったりする。

上に光の三原色とその基本的な混合を示した。緑と赤を 1:1 で混合すれば黄色になり、赤と青を 1:1 で混合すればマゼンタになり、緑と青を 1:1 で混合すればシアンになる。また全ての光を混

合すれば白になる。RGB 法は、これらの赤と緑、青の三つの光の強さを記述することによって、色を記述する。turtle の color においては、それぞれの強さは 0 から 1 で表され、赤、緑、青の三つの数字によって指定する。例えば、kame2.color(1,0,0)とすれば、赤=1, 緑=0, 青=0 となり、kame2 は赤となる。kame2.color(1,1,0)は赤=1, 緑=1 で黄色である。(0,0,0)であれば turtle は黒になる。turtle を実際に様々な色に変えてみよう。(0.1,0.6,0.2)のような小数の指定も可能であり、その微調整によって全ての色を再現できる。

４　画面の保存方法

さて、ここまでで turtle の使い方の説明は終わるが、課題に移る前に、turtle で描いた絵を保存する方法をここで述べることにする。そのためには MacOSX 標準のソフトである、グラブというソフトを用いる。グラブは、Finder の中のアプリケーションの中のユーティリティにある。グラブを見つけたら（ハサミが書かれたアイコンである）それを Dock 上までドラッグアンドドロップすると、Dock 内に登録され、以降使いやすくなる。Dock 上に登録したグラブをクリックすると、メニューバーがグラブのものに変化する。メニューのなかの取り込みをクリックして、ウィンドウを選択すると、画面上に指示がでるのでその通りにすれば指定したウィンドウの画像が取り込める。あとは、メニュー上のファイルから保存を選び、好きなファイル名で保存すれば良い。

課題　１：　正五角形を描け

Turtle を使って正五角形を描け。そのために用いた全てのコマンドと結果の画像をレポートせよ。

課題　２：　絵を描こう

Turtle を使って、自由な絵を描こう。ただし、20 回以上のコマンド入力を行い、色を三色以上用いること。

５　電子メールの使用

たいていの人が電子メールを使ったことがあると思うが、実習用のシステムではまだ使ったことがない人が多いだろう。課題の提出はメールで行うので、ここで設定しておこう。メールの設定の仕方は、情報メディア教育システムのホームページの”1. 利用法”の枠の中の一番上にある”基本的な使い方”をクリック、開いたページの中のリストの中の”メール”をクリックすれば、そこに書いてある。また、何かの電子メールソフトを使ったことがあれば使用方法はわかると思うが、http://mozilla.jp/support/thunderbird/の中の使い方ガイドに基本的な使い方はまとまっている。分からない場合は周りの人に聞いてみよう。

課題　３：　メールを出そう

となりや近くの人に、課題２で書いた絵を保存した画像ファイルを添付して、メールを出してみよう。(提出不要)

第三回 Unix, Emacs の基本的な使い方

今回の目的

今回は、python のプログラムを書くための基本的な準備を行う。プログラムはテキストファイルである。まず xterm 内でのファイルの扱い方、テキストファイルを作成するためのテキストエディタ Emacs の使い方を学ぶ。

１　UNIX の基本的な使い方

第一回で述べたように、MacOSX は UNIX と呼ばれる OS のグループの一つである。プログラムを作成する前に、まずは最低限のファイルの扱い方を知らなくてはならない。UNIX のごく基本的なことについて解説しよう。

１－１　Unix のファイルシステム

UNIX では情報をファイルという形で記憶装置に保存する。基本的に、コンピュータで扱うすべての情報はファイルとして扱われる。UNIX では、プログラムやデータだけでなく、入出力のある周辺機器（通信機器などのデバイス）もファイルとして扱う。

ひとつのファイルには必ずひとつの名前がつく。後から見て情報の内容がわかるようなファイル名をつけることが大切である。ファイル名に使える文字には制約があり、英数字、記号「.（ドット）」「-（ハイフン）」「_（アンダーバー）」を組み合わせて名前にするのが一般的である。（「/（スラッシュ）」は UNIX のファイル名の途中に使ってはいけない。日本語をファイル名に使うのも避ける。また、ドットを先頭にしたファイル名は環境設定などのために使う特殊なファイル（ドットファイルと呼ばれる）のファイル名にすることが多いので、ふつうのファイルの名前の先頭にドットを使うのは避ける（ドットはふつうは拡張子の前に使う）。また、UNIX ではファイル名においても大文字と小文字は区別される。UNIX は小文字をよく使うシステムである。ファイル名も小文字が基本。大文字を使うことも可能だが、それは README などのように、特に注目してほしいファイルの名前に使うことが習慣になっている。

ファイルの数が多くなると、いくつかのファイルをまとめて整理するためのディレクトリが必要になる（ディレクトリは、イメージとしては、ファイルを箱に入れてまとめて整理するための、箱のようなものである；windows や MacOSX では「フォルダ」と呼ばれている）。ディレクトリも、ファイルと同様に、ひとつのディレクトリにひとつの名前がつく。ディレクトリの中にさらに別のディレクトリを格納することもできる。

通常 MacOSX では、ファイルの管理は Finder で管理されている。Finder の中の一つ一つのフォルダがそのままディレクトリに対応する。

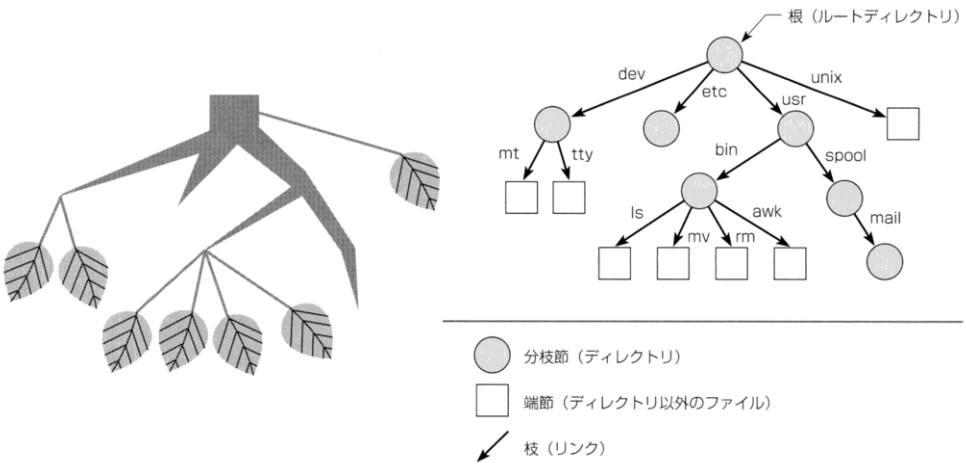
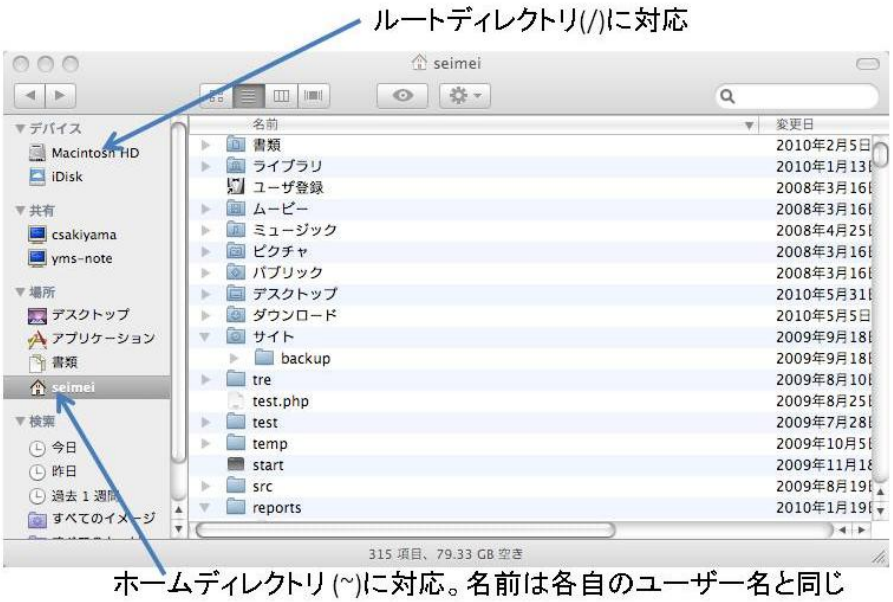


図２　本物の木（左）とファイルとディレクトリの木（右）

Windows でも同じであるが、MacOSX を含む UNIX では、ファイルの名前を「ツリー構造（木構造）」呼ばれる考えかたを使って整理している（図２）。ツリー構造とはひとつの根元から複数の要素に枝分かれしていく構造のことで、ある節から別の節までの道が１本しかない特徴をもつ。これは本物の木の枝が交わらないのと似ている。

UNIX で作業をするうえで覚えておくべき特殊なディレクトリが３種類ある。

- **ルートディレクトリ**
  ツリー構造のいちばん根元にあたるディレクトリ。記号「/（スラッシュ）」で表現される。UNIX では、１台のコンピュータには必ずひとつのルートディレクトリがある。Finder においては、”デバイス”の中の Machitosh HD がそれに当たる（次ページの図）。ただし、Finder からは/usr, /var などいくつかのディレクトリは隠されていて見えない仕様になっている。
- **カレントディレクトリ（作業ディレクトリ、ワーキングディレクトリ）**
  現在、ユーザが作業を行う場所として着目しているディレクトリ。自分がそのディレクトリの中にいるというイメージをもつとよい。「ユーザが現在作業中のディレクトリ」という意味で working directory ともいう。
- **ホームディレクトリ**
  ユーザがログインしたときに最初にいるディレクトリで、ユーザの作業の拠点となるディレクトリ。MacOSX でターミナルを起動すると最初はホームディレクトリにいる。各ユーザは、自分のホームディレクトリの中に自由にファイルを作ったり消したりすることができる。自分のホームディレクトリの外にあるファイルの利用は必要に応じて（UNIX のシステム管理者によって）制限されている。MacOSX においては、Finder の中の”場所”において家の形で示されている場所がホームディレクトリである（次ページの上図）。ホームディレクトリの名前は、各自のユーザー名と同じである。



１－２　パス名

UNIX では、ファイルやディレクトリの名前を表現するために、ツリー構造で考えてそのファイルやディレクトリにたどり着くための道順を示す。このような名前を「パス名 (path name)」という。パスとは「道」のことである。たとえば前ページ図２ではルートディレクトリから「usr」、「bin」、「ls」の矢印をたどると左から３番目のファイルにたどり着く。そして、このたどりかたで他のファイルやディレクトリには決してたどり着かないことがわかる。すなわち、この「たどりかた」を使ってファイルやディレクトリを一意的に指定できる。

ルートディレクトリから出発する方法で表記するパス名を「絶対パス名(absolute path name)」という。絶対パス名は、ルートディレクトリを表す「/」の後にたどった枝の名前を並べ、間に区切りとして「/」をはさんだものを使う。たとえば、先ほどの例のファイルは「/usr/bin/ls」と指定される。

カレントディレクトリからたどるパスでファイルやディレクトリを指定するパス名もあり、これを「相対パス名(relative path name)」という。パスの先頭を「/」ではじめなければ相対パス名となる。たとえばカレントディレクトリが/usr のとき、相対パス名で「bin/ls」は絶対パス名で「/usr/bin/ls」の意味となる。

１－３　基本的なコマンド群

ここでは、ファイルやディレクトリの操作方法のうち、比較的よく使うものを説明する。テキストエディタの説明後にもう少しコマンドの説明を追加する。

■　カレントディレクトリの確認

カレントディレクトリを知るためには、コマンド pwd (print working directory の頭文字をとったコマンド名)を使う。

書式：　pwd

このコマンドを実行すると、カレントディレクトリが絶対パス名で表示される。また、初期状態なら、プロンプトのなかにもカレントディレクトリが表示されている。

■　ディレクトリ内にあるファイルの確認

ディレクトリの中にあるファイルの名前を一覧するには、コマンド ls （list の省略形と思えば覚えやすい）を使う。

書式： ls ［オプション］［ディレクトリ名、またはファイル名］

このコマンドの引数には、一覧を表示させたいディレクトリの名前を書く。名前は絶対パス名で書いても相対パス名で書いても良い。引数を省略すると、カレントディレクトリの中にあるファイルの名前の一覧が表示される（このテキストでは省略可能な引数を大括弧［ ］で囲んで示す。）

引数にファイル名を指定した場合、そのファイル名と一致するファイルが表示される。ディレクトリ名、ファイル名ともに複数指定することもでき、二種類を混在させることもできる。また、オプションに-l をつけるとファイルの様々な属性を表示させることができる(後述)。

■ カレントディレクトリの移動

自分のいる場所から別のディレクトリに移動する（カレントディレクトリを変更する）には、コマンド cd（change directory の頭文字）を使う。

書式： cd ［移動先ディレクトリ名］

たとえばホームディレクトリからその下の Library に移るには、

cd Library を実行する。

ディレクトリの指定を簡単に行うために、UNIX には次の３つの記号が用意される。

記号	説明
.	カレントディレクトリ
..	カレントディレクトリのひとつ上のディレクトリ
~	ホームディレクトリ

これらの記号を使うと作業がはかどる。たとえばひとつ上のディレクトリに移るには「cd ..」を実行すればよく、３つ上のディレクトリには「cd ../../..」で移ることができ、ホームディレクトリには「cd ~」で戻れる。（cd は引数なしで実行するとホームディレクトリに戻るが、これは cd ~と同じこと。）

これは、MacOSX の Finder や windows におけるフォルダを開く作業に相当する。

■ ディレクトリの作成

ディレクトリを作るには、コマンド mkdir （make directory の意味）を使う。

書式： mkdir 作成するディレクトリ名

引数で指定するディレクトリ名は、相対パス名か絶対パス名で記述する。ディレクトリ名を複数指定したばあい、複数のディレクトリが同時にできる。MacOSX の Finder や windows において、新しいフォルダの作成に相当する。

■ ディレクトリの削除

ディレクトリはコマンド rmdir（remove directory の意味）を使って削除することができる。ただし、ディレクトリの中にファイルが残っている場合には、削除できない。そのときには、後述の rm をオプション-r とともに用いる。

書式： rmdir 削除するディレクトリ名

削除するディレクトリ名も相対パス名か絶対パス名で表記する。この場合も複数のディレクトリを指定できる。

では、X11 アイコンをクリックして xterm を起動し、実際に試してみよう。

```
% cd                #ホームディレクトリに戻る
```

```
% mkdir 3rdweek  #ディレクトリ3rdweekを作る
```
Finderからホームディレクトリをチェックしてみよう。1stweekがFinderからもフォルダとして見えるはずである。
```
% cd 3rdweek/    #ディレクトリ1stweekに入る
% ls              #lsしても何もない（まだディレクトリの中が空だから）
% mkdir test     #ディレクトリtestを作る
% ls
test  #testの存在を確認
% cd test  #testの中に移る。
% ls      #testの中には何もない。
% cd ..     #一つ上のディレクトリに移る
% rmdir test  #ディレクトリtestを削除
% ls      #何もなくなった
```

■　ファイルのコピー

ファイルの複製を作るには、コマンド cp（copy の意味）を使う。

書式：　cp　コピー元のファイル名　コピー先のファイル名

ただし２番目の引数としてコピー先のファイル名のかわりに既存のディレクトリの名前を指定すると、そのディレクトリの中に、コピー元のファイルの複製を作成する。このとき、複製のファイル名はもとのファイル名と同じものになる。コピー元は複数のファイルまたはディレクトリを指定できるが、コピー元を複数指定する場合は、コピー先は既存のディレクトリ名でなくてはならない。

ディレクトリを中身ごとコピーするには、以下のようにオプション-r を用いる。

書式：　cp -r コピー元のディレクトリ名　コピー先のファイル名

■　ファイル名の変更と保存場所の移動

ファイルの保存場所を変更したり名前を変更するには、コマンド mv（move の意味）を使う。

書式：　mv　操作元ファイル名　操作先ファイル名

このコマンドによってファイル名が変更されるか、それともファイルが移動されるかは、操作先ファイル名として何を指定するかによって変わる。操作先ファイル名が既存のディレクトリの名前だと、そのディレクトリの中に操作元ファイルを移動する。操作元ファイルがディレクトリの名前の場合、ディレクトリが中身ごと移動する。操作元は複数のファイルが指定できるが、その場合操作先はディレクトリでなくてはならない。

■　ファイルの削除

ファイルを削除するには、コマンド rm（remove の意味）を使う。

書式：　rm　削除するファイル名

UNIX では、いちど削除したファイルは、どのようなことをしても元に戻せない。消す前に、本当に不要なファイルかどうかを確認すること。UNIX のユーザーなら必ず何度か重要なファイルをこのコマンドで消したことがあるはずである。rm にオプション -i を付けて実行すると、本当に削除するか確認を求められるので、オプションなしの rm でいきなりファイルを消すより安全だが、ファイルが多い場合は面倒なことになる。以下のように-r オプションをつけて、ディレクトリ名を指定すると、ディレクトリを中身ごと消去する。rm も複数のファイル名やディレクトリ名を指定できる。

書式：　rm -r　削除するディレクトリ名

■　空ファイルの作成

書式：　touch ファイル名

ファイル名が存在しなければ、空のファイルを作成する。存在すれば、更新時刻をコマンド実行時刻に変更する。

```
xterm内で試してみよう
% cd
% cd 3rdweek
% touch abc.dat     #空ファイルabc.datを作成
Finderから1stweekの中をチェックして、abc.datができているのを確認しよう。
% ls
abc.dat
% cp abc.dat def.dat     #abc.datをdef.datにコピー
% ls
abc.dat def.dat
% mv def.dat ghi.dat     #def.datをghi.datに移動
% ls
abc.dat ghi.dat
% mkdir test
% cp abc.dat ghi.dat test  #abc.dat, ghi.datの二つのファイルをディレクトリtestにコピー
% cd test/
% ls
abc.dat ghi.dat     #testの中に二つのファイルがコピーされている
% cd ..
% ls
abc.dat ghi.dat test
% rm abc.dat  #abc.datを削除
% ls
ghi.dat test
% rm test/
rm: test/: is a directory   #-rオプションをつけないと、ディレクトリは削除できない。
% rm -r test
% ls
ghi.dat
```

■　コマンドの場所を表示

書式：　which コマンド名

Unix のコマンドは全て一つ一つ別のプログラムである。起動されるプログラムが実際にどこにあるのかを探すのが、which コマンドである。

例

```
% which ls
/bin/ls
```

プログラムファイル/bin/ls がコマンド ls によって起動されているのがわかる。

他に有用なコマンドが多数存在するが、本実習では紹介しきれない。興味があれば、
http://itpro.nikkeibp.co.jp/article/COLUMN/20070613/274690/
などのサイトを見て勉強してほしい。また、あるコマンドの使い方を詳しく知りたいときは man コマンドを使うこともできる。

書式:　man コマンド名

例:

```
% man ls
LS(1)                    BSD General Commands Manual                    LS(1)

NAME
     ls -- list directory contents

SYNOPSIS
     ls [-ABCFGHLPRSTW@abcdefghiklmnopqrstuwx1] [file ...]

DESCRIPTION
     For each operand that names a file of a type other than directory, ls
     displays its name as well as any requested, associated information.  For
     each operand that names a file of type directory, ls displays the names
     of files contained within that directory, as well as any requested, asso-
     ciated information.

     If no operands are given, the contents of the current directory are dis-
     played.  If more than one operand is given, non-directory operands are
     displayed first; directory and non-directory operands are sorted sepa-
     rately and in lexicographical order.

     The following options are available:

     -@      Display extended attribute keys and sizes.
:
```

man の画面では、上下の矢印でスクロールする。q を押すと終了する。

１－４　xterm 上のディレクトリと Finder で見えるフォルダの関係

１－１で述べたように、Finder からはいくつかのディレクトリは見えない。また、見えていても名前が異なっている場合がある。主な対応は以下の通り。ただし、~はホームディレクトリを表す。まず、Machintosh HD の下。

xterm	Finder
/	Machintosh HD
/Applications	アプリケーション
/System	システム
/Users	ユーザ
/usr, /var, /bin, /dev など	見えない

次に、ホームディレクトリの下

xterm	Finder
~/Desktop	デスクトップ
~/Documents	書類
~/Downloads	ダウンロード
~/Library	ライブラリ
~/Movies	ムービー
~/Music	ミュージック
~/Pictures	ピクチャー
~/Public	パブリック
~/Sites	サイト

１－５　シェル

UNIX で OS の核となるソフトウェアをカーネル(kernel)と呼ぶ。UNIX はカーネルと、その上で動作する数多くのプロセス（プログラム）で構成される。MacOSX も同様である。

カーネルはディスクやメモリの管理、周辺機器などのハードウェアの制御、プロセスの管理、プロセス間またはプロセスとハードウェアの間の情報通信処理など、OS の基本的な機能を提供するものである。

ユーザとカーネルの間の情報のやりとりは「シェル」と呼ばれるプログラムが担当している。シェル(shell)は日本語では「貝殻」を意味するが、これは OS の中心部（カーネル）をおおって対話的に扱えるようにするもので、ユーザから見るとカーネルの外側の殻にあたる（図１）。シェルはユーザからの入力を受け取ってそれを解釈してカーネルにプロセス実行などの処理の指示を出し、その結果をユーザに返す機能をもつ。

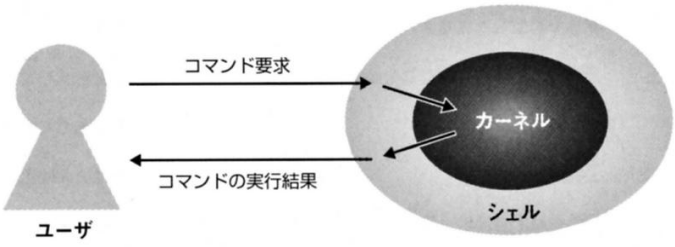


図１　シェルとカーネル
シェルはユーザとカーネルの間の仲介役をしている。

UNIX には数種類のシェルがあり、この実習では、tcsh を使う。プロンプト記号はシェルによって異なり、先ほどのプロンプトの最後にある「%」は C シェルあるいは tcsh のプロンプト記号である。また、このプロンプト記号が出ている行をコマンドラインと呼ぶ。xterm を立ち上げると、その上で tcsh が立ち上がる。複数のターミナル上の tcsh はそれぞれ独立に動く。

また、tcsh もコマンドの一つに過ぎず、次のようにして xterm 上から新たな tcsh を起動することもできる。

```
% tcsh
%
```

tcsh 起動後、コマンドを入力した場合、そのコマンドは新たに起動した tcsh が処理を担当する。

１－６ コマンドライン編集とコマンドライン補完、ヒストリ機能

UNIX でコマンドライン上にタイプしたコマンドは、return キーを押して実行する前であれば、自由に編集することができる。

コマンドライン補完機能を使うと、コマンドをタイプするのが楽になる。途中までコマンドやファイル名を打ち込んで tab を押すと、それまで打ち込んだ内容と一致する候補が一つであれば勝手に補完してくれるし、複数存在すれば、候補のリストを表示してくれる。

また、tcsh や bash には、前に実行したコマンドを呼び出す「ヒストリ機能」がある。上向き矢印キー（↑）を押すと、前に実行したコマンドをひとつずつさかのぼってコマンドラインに表示する。呼び出されたコマンドはそのまま実行することはもちろん、修正して実行することができる。戻りすぎた場合には下向き矢印キー（↓）でひとつ先に進む。

さて、ここでコマンドライン補完機能を体験してみよう。

/opt/local/Library/Frameworks/Python.framework/Versions/2.5/lib/python2.5/site-packages/matplotlib/mpl-data/matplotlibrc を自分のホームディレクトリにコピーすることを考える。

```
% cp /o
```

まで打って、tab を打つと、

```
% cp /opt/
```

と補完される。これは、ルートディレクトリ/の下には、o で始まるのが/opt しかないためである。この調子で、さらに続けて l を打ち、

% cp /opt/l
としたところで tab を打つと、
% cp /opt/local
と補完される。もし、複数候補がある場合には、候補のリストが表示されるので、それを見ながら打ち込んでいけば良い。

課題１ コマンドライン補完機能の習得 (提出不要)

実際に、コマンドライン補完を使って、
/opt/local/Library/Frameworks/Python.framework/Versions/2.5/lib/python2.5/site-packages/matplotlib/mpl-data/matplotlibrc
を、自分のホームディレクトリにコピーせよ。

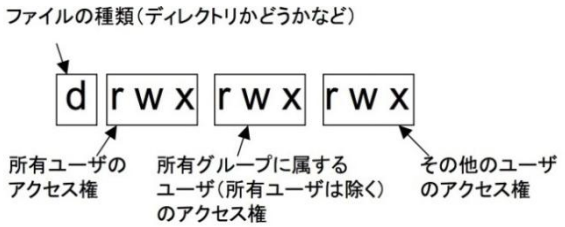
１－７　ファイルとディレクトリの属性

UNIX のファイルとディレクトリは、所有者、更新された日付などの属性(attribute)を持っている。「ls -l」コマンドを実行すると、たとえば以下のように、カレントディレクトリのファイルやディレクトリの属性が表示される。

```
% ls -l
total 1312
-rw-r--r--  1 seimei  staff     197 Jul 19 12:18 1PNE.xtal
-rw-r--r--  1 seimei  staff  345789 Jul 24 18:49 2BTF.pdb
-rw-r--r--  1 seimei  staff  106511 Jul 25 12:10 2BTF_chainP.pdb
-rwxr-xr-x  1 seimei  staff     184 Jul 24 11:48 pdbset.com
-rw-r--r--  1 seimei  staff  202528 Jul 24 11:35 r1pnesf_freeRflag.mtz
drwxr-xr-x  2 seimei  staff      68 Sep  5 19:51 unix_training
```

行単位にファイルやディレクトリの属性が表示されており、行の左から、モード（最初の１０個の文字)、リンク数、所有者、グループ名、ファイルの大きさ、更新時刻、名前の順で並んでいる。

上記「-rw-r--r--」のような１０個の文字で表される「モード」はファイルの型とファイルへのアクセス権を表示するものである。これは UNIX がマルチユーザシステムであることに関係しており、UNIX では「誰が」「何を」することができるかをファイルごとに設定することができる。



いちばん左の記号はファイルの種類を表している。この記号が「d」であればディレクトリ、「-」であれば通常のファイルを意味する。

次の３つの記号（左から２番目から４番目までの記号）は、そのファイルの所有ユーザのアクセス権である。「r」「w」「x」はそれぞれ読み込み、書き込み、実行の権限を表しています。これらの記号が書かれていればそのアクセスは許可されており、これらの記号を書く位置に「-」が書かれていればそのアクセスは禁止されている。たとえば、上記の 2BTF.pdb の左から 2~4 番目の記号は「rw-」だから、このファイルの所有者 seimei は、このファイルの読み込みと書き込みはできるが、このファイルをコマンドとして実行することはできない。モードの左から 5~7 番目

の記号はファイルを所有するグループに属するユーザ（ただし所有ユーザは除く）のアクセス権を、モードの左から 8~10 番目の記号はその他のユーザのアクセス権を表す。

■　アクセス権の変更

ファイルを所有するユーザであれば、アクセス権を変更することができます。アクセス権の変更にはコマンド chmod（change mode の略）を使う。

書式：　chmod　セット　ファイル名

chmod は引数で与えた「セット」の表記にもとづいて、引数で指定したファイルのアクセス権を変更する。「セット」では現在のアクセス権のうち修正すべき箇所を３種類の記号（ユーザの区分、アクセスを許可するか禁止するか、アクセスの種類）で表記する（アクセス権全体を２進数のビット列と考え、それを８進数３桁の数字で表記するやり方もあるが、ここではその説明は割愛する）。まず、対象とするユーザの区分を u（ユーザ）、g（グループ）、o（その他）の組み合わせ（または a（全ユーザ））で記述し、次に、アクセスを許可する場合は「+」、禁止する場合は「-」を記述する。最後にアクセスの種類を r（読み込み）、w（書き込み）、x（実行）の組み合わせで記述する。書き込みとは、そのファイルを変更したり、削除したりすることである。実行とは、それをプログラムとして実行するということである。プログラムは全て実行可能の属性を持たなくてはならない。ただし、ディレクトリの場合、実行可能とは、そのディレクトリにカレントディレクトリを移せるという意味になる。これら３種類の記号は、間に空白を入れずに連続して記述する。たとえば先程の 2BTF.pdb というファイルをその他に書き込み禁止にするには

```
% chmod o-w 2BTF.pdb
```

を実行すればよい。また、ユーザ区分を省略した場合、全ての人に対するアクセス権を変更する。

```
% chmod -w 2BTF.pdb
```

であれば、自分を含む全てのユーザーが、このファイルを変更したり削除したりできなくなる。

１－８　ワイルドカード

cp や rm を用いる場合、特定の拡張子のものだけをコピーしたり削除したりしたい場合は多い。その場合はワイルドカードである、*や?を用いる。

１－８－１ ?は任意の一文字を表す

たとえば、file?.dat とすると、fileA.dat, file0.dat、filep.dat のように、?の位置に任意の一文字が入るファイルを指定できる。?を複数入れることもでき、file.???の場合は、file.ext, file.dat, file.txt, file.csv など最後が任意の三文字のファイルを指定できる。

１－８－２ *は任意の文字列を表す

＊は 0 文字以上の文字列を表す。file*.dat とすると、file.dat, file12.dat, filesajieyhawr.dat など、最初が file で最後が.dat の全てのファイルを指定できる。?と同時に使うこともでき、?file*.dat ならば、0fileabcde.dat のように、file の前に一文字あって、最後が.dat であるファイルを指定できる。

１－８－３ [...]は指定文字のいずれかを表す。

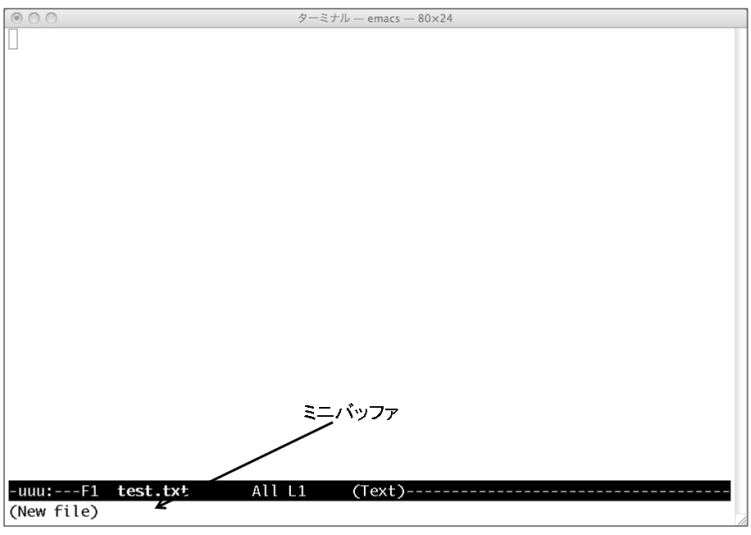
指定した文字のいずれかを含んだファイルを表したい場合には、[...]を利用する。たとえば、[ABC]であれば、A,B,C のいずれか一文字を表す。また、[0-9]なら、0 から 9 の数字のいずれか、[A-G]なら A,B,C,D,E,F,G のいずれかというような指定も可能である。[A-Za-z]ならば、すべてのアルファベットを表す。

例:
abcde.py
abcde.c
fgcde.py
abcdf.py
abddf.py
の５つのファイルが存在するとしよう。
rm abc*
とすると、abc*は、abc でスタートするすべてのファイル名と一致するから、abcde.py, abcde.c, abcdf.py の三つが消去される。
rm abc*.py とすると、消去されるのは、abcde.py, abcdf.py である。
rm *cde*とすると、abcde.py, abcde.c, fgcde.py が消去される。
rm abcd?.py であれば、 abcde.py と abcdf.py が消去される。
rm ab[cd]df.* なら、三番目の文字が c か d のいずれかのときに一致するので、 abcdf.py と abddf.py の二つが消去される。

参考文献:
http://itpro.nikkeibp.co.jp/article/COLUMN/20070514/270907/?ST=lin-beginners

２　テキストエディタ

ここまでで xterm 上におけるファイルの扱い方の基本を学んだ。ではいよいよプログラムを書くためのテキストエディタの使い方を学ぼう。Microsoft Word のようなワードプロセッサ（ワープロ）で作成する文書には文字とそのレイアウト（文字の形や色、大きさ、行の間隔、段落の配置など）の情報が含まれる。これに対してテキストファイルは文字情報だけで構成されるファイルである。ただし、「改行」のような一部のレイアウト情報は、文字のひとつとして扱うことができる)。プログラムを書くときは必ずテキストファイルで書かれるし、Unix 系 OS においては OS やユーザーのあらゆる設定はテキストファイルで書かれている。 テキストファイルはテキストエディタとよばれるコマンドを使って作成、編集する。このテキストエディタに何を使うかでプログラミング環境は大きく異なる。慣れているようであれば、今後の実習では、MacOSX 標準で付属するテキストエディタである“テキストエディット”を使っても良いが、今回は Unix 系システムで事実上の標準エディタの一つ emacs(もう一つは vi)を紹介しよう。emacs はプログラミング中に括弧の対応やインデントをチェックしてくれるので、非常に便利である。emacs の起動方法は、



書式: emacs ファイル名
である。ファイルが存在

していればそのファイルを開き、存在していなければ新しいファイルを作成する。
% emacs test.txt
とすると前ページ図のようになる。Emacs では、キーボードショートカットで様々な操作を行う。一番下のミニバッファが重要な役割をはたす。

２－１　Undo をする

Control キーを押しながら_を押す。
以後コントロールキーと同時に何かキーを押す場合、C-キーと表記する。Undo の場合は、C-_である。

２－２　ファイルをセーブする

C-x C-s
すなわち、Control キーを押しながら x を押した後、Control キーを押しながら s を押す。素早く押す場合は、Control キーを押しっぱなしにして、xs とタイプすれば良い。

２－３　別のファイルを開く

C-x C-f
ミニバッファに Find file:と表示されるので、その後に読み込みたいファイル名を入力する。ここでは tcsh と似た tab キーによる補完が使える。途中までファイル名を入力して tab を押すと、候補が一つしかない場合は補完され、複数ある場合は、ウィンドウが上下に分割され、下側のウィンドウに候補が表示される。

２－４　ファイルを別名で保存する

C-x C-w
ミニバッファに保存先を入力する。C-x C-f と同様の補完が使える。

２－５　文字列をカットする。

カットしたい文字列の最初にカーソルを合わせ、control キーを押しながらスペースキーを押す。次にカットしたい文字列の最後の文字の次の位置にカーソルを合わせ、C-w
ただし、MaxOSX では、Control キーを押しながらスペースキーを押すという動作に別の役割が当てられているので、まずそれを変更しなくてはないらない。変更方法は 2-15 を参照。

２－６　文字列をコピーする。

マウスで文字列を選択するだけでよい。

２－７　文字列をペーストする。

中ボタン。ただし、挿入場所はマウスポインタがある場所ではなく、カーソルがある場所になる。

２－８　特定の行に飛ぶ

ESC を押した後、g を二回押す。ミニバッファに Goto line: と表示されるので、そのあとに、行きたい行番号を入力する。

２－９　検索する

C-s
ミニバッファの中に I-search: が表示されるので、その後に検索ワードを入力。Emacs の検索はインクリメンタルサーチと言って、一文字入力するごとにそれまでの入力に一致する語句を検索する。同じ条件で次の語を検索する場合はもう一度 C-s を押せば良い。入力なしで次の語が検索される。

２－１０　置換する

エスケープキーを押した後%
ミニバッファの中に Query replace: と表示されるので、その後に置換前の語を入力してリターンキーを押す。その後に with: と表示されるので、置換語の語を入力してリターンキーを押す。すると、最初の置換候補にカーソルが飛ぶ。ここで、それを置換したければ y を、したくなければ n を押す。押すと、また次の置換候補にカーソルが飛ぶ。これをファイルの最後まで繰り返す。途中でやめたければ、C-g を押す。

２－１１　先頭、最後に飛ぶ

エスケープキーを押したあと、<を押すと先頭へ、
エスケープキーを押したあと、>を押すと最後へ飛ぶ。

２－１２　一画面ずつ上下する

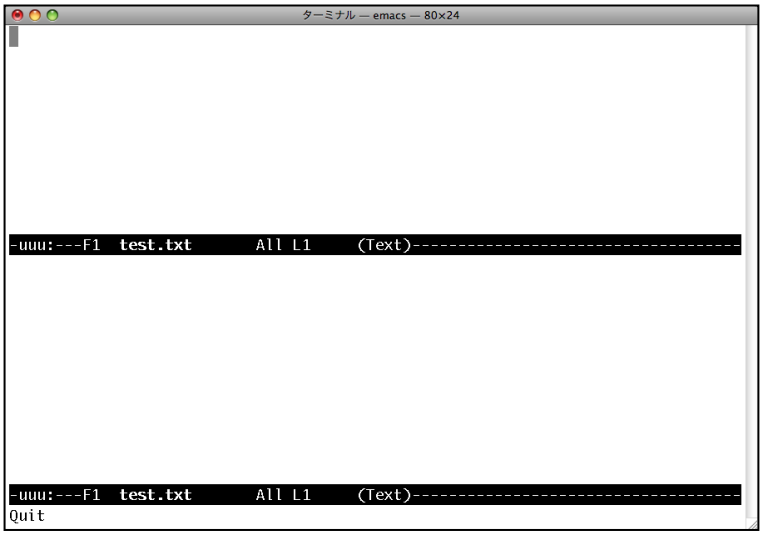
一画面分下に進むときは C-v。一画面分上に進むときは、エスケープキーを押した後 v を押す。

２－１３　終了する

C-x C-c
変更後セーブされていないときは、ミニバッファの中でセーブするか聞いてくるので、y か n で答える。

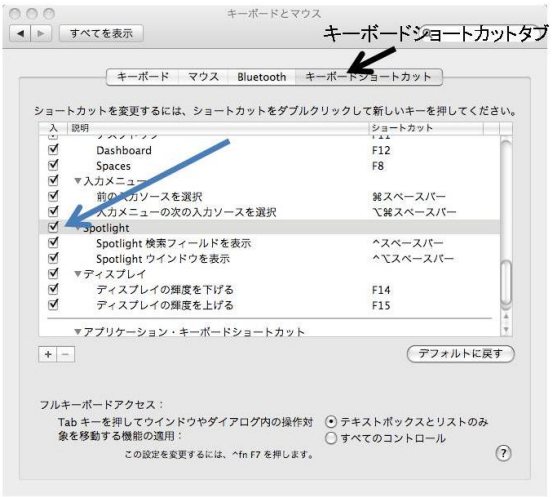
２－１４　トラブル時には

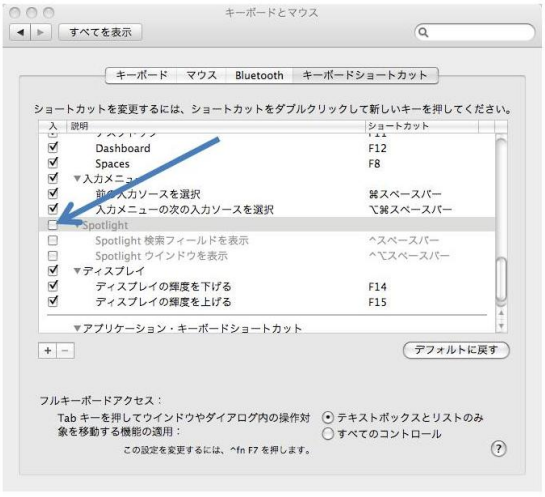
何か困ったときは、C-g を何回か押せばたいてい普通の状態に戻る。C-g は現在の操作を中止するという意味である。右図のように意図せずウィンドウが分割されてしまったときは、C-x 1 とすれば良い。



参考文献
http://sourceforge.jp/magazine/09/04/06/1138226

２－１５　カットための設定変更





Emacs では、文字列のカットにおいて、６－５のように control キーを押しながらスペースキーを押すという動作が必要とされるが、これは MacOSX では Spotlight という別のアプリケーションの起動に割り振られている。このままでは不便なので、MacOSX の設定を変更する。第一回でマウス中ボタンの設定を変えたときと同様に、システム環境設定のウィンドウの中の”キーボードとマウス”をクリックする。すると、キーボードとマウスの設定画面がでてくるのでキーボードショートカットタブ(左上図)をクリック。キーボードショートカットのリストがでてくるので、スクロールして、Spotlight の項を探す。そのチェックを解除して(右上図)、ウィンドウを閉じれば終了。

課題２ テキストエディタ習得

提出不要

インターネットから適当な英文を test.txt にコピーし、2-1 から 2-15 の操作をすべて試せ。

３　基本コマンド続き

■　テキストファイルの表示

書式：　cat　ファイル名

テキストファイルの表示を行う。もし、ファイルが長い場合は上のほうは切れてしまうので、その場合は cat のかわりに less を使うほうがいいだろう。

書式：　less　ファイル名

less でテキストを表示中は、上下の矢印でスクロールする。q を押すと終了する。

使用例

```
% cat test.txt
```

または

```
% less test.txt
```

■　シンボリックリンクの作成

書式：　ln -s　リンク元　リンク先

シンボリックリンクを作成する。シンボリックリンクは、ウィンドウズにおけるショートカットや MacOS におけるエイリアスと似たもので、あるファイルやディレクトリに別の名前を与えて、ユーザーやプログラムがその名前をファイル本体と同様に扱える仕組みである。たとえば、A というテキストファイルがあったときに、

```
% ln -s A B
% cat B
```
とすると、A の内容を見ることができる。しかし、ファイル B はコピーではなく、リンクである。したがって、A の内容が変われば cat B で見られる内容も変わる。-r オプションは使えないが、他は cp と同じように使える。A がプログラムであれば、A を B という名前のプログラムとして起動できる。ディレクトリであれば、B に入ることで、A の中のファイルにアクセスできる。

課題３: シンボリックリンク

提出不要
```
% ln -s test.txt test2.txt
```
とし、test.txt を emacs で変更したときに、test2.txt も変更されることを確かめよ。

４ 環境変数と.cshrc.local

ls のようなコマンドを入力したときに、コンピュータはどうやってその場所を探すのだろうか？ファイルシステム全体を探していたら時間がかかりすぎる。実際にはコンピュータはサーチするディレクトリのリストを持っていてその中から探すのである、どこから探すのかは、echo $PATH と入力するとわかる。
```
% echo $PATH
/mdhome/media/center/share/bin:/mdhome/share/Darwin/bin:/usr/bin:/bin:/usr/sbin:/sbin
:/usr/local/bin:/usr/X11/bin:/opt/local/bin:/opt/local/sbin:/usr/local/win/bin:/usr/l
ocal/bin:/bin:/usr/bin:.:/mdhome/media/center/cmd
```
:で区切られたディレクトリのリストが示されている。このリストの前から順番に検索して、最初にヒットしたプログラムを用いているのである。このコマンドを探すためのパスの範囲をサーチパスと呼ぶ。ここで、echo $PATH は、環境変数 PATH の中身を表示せよという意味である。環境変数とは、システムの挙動を定義する変数のことで、ユーザーが変更することもできる。他にホームディレクトリの位置を示す HOME, OS の種類を表す OSTYPE, ホストの名前を表す HOSTNAME など多くの変数が存在する。環境変数名の前に$をつけると、コマンド実行時に、変数の中身におきかわって処理される。
たとえば、
```
% cd $HOME
```
とするとホームディレクトリに戻ることができる。ちなみに、これは
```
% cd
```
と同じ動作になる。tcsh の場合は、
```
% setenv 環境変数名 セットする内容
```
とすることで、環境変数を置き換えることができるが、シェルが立ち上がるたびにリセットされてしまう。環境変数を常に変えておきたい場合には、シェルが立ち上がるときに読み込む設定ファイルを用いる。tcsh の場合はホームディレクトリ上の.tcshrc がそれにあたる(最初の.もファイル名に含まれる)。tcsh が起動するとまず.tcshrc の中が実行され、そのなかに環境変数の設定があれば適用される。しかし、実習用のコンピュータでは.tcshrc が書き込み禁止になっており、かわりに.cshrc.local を用いる。.cshrc.local は、.tcshrc の中から呼び出されるようにすでに設定されている。.cshrc.local は最初は存在しないので、テキストエディタを用いて作成する。たとえば、自作のプログラムを入れるディレクトリを~/com (ホームディレクトリ下の com という意味) にして、ここをまっさきに検索させたい場合は、テキストエディタを用いて、この.cshrc.local の中に
```
setenv PATH ~/com:$PATH
```
と書けばよい。ただし、.cshrc.local の最後には必ず改行を入れておくこと。この行で、いままでのサーチパスの先頭に~/com を足して、これを新たなサーチパスに設定している。こうすると、~/com の中に自作プログラム等を入れておけば、そのプログラムはどのディレクトリからでも実行できるようになる。注意としては、この設定はシェルが立ち上がるときに読み込まれる。そのため、.cshrc.local を設定したあと、最初に起動した tcsh からその設定が反映される。つ

まり、xterm の上でアップルキー+N を押して新しい xterm を起動するか(このとき新しい tcsh が新しい xterm の上で起動する)、xterm 上で
% tcsh
%
のようにして、新しい tcsh を起動しないと、設定の変更は反映されない。

課題４: 環境変数と.cshrc.local

提出不要
ディレクトリ~/com を作成せよ。また、テキストエディタを用いてホームディレクトリに.cshrc.local を作成し、その中に、
setenv PATH ~/com:$PATH
と書け。もし、すでに.cshrc.local が存在している場合は、それを開き、最後にこの行を加えよ。新たに xterm(または tcsh)を起動し、echo $PATH を実行して、PATH に~/com が含まれていることを確認せよ。

補足 １ .で始まるファイルの表示

.tschrc や、.cshrc.local のような”.”(ドット)で始まる名前を持つファイルはオプションなしの ls では表示できない。この場合は、ls の代わりに ls –a を用いる。-a の a は all の略で、全てのファイルを表示するという意味である。たとえば、
% mkdir test
% cd test
% touch .tcshrc
% ls
%　　#何も表示されない。
% ls –a
.　　..　　.tcshrc　　#.tcshrc が表示された。
　　#カレントディレクトリ”.”、親ディレクトリ”..”も表示されている。

補足２　.cshrc.local の設定がうまくいかないとき

綴り等内容に間違いがない場合、以下の二つのどちらかが原因である場合が多い。

１　.cshrc.local の最後に改行が入っていない。

.cshrc.local の最後には改行が必要である。これが入っていないと、最後の行の内容が反映されない。たとえば、以下のように cat で.cshrc.local の中身を表示したときに、
% cat .cshrc.local
setenv PATH ~/com:$PATH%
のように中身のすぐあとに改行なくプロンプトが出る場合は、最後に改行が入っていない。この場合は、emacs で.cshrc.local を開き、最後の行の最後にカーソルを持って行って、リターンキーを押して保存すればよい。
% cat .cshrc.local
setenv PATH ~/com:$PATH
%
のように、cat で表示した際に、プロンプトの前に改行が入っていれば大丈夫。

２ 全角文字が入っている。

一般に UNIX の設定ファイルには全角文字を入れてはいけない。ことえり等漢字変換ソフトが起動したまま入力すると、全角文字として入力される。必ず英数キーを押して、半角文字で入力すること。

第四回 プログラミング事始め I

今回の目的

ごく簡単なプログラムが書けるようになること。

１　プログラム入門

ここまでで、python を便利な電卓として使ったり、turtle で絵を描くことができるようになった。しかし、より複雑なことをコンピュータにやらせようと思ったら、プログラムを組むことは避けて通れない。ここで、まず python のごく簡単なプログラムをくんで実行してみよう。テキストエディタを用いて、以下のような内容を持つ hello.py を作成する。以降プログラムコード（プログラムを表現するテキスト）は、網掛けと囲み線で表現することにする。網掛けと囲み線の中だけをテキストエディタで入力すればよい。

hello.py:
```
#!/usr/bin/env python
a=”Hello, world!”
print a
```

一行目は、このファイルは python によって実行することを指定している。一般に unix 系の OS では、テキストファイルの最初に、
#!プログラム名
とすると、それ以降のテキストをすべてプログラム名のプログラムで実行する。このとき、プログラム名の指定は絶対パス名にしなくてはならない。しかし、プログラムが置かれている絶対パスはシステムによって異なるので、/usr/bin/env を使うやりかたが開発された。ここではあまり難しく考えずに、#!/usr/bin/env プログラム名とすると、そのプログラムでそれ以降のテキストを実行すると考えて良い。ここに別の言語、たとえば#!/usr/bin/env perl や#!/usr/bin/env ruby とすると、そのファイルは指定した言語(perl や ruby)によって実行される。本実習ではすべて python を使うので、プログラムの先頭に必ず#!/usr/bin/env python を入れる。

二行目以降は、python のプログラム。二行目で、a に”Hello World!”を代入し、三行目で a の内容を print を用いて表示している。これをセーブしたあと、hello.py が存在するディレクトリで、コマンドプロンプトから
```
% chmod +x hello.py
```
を実行し、**hello.py をプログラムとして実行可能にする**(前回テキストを参照)。**今後作成したプログラムはかならずこのように実行可能にすること。そうしないと動かない。**これで、プログラム hello.py の実行準備は完了。実際に実行してみる。

```
% ./hello.py
Hello, world!
```

のように、Hello, world!という文字列が表示されたはずである。

ここで、print は、
print　変数または数値、文字列
で、変数や文字列の内容を適切に表示してくれる命令である。カンマで区切って、複数の変数や文字列、数値を表示することもできる。たとえば、
print 1,”test”,2 とすると、
出力は、

1 test 2
になる。

Python の対話モードでは、
```
>>>a=”Hello, world!”
>>>a
‘Hello, world!’
```
のようにただ変数名だけを入力するとその中身を表示してくれたが、プログラムにおいては、明示的に print 文を用いないと表示してくれない。
hello.py にもう少し命令を追加してみよう。

hello.py:
```
#!/usr/bin/env python
a=″Hello, world!″
print a
b=2+3j
c=2-3j
print b*c
```

最後の３行を追加した。これを実行すると、出力は、
```
Hello, world!
(13+0j)
```

プログラムを書くと、このように複数の命令を上から順番に自動的に実行してくれる。

２　環境変数 PATH とプログラムの実行

プログラムがカレントディレクトリに存在していれば、
```
% ./hello.py
```
で実行できるが、離れたディレクトリに居る場合、いちいちプログラムが存在するディレクトリを指定するのは不便である。よく使うプログラムは、第３回で述べた環境変数 PATH に含まれたディレクトリに置いておくと、ディレクトリ指定なしで実行できる。第３回では、PATH に ~/com を足したので、たとえばそこにコピーすれば、どこからでも簡単に hello.py を実行できる。試してみよう。
```
% cp hello.py ~/com
% cd
% cd Movies    #Movies には hello.py は無い。
% hello.py      # ディレクトリ指定なしでの実行
Hello, world!
(13+0j)
```
うまく実行できた。

３　コマンドラインからの変数入力

命令文の羅列でプログラムを書くことができるが、それだけだと毎回同じ処理しかできない。プログラムを使う際に何らかのパラメータを入力するのが普通である。
もっとも簡単なのはコマンドライン引数を用いる方法である。コマンドライン引数とは、コマンドラインからプログラムを実行するときに、プログラム名のあとに続けるパラメータのことである。たとえば、
```
% cp hello.py hello2.py
```
とした場合、hello.py と hello2.py は cp のコマンドライン引数である。このコマンドライン引数を python で用いるためには、sys モジュールの sys.argv を用いる。まずは、サンプルプログ

ラム args.py を見てみよう。

args.py:
```
#!/usr/bin/env python
import sys
print sys.argv
print 0,sys.argv[0]
print 1,sys.argv[1]
print 2,sys.argv[2]
print 3,sys.argv[3]
```

これを以下のように実行してみる。
```
% chmod u ./args.py #args.py を実行可能にする。
% ./args.py 1 2 ab
```

結果は、
['./args.py', '1', '2', 'ab']
0 ./args.py
1 1
2 2
3 ab
のようになるだろう。この例を詳しく見てみよう。プログラム二行目で sys モジュールをインポートしている。これによって、sys.argv が使用可能になる。sys.argv はリストで、sys.argv[0]にはプログラム自身の名前‘./argv.py’が、sys.argv[1]には一番目の引数'1'が、sys.argv[2]には二番目の引数‘2’が、以降 sys.argv[n]には n 番目の引数がすべて文字列として格納されている。ここで、sys.argv の要素はすべて文字列なので、足し算のプログラムを作ろうとして、

add.py:
```
#!/usr/bin/env python
import sys
print sys.argv[1]+sys.argv[2]
```

のようなプログラムを作り、
```
% add.py 1 2
```
を実行すると、出力は
12
となる。これは、sys.argv の要素が文字列であるため、'1'と'2'を文字列として足して'12'が出力されたためである。正しい足し算のプログラムを作るには、三行目は
```
print float(sys.argv[1])+float(sys.argv[2])
```
としなくてはならない。こうすると、正しく 3.0 が出力される。関数 float は、文字列や整数を実数型に変換する関数で、第一回の実習で紹介した。

課題１ 割算プログラム
div.py 数値 1 数値 2
と実行すると数値 1/数値２が出力されるプログラム div.py を作れ。また、その実行例をレポートせよ。

４ if による条件分岐とブロック文

たいていのプログラムは入力やプログラム内部の状態によって実行内容を変化させなくてはならない。ある場合には、A を処理し、そうでない場合は B を処理するというような条件による処理の変化を条件分岐と呼ぶ。python では他のほとんどの言語と同様に、if 文を用いる。まず、以下のようなテストプログラム if.py を作ってみよう。

if.py:
```
#!/usr/bin/env python
import sys
a=float(sys.argv[1])    #コマンドライン引数を実数に変換
if a>0:     #もし、a が正ならば、下の二行を実行。そうでなければ下の二行は実行しない。
        print "Positive"
        print  "Message 1"
print  "End"
```

if a>0:のあと二行の print の前の空白(インデント)は、完全に長さを一致させること。空白の長さ自体は何文字分でも良いが、見やすさを考えるとある程度長い方が良い。スペースキーではなく、tab キーを使うと、容易にインデントを一定に保てる。このプログラムは、
if.py 数値
とすると、数値が正なら”Positive”と”Message1””End”と表示する。そうでなければ”End”とだけ表示する。
たとえば、

```
% ./if.py 1.0
Positive
Message 1
End

% ./if.py -1.0
End   #”Positive”と”Message1”は表示されない。
```

if 文のもっとも簡単な使い方は、
if 条件式 1:
        ブロック 1
である。条件式 1 が満たされれば、ブロック 1 を実行、そうでなければブロック 1 は実行されない。ブロックとは文の集合のことである。同じインデント幅(行頭の空白)をもった文の集合が 1 ブロックとみなされる。通常は、tab を使って、インデントをつくる。if.py の場合、ブロック 1 は

```
        print "Positive"
        print  "Message 1"
```

である。また、条件式 1 は、a>0 である。まず、入力した数値を 4 行目で実数型に変換し、その結果を代入した a が正であればブロック 1 を実行、そうでなければブロック 1 は実行しない。このブロックはインデント幅だけで定義されるので、インデント幅がブロックの中で違うとエラーになる。たとえば、if.py のなかの

```
        print "Positive"
        print  "Message 1"
```

を、

```
        print "Positive"
         print "Message1"
```

のように、インデント幅を変えてしまうとエラーになる。この仕様は python 独自のもので、慣れないとわかりにくいところである。あとでもう少し詳しく見てみよう。

また、ブロック文のなかにさらに if 文を入れて多重ブロックを作ることもできる。if.py の 8 行目と 9 行目の間に三行挿入して、以下のようにする。

if.py:
```
#!/usr/bin/env python
import sys
a=float(sys.argv[1])
```

```
if a>0:
        print "Positive"          #if a>0:のためのブロック文。a>0 なら実行
        print "Message1"        #同上
        if a>10:                  #同上
                print "Over 10"  # if a>10:のためのブロック文 a>10 なら実行
                print "Large!"   #同上
print "End"
```

ここで、if a>10:は if a>0:のブロック文の一部で、print “Positive”などと同じインデント幅を持つ。if a>10:のためのブロック文は、if a>10:よりも大きなインデント幅をもたなくてはならない。ここでは、tab を二回打つことによって、必要なインデント幅を作る。
このプログラムの動作は、入力した数字が 0 以上ならば

```
        print "Positive"
        print "Message1"
```

を実行。さらに 10 以上ならば

```
                print "Over 10"
                print "Large!"
```

を実行する。理屈より習うより慣れろである。まずは使ってみよう。

課題 2　if をつかったプログラミング

入力した値が負であれば、”Negative”と表示し、-20 以下であればそれに加えて”Below -20”と表示するプログラム negative.py を書き、その実行結果をレポートせよ。
プログラムを書き始めるとエラーはつきものである。本テキストの最後に、デバッグ(プログラムミス修正)について簡単に書いたので、そちらも見て欲しい。

５　条件式と True, False

ここでは、if 文の中で用いる条件式をもう少し詳しく見てみよう。条件式には以下の比較演算子が用いられる。
A > B　A が B より大きければ真
A < B　A が B より小さければ真
A >= B A が B 以上であれば真
A <= B A が B 以下であれば真
A == B A と B が等しければ真
A != B A と B が等しくなければ真

ここで、A==B は、A=B ではないことに注意。A=B では B に A を代入するという意味になる。また、二つ以上の条件式を組み合わせるには、and, or を用いる。
A == B and C == D　A と B が等しく、C と D が等しければ真
A == B or C == D A と B が等しいか、C と D が等しければ真
また、条件式の真偽を入れ替える not も使える。
not (A==B)　A と B が等しくなければ真

たとえば、
not ((A > B) and (C < D)) であれば、(A>B かつ C<D)が満たされないときに真になる。

また、シークエンス(リスト、文字列など、複数の要素をひとまとめに扱う変数型の総称)のための比較演算子に、
A in B　シークエンス B の中に要素 A が存在すれば真
A not in B シークエンス B の中に要素 A が存在しなければ真。not (A in B) と同じ
というものもある。比較演算子を用いた条件式は真のときに”True”、偽のときに”False”を返し、”True”のときだけ if 文はそのブロック文を実行する。

具体的に見てみよう。以下の compare.py を作成してほしい。この compare.py には、4 つの引数をそれぞれ a, b, c, d に代入し、a>b, a<=b, a==b, a!=b, not ((a>b) and (c<d)), a in [b, c, d], a not in [b, c, d]の各条件式の結果を表示するプログラムである。

compare.py

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

a=float(sys.argv[1])
b=float(sys.argv[2])
c=float(sys.argv[3])
d=float(sys.argv[4])

print "a>b",a>b
print "a<=b",a<=b
print "a==b", a==b
print "a!=b", a!=b

print "not ((a>b) and (c<d))", not ((a>b) and (c<d))
print "a in [b,c,d]",a in [b,c,d]
print "a not in [b,c,d]",a not in [b,c,d]
```

実行すると以下のようになる。

```
% ./compare.py 2 2 3 4
a>b False
a<=b True
a==b True
a!=b False
not ((a>b) and (c<d)) True
a in [b,c,d] True
a not in [b,c,d] False
```

また、比較演算子は文字列に対しても使える。以下のような comparestr.py を作って実行してみよう。comparestr.py は数値のかわりに三つの文字列を引数にとり、それぞれ a,b,c に代入する。

comparestr.py:

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

a=sys.argv[1]
b=sys.argv[2]
c=sys.argv[3]

print "a>b",a>b
print "a<=b",a<=b
print "a==b", a==b
print "a!=b", a!=b
print "a in c",a in c
print "a not in c",a not in c
```

実行例:

```
% ./comparestr.py  ba ab base
a>b True
a<=b False
```

```
a==b False
a!=b True
a in c True
a not in c False
```

文字列同士の大小比較は、辞書順に行われる。後のものほど大きいと判断される。また、数字はアルファベットより小さく、大文字は小文字よりも小さい。
また、A in B は、文字列の場合、文字列 A が文字列 B に含まれていれば真になる。

６　True と False, if 文の真実

さて、python の対話モードで少し奇妙な計算をしてみよう。

```
>>> a=0
>>> b=1
>>> a>b
False
>>> a<b
True
>>> (a<b)*1
1
>>> (a>b)*1
0
```

この一連の計算から、True は整数 1 で False は整数 0 の別表現であることがわかる。
では、if 文に評価式でなく、直接数値を渡したらどうなるだろう？以下のプログラム ifcheck.py を書いて実行してみよう。意味は見ればわかると思うが、1, 0.1, -1, 0.1, 0 のそれぞれが真ならばその数字を表示したあとに”True”を表示するプログラムである。

ifcheck.py:

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

a=1
b=0.1
c=-1
d=-0.1
e=0

print a
if a:
        print "True"
print b
if b:
        print "True"
print c
if c:
        print "True"
print d
if d:
        print "True"
print e
if e:
        print "True"
```

実行結果

```
% ./ifcheck.py
1
True
0.1
True
-1
True
-0.1
True
0
```
最後の 0 以外は、すべて真と判断されている。つまり if 文は評価式が 0 であれば偽、そうでなければ真と判断しているだけである。このことを利用してプログラムをシンプルに書けることがあり、実際のプログラミングで頻繁に用いられている。

7　ブロック文の詳細

ブロック文は以下のルールに従う。

- 字下げが同じになっている文は、そのブロックが終わらない限り、すべて同じブロックに属するとみなされる
- ブロックの終わりは、より字下げの少ない文を書くことによって表現する
- ブロックの中で作られるブロックは必ず、もとのブロックより大きく字下げされる。

たとえば、

```
x=1
if x:
        y=2
        if y:
                print 'block2'
        print 'block1'
print 'block0'
```

のようなコードであれば、

```
x=1     #ブロック 0
if x:  # if 文によるブロック 1 のスタート(if 文自体はブロック 0)
        y=2   #ブロック 1
        if y: #if 文によるブロック 2 のスタート(if 文自体はブロック 1)
                print 'block2'  #ブロック 2
        print 'block1' #ブロック 1、字下げの少ない文が入ることでブロック 2 は終了
print 'block0' #ブロック 0、字下げの少ない文が入ることでブロック 1 は終了
```

8　while 文を使ったループ

何回も同じことを繰り返すことは、人間にとっては苦痛だがコンピュータにとってはなんでもないことである。この繰り返しをコンピュータにさせるためのしくみをループ構造と呼ぶ。Python においては、for 文と while 文が存在する。ここではまず while 文を解説する。while 文は、条件式が真の間、while によって生成されるブロックを繰り返す。

```
while 条件式:
        ブロック
```

以下のプログラム while.py は while を使って入力した正数までの階乗を計算するプログラムである。

while.py

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

```
n=int(sys.argv[1])
i=2
answer=1
while i<=n:
        answer=answer*i
        i=i+1 # i に i+1 を代入することで、i に 1 を足す。=が等号だと考えると数学的に間違っているが、=は等号ではなく、代入記号なので、このような表記が可能である。
print answer
```

実行例
```
% while.py 10
3628800
```

プログラムを解説すると
```
import sys  #sys モジュールのインポート
n=int(sys.argv[1])     #コマンドラインの第一引数を整数型に変換して n に代入
i=2    #i に 2 を代入
answer=1   #answer に 1 を代入
while i<=n:             #i が n 以下の間、下のブロックを実行
        answer=answer*i     #answer に i を掛ける
        i=i+1      #i に 1 を足す。最終的に i が n より大きくなれば、ループを抜ける
print answer  #answer を表示
```

while を二重にすることもできる。

$$\prod_{i=1}^{i\leq n}\sum_{j=1}^{j\leq i} j$$

を計算する while2.py を作ってみよう。
while2.py n
のように使い、n は正の整数である。

while2.py
```
#!/usr/bin/env python
import sys

n=int(sys.argv[1])

i=1
answer=1

while i<=n:
        sum=0
        j=1
        while j<=i:
                sum=sum+j
                j=j+1
        answer=answer*sum
        i=i+1
print answer
```

使用例
```
% while2.py 4
180
```

９　turtle モジュールとプログラミング

第二回で取り上げた turtle モジュールであるが、当然ながらプログラム内でも用いることができる。まずは次のプログラム turtletest.py を作り、実行してみよう。見ればわかると思うが、turtle を前へ 100 進ませるプログラムである。

turtletest.py
```
#!/usr/bin/env python
import turtle
kame=turtle.Turtle()
kame.forward(100)
```

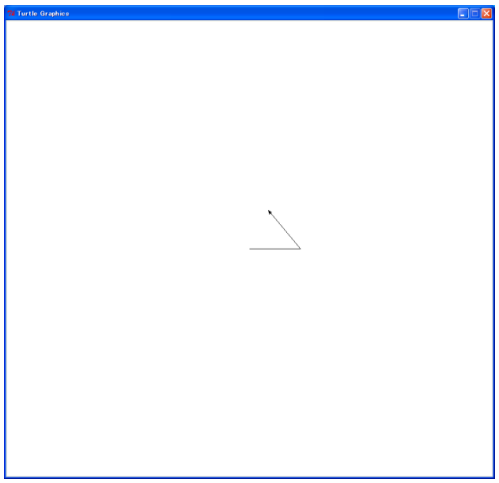
実行すると turtle が現れて進むが、終了するとウィンドウが消えてしまう。これでは困るので、最後に一行加えたのが、turtletest2.py である。

turtletest2.py
```
#!/usr/bin/env python

import turtle
kame=turtle.Turtle()
kame.forward(100)

raw_input('Press return key.') #この行を追加。
```

最後の行に用いた raw_input は、ユーザーからの入力をプログラムに伝えるための関数である。これを実行すると、turtle が動いたあと xterm 上に
Press return key.
と表示され、xterm 上でリターンキーを押すまで turtle のウィンドウが消えない。raw_input について、詳しくは次回学ぶので、今回はそういうものだと思ってもらいたい。もう一つ turtle の例を挙げよう。次の angle.py は引数で指定した角度をなす二本の直線を描く。

angle.py
```
#!/usr/bin/env python

import turtle
import sys

angle=float(sys.argv[1])　　#最初の引数(sys.argv[1])を実数型に変換

kame=turtle.Turtle()
kame.forward(100)
kame.left(180-angle)　#turtle が 180-angle だけ回転することによって、二本の直線のなす角は angle になる。
kame.forward(100)

raw_input('Press return key.')
```

```
# angle.py 50
Press return key.
```



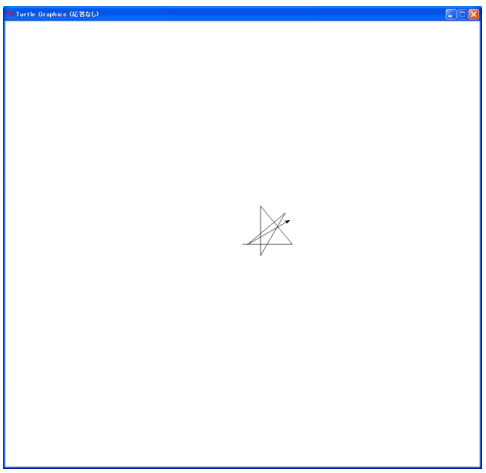
もちろん、while ループと組み合わせることもできる。つぎの anglewhile.py は、angle を 10 度づつ減らしていきながら、angle が 0 以下になるまで直線を引く。

anglewhile.py

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

import turtle
import sys

angle=float(sys.argv[1])

kame=turtle.Turtle()
kame.forward(100)

while angle > 0:
    kame.left(180-angle)
    kame.forward(100)
    angle = angle – 10
raw_input('Press return key.')
```

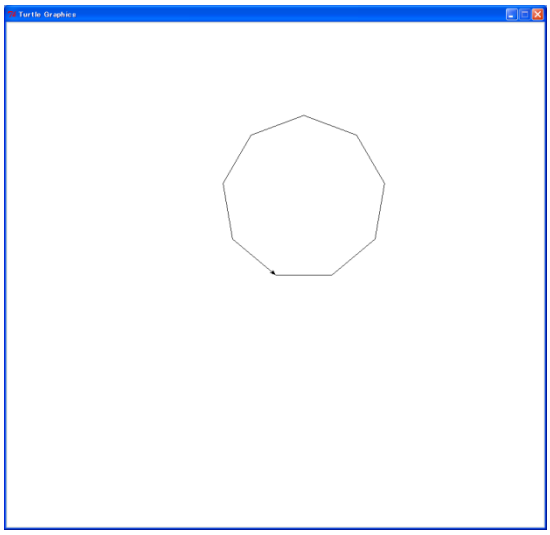
使用例

```
# anglewhile.py 50
Press return key.
```



課題 3 while 文

turtle モジュールを用いて、正 n 角形を描くプログラム polygon.py を作れ。n と各辺の長さは引数として指定されるものとする。実行例もレポートせよ。



使用例（右図）

```
# polygon.py 9 100   #一辺の長さ 100 の正 9 角形の描画をする。
Press return key.
```

補遺１　デバッグ

プログラムを書き始めるとプログラムの間違いに起因する多くのエラーメッセージをどうしても目にするようになる。このプログラムの間違いをバグ（虫）と呼び、その間違いを修正することをデバッグ（虫取り）と呼ぶ。

ここで、いくつかの典型的なエラーメッセージとその対処法について述べる。

SyntaxError

文法エラー。最も良く目にするエラー。プログラムが、Python の文法と一致しない。たとえば以下のプログラム error0.py を実行すると、

error0.py

```
#!/usr/bin/env python
a=1
if a==1
        print a
```

```
% error0.py
  File "./errortest.py", line 5
    if a==1
          ^
SyntaxError: invalid syntax
```

となる。これがエラーメッセージである。エラーメッセージ一行目をみると、errortest.py の五行目に文法エラーがありそうなのがわかる。エラーメッセージ三行目の^の位置がエラーがありそうな場所を示しており、この場合、a==1 の 1 のあたりが怪しいということになる。この場合、if の行の最後に:が無いために、エラーになっている。

5 行目を

```
if a==1:
```

とすれば、エラーは出なくなる。

全角と半角

特殊なプログラミング言語（なでしこ http://nadesi.com/ や、ひまわり http://kujirahand.com/himawari/など）をのぞいて、ほとんどのプログラミング言語は、文字列の中以外は全て ASCII 文字と呼ばれる、半角英数字と基本的な半角記号(括弧やバックスラッシュ、カンマなど）で書かれる。ことえりや ATOK が起動している場合、標準で全角文字が入力されるが、この全角文字がプログラムの中に混ざるとエラーになる。特にわかりにくいのが、全角スペースが混入している場合である。以下は、今回取り上げた加算プログラム add.py を作ろうとして失敗した例である。

add.py:

```
#!/usr/bin/env python
import sys
import pylab
print  float(sys.argv[1])+float(sys.argv[2])
```
どこが悪いのかまったくわからないと思うが、これを実行すると以下のようになる。
```
% ./add.py 1 2
File "./add.py", line 4
SyntaxError: Non-ASCII character '¥xe3' in file ./add.py on line 4, but no encoding
declared; see http://www.python.org/peps/pep-0263.html for details
```
このエラーは4行目にASCII文字では無い文字が混ざっていることを示している。実際、４行目のprintとfloatの間のスペースが全角であり、それがエラーの原因である。このようなエラーがでた場合は、その行に全角文字が混ざっていることを疑おう。

IndentationError

インデントエラー。インデント幅が正しくない。たとえば、以下のプログラム error1.py を実行すると、

error1.py
```
#!/usr/bin/env python
a=1
 if a==1:
        print a
```

```
% error1.py
  File "./errortest.py", line 5
    if a==1:
    ^
IndentationError: unexpected indent
```
となる。5 行目の if のところにエラーがありそうと言っている。実際、この場合には、if の行頭に空白があるのが、エラーの原因である。

NameError

変数名、関数名に関するエラー。たとえば、定義していないリストに対する代入や append などによって起こる。以下のプログラム error2.py を実行すると、
```
#!/usr/bin/env python
a=1
if a==1:
        b.append(a)
        print b
```

```
% error2.py
Traceback (most recent call last):
  File "./errortest.py", line 6, in <module>
    b.append(a)
NameError: name 'b' is not defined
```
6 行目の b が定義されていないと言っている。この場合、b に一度も代入していないので、まだ b が生成されていない。b を使用する前に `b=[]` として、b に空リストを代入しておくとエラーは無くなる。

また、定義していない関数やモジュールの呼び出しによっても起こる。

nameerror.py:
```
#!/usr/bin/env python
a=math.exp(10)
print a
```

これを実行すると、以下のようになる。
```
% ./nameerror.py
Traceback (most recent call last):
  File "./nameerror.py", line 3, in <module>
    a=math.exp(10)
NameError: name 'math' is not defined
```
この例では、math が定義されていないと言っている。プログラムを見ると、math モジュールが import されていない。最初に
```
import math
```
を加えれば、エラーは出なくなる。

エラーの出ないバグ
プログラムの文法上は正しくても、期待したように動かない場合がある。この場合はエラーメッセージが出ないので、デバッグは少し難しくなる。この場合、プログラムの途中に print 文を入れて、変数の変化を追うことが有効である。無限ループの例で説明してみよう。

無限ループ
while 文においては、バグによって、永久にループを抜け出せなくなることはわりと良くある。この場合、プログラムがいつまでも終わらない。
たとえば、while.py を作ろうとして、最後から二行目の i=i+1 を忘れたとする。
whileerror.py
```
#!/usr/bin/env python
import sys

n=int(sys.argv[1])
i=2
answer=1
while i<=n:
        answer=answer*i
print answer
```

これを実行するといつまでもプログラムが終わらない。**その場合は control キーを押しながら c を押すと停止する。**停止したらプログラムを見直してみよう。その際に、プログラムの途中に print 文で変数を表示させるのが有効である。

whileerror.py
```
#!/usr/bin/env python
import sys

n=int(sys.argv[1])
i=2
answer=1
while i<=n:
        answer=answer*i
        print "i", i,"answer", answer, "n",n     #追加
print answer
```

追加行で、プログラムの中で起こっていることを知るために、i と answer と n を表示させている。
これを実行すると、
```
% whileerror.py 3
i 2 answer 2 n 3
i 2 answer 4 n 3
i 2 answer 8 n 3
```

```
i 2 answer 16 n 3
i 2 answer 32 n 3
i 2 answer 64 n 3
i 2 answer 128 n 3
i 2 answer 256 n 3
i 2 answer 512 n 3
i 2 answer 1024 n 3
```
以下無限に続く
となり、i がいつまでも 2 のままであることがわかる。これから、i を増加させるのを忘れていたことが一目でわかる。このように、プログラムの途中に print 文を用いて変数を表示させてプログラム実行中の動作を追うのは、デバッグに非常に有効である。

Permission denied

権限がない。プログラム実行時にでる場合は、プログラムが実行可能になっていない場合が多い。たとえば、実行可能でないプログラム env.py を実行しようとすると、以下のように、
```
% ./env.py
./env.py: Permission denied.
```
のように、Permission denied が起こる。この場合は、
chmod +x プログラム名
を実行して、実行可能にすること。
また、プログラム一行目が間違っている場合にも起こる。本来 python プログラムの一行目は、
```
#!/usr/bin/env python
```
にならなくてはならないが、これが、
```
#!/usr/bin env python
```
になっていてりすると、プログラムが実行可能であっても Permission denied が起こる。

No such file or directory

プログラム名が間違っている場合が大半だが、プログラム一行目が間違っている場合にも起こる。たとえば、python プログラム一行目が、
```
#!/usr/bin/env pythn
```
や、
```
#!/usr/bin/env /python
```
になっていたりすると、たとえプログラム名があっていても以下のようにこのエラーになる。
env.py の一行目が以上のようなものであれば、
```
% ./env.py
env: /python: No such file or directory
```
のようにエラーになる。この場合は一行目を正しく、
```
#!/usr/bin/env python
```
とすれば良い。

補遺２　ディレクトリ

プログラムが増えてくると、だんだん整理が難しくなってくる。そのときにはディレクトリをうまく活用すればよい。たとえば、第四回用のディレクトリとして、
```
% mkdir fourth
```
のようにディレクトリ fourth をつくり、そのなかで第四回の実習のファイルの作成をすれば、ずいぶんとすっきりするだろう。

補足　モジュールとプログラム名

python では、sys や math など様々なモジュールを呼び出すが、そのモジュールの実態は sys.py や math.py のような python プログラムである。詳しくは第 7 回で学ぶが、**作成したプログラ**

ムと同じディレクトリに sys.py や math.py があると、sys や math のモジュールがうまく呼び出せない。呼び出すモジュールと同じ名前のプログラムを作らないように注意しよう。

第五回　プログラミング事始め II

今回の目的

プログラムの基本がひととおりわかるようになること。今回は前回で説明できなかったプログラムの基本を学ぶ。

１　for を使ったループ

前回はwhileを使ったループを学んだ。今回はもう一つのループ構造であるfor文から始めよう。for は以下のように使う。

for 変数 in シークエンス:
　　ブロック
シークエンスは、リストのように、多くの要素をひとまとめに扱う変数型の総称である。文字列もシークエンスである。シークエンスの中の要素を順番にひとつずつ変数に代入し、ブロックを実行する。たとえば、以下のプログラムを for.py として作成し、実行してみよう。ファイルを実行可能にするのを忘れずに。

for.py
```
#!/usr/bin/env python
a=[2,3,4,5]
for x in a:
        print x,x**3
print "out of loop"
```

実行すると、
```
% ./for.py
2 8
3 27
4 64
5 125
out of loop
```

この例では、リスト[2,3,4,5]の中から順番に一つずつとりだして、x に代入する。その上で、x と x の三乗を print 文で表示している。
文字列に対しても同じようにできる。以下のような forstr.py を作る。

forstr.py
```
#!/usr/bin/env python
import sys
a=sys.argv[1]
for x in a:
        print x
```

使用法は、
forstr.py 文字列
与えられた文字列から一文字ずつ取り出して表示する。たとえば
```
% ./forstr.py string
```

s
t
r
i
n
g
のようになる。for 文は while 文で書き換えることができる。たとえば forstr.py と同じ動作をするプログラムを while 文で書き換えるとすると、
whilefor.py
```
#!/usr/bin/env python
import sys
a=sys.argv[1]
i=0
while i<len(a):
        print a[i]
        i=i+1
```

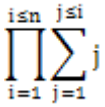
for で書いた場合に比べて若干複雑になる。len(a)は a の要素の数(この場合は文字数)を返す(第１回の実習で紹介した)。

２　range と for の組み合わせ

for 文は python においては range と一緒に使われることが多い。range は、整数列のリストを作成する(第１回のテキストを見よ)。たとえば、range(1,10)は、1 から 9 までのリスト、
[1, 2, 3, 4, 5, 6, 7, 8, 9]
を作る。以下の forrange.py を作って実行してみよう。
forrange.py
```
#!/usr/bin/env python
for i in range(1,5):
        print i
```

実行結果
```
% ./forrange.py
1
2
3
4
```

これを使うとたとえば 0 から n までループするというようなことが簡単にできる。前回取り上げた、



を計算する while2.py も for と range を使ってよりシンプルに表現できる。

while2byfor.py
```
#!/usr/bin/env python
import sys
n=int(sys.argv[1])
answer=1
for i in range(1,n+1):
        sum=0
```

```
        for j in range(1,i+1):
                sum=sum+j
        answer=answer*sum
print answer
```

このプログラムは while2.py と全く同じ動作をする。

課題１ for と range

前回の課題３で作った polygon.py を for と range を使って書き直して、polygonfor.py を作れ。実行例もレポートせよ。

３ if をもう少し。 if-elif-else 構文

if 文は
if 条件式:
　　ブロック
で、条件式が真であればブロックを実行するという使い方はすでに学んだ。しかし、条件式が偽である場合に別の処理をしたい場合は多いだろう。そのために、elif と else が用意されている。構文は、
if 条件式 1:
　　ブロック 1　#条件式 1 が True なら実行
elif　条件式 2:
　　ブロック 2　　#条件式 1 が False で、条件式 2 が True なら実行
elif　条件式 3:
　　ブロック 3　#条件式 1,2 が False で、条件式 3 が True なら実行
..
else 条件式 4　　#それまでのすべての条件式が False なら実行
　　ブロック 4
のようになる。elif の数はいくつあっても良く、無くてもよい。else は構文の最後にひとつ入れるか、入れないかのどちらかである。いままで使ってきた if 文は、elif の数 0,else なしの場合に相当する。使用例を見てみよう。以下の elif.py を書いて実行してみよう。

elif.py
```
#!/usr/bin/env python
import sys
n=float(sys.argv[1])
if n>0:
        print "Positive"
elif n<0:
        print "Negative"
else:
        print "Zero"
```

これは、
elif.py 数字
とすると、数字が正か負か 0 かを表示するプログラムである。

実行例
```
% ./elif.py -1
Negative
```

課題２ if と else

math と cmath モジュールをインポートし、入力した数字が正であれば math.sqrt を用いてその平方根を、負であれば cmath.sqrt を用いてその平方根を、0 であれば Zero と表示するプログラ

ム sqrt.py を作れ。実行結果もレポートせよ。

ヒント:実行例
% sqrt.py -1
1j
% sqrt.py 0
Zero

4　break と continue

For 文や while 文が作るループの最中に、条件によってループをその場で終わらせたい場合がある。そのときに使えるのが break である。break 文は、実行されると、その時点で実行中のループの中で、最後にスタートしたループを終わらせる。以下の例 break.py を見てみよう。

break.py
```
#!/usr/bin/env python
import sys
n=int(sys.argv[1])
a=[1,3,5,7,9]
for i in a:
        if i==n:
                break
        print i
print "END"
```

このプログラムは、入力した数字がリスト a の中から取り出した要素と一致するまで、リスト a の要素を表示し、最後に”END”を出力する。
実行例
% ./break.py 5
1
3
END

実行例の場合、3 までは表示されるが、5 になると i==n が真になり、break で for i in a:のループが終了し、最後の文 print “END”が実行される。

一方、continue はブロックの残りを飛ばして、次の反復に移る。for ループを終了させることはない。break.py の break を continue に変えてみよう。

continue.py
```
#!/usr/bin/env python
import sys
n=int(sys.argv[1])
a=[1,3,5,7,9]
for i in a:
        if i==n:
                continue
        print i
print "END"
```

実行例
% ./continue.py 5
1

```
3
7
9
END
```
今度は 5 以外はすべて表示される。i==n が真になると、continue 文によってブロックの残り(print i)をとばして、次の繰り返しに進むから 5 だけが表示されないのである。

５　for, while 文の else

if だけでなく、for, while においても else が使える。これは先述の break と組み合わせることによって意味を持つ。while の場合には、

```
while 条件式:
        ブロック１
else:
        ブロック２
```
となり、条件式が満たされないときに、ブロック２が実行される。break がブロック１の中で実行されると、else 後のブロック 2 も無視される。
つまり、
```
while 条件式 1:
        ブロック１
        if 条件式 2:
                break
else:
        ブロック２
ブロック３
```
とすると、条件式 2 が満たされた場合、ブロック２も実行されずに一気にブロック３が実行される。

For の場合は、
```
for 変数　in シークエンス:
        ブロック１
else:
        ブロック２
```
となり、for のループが終わったあと、ブロック２が実行される。この場合も同様にブロック１の中で break が実行されると、ブロック２も実行されない。

以下の else.py で実際に見てみよう。
else.py
```
#!/usr/bin/env python
import sys

n=int(sys.argv[1])

a=[1,3,5,7,9]

for i in a:     #for 文の例
        if i==n:
                break    #break が実行されると、else のブロックもスキップされる。
        print i
else:
        print "Not Found (For)"  #break が実行されなかったときだけ実行される。
print "END(For)"
```

```
i=0   #while 文の例
while i < len(a):
        if a[i]==n:
                break  #break が実行されると、else のブロックもスキップされる。
        print a[i]
        i=i+1
else:
        print "Not Found (While)"  #break が実行されなかったときだけ実行される。
print "END(While)"
```

このプログラムは、前半が for と else の組み合わせ例、後半が while と else の組み合わせ例である。前半と後半は同じ動作をする。引数の数字がリスト a に含まれている場合:

```
% ./else.py 5
1
3
END(For)
1
3
END(While)
```

break が実行されると else 以降も実行されないので、”Not Found”は表示されない。

含まれていない場合:

```
% ./else.py 4
1
3
5
7
9
Not Found (For)
END(For)
1
3
5
7
9
Not Found (While)
END(While)
```

この場合は break が実行されないので、ループ終了後 else 以降が実行され、”Not Found”が表示される。

６　少し実際的なプログラム

ここまでで、リストの使い方(第 2 回)と、プログラムのためのすべての基本的な構文(第 4 回、第 5 回)を学んで来た。いままで学んで来たものを使ってもう少し実用的なプログラムの例を見てみよう。ここでは、ある正数 n を与えられたときに n 以下の素数を探索するプログラム prime.py を示す

prime.py

```
#!/usr/bin/env python
import sys
n=int(sys.argv[1])
prime=[]
for i in range (2,n+1):
```

```
        for p in prime:
                if (i % p == 0):
                        break
        else:
                prime.append(i)
print prime
```

実行例

```
% prime.py 100
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83,
89, 97]
```

このプログラムはいままで学んで来たものしか使っていない。i % p は i を p で割ったときの余りである。また、prime.append(j)はリスト prime に新たな要素 j を加えるという意味である。最初の prime=[]は、第１回で学んだように、代入によって変数は始めて生成する。存在しないリストに append はできないので、最初に空リストを代入することによってリスト prime を生成するのである。

課題３ 素因数分解

与えられた整数を素因数分解するプログラム factor.py を作れ。実行例もレポートせよ。
ヒント: 素因数分解のアルゴリズム
数値 n を素因数分解するとする。また、n 以下の素数のリスト prime が用意されているものとする。(prime は、prime.py と同様に作成できる。) また、約数はリスト factor に記録するとする。
1 while を使って、n が prime の中に存在しない(つまり、n が素数でない)間ループする。
2 for を使って prime のリストから順番に変数 p に取り出す。
3 n % p==0 であれば、p は n の約数。factor に p を追加。n=n/p として 2 のループを抜ける。
4　1 に戻り、n が prime の中に存在するかを再評価。存在すれば 5 へ。しなければループ続行。
5 factor に最後の約数として n を追加して、factor を表示して終了。
実行例

```
% ./factor.py 120
[2, 2, 2, 3, 5]
% ./factor.py 127
[127]
```

7　raw_input 文による変数入力

プログラムになんらかの入力を与える方法は、いままで使ってきた sys.argv を使ったコマンドライン引数だけではない。ここでは raw_input を使った変数入力について学ぼう。
先ほどの prime.py を、raw_input を使って書き換えたのが以下のプログラムである。

primeinput.py

```
#!/usr/bin/env python
input=raw_input('n=?')
n=int(input)
prime=[]
for i in range (2,n+1):
        for p in prime:
                if (i % p == 0):
                        break
        else:
                prime.append(i)
print prime
```

変更点は、2 行目と 3 行目だけである。これを実行すると、
```
% ./primeinput.py
n=?
```
とでてくる。ここで数字を入れてリターンキーを押すと、その数字に対する結果が表示される。
raw_input の書式は簡単で、
変数=raw_input(‘message’)
とすると、画面に message が表示されて、入力待ちになる。ユーザーが何か入力してリターンキーを押すとその内容が文字列として変数に代入される。
このリターンキーが押されるまで待つという動作を利用したのが、前回の turtle モジュール使用時に用いた
```
raw_input('Press return key.')
```
である。この場合変数が省略されているので、入力した内容は無視される。

課題４ raw_input

課題３のプログラムを raw_input を使って書き換えてレポートせよ。

第六回 データ入出力

今回の目的

データの入出力ができる。

１　ファイル入出力

コマンドライン引数や raw_input によって入力できるデータの量はさして多くない。自動計測によって得られたデータのような大きなデータに関しては、ファイルからプログラムに直接読み込まなくてはならない。また、結果もファイルに書き出したいだろう。今回は、そのための仕組みであるファイル入出力の基本を学ぶ。まず、簡単な例として、任意のテキストファイルをコピーするプログラム copytest.py は以下のようになる。

copytest.py
```
#!/usr/bin/env python
import sys

infile=sys.argv[1]
outfile=sys.argv[2]

fin=open(infile,'r')
fout=open(outfile,'w')

for line in fin:
        fout.write(line)

fin.close()
fout.close()
```

使用例
```
% ./copytest.py copytest.py test.py
```
とすると、copytest.py の内容が test.py にコピーされる。

　このプログラムにはファイル入出力のすべての基礎が入っている。ファイル入出力は、紙のファイルの保管庫での作業のアナロジーで考えるとわかりやすい。ファイルの中をどうにかするためには、まず目的のファイルを保管庫の棚からとってきて開かなくてはならない。これが open である。ファイルを開いたら、中の紙を読んだり書いたりすることができる。作業が終わったらファイルを閉じて、もとのところにもどす。これが close である。まず、open を見てみよう。
書式は、
変数=open(ファイル名，‘モード’)
である。open はファイルオブジェクトを生成し、左辺の変数に代入する。プログラム中のファイルに対する作業はすべてこのファイルオブジェクトに対して行われる。ファイルオブジェクトは、プログラムがファイルにアクセスするための窓のようなものである。モードには、書き込み用の w，読み込み用の r，元のファイルに追加する a，読み書き両方の r+の４つのモードがある。普通は r と w だけ使えれば良い。
```
fin=open(infile,'r')
```
は、変数 infile に書かれているファイル名のファイルを読み込みモードで開き、その結果できたファイルオブジェクトを fin に代入する。

```
fout=open(outfile,'w')
```
は、変数 outfile に書かれているファイル名のファイルを書き込みモードで開き、その結果できたファイルオブジェクトを fout に代入する。
読み込みモードで開いたファイルオブジェクトは、for 文の中ではファイルの中の一行を一要素とするシークエンスとして振る舞う。そのため、
```
for line in fin:
        fout.write(line)
```
では、fin から(つまり infile から)変数 line に一行ずつ読み込んでいることになる。ただし、ファイルオブジェクトにおいては、リストで行うように、fin[0]のようにインデックスで直接に要素(ここでは行)を呼び出すことはできない。
書き込みモードで開いたファイルオブジェクトには、以下のようにして書き込みができる。
ファイルオブジェクト名.write(文字列)
このようにすると、ファイルオブジェクトが示すファイルに文字列が書き込まれる。
```
fout.write(line)
```
は、fout に文字列 line を書き込む。二行全体では、fin から変数 line に一行づつ読み込んで、fout に line を書き込むという作業をしている。
最後に close をする。書式は、
ファイルオブジェクト.close()
である。実はプログラムの終わりにはすべてのファイルオブジェクトは自動的に close するのだが、大きなプログラムにおいてはファイルオブジェクト名が重なったり余分なメモリを消費するのを避けるためにこまめに close をするほうがいい。これも紙のファイルと同様である。ファイルを机の上に出しっぱなしではそのうち収集がつかなくなる。

課題１： ファイル入力テスト

提出不要
copytest.py の fout.write(line)の前に一行 print line を加えて実行して、for ループのなかで line の中身の変化を確かめよ。

２　ファイル出力のための型変更

ファイルオブジェクト.write によって、ファイル出力する場合、文字列しか書き出すことができない。たとえば、実数を出力しようとして、書き込み可能なファイルオブジェクト fout にたいし、
fout.write('1.0')
とすれば問題ないが、('1.0'は文字列)
```
fout.write(1.0)
```
とすると、以下のエラーがでる。
TypeError: argument 1 must be string or read-only character buffer, not float
これは、fout.write の引数は文字列しか使えないという意味のエラーである。実数型の変数の内容をファイルに書き出したい場合は、それを文字列に変換する必要がある。
たとえば、
```
a=1.0
fout.write(a)
```
であればエラーになるが、
```
a=1.0
fout.write(str(a))
```
として、str 関数(第１回参照)を用いて、a を文字列に変換してしまえば、書き出すことができる。
同様に、リスト型等他の型の場合でも、
a=[1,2,3,4]
fout.out(str(a))
のように str 関数を用いて書き出すことができる。

３　改行文字

　ファイルオブジェクト.write で文字列を書き出す場合、print 文とは異なり、改行は自動的に挿入されない。どういうことか、以下のプログラム writetest.py で見てみよう。このプログラムは、引数で指定したファイルに、1.0 と 2.0 を書き出すプログラムである。
writetest.py:
```
#!/usr/bin/env python

import sys

outfile=sys.argv[1]
fout=open(outfile,'w')

a=1.0
fout.write(str(a))
b=2.0
fout.write(str(b))
fout.close()
```

このプログラムを実行する。
```
% writetest.py test.out
```
できたファイル test.out をテキストファイルの中を見るプログラム cat (第一回参照)で見てみる。
```
% cat test.out
1.02.0
```
1.0 と 2.0 がつながってしまっているのがわかるだろう。これは、str(a)の文字列'1.0'の直後にそのまま str(b)の'2.0'の文字列が出力されたからである。これでは、多数のデータを書き出すとわけがわからなくなる。
　実は、python を含むほとんどのプログラミング言語で、「改行文字」というものが\n（バックスラッシュ+n、バックスラッシュは MacOSX ターミナル環境と Linux 環境、Windows+cygwin 環境では¥キーを押すことで入力される）として、定義されている。この改行文字を文字列の中に含めると、その文字列が表示されたり出力されたりするときに、そこで改行が行われる。Python の対話モードで確かめてみよう。
```
 >>> a='abc\nde'   #abc と de の間に改行文字
 >>> print a
 abc
 de
```

ファイル出力の際にも同様に、明示的に\n を出力させれば良い。前出の writetest.py の場合は、
```
fout.write(str(a))
```
を、
```
fout.write(str(a)+' \n')
```
とすることで、a の内容を出力したあとに改行を入れることができる。str(a)+' \n'は第１回で説明した文字列同士の足し算で、str(a)の最後に改行文字' \n'を足すという意味である。全体では、以下のように変える。
writetest.py:
```
#!/usr/bin/env python

import sys

outfile=sys.argv[1]
fout=open(outfile,'w')

a=1.0
```

```
fout.write(str(a)+'\n')   #変更
b=2.0
fout.write(str(b)+'\n')   #変更
fout.close()
```

変更後の実行例
```
% ./writetest.py test.out
% cat test.out
1.0
2.0
```
確かに、1.0 と 2.0 のあとに改行が入っていて見やすくなっている。

４　データ解析入門

ファイルの入出力の基礎ができたら、次は簡単なデータ解析をしてみよう。まずデータ解析のための簡単なデータ test.dat をテキストエディタで用意しよう。

test.dat
```
2.5
3
2.1
1.2
-2
4.1
```

このような一行に数字が一つ書かれているファイルから、全体の平均値を求めるプログラム avr.py は下のようになる。

avr.py
```
#!/usr/bin/env python
import sys

infile=sys.argv[1]
outfile=sys.argv[2]

fin=open(infile,'r')
fout=open(outfile,'w')

sum=0 #sum はデータの合計値
n=0 #n はデータの数
for line in fin:
        sum = sum + float(line)   #sum に読み込んだデータを足していく
        n=n+1   #データを読み込むたびに、n に 1 を足す。
avr=sum/n   #合計値 sum をデータ数 n で割って平均を求める。
out="avr: "+str(avr) +'\n'
fout.write(out)

fin.close()
fout.close()
```

使用例
```
% ./avr.py test.dat testavr.dat
% cat testavr.dat
avr: 1.81666666667
```

となる。結果はファイル testavr.dat に記録され、それを cat(第三回参照)で表示させている。

課題　2：最小、最大値

test.dat を読み込み、その中の最小値と最大値、データの数を求め、表示する minmax.py を作れ。test.dat を使った実行例も示せ。

５　複数列の読み込み

次に入力データが複数列にわたる場合を考えよう。以下のようなデータ test2.dat を作成しよう。(具体的な数値はまったく同じでなくてもよい。)

test2.dat

```
100 100 20
-200 -100 50
150 -300 100
0 0 400
-400 0 10
```

この場合、一行に複数のデータが含まれているので、avr.py のときのように、読み込んできた行を line に格納して float(line)としてもうまく数値を取り出せない。その場合にはまず一行の中の数値を分けなければならない。この場合は文字列に対するメソッド split を用いる。使用方法は、各データの区切りがスペースかタブである場合には、

文字列名.split()

とすれば、各データをリストの要素としてとりだせる。Python の対話モードで少し実験してみよう。

```
>>> a="0.1 2.5 0.5"  #空白で区切られた三つの数値が入った文字列
>>> b=a.split()
>>> b
['0.1', '2.5', '0.5']
```

0.1, 2.5, 0.5 がリスト b の各要素として収まっているのがわかる。

では、この split を使って実際に test2.dat のような三列のデータを読み込むプログラム data3.py を作ってみよう。

data3.py

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

infile=sys.argv[1]

fin=open(infile,'r')

data0=[]
data1=[]
data2=[]

for line in fin:
        linedata=line.split()
        data0.append(linedata[0])
        data1.append(linedata[1])
        data2.append(linedata[2])
fin.close()

print "data0: ",data0
```

```
print "data1: ",data1
print "data2: ",data2
```

使用例:
```
% ./data3.py test2.dat
data0:  ['100', '-200', '150', '0', '-400']
data1:  ['100', '-100', '-300', '0', '0']
data2:  ['20', '50', '100', '400', '10']
```

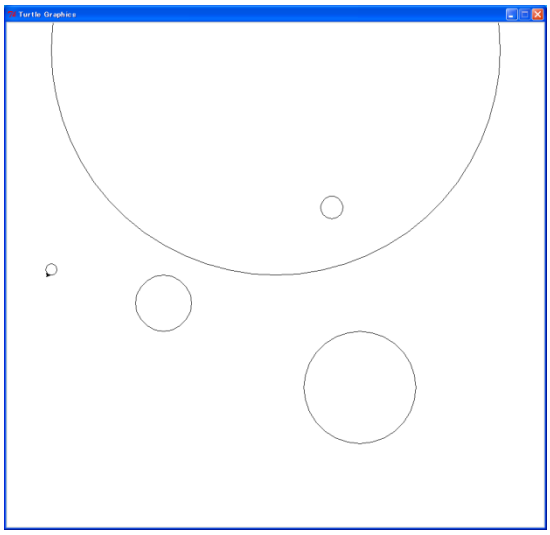
このプログラムでは、1 列目をリスト data0 に、2 列目をリスト data1 に、3 列目をリスト data2 に保存し、最後にそれぞれのリストを表示している。

課題　3　複数列データ読み込み

turtle モジュールを使い、test2.dat のデータの各行の最初の二つをスタート地点の x 座標、y 座標、三つ目を半径とする円を描くプログラム circles.py を書け。

使用例
```
# ./circles.py test2.dat
Press return key.
```

Turtle モジュールの up, down, circle, goto を用いると容易である。



第七回　関数とメソッド

今回の目的

ある程度大きな規模なプログラムを書く基礎がわかる。
同じプログラムコードを何度も書かなくて良くなる。

１　関数の基礎

プログラムを書いていると同じことを何回も書かなくてはならない場合がある。その場合は何回もでてくる一連の文をひとまとめにして関数を定義することでプログラムをシンプルにすることができる。いままで使ってきた len, range, int, float, str, print, open もすべて関数である。関数は以下のように定義する。

```
def 関数名（引数列):
        ブロック
```

Python においては、関数を使用する前に定義しなくてはならない。
いつも通りまずは簡単な例を見てみよう。以下の func.py を作る。

func.py
```
#!/usr/bin/env python
import sys

def sqradd(a,b):
        a=a*a
        b=b*b
        c=a+b
        return c

print "sqradd(1,2): ",sqradd(1,2)
x=1.5
y=2.5
print "sqradd(x,y): ",sqradd(x,y)," x= ",x,"y= ",y
a=-2
b=-3
print "sqradd(a,b): ",sqradd(a,b)," a= ",a,"b= ",b
```

この func.py の中で、
```
def sqradd(a,b):
        a=a*a
        b=b*b
        c=a+b
        return c
```
が関数の定義である。この場合、引数列は、a と b の二つである。sqradd(1,2)として呼び出す場合を例にとってその動作を説明しよう。まず、a に 1 が、b に 2 が代入されて関数の中の計算が行われる。このとき、変数 a と b は関数が呼び出されたときに生成し、関数の処理が終わると

消滅する。4行目で別の変数 c がでてくるが、これも関数の中だけの変数である。最後の return c で、c の内容が関数の結果として出力される。関数の中で生成される変数 a, b, c は、関数の外には影響を与えない。このような変数をローカル変数と呼ぶ。
sqradd(x,y)の場合は、a に x の中身が、b に y の中身が代入される。

このプログラムを実行すると以下のようになる。
```
% ./func1.py
sqradd(1,2):  5
sqradd(x,y):  8.5  x=  1.5 y=  2.5
sqradd(a,b):  13  a=  -2 b=  -3
```

きちんと関数内変数 c が return によって、関数から出力されているのがわかる。ここで、三つ目の結果を見てみよう。sqradd(a,b)を実行しても、実行後の a, b の内容は変化していない。これは、関数が呼び出されたときに生成される a,b は、関数の中だけの変数で、たとえ名前が同じでも関数の外の変数とは干渉しないことを示している。この性質のおかげで、関数の中で使われている変数名を、関数の外では気にする必要がない。
return で関数の外に返される変数は、実数でも文字列でもリストでも何でも良い。たとえば、プログラム中の
```
        return c
```
を
```
        return [a,b,c]
```
のようにリストに変えると、実行結果は
```
% ./func.py
spradd(1,2):  [1, 4, 5]
sqradd(x,y):  [2.25, 6.25, 8.5]  x=  1.5 y=  2.5
sqradd(a,b):  [4, 9, 13]  a=  -2 b=  -3 [5.0, 6.0, 11.0]
```
のようになり、関数が関数内変数[a,b,c]の内容をリストとして出力しているのがわかる。たとえば、sqradd(1,2)においては、関数スタート時には a=1, b=2 だったのが、
```
        a=a*a
        b=b*b
        c=a+b
```
の計算を経て、return の時点では、a=1, b=4, c=5 になっている。リストの中には異なる型のデータが混在していても良いので、このリストを return する方法を用いると、複数の変数の内容を無理なく出力することができる。実際に関数を使ってみよう。たとえば、あるリストの中の数値の平均を返す関数 average は、以下のようになる。
```
def average(inlist):
        sum=0
        for i in inlist
                sum = sum+i
        sum = sum/len(inlist)
        return sum
```
これを使って、前回で取り上げた avr.py と同じ動作をするプログラム avrfunc.py を作ると、

avrfunc.py
```
#!/usr/bin/env python
import sys

def average(inlist):  #関数 average の定義
        sum=0
        for i in inlist:
                sum = sum+float(i)
        sum = sum/len(inlist)
        return sum
```

```
infile=sys.argv[1]
outfile=sys.argv[2]

fin=open(infile,'r')
fout=open(outfile,'w')

data=[]
for line in fin:
        data.append(line)
avr=average(data)  #平均値の計算。関数が定義してあると、一行で計算が終わる！
out="avr: "+str(avr) +'\n'
fout.write(out)

fin.close()
fout.close()
```

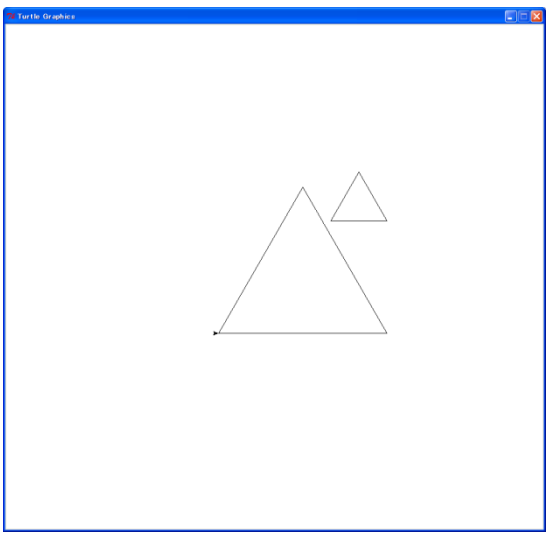
前回の avr.py と同じ動作をすることを確認してみよう。

1-2 turtle モジュールを使った例

さて、また turtle モジュールを使って練習してみよう。三角形を描く関数 triangle を作ってみる。

triangle.py

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

import turtle
import sys

def triangle(tur, x, y, length):
    tur.up()
    tur.goto(x,y)
    tur.down()
    tur.forward(length)
    tur.left(120)
    tur.forward(length)
    tur.left(120)
    tur.forward(length)
    tur.left(120)


kame=turtle.Turtle()
triangle(kame, 100,100,100)
triangle(kame, -100,-100,300)

raw_input('Press return key.')
```



turtle.Turtle()で生成される turtle も変数であり、関数の引数として用いることができる。

使用例:

```
#./triangle.py
Press return key.
```

課題１　多角形

triangle の代わりに正多角形を作る関数

polygon (tur, n, x, y, length)
を定義せよ。ここで、tur は turtle、n は生成する正多角形の頂点の数、x, y は始点の位置、 length は一辺の長さである。また、関数 polygon を使って triangle.py のような簡単なプログラム polygonfunc.py を書け。その実行例も報告せよ。

２　モジュール

汎用的な関数は、一つのプログラムだけではなく、いくつものプログラムで使いたい。そのときに、それぞれのプログラムの中で def を用いて定義するのは、煩雑なだけでなく、もし将来その関数にバグが見つかった場合、その修正はほとんど不可能になる。そのために、python では、関数や後述のメソッドを複数のプログラムから呼び出すための仕組みとしてモジュールを用意している。いままで、math, cmath, sys などのモジュールを使ってきた。今回はモジュールの使い方、作り方について学ぶ。モジュールの作り方は簡単である。先ほどの関数 average をモジュールから使えるようにするためには、モジュール名を liststat とすると、ファイル名 liststat.py で、以下のようなファイル

liststat.py
```
def average(inlist):
        sum=0
        for i in inlist:
                sum = sum+float(i)
        sum = sum/len(inlist)
        return sum
```

を作れば良い。liststat.py と同じディレクトリで python の対話モードで試してみよう。
```
>>> import liststat
>>> list=[1,2,3,4]
>>> liststat.average(list)
2.5
```

このように、自作モジュール liststat が import され、その中の関数 average が、liststat.average として呼び出せることがわかる。一般的には、ファイル名が、モジュール名.py のファイルの中に関数を書いておくと、import モジュール名で import でき、モジュール名.関数名で中の関数が使用できるようになる。他のモジュールのように、import liststat as ls のように省略名でインポートすることもでき(第１回のテキスト参照)、この場合は ls.average で使える。

しかし、ここで問題がある。liststat.py と違うディレクトリで python を起動して同じことをするとエラーになる。これは、python が liststat.py がどこにあるのかわからなくなるからである。python がモジュールを探す際には、まずカレントディレクトリを探し、無い場合は環境変数 PYTHONPATH の中にあるディレクトリを探す。そこにもない場合は、インストール時に決まるディレクトリのリストが使われる。math, pylab 等もモジュールはインストール時に決まるディレクトリの中(実習用コンピュータの場合、/opt/local/lib/python2.5 とそのサブディレクトリ）に入っているが、このディレクトリは共用の Mac ではユーザーはいじれない。モジュール liststat をいつでも使えるようにするためには、以前 PATH を.cshrc.local の中で設定したように、PYTHONPATH を設定し、設定したディレクトリの中に liststat.py をコピーすればよい。

ホームディレクトリに mkdir でディレクトリ pythonlib を作り、そこに自作のモジュールを入れるようにしよう。liststat.py を pythonlib にコピーし、.cshrc.local に以下の一行を追加する。
```
setenv PYTHONPATH ~/pythonlib
```

これで、新しいターミナルを立ち上げるか、ターミナル上で
```
% tcsh
```
として tcsh を新たに起動すれば、どのディレクトリからでもモジュール liststat を呼び出せる

ようになる。実際に、liststat.py を pythonlib にコピーし、.cshrc.local を編集して、liststat.py の無いディレクトリから python を立ち上げても liststat をインポートできるか試してみよう。第三回で述べたように、新しく xterm か tcsh を起動しないと、.cshrc.local の設定変更を反映されないことに注意。

では、モジュール liststat を使って、avrfunc.py を書き換えて、avrfuncmodule.py を作ろう。関数の定義をなくして、かわりに import liststat を書く。あとは、average を liststat.average に変えれば良い。

avrfuncmodule.py
```
#!/usr/bin/env python
import sys
import liststat #自作モジュールのインポート

infile=sys.argv[1]
outfile=sys.argv[2]

fin=open(infile,'r')
fout=open(outfile,'w')

data=[]
for line in fin:
        data.append(line)
avr=liststat.average(data) #モジュール liststat の中の関数 average を呼び出す。
out="avr: "+str(avr)
fout.write(out)

fin.close()
fout.close()
```

ここで一つ注意点がある。モジュールの中で必要とされるモジュールは、モジュールの中で import する必要がある。たとえば、math.sqrt を使わなくてはならない関数をモジュール化する場合、以下の例のように、
sqrfunc.py
```
import math
sqradd(a,b):
        c=math.sqrt(a*a+b*b)
        return(c)
```
モジュールファイル sqrfunc.py の先頭に
```
import math
```
を入れなくてはならない。これが無いとエラーになる。

課題２　モジュール

上で述べたように.cshrc.local を設定せよ。テキストファイル polygondef.py を作り、そこに課題１の polygonfunc.py から関数 polygon の定義をコピーして、できたファイルを~/pythonlib にコピーすることで、モジュール polygondef を作成せよ。その上で、polygonfunc.py を、モジュール polygondef をインポートする形に書き換え、レポートせよ。

補足: モジュールの検索順

モジュールの検索順は、呼び出し側のプログラムがあるディレクトリ->PYTHONPATH。プログラムがあるディレクトリに liststat.py がある状態で、pythonlib の中の liststat.py だけを書き

換えると、プログラムがあるディレクトリの中の古い liststat.py が呼び出される。

３　メソッドと変数型

さて、関数以外に似たものとしていままでメソッドも使ってきた。たとえば、リストにおける append や、文字列における split などがそうである。このメソッドも自分たちで定義することができる。

メソッドを主に用いる言語はオブジェクト指向言語（Java, ruby など)と呼ばれる。逆に関数を主に用いる言語（C 言語、fortran, perl など）は手続き型言語と呼ばれる。Python はどちらの特徴も兼ね備えており、手続き型言語としてもオブジェクト指向言語としても使える。一般にクラスごとに高い独立性で開発ができるオブジェクト指向言語は、大規模なプログラムの開発を複数のプログラマで行う場合に特に向いている。小さいプログラムの場合は手続き型言語のほうが使いやすい。生物学の研究者がそれほど大きいプログラムの開発を行うことは多くないので、本実習ではオブジェクト指向についてはさわりだけにとどめておこう。時間がなければこの項は読み飛ばしても良い。(課題もないことだし。)

オブジェクト指向言語におけるオブジェクトとは、特定のルールに従うデータの固まりと、それを扱うための関数をセットにしたものである。この関数をメソッドと呼ぶ。実はいままで出てきた変数はすべてオブジェクトであり、データとそれを扱うためのメソッドの組み合わせである。メソッドをつかって、そのオブジェクトのデータを処理するには、

オブジェクト名.メソッド

とする。たとえば、リスト a に対して、a.append(1)として 1 をリスト a に加える場合、オブジェクト名は a で、append がメソッドである。オブジェクトには種類があり、これをクラスと呼ぶ。いままで出てきた文字列型、複素数型、リスト型などは(ユーザー定義のクラスとは若干の違いがあるが)すべてクラスと見なしても良い。メソッドはクラスごとに定義される。リスト型であればどのオブジェクトもリスト型用のメソッド（append, expand など)を持っている。

ただ一ついままでの変数型とユーザーが定義するクラスとの違いは、ユーザー定義クラスの場合、そのクラスに属するオブジェクトのデータ(メンバと呼ぶ)にアクセスするときに、

クラス名.メンバ名

の形をとることである。これは実際に例を見てもらったほうが早いだろう。

ここで、例として x,y,z 座標を一つにまとめて扱うクラス coord3 を作ってみよう。また、メソッドとして、そのオブジェクトの 0 点からの距離を返すメソッド dist と、別の coord3 オブジェクトとの距離を返すメソッド distfrom を実装しよう。クラス coord3 は、x, y, z の三つのメンバを持つものとする。以下のファイル coord3.py を環境変数 PYTHONPATH に書かれている場所、本実習ではさきほど書いた ~/pythonlib に作れば、どこからでもモジュール coord3 を import して、クラス coord3 が使えるようになるのは、liststat.py のときと同様である。

coord3.py

```
import math
class coord3:
        def __init__(self, x, y, z):
                self.x = x
                self.y = y
                self.z = z

        def dist(self):
                dist=self.x**2+self.y**2+self.z**2
                if dist>0:
                        dist=math.sqrt(dist)
                return dist

        def distfrom(self,other):
                dist=(self.x-other.x)**2+(self.y-other.y)**2+(self.z-other.z)**2
```

```
                if dist>0:
                        dist=math.sqrt(dist)
                return dist
```

まずは、理屈抜きに使ってみよう。coord3 をインポートすると、クラス coord3 が使えるようになる。クラス coord3 のオブジェクトを生成するには、coord3.coord3 を用いる。クラス coord3 は x,y,z の三つのデータ(メンバ)を持ち、a=coord3.coord3(1,2,3)とすると、a の三つのメンバ a.x, a.y, a.z にそれぞれ 1,2,3 が渡される。オブジェクトを生成するためのメソッド名は常にクラス名と同じである(coord3.coord3 の前半の coord3 はモジュール名、後半の coord3 がメソッド名である)。

```
>>> import coord3
>>> a=coord3.coord3(1,2,3)  #クラス coord3 を生成して a に代入。初期データは x=1, y=2, z=3
>>> a.x
1
>>> a.y
2
>>> a.z
3
```
a.x では、a がオブジェクト名、x がメンバ名である。実際に各メンバに 1,2,3 が代入されているのがわかる。また、メソッド dist, distfrom を使ってみよう。
```
>>> a.dist()
3.7416573867739413
```
(座標(1,2,3)と原点の距離)
a.dist は、a.x
```
>>> b=coord3.coord3(2,3,4)
>>> a.distfrom(b)
1.7320508075688772
```
(座標(1,2,3)と座標(2,3,4)の距離)

では、もう少し中身を見てみよう。まず、クラスの定義から。
```
class クラス名:
        ブロック
```
でクラスの定義ができる。ブロックの中にメソッドなどのそのクラスの内容が定義される。ここではクラス名 coord3 に対して、__init__, dist, distfrom の三つのメソッドが定義されている。__init__は特殊なメソッドで、クラスのデータ構造を定義し、その初期化メソッドを提供する。クラスを定義するときにはほとんど常に必要とされる。__init__は、オブジェクト生成のためにクラス名のメソッドが呼び出されるときに、呼び出される。つまり、前の例で、
```
>>> a=coord3.coord3(1,2,3)
```
としたときに呼び出されているメソッドは、
```
        def __init__(self, x, y, z):
                self.x = x
                self.y = y
                self.z = z
```
である。関数の定義とほぼ変わらないが、呼び出すときの引数は全部で三つなのにこのメソッド定義ではもう一つの引数 self が存在している。self はメソッド特有のもので、外部から与えられる引数ではなく、生成されるオブジェクトそれ自体を示す。つまり、__init__メソッドの中の self.x = x は、生成されるオブジェクトの x メンバに x を代入するという意味である。Python においては代入することによって変数が生成されるのだが、この__init__メソッドの中では、オブジェクトのメンバを生成する。Python においては、このオブジェクトメンバの生成によって、クラスのデータ構造を定義する。つまり、__init__の中で、そのクラスに持たせたいメンバの数だけ代入を繰り返せば、それがそのままクラスのデータ構造になる。関数のローカル変数とは異なり、ここで生成されたオブジェクトのメンバは、そのオブジェクトが存在する限り存在し続ける。ここでは、self.x、self.y、self.z の三つに対する代入が行われているので、coord3 のメ

ンバは、x,y,z の三つになる。
もし、これを少し変更して、

```
        def __init__(self, x, y, z, t):
                self.x = x
                self.y = y
                self.z = z
                self.t = t
```

とすると、クラス coord3 は４つのメンバ x,y,z,t を持つことになる。

他の二つのメソッドは標準的なメソッドで、オブジェクト名.メソッド名で呼び出される。まずは、dist を見てみよう。

```
        def dist(self):
                dist=self.x**2+self.y**2+self.z**2
                if dist>0:
                        dist=math.sqrt(dist)
                return dist
```

ここでも self が使われている。ここでの self もオブジェクトそれ自体を表す。たとえば、a.dist()とした場合には、self=a となる。この self も__init__のときと同様に外部からの引数ではないので、実際に dist をメソッドとして呼び出すときは、a.dist()のように、引数なしになる。あとは、普通の関数と同じである。

一方、

```
        def distfrom(self,other):
                dist=(self.x-other.x)**2+(self.y-other.y)**2+(self.z-other.z)**2
                if dist>0:
                        dist=math.sqrt(dist)
                return dist
```

は、self 以外に引数 other があり、呼び出すさいには引数を一つだけ受け取る。

オブジェクト指向言語の利点は、クラスの高い独立性にある。データ構造とそれに対する演算の双方を持ったクラスは、プログラムの他の部分とまったく独立に開発ができ、多くのプログラムで使い回すのも簡単である。関数の場合、データ構造に変更が加えられると関数も変更しなくてはならず、かならずしもデータ構造と関数が一緒に定義されているとは限らないため、大規模な開発においては、その修正は極めて煩雑になる。クラスの場合は、クラス定義の中だけをチェックすれば良い。

第八回 ハッシュとシークエンス

今回の目的

ハッシュについてわかる。
アミノ酸シークエンス解析の基礎がわかる。

１　ハッシュ

今回は python のもう一つの変数型であるハッシュについて学び、それを使ってアミノ酸シークエンスを処理してみよう。ハッシュはリストに似ているが、リストのインデックスは整数に限られていた。ハッシュでは、整数のかわりに文字列や実数をインデックスのように使うことができる。これをキーと呼ぶ。このような形式を連想記憶配列または連想配列とも呼び、最近の多くの言語でサポートされている。まずは、python の対話モードでハッシュとはどんなものかを見てみることにする。

```
>>> a={}          #空ハッシュ a の生成。リストの場合は、空リストを生成するのに大括弧を用いて a=[]のようにしたが、ハッシュの場合は中括弧、a={}のようになる。
>>> a['test']=12      #a['test']の生成。この場合、キーは'test'。リストの場合は存在しないインデックスに代入するとエラーになったが、ハッシュの場合は代入するだけで新たな要素を生成できる。
>>> a['test']    #a['test']にちゃんと 12 が入っている。
12
>>> k='abc'  #キーは変数でも良い。
>>> a[k]='def'
>>> a['abc']
'def'
>>> a[2.3]=[2,3,4]  #キーは実数でも良い。また、リスト同様、要素はリストやハッシュを含めて、どんな型でもよい。（もちろん、ユーザーが定義したクラスのオブジェクトでも良い。）
>>> a[2.3]
[2, 3, 4]
```

a を表示すると以下のようになる。

```
>>> a
{'test': 12, 2.2999999999999998: [2, 3, 4], 'abc': 'def'}
```

中括弧{}の中に、キーと値のペアが、キー:値の形で,(カンマ)に区切られて並んでいる。順番は、代入順とは異なる。この順番には特に意味はない。2 番目のペアのキーは、代入時に 2.3 だったのが表示すると 2.2999999999999998 になっているが、これは第 1 回で述べた十進数と二進数のずれに起因する誤差。この誤差が嫌であれば、実数は文字列に変換してキーにすれば良い。

```
>>> a[str(2.3)]=[5,6,7]
>>> print a
{'test': 12, 2.2999999999999998: [2, 3, 4], '2.3': [5, 6, 7], 'abc': 'def'}
```

２　ハッシュの関数とメソッド

ハッシュには、以下の関数とメソッドがある。ハッシュを a としたばあい、

```
len(a)   a 内の要素数
del a[k]  a[k]の削除
a.clear()  a から全要素削除
```

a.has_key(k)　a[k]が存在すれば True そうでなければ False
a.keys()　 a のキーのリスト
a.values()　a の値のリスト

さきほどの続きで、
```
>>> a
{'test': 12, 2.2999999999999998: [2, 3, 4], '2.3': [5, 6, 7], 'abc': 'def'}
```
の場合を例として見てみよう
```
>>> len(a)
4          #値とキーをあわせて 1 つとみなす。
>>> a.has_key('test')
True
>>> a.has_key('check')
False
>>> a.keys()
['test', 2.2999999999999998, '2.3', 'abc']
>>> a.values()
[12, [2, 3, 4], [5, 6, 7], 'def']
>>> del a['2.3']
>>> a
{'test': 12, 2.2999999999999998: [2, 3, 4], 'abc': 'def'}  #'2.3':[5,6,7]のペアが消えている
>>> a.clear
>>> a
{}
```

３　ハッシュを使ったループ

ハッシュの要素ごとにループをするには、keys メソッドを使えば簡単である。以下に簡単な例 dirloop.py を示す。
dirloop.py:
```
#!/usr/bin/env python
a={}
a['a']=1
a['b']=2
a['c']=3
for key in a.keys():  #a のキーのリストを a.keys()で取り出し、そのリストについてループ
        print key,a[key] #キーと値のペアを表示
```

実行例：
```
% ./dirloop.py
a 1
c 3
b 2
```

４　シークエンスデータとハッシュ

現在の生物学は DNA シークエンス、蛋白質のアミノ酸シークエンス、蛋白質構造などの膨大なデータベースの有効活用が必要不可欠である。このうちシークエンスデータを扱うのに、ハッシュは便利である。通常アミノ酸シークエンスは一文字表記のアミノ酸の羅列で表記される。たとえば以下のシークエンスはウサギ骨格筋アクチンを表記したものである。このシークエンスが入ったテキストファイル rabbit.txt は分子第三講座ホームページ

http://str.bio.nagoya-u.ac.jp:8080/Plone の講義資料からダウンロードできる。

DEDETTALVCDNGSGLVKAGFAGDDAPRAVFPSIVGRPRHQGVMVGMGQKDSYVGDEAQSKRGILTLKYPIEHGIITNWDDMEKI
WHHTFYNELRVAPEEHPTLLTEAPLNPKANREKMTQIMFETFNVPAMYVAIQAVLSLYASGRTTGIVLDSGDGVTHNVPIYEGYA
LPHAIMRLDLAGRDLTDYLMKILTERGYSFVTTAEREIVRDIKEKLCYVALDFENEMATAASSSSLEKSYELPDGQVITIGNERF
RCPETLFQPSFIGMESAGIHETTYNSIMKCDIDIRKDLYANNVMSGGTTMYPGIADRMQKEITALAPSTMKIKIIAPPERKYSVW
IGGSILASLSTFQQMWITKQEYDEAGPSIVHRKCF

このような、アミノ酸一文字表記を並べた表記方法を **FASTA フォーマット**と呼び、タンパク質一次構造解析の基本フォーマットである。さて、このシークエンスから、どのアミノ酸が何残基あるのか数えるプログラム aminocount.py を書いてみよう。

aminocount.py

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

infile=sys.argv[1]
fin=open(infile,'r')

count={}
for line in fin:
        for let in line:
                if count.has_key(let):              #もし、count[let]が存在するなら
                        count[let] = count[let] + 1    #1 を足し、
                else:                               #そうでないなら
                        count[let] = 1              #count[let]を生成
print count
```

実行例

```
% ./aminocount.py rabit.dat
{'A': 29, 'C': 5, 'E': 28, 'D': 22, 'G': 28, 'F': 12, 'I': 30, 'H': 9, 'K': 19, 'M': 16,
'L': 26, 'N': 12, 'Q': 11, 'P': 19, 'S': 23, 'R': 18, 'T': 27, 'W': 4, 'V': 21, 'Y': 16}
```

結果をアルファベット順に並べたければ、print count のかわりに最後に４行加えて以下のようにする

aminocount.py

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

infile=sys.argv[1]
fin=open(infile,'r')

count={}
for line in fin:
        for let in line:
                if count.has_key(let):
                        count[let] = count[let] + 1
                else:
                        count[let] = 1

keylist=count.keys()
keylist.sort()
for key in keylist:
```

```
        print key,count[key]
```

最後から三行目で使われている、リスト用のメソッド sort()については、第一回のとき紹介したが、そのときに述べなかった注意点がある。リスト名.sort()はリストの中身を変えるだけで何も出力しない。したがって、リスト名.sort()を、リストを必要とする場所に直接は書けない。たとえば、

```
for key in keylist.sort():
        print key,count[key]
```

とするとエラーになる。

実行例

```
% ./aminocountsort.py rabit.txt
A 29
C 5
D 22
E 28
F 12
G 28
H 9
I 30
K 19
L 26
M 16
N 12
P 19
Q 11
R 18
S 23
T 27
V 21
W 4
Y 16
```

課題１：タンパク質の電荷

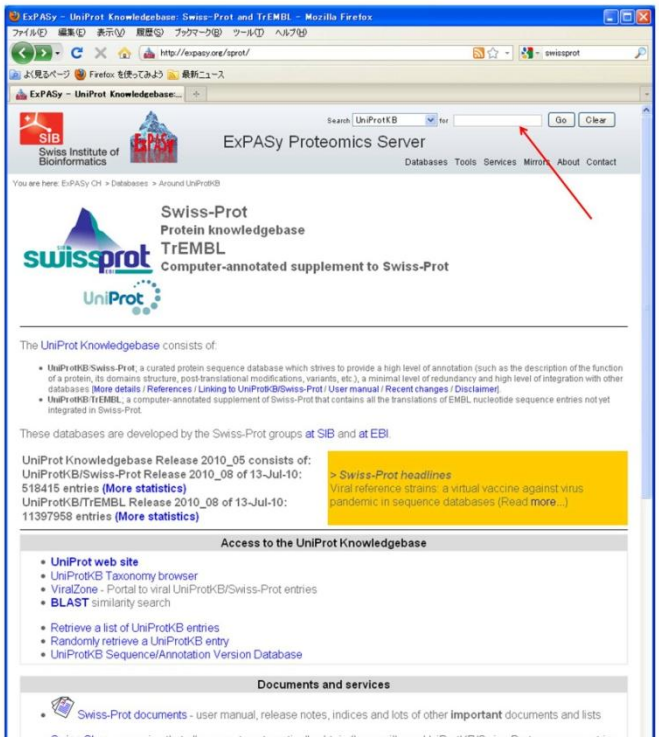
与えられたシークエンスの中に、正電荷を持つアミノ酸、負電荷を持つアミノ酸がそれぞれいくつあるのか数えるプログラム chargecount.py を作成せよ。実際に rabbit.dat に関して数えてみよ。

課題２　タンパク質の分子量

与えられたシークエンスからタンパク質の分子量を計算する、proteinweight.py を作成せよ。各アミノ酸の一文字表記とその分子量は、ホームページ http://str.bio.nagoya-u.ac.jp:8080/Plone の講義データから、aminoweight.txt をダウンロードすれば、その中に書いてある。このテキストファイルをプログラムに読み込むようにしても良いし、プログラムの中にこれらの値を書き込んでしまっても良い。ただし、アミノ酸が重合するさいには、水分子(分子量 18.02)が抜けることを考慮せよ。

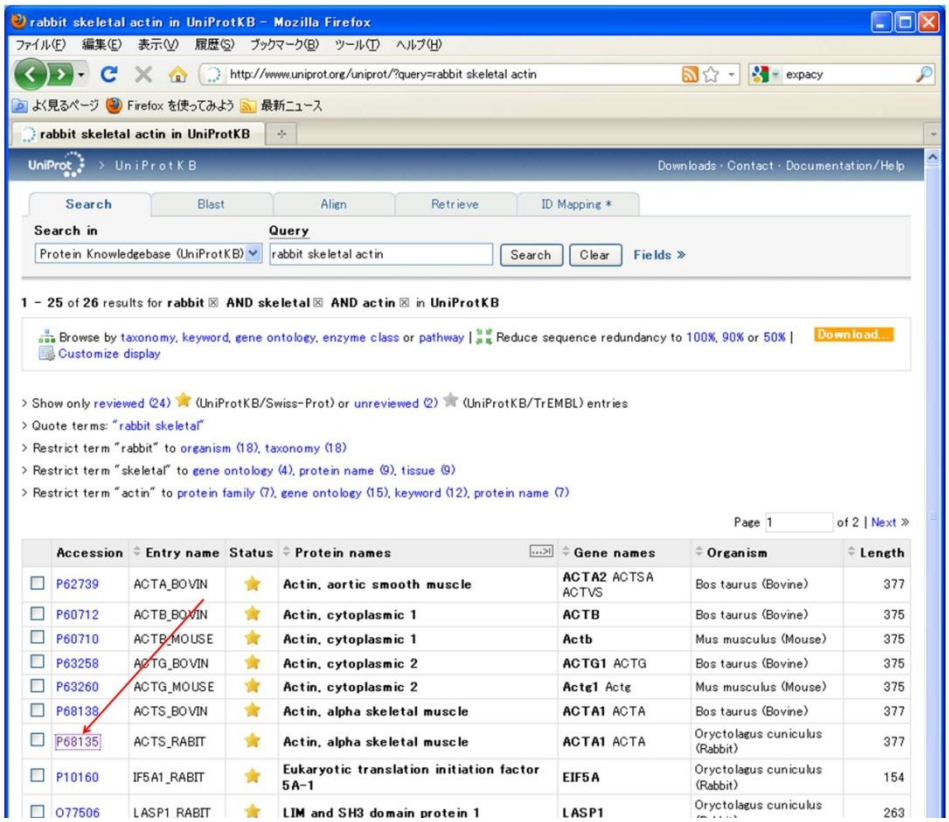
アミノ酸シークエンスから分子量を計算してくれるインターネット上のツールも存在する。
http://expasy.org/tools/pi_tool.html
これを用いて結果が大きく間違っていないことを確かめても良い。ただし、各アミノ酸の分子量は文献によって異なる(どこの地域の同位体比率を用いるかによっても異なる)ので、一残基あたり計算が 0.01 程度ずれていても問題はない。

5　シークエンスデータベース

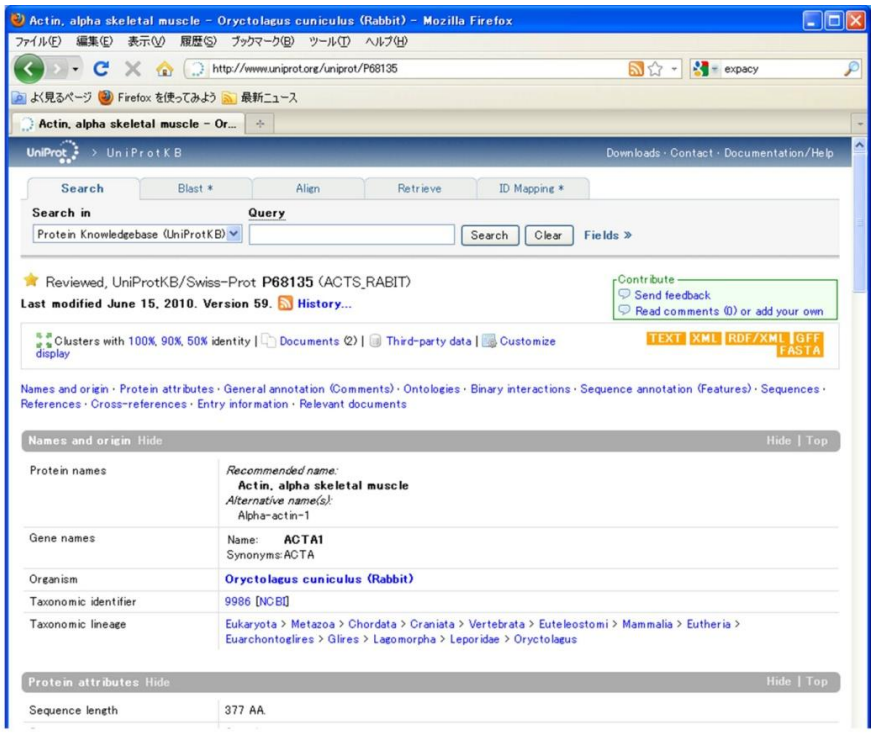


すでにわかっているアミノ酸配列のシークエンスは、インターネット上のデータベースに納められている。ここでは、ExPASy (http://expasy.org/sprot/) を使ってシークエンスを調べる方法について紹介しよう。アドレス http://expasy.org/sprot/にアクセスすると、左のような画面になる。

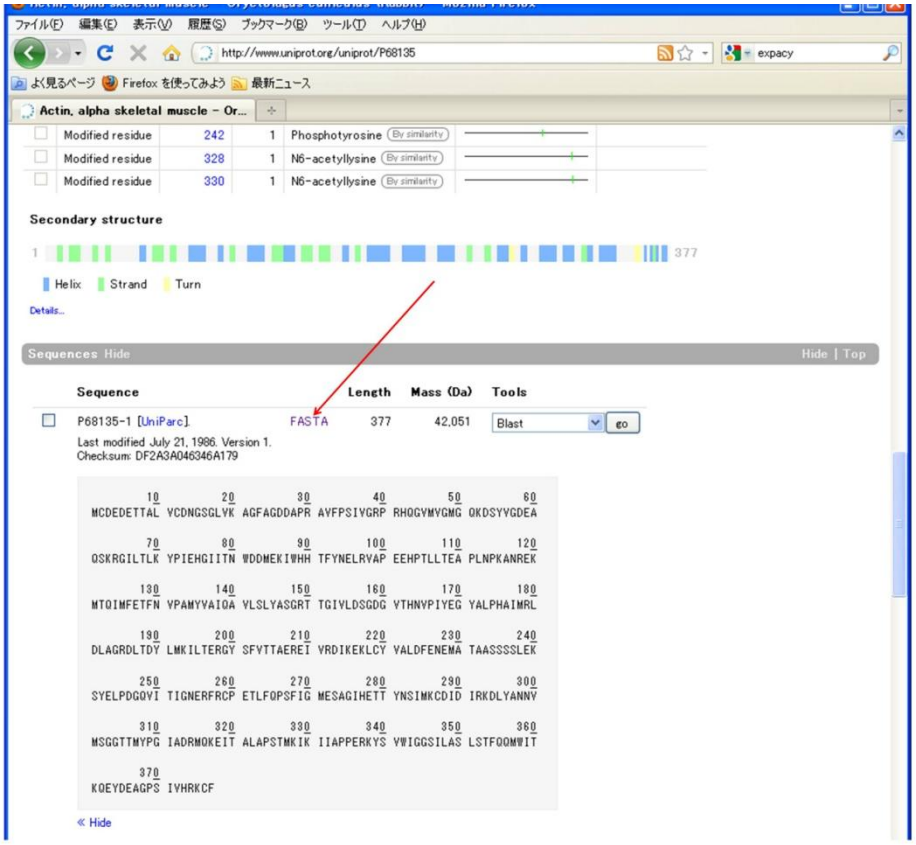
ここで、画面右上の検索窓(赤矢印)に検索ワードを入れると検索ができる。試しに、ウサギ骨格筋アクチンで調べて rabbit.txt と同じデータが得られるか試してみよう。検索窓に、rabbit skeletal actin と入力したときの検索結果を左下図に示す。

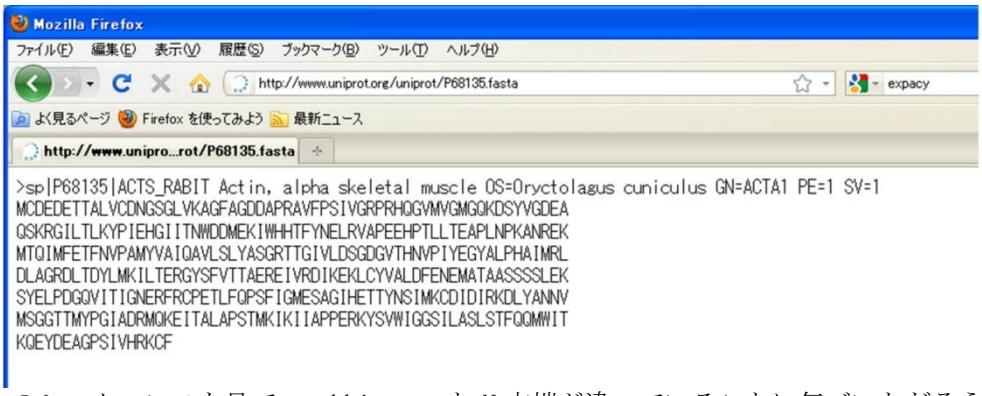


検索結果の中で、rabbit の skeletal muscle の項目を探して、Accsssion の列の番号(赤矢印)をクリックすると右図のようになる。Protein name に actin, alpha skeletal muscle とあり、Organisum に rabbit と書かれているので、望みのデータが得られたことがわかる。スクロールすると、シークエンスが書かれている(下図)。



シークエンスだけを得るには、矢印で示した青字の FASTA(赤矢印)をクリックすればよい。クリックした結果が次ページ画像。これをカットアンドペーストでテキストファイルにコピーして保存すれば、python で使用可能な形式になる。





さて、このシークエンスを見て、rabbit.txt と N 末端が違っていることに気づいただろうか？データベースのデータはほとんどが遺伝子を元に得られているが、実際のタンパク質の多くは翻訳後修飾を受け、遺伝子上のアミノ酸配列と異なっている。ウサギ骨格筋アクチンの場合、翻訳後、N 末端の M と C が失われ、新たな N 末端となる D がアセチル化されている。実際にデータベースを用いてタンパク質を扱う場合は、この翻訳後修飾を考慮に入れなければならない。翻訳後修飾については、既にわかっているものについてはデータベースのそれぞれのデータに書いてあることがある。このアクチンのデータについては、FASTA をクリックする前のページに Molecule processing の項があり、そこに載っているので、探してみよう。

課題３　データベースシークエンスの処理

expacy からなんでも良いから蛋白質のシークエンスを二つとってきて、それを課題１，課題２で書いたプログラムで処理せよ。翻訳後修飾は無視してよい。シークエンスは大文字で書かれる場合と小文字で書かれる場合がある。必要があれば、小文字にも対応できるようにプログラムを修正すること。その結果が、http://expasy.org/tools/pi_tool.html の結果とほぼ一致することを確かめても良い。
また、シークエンスの改行や空白に注意。改行や空白は一文字として数えられるので、水の質量を計算するときに真の値とずれが生じることに注意せよ。たとえば、
ACD
EFG
は、ACD と EFG の間に改行があるので、文字数を数えると７文字になる。このずれは、読み込まれた文字が改行(“\n”)や空白(” “)の場合にカウントしないようにプログラムすれば生じなくなる。

小文字にも対応するには、大文字と小文字を変換する文字列用のメソッドが便利である。
大文字を小文字に変換
文字列変数名.lower()
小文字を大文字に変換
文字列変数名.upper()
例:

```
>>> b="TesT"
>>> b.upper()
'TEST'
>>> b.lower()
'test'
```

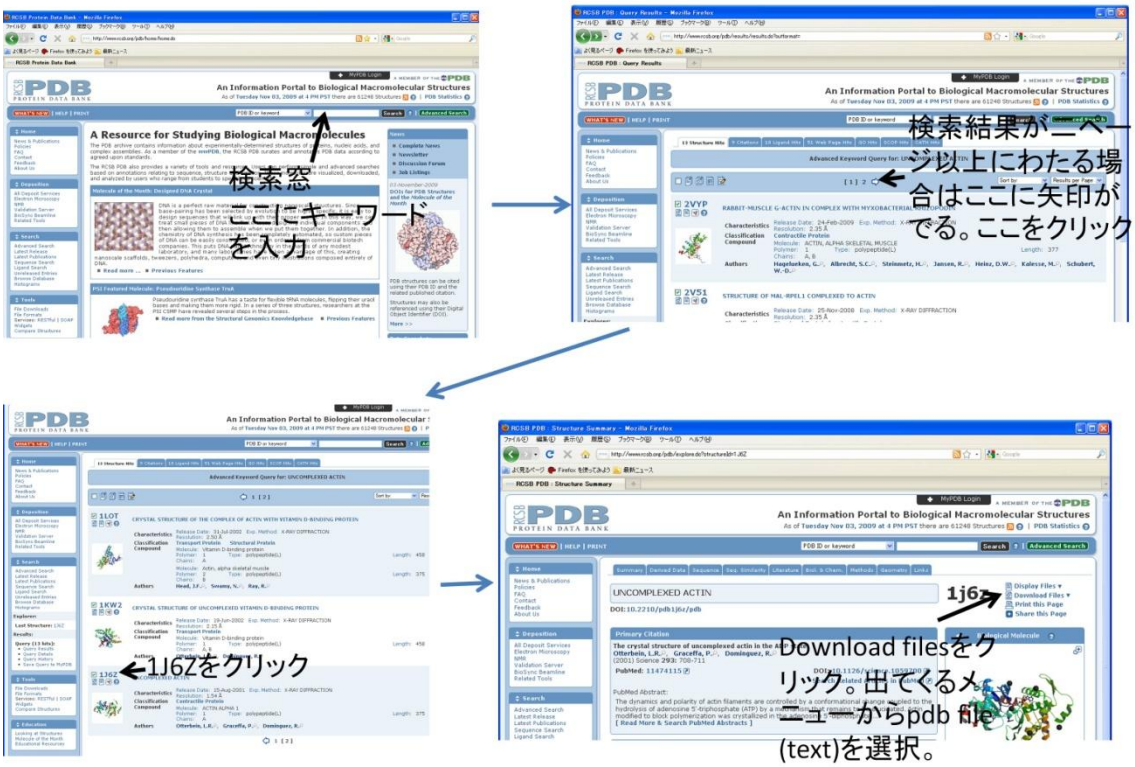
第九回　pdb フォーマットと蛋白質構造

今回の目的

蛋白質構造を記述する pdb フォーマットの基礎がわかる。簡単な解析ができる。

Pdb database

X 線結晶解析や NMR などで解かれた生体高分子の原子座標構造を表すための共通フォーマットが策定されており、これを pdb フォーマットと呼ぶ。pdb フォーマットで書かれた原子座標構造を pdb ファイルと呼ぶ。いままで決定された構造の pdb ファイルは、http://www.rcsb.org/pdb/home/home.do
からダウンロードできる。試しに、ウサギ骨格筋アクチン構造のひとつ、1J6Z.pdb をダウンロードしてみよう。このアドレスにアクセスすると、下のようなページになるので、矢印に示した検索窓に、uncomplexed actin と入力してみよう。すると様々な構造がでてくる。この検索結果の二ページ目に 1J6Z があるので、それをクリック。開いたページの中の Download files をクリックし、出てくるメニューから pdb file (text) を選択すると、保存場所を聞かれるので、指定すれば、ダウンロードができる。



Pdb file とは

Pdb ファイルは、厳密にフォーマットが定められたテキストファイルである。PDB ファイルの基本単位は行となっており、各行の先頭のキーワードがその行がもつ情報が何であるかを表している。Pdb ファイルには、以下のような情報が含まれている。

- HEADER: このファイルに含まれている蛋白質の総称名とこのファイルが PDB に登録された日付が書いてある。また、このデータの ID もしめされている。
- COMPND: この蛋白質の名前。
- SOURCE: この蛋白質が得られたソース（起源）。

- AUTHOR: データを PDB に登録した人の名前。
- JRNL: この蛋白質の X 線結晶構造解析の論文。
- REMARK: この構造解析に関する種々の情報。
- SEQRES: 3 文字表記のアミノ酸コードでこの蛋白質のアミノ酸配列を示す。
- FORMUL: この結晶に含まれる他の分子（低分子）に関する情報。
- HELIX: αヘリックスをとるアミノ酸の位置が残基番号で示される。βシートやターンの位置を示すためには同様に SHEET や TURN というキーワード名が使われる。
- SSBOND: 分子内の S-S 架橋の位置。
- CRYST: 格子定数、空間群および単位格子中に含まれる蛋白質分子の数。
- ORIGX および SCALE: 単位格子中で空間群の対称操作で関係づけられる蛋白質分子の座標を計算するための変換マトリックス等。
- ATOM: 各原子の座標に関するデータが入っている。各カラムの意味は以下の通り:
  カラム 1-4　ATOM
  7-11　Atom serial number　←原子通し番号
  13-16　Atom name　←原子名
  17　Alternate location indicator
  18-20　Residue name　←残基名
  22　Chain identification　←ポリペプチド鎖名
  23-26　Residue sequence number　←残基番号
  31-38　X (Orthogonal Å coordinates)　←原子の直交Å座標値
  39-46　Y (Orthogonal Å coordinates)
  47-54　Z (Orthogonal Å coordinates)
  55-60　Occupancy　←占有率
  61-66　Temperature factor　←温度因子
  73-76　ID　←識別コード
  77-80　Sequential number　←通し番号
  原子名は IUPAC-IUB 命名法にしたがっているが、ギリシャ文字が使えないので、α、β、γ、δ、ε、ζ、ηに対しては、A、B、G、D、E、Z、H を対応させている。たとえば CA は C_α（αカーボン）を表す。また、カラム 1-4 とは、その行の 1-4 文字目という意味。
- HETATM: 蛋白質以外の原子の座標がここに示される（水分子や ATP など）。内容は全く ATOM と同じ。
- CONECT: 蛋白質分子ではない原子間の結合を表す。対になっている番号の原子間に結合があることを示す。
- TER: 一つのポリペプチド鎖の終了を示す。
- END: このファイルの終了を示す。

先ほどダウンロードした 1J6Z.pdb の中を emacs でみてみよう。まず最初に、構造の由来が書かれているヘッダー部分がある。ここには、何の分子の構造で、どの手法を用いたか、どの雑誌のどの論文に最初に載せられたかなどが書かれている。

```
HEADER    CONTRACTILE PROTEIN                     15-MAY-01   1J6Z
TITLE     UNCOMPLEXED ACTIN
COMPND    MOL_ID: 1;
COMPND   2 MOLECULE: ACTIN ALPHA 1;
COMPND   3 CHAIN: A
SOURCE    MOL_ID: 1;
SOURCE   2 ORGANISM_SCIENTIFIC: ORYCTOLAGUS CUNICULUS;
SOURCE   3 ORGANISM_COMMON: RABBIT;
```

```
SOURCE    4 OTHER_DETAILS: MUSCLE
KEYWDS     ACTIN, TETRAMETHYLRHODAMINE-5-MALEIMIDE, ADP-STATE
EXPDTA     X-RAY DIFFRACTION
AUTHOR     L.R.OTTERBEIN,P.GRACEFFA,R.DOMINGUEZ
REVDAT    2   25-MAR-03 1J6Z    1       FORMUL REMARK
REVDAT    1   15-AUG-01 1J6Z    0
JRNL         AUTH   L.R.OTTERBEIN,P.GRACEFFA,R.DOMINGUEZ
JRNL         TITL   THE CRYSTAL STRUCTURE OF UNCOMPLEXED ACTIN IN THE
JRNL         TITL 2 ADP STATE
JRNL         REF    SCIENCE                       V. 293   708 2001
JRNL         REFN   ASTM SCIEAS  US ISSN 0036-8075
```

次に続く REMARK の項には、構造の分解能、実験の詳細、結晶が持つ対称性などが書かれている。次の SEQRES の項には、この蛋白質のシークエンスが書かれている。その先にある ATOM、HETATM の項には、実際の構造データが書かれている。ATOM にはポリペプチドの座標が、HETATM は ATP や金属イオンなどのポリペプチド以外の座標が書かれている。
最初のカラムから、"ATOM"または"HETATM", 原子の通し番号、原子の名前、残基の名前、分子の ID, 残基番号、その原子の x 座標、y 座標、z 座標、占有率、温度因子がかかれている。このファイルでは、最後に原子の種類が書かれているが、これはオプションである。たとえば、このファイルの ATOM の項の二行目

```
ATOM      2  CA  GLU A   4      -14.630  11.239   6.884  1.00 45.46           C
```

は、このファイルの二番目の原子で、Cα原子、グルタミン酸、chainIDはA、残基番号は4, 座標は、(-14.630, 11.239, 6.884)（単位はÅ）であることを表している。占有率と温度因子は、結晶学特有のパラメータなので、今回は省略する。

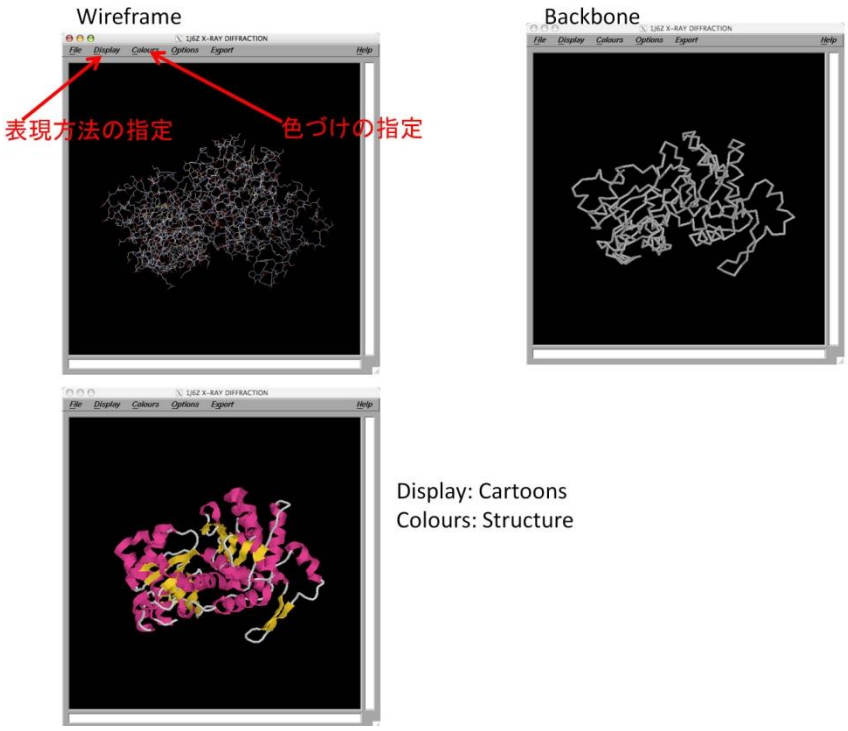
Pdb viewer rasmol

Pdb ファイルを三次元的に見るプログラムは多数あるが、もっとも歴史がありどの OS でも動くプログラムが rasmol である。rasmol の使い方は簡単で、

% rasmol pdbfilename

で起動する。たとえば、先ほどの 1J6Z.pdb であれば、

% rasmol 1J6Z.pdb



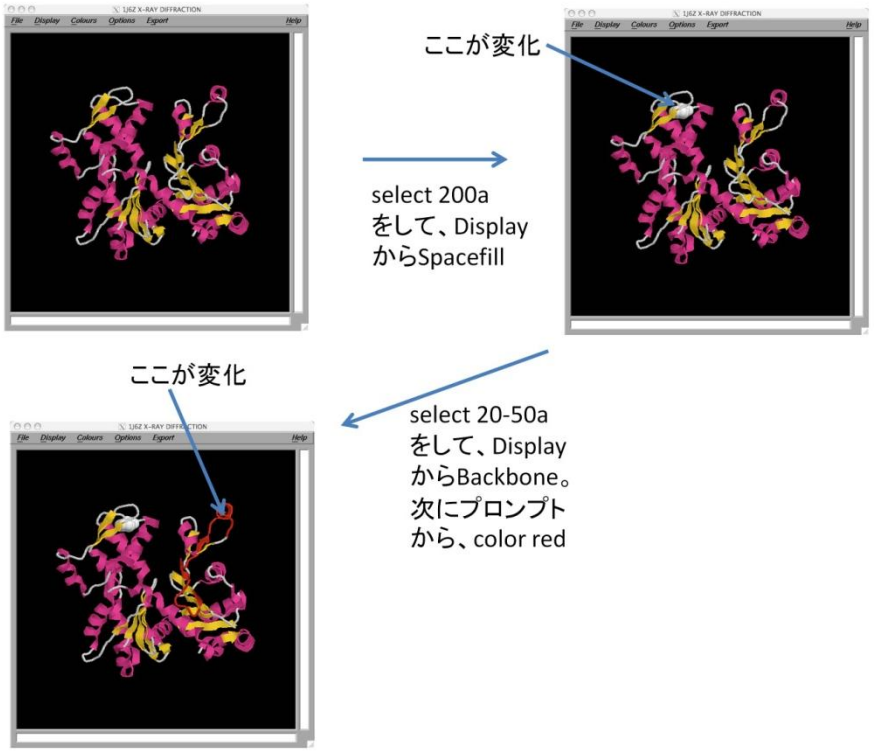
である。最初はワイヤーフレームと呼ばれる形式で表示されていると思う。様々な表現方法を指定できる。表現方法は、ウィンドウ上のメニューの Display で選べる。Backbone や Cartoons などにしてみよう。また、色のつけかたはメニューの Colours で選べる。また、画面の中で左ボタンを押しながら、マウスを動かすと構造が回転する。右ボタンを押しながら動かすと、平行移動する。

一方、rasmol を起動した xterm の上には、

RasMol>

という表示がでている。これは rasmol のプロンプトで、ここにいろいろなコマンドを打ち込むことがで

きる。また、rasmol からの情報もここに表示される。たとえば、画面の中の蛋白質構造の上をクリックすると、
Atom: CA 1179　Group: GLY 156　Chain: A
のような表示が現れる。これは、クリックした原子が Cα で、1179 番目の原子。残基番号 156 のグリシンで、Chain ID が A であることを示している。Chain ID は、pdb ファイルの中に複数のポリペプチド鎖が含まれている場合、何番目のポリペプチド鎖かを指定する。コマンドプロンプトから、蛋白質の特定の部分を指定して、そこだけ表現方法を変えたり、色を変えたりすることもできる。たとえば、Chain Id が A の残基番号 200 の残基を指定するには、rasmol のプロンプトから、select 200a とすれば良い。
RasMol> select 200a
11 atoms selected!
この上で、rasmol のメニュー上の Display から Spacefill を指定すると、残基番号 200 のアミノ酸残基だけが Spacefill モデル(空間充填モデル)の表現に変化する。もし、Chain ID A の 20-50 番目の残基を一気に指定したければ、select 20-50a とする。
RasMol> select 20-50a
226 atoms selected!
この領域の色を変えたければ、コマンド color を使う。
RasMol> color red
そうすると、20-50a の領域が赤に変化する。また、select は,で区切って複数の領域を指定できる。たとえば
RasMol> select 20-50a, 200a, 300-320a,350a
とすれば、chainID A の 20-50,200,300-320,350 の残基を指定することになる。
ズームもできる。そのためには、ズームする中心の残基を center で指定し、コマンド zoom を用いる。zoom においては、100 が等倍である。たとえば、157a を中心に、二倍に拡大したければ、
RasMol> center 157a
RasMol> zoom 200
となる。また、zoom や水平移動、回転状態を最初の状態に戻したければコマンド reset を用いる
RasMol> reset



Rasmol のマニュアルは
http://www.rasmol.org/software/RasMol_2.7.5_Manual.html
にある。これを参考にいろいろ自分で試してみよう。

複数のポリペプチド鎖がある場合

複数のポリペプチド鎖がある構造の例として、分子第三講座 http://str.bio.nagoya-u.ac.jp:8080/Plone の講義資料のページから、F-actin 構造をクリック、factinn6.pdb をダウンロードしてみよう。これは、アクチンが重合したときの構造を示している。このファイルには 6 分子のアクチンが含まれており、chainID はそれぞれの分子に B,C,D,E,F,G がわりあてられている。rasmol で開いてみよう。Rasmol のメニューの Colours から、Chain を指定すると、分子ごとに別の色がわりあてられ、見やすくなる。

Select いろいろ

ここで、select についてもう少し見てみよう。chainID を省略すると、全てのポリペプチド鎖に対する指定になる。

```
RasMol> select 10
RasMol> color green
```

とすると、全てのポリペプチド鎖の残基番号 10 の残基が緑色になる。chainID C のポリペプチド鎖の残基全てを指定したければ、

```
RasMol> select *c
```

とする。ファイル内の全ての原子を指定したければ、

```
RasMol> select all
```

でよい。

課題１　rasmol の習得

factinn6.pdb の中の分子 D の N 末端と C 末端の残基を白の spacefill で表示せよ。その場所がわかりやすいような図を作り、レポートせよ。

Pdb file の操作

テキストエディタによる操作

Pdb ファイルはテキストファイルにすぎないので、中を直接 emacs などのテキストエディタで直接いじることができる。簡単な例を見てみよう。emacs で 1J6Z.pdb の ATOM 項最初の行

```
ATOM      1  N   GLU A   4     -16.021  11.749   6.951  1.00 46.20           N
```

を

```
ATOM      1  N   GLU A   4     -56.021  11.749   6.951  1.00 46.20           N
```

に変えて、1J6Z2.pdb として保存、rasmol で開き、spacefill で表示してみよう。原子が一個だけ離れているのが見えるはずである。離れた原子をクリックすると確かに残基番号 4 の原子 N であることがわかる。この座標を適当に変えてどう動くか試してみよう。

Python による Cα原子の抜き出しと find メソッド

テキストファイルである以上、python による操作は簡単である。試しに、pdb ファイルから Cα原子だけを抜き出すプログラム caonly.py を書いてみよう。原子名はカラム 13-16、つまり 13 から 16 文字目に書かれている。従って、13-16 文字目をスライスによって抜き出すことで、簡単に原子名を抜き出すことができる。また、このプログラムのために使うと便利な文字列用メソッド find を紹介しよう。

変数名.find('探す文字列')

とすると、変数の中で、'探す文字列'がスタートする index を返す。存在しなければ-1 を返す。Python の対話モードで見てみよう。

```
>>> a="abcdef"
```

```
>>> a.find('bc')
1   #'bc'は index 1(二文字目)からスタートする。
>>> a.find('ef')
4   #'ef'は index 4(五文字目)からスタートする。
>>> a.find('z')
-1   #'z'は a の中には存在しない。
```
これを使って、caonly.py を書くと以下のようになる。

caonly.py
```
#!/usr/bin/env python

import sys
infile=sys.argv[1] #infile:読み込むpdbファイル
outfile=sys.argv[2] #outfile:書き出すpdbファイル
fin=open(infile,'r')
fout=open(outfile,'w')

for line in fin:
    head=line[0:6]
    if head.find('ATOM') != -1:  #1文字目から6文字目まで(head)の間にATOMが含まれているか。Cαはの項の中だけから探せば良い。
        atomname=line[12:16]  #13から16文字目を取り出す。
        if atomname.find('CA') != -1:  #取り出した中にCAが含まれていれば
            fout.write(line)    #foutに書き出す。
fin.close()
fout.close()
```

実際に
```
% caonly.py 1J6Z.pdb 1J6ZCa.pdb
```
として、その結果が Cα原子しか含まれていないことを確かめてみよう。

課題２特定 Chain ID の抜き出し

Pdb ファイルから特定の chain ID の原子だけを抜き出すプログラム singlechain.py を書け。それを用いて、実際に factinn6.pdb から chainID が D の原子だけ抜き出し、その結果もレポートせよ。

Python による分子間インタフェースの同定

ここまでで見たように、python を使えば pdb ファイルをわりと簡単に変更したり解析したりできる。ここでは少し実用的なプログラムとして、pdb ファイルから分子間の結合部位を取り出すプログラム pdbinterface.py を書いてみよう。

```
% pdbinterface.py pdbファイル名 chainID1 chainID2 threshold  output
```

とすると、pdbファイルの中のchainID1で示される分子のある残基Aのなかのどれかの原子が、chainID2で示される分子のある残基Bの中のどれかの原子とthreshold(Å)以内の距離に存在する場合、AとBの二つの残基は、結合部位に存在すると判断する。そのような残基のリストをoutputに出力する。たとえば、
```
# ./pdbinterface.py factinn6.pdb C E 4  CEinterface.txt
```
とすると、factinn6.pdbの中で、C分子とE分子の結合部位にある残基を4ÅのthresholdでCEinterface.txtという意味になる。

pdbinterface.py
```
#!/usr/bin/env python

import sys
```

```
infile=sys.argv[1] #pdb file名
mol1=sys.argv[2] #分子1のID
mol2=sys.argv[3] #分子2のID
th=sys.argv[4]  #threshold
out=sys.argv[5] #出力ファイル

th=float(th)*float(th)  #計算の都合で、thresholdを二乗にする。
f=open(infile,'r')
x=[]
y=[]
z=[]
res=[]
mol=[]
flag={} #結合部位に存在する残基名を保存するハッシュ

for line in f: #全ての原子のx座標、y座標、z座標、残基番号(res)、chainID(mol)をリストに保存する。
    name=line[0:6]
    if (name.find('ATOM') != -1):
        x.append(float(line[30:38]))
        y.append(float(line[38:46]))
        z.append(float(line[46:54]))
        res.append(line[22:26])
        mol.append(line[21:22])
f.close()

for n in range(0,len(x)):
    for m in range (n+1,len(x)): #保存された全ての原子のペアについてループする。
        if ((mol[n].find(mol1) != -1 and mol[m].find(mol2) != -1) or (mol[n].find(mol2) != -1 and mol[m].find(mol1) != -1)): #上の行とこの行の間には改行なし。あわせて一行。
# もし、mol[n]とmol[m]が、mol1, mol2またはmol2, mol1の組み合わせであれば、結合部位かどうかを判定する。
            d = (x[n]-x[m])**2 +(y[n]-y[m])**2 + (z[n]-z[m])**2
            if (d < th):  #原子間の距離の二乗を計算し、thより小さいかを判定。
                resmol1=res[n]+mol[n] #小さければ結合部位と判断。
                resmol2=res[m]+mol[m]
                flag[resmol1]=1
                flag[resmol2]=1
```

結合部位であれば、flag[残基番号+chainID]に1を代入する。こうすることで、flag[残基番号+chainID]が生成される。

```
f=open(out,'w')
for key in flag.keys():
    f.write(key+"¥n")
f.close()
```

#存在する flag のキーを出力する。

実際に

```
# ./pdbinterface.py factinn6.pdb C E 4  CEinterface.txt
```

を実行すると、CEinterface.txt は以下のようになる。

```
40C
  42C
 244C
 291E
 287E
```

```
242C
323E
169E
 39C
167E
205C
 62C
171E
204C
325E
 45C
 41C
…
```

Rasmol のスクリプトと、残基リストの表現

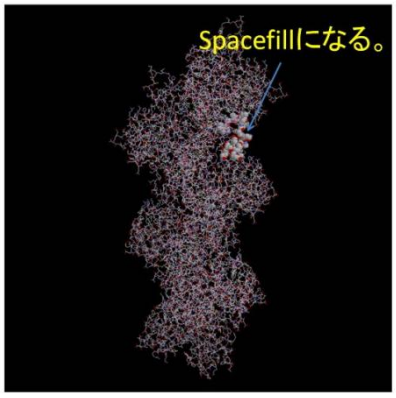
これだとわかりにくいので、rasmol上で見られるようにしよう。rasmolには、コマンドを羅列したスクリプトを読み込むことができる。たとえば、以下のようなtest.rasを作ってみよう。

```
select 40-60E
spacefill
```

これは、40-60E を spacefill で表示するという意味である。
その上で factinn6.pdb を rasmol で開き、test.ras を読み込んでみよう。rasmol におけるスクリプトの読み込みは、source コマンドを用いる。

```
% rasmol factinn6.pdb
RasMol> source test.ras
```

こうすることで、図のように 40-60E が spacefill 表示になることを確かめよう。



結合部位のリストに関して、このようなスクリプトを書いて spacefill で表示させるようにしたらわかりやすいだろう。そのようなスクリプト makespacefill.py は簡単に書ける。
makespacefill.py 入力ファイル　出力ファイル
入力は pdbinterface.py の結果ファイル、出力ファイルは rasmol のスクリプトで、結合部位の残基を spacefill で表示させる。

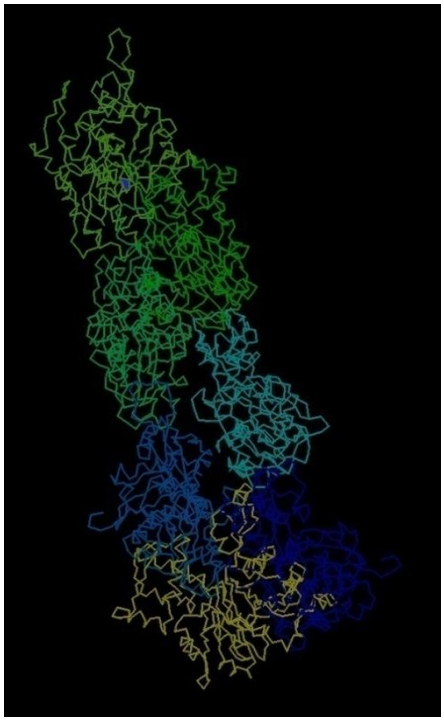
makespacefill.py

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

import sys
infile=sys.argv[1]
outfile=sys.argv[2]

fin=open(infile,'r')
fout=open(outfile,'w')

for line in fin:
    fout.write('select '+line)
    fout.write('spacefill'+'¥n')
fout.close()
fin.close()
```

このプログラムを使って、先ほどの CEinterface.txt をスクリプトに変換しよう。

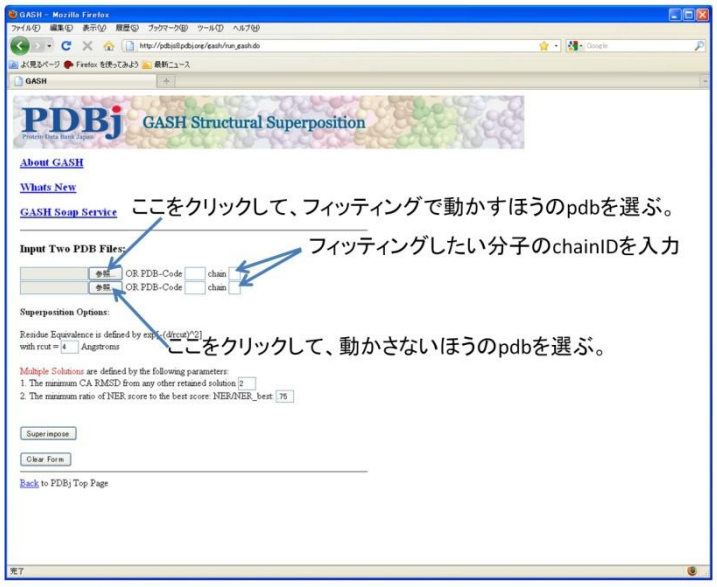
```
% makespacefill.py CEinterface.txt CEinterface.ras
```

あとは、
% rasmol factinn6.pdb
RasMol> source CEinterface.ras
とすると、分子 C と E の結合部位に存在する残基が spacefill で表示される。

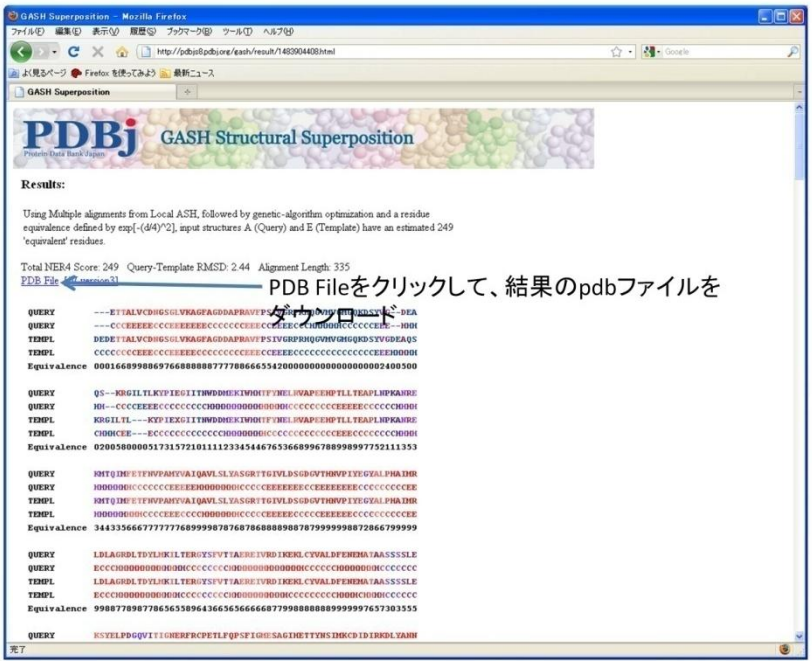


Structural alignment

似た二つの蛋白質構造を比べてみたい場合は良くある。今回はアクチンフィラメント構造である factinn6.pdb の中のアクチン分子と、単量体アクチンである 1J6Z.pdb のアクチン分子の間の構造変化を調べてみよう。二つの分子間の構造変化を調べるためにはまずは、二つの構造を重ねて表示しなくてはならない。そのために一番簡単な方法は、二つの pdb ファイルを結合することである。cat を使うと二つのファイルを結合できる。
cat ファイル名１　ファイル名２　…　> 出力ファイル名
のようにすると、いくつのテキストファイルでも一つのテキストファイルにまとめることができる。たとえば、
% cat 1J6Z.pdb factinn6.pdb > factinn6.1J6Zadd.pdb
とすると、1J6Z.pdb と factinn6.pdb の内容が factinn6.1J6Zadd.pdb の中にコピーされる。まずはこれを rasmol で表示してみよう。
右上図は、メニューの colour→Chain で色づけしてある。黄色が chainA つまり 1J6Z.pdb であるが、ほかの factinn6.pdb の分子と位置がまったくあっていない。比較するためには、比較したい分子同士をまず同じ位置と方向にそろえなければならない。この位置と方向を合わせることを structral alignment と呼ぶ。



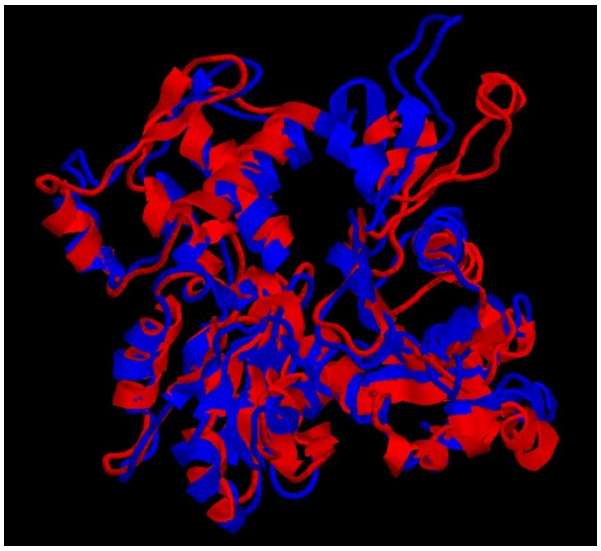
様々なソフトウェアが structural alignment をサポートしているが、今回は web 上で使える PDBj の GASH というアラインメントを試してみよう。使い方は簡単である。
http://pdbjs8.pdbj.org/gash/run_gash.do
にアクセスすると、右下図のようになる。上側は、アラインメントで動かすほうの pdb ファイル、下側はアラインメントで動かさないほうの pdb ファイルである。上側の構造を下側の構造に合わせてくれる。しばらく待つと次ページのような画面になる。いくつかの候補が表示されるが、上にでてくるほどスコアが高いので、今回は、一番上の結果の pdb ファイルをダウンロードしよう。



結果のファイルには、フィッティングで動かしたほうの構造の chainID は B で、動かさないほうの chainID は A になっている。Rasmol で表示すると左下図のようになる。このとき、メニューから Display で Cartoons を選び、Colour から Chain を選んで表示した。青が chain A、赤が chain B である。二つの分子の位置がちゃんと合っているのがわかる。

課題 3　GASH を実際に使う。

GASH を用いて、factinn6.pdb の分子 E に対して、1J6Z.pdb の位置と方向を合わせよ。
その結果を rasmol を使って表示し、二つの位置があっていることを示せ。



構造の比較

Structural alignment が終わったら、いよいよ構造変化部位を同定しよう。今回の場合は同じ蛋白質の違う構造を比べるので、アミノ酸シークエンスは同じである。Cα原子だけを比べることにする。そのためのアルゴリズムの例としては以下の通り。入力パラメータは、pdb ファイルと、比べる分子の chainID(二つ)、それ以上位置にずれがある場合に構造変化していると決定するための閾値である。pdbinterface.py が参考になるだろう。

1 全ての Cα の座標とその chainID、残基番号をリストに保存する。
2　比べる分子間で、残基番号が同じ物同士で位置を比べ、そのずれが threshold 以上であれば、その残基名を保存する。
3 最後に 2 で保存された全残基名を出力する。

課題４　構造変化部位の同定

実際に、構造変化部位を同定するプログラム structchange.py を書き、課題 3 の GASH の結果に対して実行することで、アクチンが単量体からフィラメント構造になるときに変化が大きい部位を決定せよ。また、その部位を rasmol を用いてわかるように表示せよ。

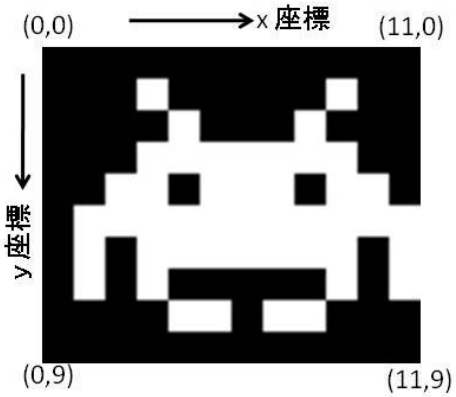
第十回　画像ファイルの扱い

今回の目的

Python から、画像ファイルを作成、解析、操作する方法を学ぶ

１　画像表現の基本

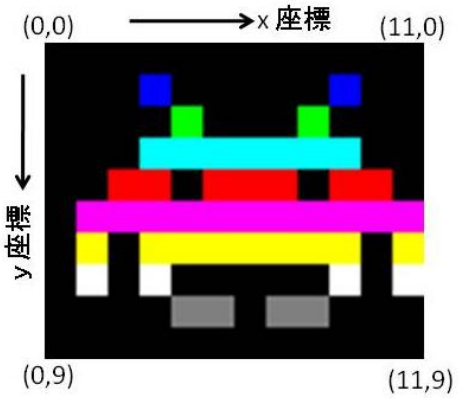
コンピュータにおいて、画像は、ピクセルと呼ばれる点の集合で記録される。実際には、x,y の二次元座標平面の各ピクセルに色、または明るさを数値として設定することによって表現される。たとえば、下のような簡単なインベーダーの絵を見てみよう。このファイルは http://str.bio.nagoya-u.ac.jp:8080/Plone の講義資料のページから invader.tif としてダウンロードできる。



この絵は横 12 ピクセル、縦 10 ピクセルの絵である。左上を原点にとり、x 座標、y 座標が設定される。これを白黒の 8 bit 画像で保存した場合、コンピュータの内部においては、右のような数表として保存される。後で述べるが、8bit 画像の場合、最大値は 255 で、値が大きいほどピクセルは明るくなり、0 が黒で 255 が白である。

y座標＼x座標	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	255	0	0	0	0	0	255	0	0
2	0	0	0	0	255	0	0	0	255	0	0	0
3	0	0	0	255	255	255	255	255	255	255	0	0
4	0	0	255	255	0	255	255	255	0	255	255	0
5	0	255	255	255	255	255	255	255	255	255	255	255
6	0	255	0	255	255	255	255	255	255	255	0	255
7	0	255	0	255	0	0	0	0	0	255	0	255
8	0	0	0	0	255	255	0	255	255	0	0	0
9	0	0	0	0	0	0	0	0	0	0	0	0

では、カラーの場合どうなるだろうか？インベーダーに色を付けた次ページの絵を見てみよう。座標は白黒のときと同じである。この画像は invadercolor.tif としてダウンロードできる。



y座標 ↓ / x座標 →	0	1	2	3	4	5	6	7	8	9	10	11
0	R:0 G:0 B:0	R:0 G:0 B:0	R:0 G:0 B:0	R:0 G:0 B:0	R:0 G:0 B:0	R:0 G:0 B:0	R:0 G:0 B:0	R:0 G:0 B:0	R:0 G:0 B:0	R:0 G:0 B:0	R:0 G:0 B:0	R:0 G:0 B:0
1	R:0 G:0 B:0	R:0 G:0 B:0	R:0 G:0 B:0	R:0 G:0 B:255	R:0 G:0 B:0	R:0 G:0 B:0	R:0 G:0 B:0	R:0 G:0 B:0	R:0 G:0 B:0	R:0 G:0 B:255	R:0 G:0 B:0	R:0 G:0 B:0
2	R:0 G:0 B:0	R:0 G:0 B:0	R:0 G:0 B:0	R:0 G:0 B:0	R:0 G:255 B:0	R:0 G:0 B:0	R:0 G:0 B:0	R:0 G:0 B:0	R:0 G:255 B:0	R:0 G:0 B:0	R:0 G:0 B:0	R:0 G:0 B:0
3	R:0 G:0 B:0	R:0 G:0 B:0	R:0 G:0 B:0	R:0 G:255 B:255	R:0 G:255 B:255	R:0 G:255 B:255	R:0 G:255 B:255	R:0 G:255 B:255	R:0 G:255 B:255	R:0 G:255 B:255	R:0 G:0 B:0	R:0 G:0 B:0
4	R:0 G:0 B:0	R:0 G:0 B:0	R:255 G:0 B:0	R:255 G:0 B:0	R:0 G:0 B:0	R:255 G:0 B:0	R:255 G:0 B:0	R:255 G:0 B:0	R:0 G:0 B:0	R:255 G:0 B:0	R:255 G:0 B:0	R:0 G:0 B:0
5	R:0 G:0 B:0	R:255 G:0 B:255	R:255 G:0 B:255	R:255 G:0 B:255	R:255 G:0 B:255	R:255 G:0 B:255	R:255 G:0 B:255	R:255 G:0 B:255	R:255 G:0 B:255	R:255 G:0 B:255	R:255 G:0 B:255	R:255 G:0 B:255
6	R:0 G:0 B:0	R:255 G:255 B:0	R:0 G:0 B:0	R:255 G:255 B:0	R:255 G:255 B:0	R:255 G:255 B:0	R:255 G:255 B:0	R:255 G:255 B:0	R:255 G:255 B:0	R:255 G:255 B:0	R:0 G:0 B:0	R:255 G:255 B:0
7	R:0 G:0 B:0	R:255 G:255 B:255	R:0 G:0 B:0	R:255 G:255 B:255	R:0 G:0 B:0	R:0 G:0 B:0	R:0 G:0 B:0	R:0 G:0 B:0	R:0 G:0 B:0	R:255 G:255 B:255	R:0 G:0 B:0	R:255 G:255 B:255
8	R:0 G:0 B:0	R:0 G:0 B:0	R:0 G:0 B:0	R:0 G:0 B:0	R:127 G:127 B:127	R:127 G:127 B:127	R:0 G:0 B:0	R:127 G:127 B:127	R:127 G:127 B:127	R:0 G:0 B:0	R:0 G:0 B:0	R:0 G:0 B:0
9	R:0 G:0 B:0	R:0 G:0 B:0	R:0 G:0 B:0	R:0 G:0 B:0	R:0 G:0 B:0	R:0 G:0 B:0	R:0 G:0 B:0	R:0 G:0 B:0	R:0 G:0 B:0	R:0 G:0 B:0	R:0 G:0 B:0	R:0 G:0 B:0

2　bit 表記

さて、1 で何の説明もなく 8 bit という表現を使ったが、bit というのはデータの量を示す単位である。第一回で、コンピュータの内部では全てが二進数で表記されていると述べたが、それに対応してデータの量も、そのデータが二進数何桁で表現されているかで表され、そのときに用いる単位が bit である。たとえば、8 bit といえば、二進数 8 桁の意味であり、16 bit といえば、二進数 16 桁の意味である。8 bit のデータがあれば、
00000000 (二進数表記)　= 0 (10 進数表記)
11111111 (二進数表記) = 255 (10 進数表記)

であるから、0 から 255 までの値を表現できる。したがって、1 ピクセルあたり 8 bit の 白黒のインベーダー画像においては、0 が黒(最小値)、255 が白(最大値)であったのである。1 ピクセルあたり 16 bit であれば、0 から 65535 まで表現でき、この場合白は 65535 になる。多くの場合白黒画像であれば、1 ピクセルあたり 8bit または 16bit で表現され、それぞれ 8bit グレースケール画像、16bit グレースケール画像または、単に 8 bit 画像、16 bit 画像と呼ばれる。カラーでは、現在では R, G, B それぞれに 8 bit をわりあて、一ピクセルあたり 24 bit で表現することが多い。この場合表現できる色の数は、2 の 24 乗で、16777216 色となる。
ちなみに、8 bit = 1 byte であり、byte はファイルの大きさやハードディスクの大きさを表現するときに用いられる単位である。

3　ファイルフォーマット

画像を扱うさい、そのままではファイルサイズがどうしても大きくなりがちである。そのため、画像の特徴をなるべく残したままファイルサイズを小さくする様々な圧縮アルゴリズムが存在し、それに対応して様々なファイルフォーマットが存在する。圧縮アルゴリズムには、完全に元のデータを復元できる可逆圧縮と、一部のデータが失われ、元のデータを復元できない非可逆圧縮がある。あとでその画像ファイルを数値的に解析しようとする場合、決して画像を非可逆圧縮形式で保存してはならない。Jpeg や gif フォーマットが非可逆圧縮の典型である。一方、zip 圧縮や lzh 圧縮などが、可逆圧縮の典型例である。また、このような圧縮を行わないフォーマットも存在する。それぞれに特徴があり、上手に使い分ける必要がある。それぞれのフォーマットは、ファイル名の最後に付ける拡張子で区別する。以下に主ないくつかのフォーマットを説明する。

Tiff フォーマット

汎用の画像フォーマット。ほとんどの画像操作ソフトウェアで扱うことができる。拡張子は.tiff または.tif。非常に自由度が高く、非圧縮、圧縮どちらでも使え、圧縮アルゴリズムも様々な手法が使える。科学データとして画像を撮る場合は、ほとんどの場合非圧縮または可逆圧縮の tiff フォーマットで保存する。

Bmp フォーマット

ビットマップ形式と呼ばれる。非常にシンプルな非圧縮形式のフォーマットで、主にマイクロソフトの windows OS で用いられる。拡張子は.bmp

jpeg フォーマット

写真の保存に適した非可逆圧縮フォーマットで、デジタルカメラの画像保存やインターネットを通じての写真の公開に広く使われる。色数の少ないシンプルな CG (Computer graphics)や線画の保存には向かない。非可逆圧縮のため、jpeg で保存することで、もとのデータの一部が失われるが、ファイルサイズの圧縮率が高い。

gif フォーマット

CG の保存に適した非可逆圧縮フォーマット。特にシンプルな画像の圧縮率に優れる。写真など、色の種類が多い画像の保存には向かない。拡張子は.gif。

画像ファイルの読み込み、書き込みの仕方はそれぞれの画像フォーマットごとに異なり、それが画像を処理するプログラムを書くことを困難にしている。しかし、python の python Imaging Library モジュールを使うと、フォーマットの差異をあまり意識せずにプログラムが書ける。Python imaging library は python 標準のモジュールでは無いので、別途インストールする必要がある。実習用コンピュータには既にインストールされているが、他のコンピュータで使用する場合には、巻末の自分のコンピュータでの環境構築の項を参考にインストールすること。

4　Python Imaging Library

では、実際に使ってみよう。以下のプログラムは読み込んだ 8bit 白黒画像の階調を反転する

プログラム imageinverse.py である。

```
#!/usr/bin/env python
import sys
import Image   # python Imagign library のインポート

infile=sys.argv[1]
outfile=sys.argv[2]

im=Image.open(infile)    #画像ファイルの読み込み

(sizex,sizey)=im.size   # im.sizeは画像imのサイズを表す
for y in range (0,sizey):
    for x in range (0,sizex):
        data=im.getpixel((x,y))   imの座標(x,y)のピクセルの値を読む
        im.putpixel((x,y),255-data)   座標(x,y)に255-dataを書き込む。
im.save(outfile,"TIFF")   #画像ファイルの書き込み
```

invader.tiff に対して使ってみる。

```
% imageinverse.py invader.tif invaderinv.tif
%
```

できた invaderinv.tif を finder からダブルクリックして開いてみよう。白と黒が反転しているのがわかる。

５　画像の読み込みと書き込み

では、さきほどのプログラムを少し詳しく見てみよう。まず、画像ファイルの読み込みは非常に簡単である。

画像オブジェクト名 = Image.open(ファイル名)

たとえば、さきほどの例では、

```
im=Image.open(infile)
```

は、ファイル名 infile の画像ファイルを、画像オブジェクト im に読み込んでいる。画像オブジェクトは変数の型の一種と思って良い。ここで、読み込む画像フォーマットの種類を指定しなくても良いことに注意。フォーマットの種類は自動的に認識される。書き込みは、

画像オブジェクト名.save(ファイル名，ファイルフォーマット名)

```
im.save(outfile,"TIFF")
```

は、ファイル名 outfile に tiff フォーマットで、画像オブジェクト im を書き込むという意味である。つまり、python で読み込んだファイルを任意のフォーマットに簡単に変換することができる。

http://www.pythonware.com/library/pil/handbook/

に使用できるフォーマットの一覧を見ることができる。実際に python の対話モードで試してみよう。

```
% python
>>> import Image
>>> im=Image.open("invader.tif")    #invader.tifを読み込み
>>> im.save("invader.gif","GIF")   # gif形式で保存
>>> im.save("invader.bmp","BMP")   # bmp形式で保存
>>> im.save("invader.jpg","JPEG") # jpeg形式で保存
```

実際に出力されたファイルを開いて確認してみよう。Jpeg形式は若干もとの画像と違って見えるはずである。これはjpeg形式が非可逆圧縮であり、このようなシンプルな画像にその圧縮アルゴリズムが向いていないことを示している。一方gif形式は元の画像と同じように見えるが、これ

は圧縮アルゴリズムがこのような画像に向いているからである。元の画像が写真のような多数の色が使われているような画像の場合、反対にgif形式のほうが元画像と違って見えるはずである。

課題１　フォーマット変換プログラム

画像ファイルを python imaging library が扱える任意のフォーマットに変換するプログラム formatchange.py を書け。インターネットから適当な画像をダウンロードし、その画像に対してフォーマットの変換を実際に行ってみよ。変換前と後の画像もレポートせよ。

使用例:
% formatchange.py test.jpg test.gif GIF
　　# test.jpg を GIF 形式に変換し、test.gif に出力する。
%

５　画像の属性

Image.open で生成される画像オブジェクトは、各ピクセルの値以外にも情報を持っている。その中で特に重要なのは、画像サイズとモードである。

５－１　画像サイズ

画像サイズは以下のようにして得られる。
画像オブジェクト名.size
得られる値は(横方向のサイズ, 縦方向のサイズ) の形式になっている。単位はピクセル。このセットは tupple と呼ばれる形式だが、リスト形式と同じと思ってもらって良い。したがって、imageinverse.py における、
(sizex,sizey)=im.size
は、sizex に横方向のピクセル数,sizey に 縦方向のピクセル数を代入している。

５－２　モード

画像のモードとは、その画像が白黒なのか、カラーなのか、各ピクセルが何 bit の情報を持っているのかを表している。主なモードは以下の通り。
画像オブジェクト名.mode
“1”: 白黒二値画像。白と黒だけで表現。一ピクセルあたり 1 bit で表現。
“L”: 8bit グレースケール。色情報はない。
“RGB”: RGB カラー。一つの色ごとに 8bit。
“CMYK”: CMYK カラー。RGB とはまた別の色指定の方法。

試してみよう。
% python
>>> import Image
>>> im=Image.open("invader.tif")
>>> im.mode
>>> “L”
>>> im2= Image.open("invadercolor.tif")
>>> “RGB”

６　ピクセル操作

では、各ピクセルを実際に操作してみよう。ピクセルの値の読み込みは、
画像オブジェクト名.getpixel((x 座標, y 座標))　である。プログラム imageinverse.py 中の、
　　data=im.getpixel((x,y))
では、画像オブジェクト im の(x, y)座標のデータを data に代入している。書き込みは、
画像オブジェクト名.putpixel((x 座標,y 座標),書き込む値)
である。プログラム imageinverse.py 中の、

```
            im.putpixel((x,y),255-data)
```
では、画像オブジェクト im の(x,y)座標に、値 255-data を書き込んでいる。対話モードで試してみよう。

```
% python
>>> import Image
>>> im=Image.open("invader.tif")
>>> im.getpixel((0,0))
>>> 0
>>> im.putpixel((0,0),255)
>>> im.save("invaderchange.tif","TIFF")
```

これで、invaderchange.tif を見てみよう。(0,0)に当たる左上のピクセルが白に変わっているはずである。

課題 2 pixel の値の操作

Python の対話モードを用いて、invader.tif の pixel の値を三つ以上操作して、invaderchange2.tif を作れ。その作成のために必要な全てのコマンドと結果のファイルをレポートせよ。

7　カラー画像の取り扱い

カラー画像の場合は、1 ピクセルごとに R,G,B それぞれの色成分を持っている。従って、getpixel の返り値は(赤成分, 緑成分, 青成分)の三つの要素を持つ。同様に、putpixel においても、値を(赤成分, 緑成分, 青成分)のように指定する。では、実際に見てみよう。冒頭の imageinverse.py をカラー画像向けに変更すると以下のようになる。

colorimageinverse.py

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

infile=sys.argv[1]
outfile=sys.argv[2]

im=Image.open(infile)

(sizex,sizey)=im.size
for y in range (0,sizey):
    for x in range (0,sizex):
        (red,green,blue)=im.getpixel((x,y))
        im.putpixel((x,y),(255-red,255-green,255-blue))
im.save(outfile,"TIFF")
```

ピクセルの値が三成分を持つようになっただけで、ほとんど同じである。

これを invadercolor.tif に対して使用する。

```
% colorimageinverse.py invadercolor.tif invadercolorinv.tif
%
```

invadercolorinv.tif:

では実際に対話モードを使ってやってみよう。
元の画像は invadercolor.tif を用いる。

```
% python
>>> import Image
>>> im=Image.open('invadercolor.tif')
>>> im.getpixel((5,5))
(255, 0, 255)
>>> im.putpixel((5,5),(127,0,127))
>>> im.save('invadercolorchange.tif',"TIFF")
```

invadorcolorchange.tif を見ると、座標(5,5)の紫が薄くなっているのがわかるだろう。

課題 3 カラー画像の pixel の値の操作

Python の対話モードを用いて、invadercolor.tif の pixel の値を三つ以上操作して、invadercolorchange2.tif を作れ。その作成のために必要な全てのコマンドと結果のファイルをレポートせよ。

８　カラー画像の各色要素の分解と合成

これは非常に簡単である。
(赤成分, 緑成分, 青成分) = 画像オブジェクト名.split()
とすると、カラーの画像オブジェクトは、赤成分、緑成分、青成分の三つの画像オブジェクトに分解される。それぞれの画像オブジェクトは通常の白黒画像のオブジェクトとまったく同じように使用できる。合成するには、
出力画像オブジェクト名=Image.merge('RGB', (赤成分,緑成分,青成分))
とすればよい。それぞれの成分は同サイズの白黒画像オブジェクトである。少し試してみよう。

```
% python
>>> import Image
>>> im=Image.open('invadercolor.tif')
>>> (blue,red,green)=im.split()   #im を各色要素の白黒画像オブジェクトにわける。
>>> blue.putpixel((0,0),255)
>>> green.putpixel((1,0),255)
>>> red.putpixel((2,0),255)
>>> im=Image.merge('RGB',(red,green,blue))   #三つの画像オブジェクトを一つのカラー画像に merge
>>> im.save('invadersplitmerge.tif',"TIFF")
```

結果の invadersplitmerge.tif を見ると、座標(0,0)が青に、(1,0)が緑に、(2,0)が赤に変化しているのが分かる。

課題 4 カラー画像の各色要素への分解

カラー画像の R, G, B それぞれの色成分を抜き出し、三つの画像に分けるプログラム imageseparate.py を書け。それを用いて、インターネットから適当なカラー画像をダウンロードし、三つの成分に分けてみよ。インターネット上の画像はほとんどが RGB カラーで表現されているが、たまに CMYK カラーの画像も存在する。その場合は本実習の方法はうまくいかないので、別の画像を試すこと。

使用例：
% imageseparate.py invadercolor.tif
%
出力
red.invadercolor.tif

green.invadercolor.tif

blue.invadercolor.tif

また、分かれた三つの画像から元の画像を合成するプログラム imagemerge.py を書き、実際の合成してみよ。また、合成する順番を変えるとどうなるかレポートせよ。
使用例:
% imagemerge.py red.invadercolor.tif green.invadercolor.tif blue.invadercolor.tif invaderout.tif
%
この場合の出力ファイル invaderout.tif は元の invadercolor.tif と同じになるように作る。そのように作った場合、順番を変えて、
% imagemerge.py green.invadercolor.tif red.invadercolor.tif blue.invadercolor.tif invaderoutchange.tif
%
出力ファイル invaderoutchange.tif は元の invadercolor.tif と違う色になるはずである。

９　画像の新規作成

画像を 0 から新しく生成することもできる。方法は簡単である。
画像オブジェクト名=Image.new(モード, (横サイズ,縦サイズ),default value)
である。生成した画像は、全ピクセルが Default value で埋められる。カラーであれば、ピクセル値は三要素を持つので、
im=Image.new("RGB",(sizex,sizey),(0,0,0))
とすれば、一面黒の画像が得られる。
8bit モノクロ画像であれば、同様の画像を得るには、
im=Image.new("L",(sizex,sizey),0)
である。生成したあとは、いままでと全く同じに使うことができる。実際に試してみよう。

次のプログラム makesphere.py は、指定した大きさの画像の中心に、指定した半径 r の球の投影像を描く。球の投影像は、以下のように計算される。
画像の中心を(cx,cy)とすると、

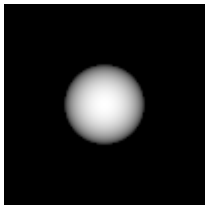
$$V(x,y)=\begin{cases}r^2<(x-cx)^2+(y-cy)^2\text{のとき: }\sqrt{r^2-((x-cx)^2+(y-cy)^2)}\\ r^2>(x-cx)^2+(y-cy)^2\text{のとき: }0\end{cases}$$

この計算における最大値は、(x,y)=(cx,cy)のときに得られる r である。しかしながら、画像の各ピクセルは整数値であり、最大値が余り小さいと十分な階調が得られない。例えば r=5 であれば、V を整数として保存したときには 0,1,2,3,4,5 のいずれかになり、たったの 6 階調しか得られない。また、画像フォーマットの最大値を超える値も許されない。出力する tiff ファイルは 8bit であるから、使用できるレンジは 0 から 2^8-1 である。したがって、V に$(2^8-1)/r$ を掛ければ、8 bit をフルに使うことができる。

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

r=float(sys.argv[1])    # 半径
sizex=int(sys.argv[2]) # 縦方向の画像サイズ
sizey=int(sys.argv[3])  # 横方向の画像サイズ
out=sys.argv[4] #出力ファイル名
cx = (sizex - 1)*0.5   #画像の中心位置を計算
cy = (sizey - 1)*0.5
r2= r**2
```

```
value = (2**8-1) / r #V に掛ける値(value)を計算

im=Image.new("L",(sizex,sizey),0) #グレースケール画像として生成

for y in range (0,sizey):
    for x in range (0,sizex):
        dx = abs(x-cx)
        dy = abs(y-cy)
        d = dx**2 + dy**2
        if (d<r2):
            im.putpixel((x,y),int(value*math.sqrt(r2-d)+0.5))    #Vの計算をし,
valueを掛けて、整数に変換。結果を(x,y)座標に記録。
        else:
            im.putpixel((x,y),0)
im.save(out,"TIFF")
```

使用例:
% makesphere.py 20 100 100 r20.tif
%
r20.tif:



課題 5: 関数を使った画像生成

makesphere.py を参考にして、好きな関数から二次元画像を生成せよ。カラーを使っても良い。

自分のコンピュータでの環境構築

本講義の環境を自宅のパソコン等で再現し、そこで課題をこなせるようにするためには、以下のプログラムをインストールする必要がある。

第一回から第八回まで
python (version2.5 or 2.6) (プログラム本体)
emacs (テキストエディタ)
第九回
rasmol (タンパク質構造 viewer)
第十回
python imaging library (python 用イメージファイル編集ライブラリ)

これらは、MacOSX でも linux でも windows でもインストールでき、以上のプログラムをインストールすれば、本講義で書いたプログラムはどの OS でもそのまま実行可能である。実際、本実習書はほとんど windows の実行環境で書いている。

MacOSX

第一回から第八回までは、標準設定のままで良い。第九回、第十回のためには以下の設定が必要。

1 Xcode Tools の最新バージョンのインストール

Xcode Tools とは、アップル社提供の MacOSX 用のプログラム開発環境セットである。次に述べる MacPorts を利用するために、この Xcode Tools が必要。OS のインストール CD にも入っているが、新しいバージョンをインストールしたほうが良い。
http://developer.apple.com/jp/products/membership.html
から、ADC online membership
に登録し、ADC アカウントにログインすると、メンバーページからダウンロードできる。
ログイン後のメンバーサイトから、Downloads を選び、その次に右側の Downloads のリストから Developer Tools を選択すると現れるページから Xcode Tools のダウンロードができる。ダウンロードファイルを開くと、AboutXcodeTools.pdf のような名前のファイルが見つかるので、それを読めばインストールの方法が書いてある。

2 Macports のインストール

いままでに Unix 系システムで開発されてきた無数の資産を MacOSX に継承するために、ユーザー達が構築したソフトウェアインストールシステムが MacPorts である。これを使うと、様々なソフトウェアが簡単にインストールできる。
少しわかりにくいが、2010/8/11 現在、ホームページ
http://www.macports.org/
の右上の download をクリックして、開く Installing MacPorts のページの上から二番目の段落に、、
“dmg” disk images for Snow Leopard, Leopard and Tiger as a legacy platform, containing pkg installers for use with the Mac OS X Installer. By far the simplest installation procedure that most users should follow after meeting the requirements listed below.
という記述があるが、この中の対応する OS バージョンに合わせた青字(Snow Leopard, Leopard, Tiger のいずれか)をクリックすると最新版のダウンロードが始まる。
あとは通常のアプリケーションインストールと同じである。

3 MacPorts を使ってそれぞれのソフトウェアのインストール

ここまでくれば、ターミナルのコマンドラインから MacPorts が使える。port というコマンドを使う。まず、MacPorts のソフトウェア一覧を新しいものに更新する。
```
$ sudo port sync
```

システム管理者(root)のパスワードを聞かれるので、パスワードを入れてあげると、更新できる。
次に使いたいプログラムが MacPorts で使えるかを調べるには、
port search プログラム名
とすれば良い。たとえば rasmol なら、
```
$ port search rasmol
```
のようになる。これをインストールするためには、
```
$ sudo port install rasmol
```
とすればよい。同様に python2.5 のインストール
```
$ sudo port install python25
```
python2.5 用 Python imaging library のインストール
```
$ sudo port install py25-pil
```
Emacs のインストールは必要ないが、X window 対応の emacs をインストールしたければ、
```
$ sudo port install xemacs
```

ちなみに、インストールしたソフトをアンインストールしたければ、
sudo port uninstall ソフト名
とすれば良い。また、すでにインストールしたソフトの一覧を見るには、
port installed
とする。

4　.bashrc の設定と python へのシンボリックリンクの作成

通常のインストール状態では、tcsh ではなく bash が標準シェルである。bash において.tcshrc と同じ役割をするのは.bashrc である。.bashrc においては、setenv では無く export を使う。自分のマシンであれば、.bashrc は自由に編集できる。

MacOSX には、最初から python が入っているが、MacPort でインストールした python である、/opt/local/bin/python2.5 からでないと、そのままでは MacPort でインストールした各種ライブラリにアクセスできない。まず、以下の一行を.bashrc に追加すること(存在しない場合はこの一行が入った.bashrc を作成すればよい)。
```
export PATH=~/com:$PATH
```
このことによって、コマンド入力時に、まっさきに~/com を探すようになる。次に、
```
$ cd
$ mkdir com
$ cd com
$ ln -s /opt/local/bin/python2.5 python
```
とすれば、python 起動時には、MacPort でインストールされた python (/opt/local/bin/python2.5)が起動するようになり、インストールした各種ライブラリを呼び出せるようになる。

Linux

1　各種 Linux

Linux には、いくつもの種類があり、それぞれ少しずつ異なるソフトウェアインストールシステムを持っているが、本質的にはそれほど変わらない。ここでは、openSUSE 11 を例にとってみてみよう。ちなみに openSUSE11 は、http://ja.opensuse.org/ から無料でダウンロードできる。openSUSE は、豊富なプログラミング環境が標準で付属し、OS としての安定性、設定の容易さのバランスが良くとれている。openSUSE11 の他に有名な無料 Linux としては、
Fedora http://fedoraproject.org/ja/（先進の機能を積極的に取り入れている。その分安定性に難がある。筆者は openSUSE を使う前にはこれを使っていた。）
Debian http://www.debian.or.jp/（正統派の Linux で様々なカスタマイズがしやすい。上級者向きか。）
Gentoo http://www.gentoo.gr.jp/（ソフトウェアの管理システムに特徴があり、一定の人気がある。）
Ubuntu http://www.ubuntulinux.jp/（デスクトップ環境として使いやすい OS として定評がある。）
などがある。また、CD から起動できる Linux としては、

Knoppix http://unit.aist.go.jp/itri/knoppix/
が有名である。これは Windows が起動できなくなったときのデータレスキュー用としても定評がある。

2 openSUSE11 におけるソフトウェアインストール

以下、openSUSE11 が標準状態でインストールされているとしたときの環境構築を述べる。openSUSE では、YaST と呼ばれるソフトウェアでソフトウェア管理を含むほとんどすべての設定ができるようになっている。画面左下の”コンピュータ”をクリックすると開くウインドウの右上あたりに YaST と書かれているメニューがある。これをクリック。これで YaST が立ち上がる。”ソフトウェア”の中の”ソフトウエア管理”から、必要なソフトウェアをインストールできる。
ここから、
rasmol
emacs-x11
python-imaging (python imaging library のこと。)
を通常通りにインストールすれば良い。ちなみに python は標準状態でインストール済みである。”ソフトウエア管理”や、”ソフトウエアリポジトリ”の使い方は openSUSE の入門書に必ず書いてあるので、そちらを参考にしてほしい。

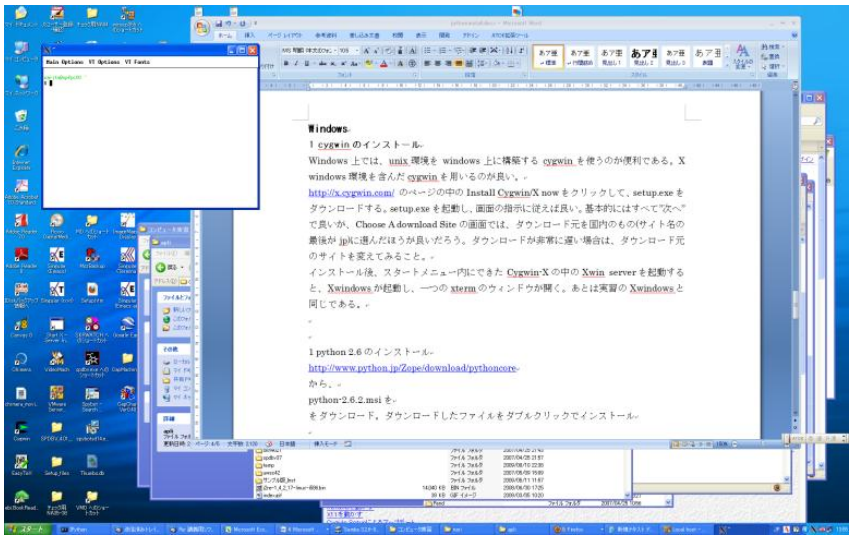
3 bashrc の設定

OpenSUSE でも標準シェルは bash である。MacOSX の項で述べたように、bash の設定ファイルは.bashrc であるが、OpenSUSE においては、特別な設定は必要ない。

Windows

1 cygwin のインストール

Windows 上では、unix 環境を windows 上に構築する cygwin を使うのが便利である。X windows 環境を含んだ cygwin を用いるのが良い。
http://x.cygwin.com/ のページの中の Install Cygwin/X now をクリックして、setup.exe をダウンロードする。



setup.exe を起動し、画面の指示に従えば良い。基本的にはすべて”次へ”で良いが、Choose A download Site の画面では、ダウンロード元を国内のもの(サイト名の最後が jp)に選んだほうが良いだろう。ダウンロードが非常に遅い場合は、ダウンロード元のサイトを変えてみること。
インストール後、スタートメニュー内にできた Cygwin-X の中の Xwin server を起動すると、Xwindows が起動し、一つの xterm のウィンドウが開く。次の画面は Xwin server 起動直後の画面で、windows の画面の中の左上に xterm のウィンドウ(本実習におけるターミナルに相当するもの)が開いている。

2 各ソフトウェアのインストール

python 2.6 のインストール
http://www.python.jp/Zope/download/pythoncore
から、
python-2.6.2.msi をダウンロード。ダウンロードしたファイルをダブルクリックでインストー

ル。

python imaging library のインストール
http://www.pythonware.com/products/pil/ から、
Python Imaging Library 1.1.6 for Python 2.6 (Windows only)をクリック。
ダウンロードしたファイルをダブルクリックでインストール。

rasmol のインストール
http://www.openrasmol.org/
から、RasMol Latest Windows installer をクリックしてダウンロード
ダウンロードしたファイルをダブルクリックでインストール。

Emacs は cygwin に最初から X window 対応版が含まれているので、インストールは必要ない。

４ .bashrc の設定

Cygwin における標準シェルも bash である。以下の一行を.bashrc に加えること。

```
export PATH=/cygdrive/c/Python26:$PATH
```

MacOSX のときと同様に、cygwin にはもとから python が含まれているが、ライブラリを追加するのは容易でない。そこで、python-2.6.2.msi からインストールされた python (/cygdrive/c/Python26/python)を先に検索させるようにしている。このようにすると、上記手順でインストールした各ライブラリを使えるようになる。

ちなみに、cygwin においては、c ドライブは/cygdrive/c としてアクセスできる。他のドライブも同様に/cygdrive/ドライブレターでアクセスする。/cygdrive/c/Python26/python は、c ドライブの下の Python26¥python と同じである。

５ python の対話モード

cygwin の xterm と windows 用の python の相性の問題で、windows 用 python は xterm 上で対話モードが起動できない。Windows 用 python 付属の対話モード用プログラムを用いる。スタートメニューの”すべてのプログラム”の Python2.6 の中の Python(command line)を選択すると、対話モードを起動できる。同じ Python2.6 の中に、IDLE というより高機能な対話モード環境もできているが、IDLE からは turtle による描画がうまくいかない。Turtle を使わないなら、IDLE を使っても良い。