Python GUI 入門
–Tkinter から PIL まで–

田中義文

なぜ Python か？

今日では多くの優秀なプログラムが公開または市販され，誰でもが通常の業務に利用できるようになった．特に Microsoft Office に代表される Word や PowerPoint は世界共通製品となり，USB メモリーチップにデータを書き込めば，どの様な場所に出張しようと，そこに PC が有れば仕事が継続できる．

一方，データベースや表計算，そして CPU 間の LAN 操作などの機能を一体化し，さらに目的に特化したプログラムとなると状況は一変する．こまごまとした仕様を作りあげてソフト業者に発注しなければならない．費用も莫大となり，満足する状況までにはなかなか至らない．その原因は，利用者が，専用プログラムには少なくとも Microsoft Office 程度の汎用機能を持った上で，種々の専用機能を実現できているだろうと勝手に想像するからである．

専用プログラムの不備な部分は，業者に訴えて，改善してもらうより他に方法がないが，時間も費用もかかる難問である．結局，個人プログラマーになって解決するのが得策である．そこで，どのプログラム言語を利用すれば最も効率的にプログラムが開発できるかを考える必要がある．

1. 言語が読みやすいこと，
2. すぐに実行できること，
3. 仕様の変更や拡張が容易であること，
4. GUI (Graphic User Interface) を装備し，利用しやすいこと，
5. データベースとしてハードディスクが取り扱えること，
6. LAN 通信ができること，
7. リモート CPU 操作ができること，
8. Linux, Windows, MAC などのプラットホームで使用できること，

などがあげられよう．これだけ出来れば，インターネットで日本全国どこでもデータベースが構築できる．

プログラム開発において，C 言語は最も動作が速いプログラムを作成するが，完成までに何度もコンパイル，デバグを繰り返さねばならない．また，GUI は X window システムを呼び出す必要がある．C 言語はプロのプログラマーでないかぎり，実用言語として利用すべきではない．

今日よく利用される Microsoft Basic は，実はオプションが複雑で使いづらいものの一つである．JAVA や Tcl/Tk は非常に優れた言語であり，Linux, Windows, MAC の各プラットホームで動作する．しかしながら，言語は冗長で type miss の繰り返しといって過言ではない．

筆者が推薦する言語は Python である．文末やブロック文を示す「;」や「 」がなく，書きやすくてタイプミスが少ない．C 言語と Python 言語を比較してみよう．以下のプログラムは”Hello World.”を 5 回繰り返し端末に表示するだけのプ

京都府立医科大学大学院医学研究科麻酔科学教室

ログラムである．

C 言語

```
#include <stdio.h>
main()
{
   int i;
   for(i=0;i<5;i++) {
      printf("Hello World.\n");
   }
}
```

Python 言語

```
#! /usr//bin/env python
for i in range(0,5):
   print "Hello World."
```

[1]

C 言語では 8 行必要であるが，Python では 3 行で実現できる．しかもインタープリータ (対話方式) であるから，コンパイル，リンクの手間が不要である．というわけで Python は上記の全ての条件を満たしている言語だといえる．Python でプログラムの見通しを立て，必要なら C 言語に書き直せばよい．始めから C 言語でのプログラム開発は労が多すぎて成功する可能性は低い．CPU に同じ仕事をさせるのに，プログラムの行数が 1/2〜1/3 になれば，それだけ間違いが少なくなり，開発効率も飛躍的に上昇する．その意味でも Python 言語を用いてオブジェクト指向に慣れることが重要である．オブジェクト指向については後述するが，プログラムのグループ化，継承ができ，短いプログラムピースを積み重ねて全体のシステムを構築する手法である．

Python のインストール

Python は無料のパブリックドメインプログラムであるから，インターネット接続できる機種であれば，いつでも導入できる．また多くの解説がホームページ上に公開されているから，悩むことは少ないと思われる．

Windows と MAC:

Google 検索で「Python インストール」を行い，そのサイトの指示に従う．コンパイル済みの実行環境が入手できるから容易である．GUI モジュールである Tkinter は装備されている．追加して，別の GUI モジュールの wxPython も「wxPython インストール」を検索してインストールする．wxPython は Tkinter よりも洗練された GUI が作成できる．最も容易にインストールできる．MAC も同様にインストール出来るはずである．

ubuntu:

ubuntu は近年最も活発に活動している Linux デベロッパーで，雑誌の付録の CDROM，ネット上で ubuntu インストーラを導入すれば容易に ubuntu Linux が動作する．Windows のディスクスペースに ubuntu を構築する方法と，ディスクを 2 分割 (パーティティション) して Windows と ubuntu とのデュアルブートを行う方法との 2 通りの選択ができる．どちらも快適に動作するが，当然 Windows 上で動作すれば使用できるディスク容量は少ない．最大 30 G バイトしか占有できないが Python の動作には十分である．ubuntu に慣れるまではこの方法で十分である．Ubuntu では Python 言語はバイナリーファイルで提供され容易に動作する．

Python インストールは端末 (gnome-terminal) を開け，以下の命令を入力する．

```
sudo apt-get install python
sudo apt-get install python-tk
```

また wxPython もバイナリーファイルで動作可能である．

[1] 枠で囲まれた部分はソースプログラム

Python 言語仕様

書式

Python のプログラム書式は C 言語と違って，改行に意味が有る．そして行頭からのインデント (空白) に意味があり，同じインデントは同一のブロック文と解釈される．このルールにより C 言語の「;」と「{ }」記号から開放されることができる．

for 文，if 文，while 文などでは条件式の後に「:」で改行し，次の行からインデントを行う．関数指定である def 文，クラス指定である class 文も同様に「:」で改行し，次の行からインデントを行ってブロック文の範囲を定める．

定数と変数

定数には整数，実数，複素数，文字列を受け付ける．これらの定数を受け付ける変数は前もっての宣言が不要である．前項で示した C 言語と Python 言語との違いがここにある．定数 i についての宣言をしていない．定数を「=」つまり，等号で変数に代入するときに，定数がどのような種類であるかを Python は自動的に判断して変数領域に書き込む．

実は Python では変数そのものがオブジェクトである．オブジェクトとは，変数がもつ定数の値，属性 (性質) を含み，更にその演算方法まで用意されている．従って，"a = 1"とプログラムに記述すると，変数 a のメモリー領域に値，属性，演算法の全てが記録される．従って，何バイトのメモリーを変数 a が消費したか利用者にはわからない．

オブジェクト指向のプログラムであるために，複素数どうしの演算も通常の「+，-，* /」演算子を用いてできる．変数が文字列どうしであれば，「+」 記号は文字列の連結を意味する．当

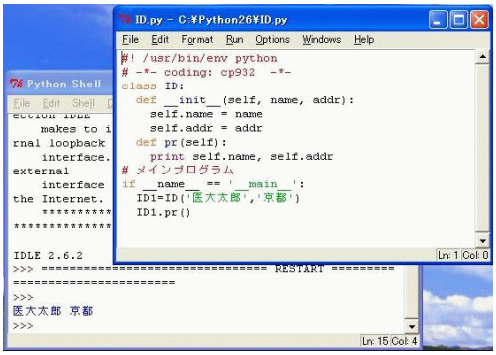

図 1 Windows での Pytho 駆動画面

然文字列の引き算は意味がないからエラーが返る．このように同じ演算子を用いて，その要求に見合った仕事ができることをオブジェクト指向では「多態性」と呼んでいる．この性質により，プログラマーは処理関数の名称を覚える量が少なくなり，エラーも防止できる．

いろいろな変数

Python では整数，実数，複素数，文字列などの単純変数の他に，リスト，タプル，辞書形式と呼ばれている複合変数を取り扱うことができる．この説明のために対話形式で Python を駆動させよう．

Linux なら端末を開き，python ↵と入力する．Windows であれば Python 2.6 IDLE をスタートプログラムで選択する．色々表示され，最後に「>>> 」とプロンプトが表れる．これでプログラムを入力すると対話形式で実行する．簡単なプログラムでも別に編集用ウィンドウを開いてそこにプログラムを書き，コピーペーストで行入力すると，入力作業が楽になる．また編集用ウィンドウで Run Module を選択すると Shell 画面に結果やエラーが表示される．

1) 単純変数

まず単純変数の演算から始めよう．

```
>>> 2*3.14
6.2800000000000002
>>> a='abc'
>>> b='def'
>>> a+b
'abcdef'
>>> (1+1j)*(1-1j)
(2+0j)
>>> from math import *
>>> log(10.0)
2.3025850929940459
```

[2]

前述の単純変数の演算は全て思う通りの結果になっている．「from math import *」は数学演算用モジュールの読み込み命令で通常使用する数学関数は全て用意されている．

2) リスト

リストは順序だった変数の集まりである．変数 (リスト項目) は何でもよい．リストは項目ごとに「,」で区切り，全体を「[ ]」で括る．「+」演算子はリストの連結ができる．

```
>>> a=[1,2,3]
>>> b=[4,5,6]
>>> c=a+b
>>> c
[1, 2, 3, 4, 5, 6]
>>> a.append(b)
>>> a
[1, 2, 3, [4, 5, 6]]
```

となる．最後の行で，変数 c ⏎を行うとその内容が返る．なお，append メソッドを使うと最後の項目にリスト b が 1 項目として追加される．

リスト項目の選択はリスト変数に [m:n] の添字をつける．m は始まりを指定し，先頭番号は 0 である．末尾は n-1 になる．上記のリスト変数 c をもちいて，

```
>>> print c[1:4]
[2, 3, 4]
>>> print c[0]
1
```

となる．

リストには追加，削除，リストの個数，検索，リスト整数の作成，など色々のメソッドが備わっている．help() ⏎と入力すると，簡単な説明が表示されるが，成書を参考にした方がよい．

また，リストはリストのネスティングができ，一次元配列，二次元配列であるマトリックスも表示，演算ができる．

リストを重ねて表記する．

```
>>> a=[[1,2,3],[4,5,6],[7,8,9]]
>>> a
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> a[2][1]
8
```

0 行，0 列から始まり，行，列の順に記述すると配列因子が指定できる．ベクトル，マトリックスの演算は関数演算プログラムを作成しなければならない．前項の Python プログラムで使った，range(0,5) はリスト関数のメソッドで，[0，1，2，3，4] が返る．従って，

```
for i in range[0, 5]
```

は，変数 i に 0，1，2，3，4 を順に代入して作業を行えというループである．

3) タプル

タプルはリストに似ているが，項目を変更することができない．リストは「[ ]」で括るが，タプルは「()」で括る．関数の引数並びはタプルと解釈される．

```
>>> (a,b)=(1,2)
>>> a
1
>>> b
2
>>> d=(a,b)
>>> d[0]
1
>>> d[1]
2
```

[2]影枠で囲まれている部分は Python 対話モードの表示を示す

と，見れば分かる．1 項目だけのタプルは d=(1,) と最後に「,」が必要である．d = 1,2,3 は括弧が省略されたタプルである．したがって，
a,b,c = 1,2,3 と記述すれば，一行で 3 項目の代入になる．

4) 辞書

辞書変数は見出しと内容を示す連想配列でハッシュ法を用いている．表記法は
{ 見出し:内容，...}
である．見出しは数値，文字列などが適当で，リストなど複合変数は指定できない．内容は確定した変数であれば何でもよく，リストを入力してもよい．

```
>>> d1={}                              #空辞書
>>> d1={'a':1,'b':2,'c':3}
>>> print d1['a'], d1['c']
1 3
>>> d1.keys()
['a', 'c', 'b']
>>> d1.values()
[1, 3, 2]
```

と，見れば分かる．辞書オブジェクトには，追加，削除，検索などのメソッドが備わっている．名前と住所とで住所録を作成できる．内容にリストを用いれば，住所，電話番号，職業など，写真ファイル名などいくらでも複雑なデータベースが構築できる．せっかく作ったデータベースの保存はというと，次に示す pickle(漬物) と shelve(本棚) というディスク入出力ルーチンがある．

a) pickle

辞書機能を系統立てて複雑にしていくとデータベースに発展する．しかし，このデータベースをハードディスクに記録できなければ実用化ができない．pickle はこのためのユーティリティで，クラスで作成したインスタンスがディスク読み書きできる．

例：家族構成を辞書型とリストで示す．

A 家には A1, A2 で構成され，B 家では B1, B2, B3 で構成される．そのデータを d とすると，
d='A':['A1','A2'],'B':['B1','B2','B3']
と表記できる．もちろん操作性のよい GUI 入力プログラムを作り，スケールの大きなリストを完成できるが，今は，この完成したデータを pickle でディスクに書き込む．

プログラム： pickle 書き込み

```
import pickle
d={'A':['A1','A2'],'B':['B1','B2',
  'B3']}
object = d
file = open('picle.dic', 'w')
pickle.dump(object, file)
file.close()
```

ファイル名 picle.dic にデータが書き込まれる．

プログラム： pickle 読み取り

```
import pickle
file = open('filename', 'r')
object = pickle.load(file)
d = object
print d
file.close()
```

pickle はファイルをオープンしたままでの追加書き込みはできない．全てを読み取り，一旦ファイルを閉じて，内容を変更し，再度ファイルを開いて，書き込む操作が必要がある．これが shelve との違いである．つまり，pickle は全てのオブジェクトを一度に書き込み，また読み込む．キー検索はできないから，(漬物) と名付けたと思われる．

b) shelve

shelve は pickle と同様に辞書のディスク保存ができるが，キーインデックスによる直接保存だと思えばよい．

例：shelve 書き込み

```
import shelve
dbase = shelve.open('filename')
object = ['A1', 'A2']
dbase['A']=object
dbase.close()
dbase = shelve.open('filename')
object = ['B1', 'B2','B3']
dbase['B']=object
dbase.close()
```

と直接ディスクに書き込む．

例：shelve 読み取り

```
import shelve
dbase = shelve.open('filename', 'r')
object=dbase['A']
print object # ['A1', 'A2'] が返る
dbase.close()
```

とキーインデックスを使って，直接ディスクから内容が取り出せる．

以上で関数やクラスをのぞいて，Python プログラミングのための基本知識は終了するが，リストと shelve のサポートのおかげて，簡便に住所録程度のデータベースが構築できるようになったことを理解してほしい．import できるモジュールは math，pickle，shelve を使用したが，これ以外に多くのモジュールがネット上に公開されている．

関数操作法

関数の概念

少し大きなスケールのプログラムを行って困ることは変数名 (ラベル) である．作成当初はその意味を記憶しているが，数日も経つと忘れるからだ．そこで，一時的に使用する変数名は全体に影響が生じないように関数の概念が生まれた．関数の使用法さえ覚えておけば．関数内の変数名は全体のプログラムに影響をあたえない．すると，一連の長いメインプログラムをステップ 1，ステップ 2，… と区切りのよい所で関数登録し，関数の呼び出しだけでメインプログラムになる．

このようにプログラムを関数で区切ると，例えば，日時や，人名など関数を越えて，引用できる変数定義が欲しくなる．そこで生まれたのがグローバル変数でメインプログラムに登録する変数名である．それに対して関数内で登録する変数を局所変数という．

1) 関数の書式

```
def 関数名 (引数 1, ..., 引数 n):
  文
  文
  return 返り値
```

と記述する．return は返り値がなければ必要ない，引数は無くてもよい．但し，関数の呼び出し側と関数定義と引数が一致しなくてはならない．例を上げると，

```
def pr():
  print 'aaa'
# メインプログラム
pr()
print 'bbb'
```

とプログラムすると，端末に

```
aaa
bbb
```

と表示される．つまり始めに pr() を実行して aaa が表示され，次にメインプログラムの print 'bbb' を出力したわけである．

引数を出力するプログラムは，

```
def pr(x):
  print x
# メインプログラム
pr('abc')
pr(123)
```

で確かめられる．つまり，メインの関数呼び出しで引数 x に対応する部分にデータを記述すると，始めに abc 次に 123 と出力する．文字変数も数値も同じ print 文でよい．これもオブジェクト指向のよい所である．引数を増やす場合は「,」

で区切る．この（ ）構造は前述のタプルである．したがって,「pr(x)」と書いても「pr (x)」と書いてもよい．

関数内の局所変数とグローバル変数について述べる．メインプログラムで定義した変数は関数内で引用できる．

```
def pr():
  print x
# メインプログラム
x=3
pr()
```

は 3 を返す．しかし，

```
def pr(x):
  x=x+1
  print 'sub',x
# メインプログラム
x=3
pr(x)
print 'main',x
```

は，始めに sub 4 を出力し，次いで，main 3 を返す．関数内の変数は局所変数であって，メインプログラムに影響を与えない．しかし，global x を関数内で宣言すると，関数内の x が加算され，メインプログラムの x を書き直す．

例：global 宣言

```
def pr():
  global x
  x=x+1
  print 'sub',x
# メインプログラム
x=3
pr()
print 'main',x
```

は，始めに sub 4 を出力し，次いで，main 4 を返す．上記の例で def pr(x) と記述すると，局所変数とグローバル変数が同じだとのエラーを知らせてくれる．

2) 様々な引数操作

関数の引数は前述のごとく数と順番を会わせる必要があるが，Tkinter などの GUI ソフトでは引数のデフォルト値を前もって準備したり，自由にオプション指定を行ったりしている．そこで，3 種類の引数設定法を述べる．

a) 可変長引数

可変長引数は引数をいくらでも受け付ける．書式は引数名の前に「*」(アスタリスク) を付ける．例を示す．

```
def fun(*arg):
  for i in arg:
    print i
# メインプログラム
x = 1; y = 2; z = 3
fun(x, y, z)
```

は，数値 1 2 3 を出力する．通常の引数と可変長引数とが混在する場合は通常引数を先に書く．

b) デフォルト引数

例えば GUI で背景色は白がデフォルトであるが，それを赤に指定する場合などに利用できる．

```
def fun(a,bg='white'):
  print a, 'bg =',bg
# メインプログラム
fun(3)
fun(5,bg='red')
```

は，始めに 3 bg = white，次いで，5 bg = red と出力する．可変長引数の場合と同じ様に通常の引数は先に指定する．

c) キーワード引数

キーワード引数はキーワードとなる変数とその値を関数に与える．書式を次に示すようにアスタリスク 2 個を引数の前に書く．

```
def fun(a,**b):
  print a,b
# メインプログラム
fun(3)
fun(5,bg='red')
```

は，始めに 3 {}，次いで，5 {'bg': 'red'} と辞書形式で出力する．可変長引数の場合と同

じ様に通常引数は先に指定する．

通常の引数を含めて 4 種の引数を示したが，これらの記述法の順番は，通常引数，デフォルト引数，可変長引数，キーワード引数の順である．

引数の戻り値についても複数になればリスト形式，そして辞書形式にすれば任意のキーワードをメインプログラムに返すことができる．

クラス操作法

クラス

クラスとは，データおよび関数の上位に存在するオブジェクトの雛型であり，代入操作によりインスタンスを作成する．したがって，クラスで作成されたオブジェクトは具体的な存在であるからインスタンスと呼ばれる．また，クラスに属するデータを処理する関数をメソッド (処理法) という．

始めにクラスのデータ部分について述べる．例えば C 言語で構造体とか，レコードと呼ばれているデータ構造がある．これは，いくつかのデータを集めて一つの構造体データとして扱うことができる．例えば，名前と住所は対の関係になっており，ID1.name とすると，1 番目の名前が呼び出され，ID1.addr とすると，その人の住所がわかる．ID1 とは上記の構造体名となる．これを Python のクラスで実行すると，

```
class ID:
  def __init__(self, name, addr):
    self.name = name
    self.addr = addr
# メインプログラム
ID1=ID(’医大太郎’,’京都’)
print ID1.name
print ID1.addr
```

となる．まず，class はクラス宣言であり，空白を置いてクラス名を書く．クラス名の次に「()」を記述してもよく，括弧の中に上位のクラス名を記述すると，指定したクラスの全機能が継承される．(上記の例では使っていないが，Tkinter の説明では他用している．) 行末には「:」を記述する．次行は__init__関数を定義する．この関数はクラスの初期設定と呼ばれ，具体的なインスタンスの性質を決める．引数を見ると，self に始まり，name と addr が続く．self は具体的なインスタンス名が引用される仮の名称として使われる．name と addr は前述の構造体メンバーに対応する．次の 2 行は引用した引数がクラスメンバーとして代入される．

メインプログラムを見ると、始めに

ID1=ID(’医大太郎’,’京都’)

と書き，インスタンス ID1 を作成する．これがクラスの代入操作である．引数はクラス構造体を作るだけだから，引用する名前と住所に対して self の接頭語を付けて代入する．__init__関数を実行することにより，この 2 行でインスタンスにデータが記録される．

次の 2 行はインスタンスの確認のための端末出力である．インスタンスメンバーは「インスタンス名．メンバー名」とピリオッドがオブジェクト書法の分離子となる．メソッドの呼び出しも同様の表記である．

対話形式で実行すると，

```
>>> class ID:
...   def __init__(self, name, addr):
...     self.name = name
...     self.addr = addr
...
>>> ID1=ID(’医大太郎’,’京都’)
>>> print ID1.name
医大太郎
>>> print ID1.addr
京都
>>>
```

と結果が返る．

クラスメソッドの作成は容易である．クラスメンバーを表示する関数 pr() を作成する．関数のルールに従って記述すればよい．しかし，第 1 引数に self を忘れずに書く必要がある．また，self で始まる変数はクラス内でグローバル変数として引用できるが，__init__関数の中で，引数や式の右辺に書かれている name や addr などは，定義した関数内でしか引用できない局所変数である．従って，pr() 関数内で print name, addr と書いても変数名が引用できないとエラーが返る．

```
class ID:
  def __init__(self, name, addr):
    self.name = name
    self.addr = addr
  def pr(self):
    print self.name, self.addr
# メインプログラム
ID1=ID('医大太郎','京都')
ID1.pr()
```

とすれば，

```
>>> class ID:
...   def __init__(self, name, addr):
...     self.name = name
...     self.addr = addr
...   def pr(self):
...     print self.name, self.addr
...
>>> ID1=ID('医大太郎','京都')
>>> ID1.pr()
医大太郎 京都
>>> ID2=ID1
>>> ID2.pr()
医大太郎 京都
>>> ID3=ID('医大次郎','東京')
>>> ID3.pr()
医大次郎 東京
>>>
```

と，端末に'医大太郎 京都' と表示する．また，ID2=ID1 のようにクラス構造体のコピーは容易である．ID3 の定義のようにいくらでもインスタンスの作成ができる．

プログラムの保存と実行

クラスの継承に話しを移す前に，クラス ID を実行するプログラムを保存して，バッチ操作ができる正しい書法を紹介する．

```
#! /usr/bin/env python
# -*- coding: euc-jp -*-
class ID:
  def __init__(self, name, addr):
    self.name = name
    self.addr = addr
  def pr(self):
    print self.name, self.addr
# メインプログラム
if __name__ == '__main__':
  ID1=ID('医大太郎','京都')
  ID1.pr()
```

第 1 行は Python インタプリータを働かすための呼び出し行である．第 2 行はこのファイルコーディングが euc-jp を使用していることの宣言である．Windows の文字コードは Shift-Jis であるから，この行は以下のように変更する．

```
# -*- coding:  cp932 -*-
```

続いて，class 定義を行う．

#の行はコメント行と解釈されて，何を書いてもよい．メインプログラムと記述しているコメント行の次の if 文は，もし，これがメインプログラムであれば，以下の文を実行せよ，という意味である．そこで，先ほど実行した命令をインデントをつけて記述する．

上記のデータに対して，クラス名と同じファイル名に拡張子「.py」を付けることが習慣的に決まっている．すなわち「ID.py」となる．Linux であればプログラムファイルに実行命令をつける．

```
chmod +x ID.py
```

として，端末から ID.py と入力すれば実行する..Windows であれば，Python26 のディレクトリに保存し，Run Module を行えば実行する．

クラスの継承

クラス ID に職業項目 (job) を追加するクラス IDJ を作成する．既に，名前，住所のクラス ID を作成しているから，そのリソースを利用して新しいクラス IDJ を作成すると開発効率がよくなる．この方法を継承という．なお，継承元となるクラス ID はスーパークラスと呼ばれている．

クラス IDJ を正しく書くと，

```
#! /usr/bin/env python
# -*- coding: euc-jp -*-
from ID import *
class IDJ(ID):
  def __init__(self, job, *ext):
    self.job=job
    ID.__init__(self, *ext)
  def pr(self):
    print self.job, self.name,
      self.addr
# メインプログラム
if __name__ == '__main__':
  ID1=IDJ('MD','医大太郎','京都')
  ID1.pr()
```

とすれば，端末に'MD 医大太郎 京都' と表示され，ほんの少しの変更だけで，職業欄が追加される．また，同じ pr() 関数で，項目表示ができている．つまり，継承元の関数がオーバーライトされて，新しくなる．このようなメソッドの変更は以前に紹介した多態性を利用している．

新しく作成した IDJ クラスを見ると，始めに，

```
from ID import *
```

と記述している．これはモジュール ID から全てのファイルを読めという意味である．次に，

```
class IDJ(ID):
```

となって，クラス ID を継承して新しいクラス IDJ を宣言している．次の初期設定関数__init__ の引数では，job，name,addr としてもよいが，せっかく継承を利用しているから，job だけを宣言し，残りの引数を*ext として可変長引数を利用した．そのことにより name，addr は*ext にタプルとして挿入される．

job をクラス変数 (self.job) に代入し，次の行は

```
ID.__init__(self, *ext)
```

となっている．この意味は，継承元の ID クラスに引数を渡して，初期設定せよという意味である．この操作でスーパークラスである ID が初期化され，self.name，self.addr がクラス内で有効になる．ext のアスタリスクを外すとエラーになる．初期設定はここまでで，pr() 関数は ID クラスを利用せずに新しく print 文を作成した．

クラス ID と IDJ の例を見て，たった 1 項目の追加のためにこのように大がかりの変更をしなくても良いのでは，と思われるかもしれないが，実用にしているプログラムは少なくとも数百行になる．この時，1 項目を増やすために全体のプログラムを再度点検して設計変更するか，それとも継承ができるオブジェクト指向で設計当初よりプログラミングしておくかを考えれば答えは自ずからきまるであろう．また，多くのプログラマーと協力して一つの巨大システムを作り上げる場合，この方法なくして，プログラム開発はあり得ない．次章で紹介する Tkinter GUI では文書エディター，図形エディターにメニューボタンを継承して利用している．短いプログラムステップの割には豊富なバリエションが実現できるから，この際クラスと継承の概念は是非習得してもらいたい．

演算子としてのクラスの利用

今までメソッドの第 1 引数に self を用いてインスタンスの作成と，メソッドの利用に当ててきたが，第 1 引数は同じクラスのインスタンスであれば self と異なる変数を指定してもよい．クラス ID の pr() 文のあとに prname(x,y) を

追加して、以下のようにプログラムする.

```
class ID:
  def __init__(self, name, addr):
    self.name = name
    self.addr = addr
  def pr(self):
    print self.name,self.addr
  def prname(x,y):
    print x.name
    print y.name
```

プログラムを走らせると,

```
>>> a=ID(’医大太郎’,’京都’)
>>> b=ID(’医大次郎’,’大阪’)
>>> ID.prname(a,b)
医大太郎
医大次郎
>>> a.prname(b)
医大太郎
医大次郎
```

となる. つまり, a.xxx(b) という形式の演算処理が出来ることがわかる. これはリストに新たな項目を追加する append() メソッドと同じ形式である. この手法は色々な場面で応用できる. 繰り返しになるが, def __init__() 関数でクラスの性質を規定し, メソッドでそのクラス演算処理を定めることができる. この応用として, x-y 平面の 2 点間の距離を求めるクラスを作成する.

```
import math
class Point:
  def __init__(self, x1, y1):
    self.x = x1
    self.y = y1
  # 2 点間の距離を求める
  def distance(p1, p2):
    dx = p1.x - p2.x
    dy = p1.y - p2.y
    return math.sqrt(
      dx * dx + dy * dy)
```

実行させると,

```
>>> a=Point(0,0)
>>> b=Point(1,1)
>>> a.distance(b)
1.4142135623730951
```

と確かに 2 点間の距離が求められている. 三角形の面積, 立体図形の距離, など種々の応用が可能である.

よく利用する関数

これまで Python の骨格となるプログラム書法を述べたが, 実際は sys, os, string, math などのモジュールを取り込んで実用的なプログラムに仕上げることになる. 本項ではよく利用する関数を取り上げて使い方を述べる.

★ 数字を文字列にする

```
>>> str(123)
’123’
```

★ ファイルを読む

a) 1 行毎に読む方法

```
file=open(’ID.py’, ’r’)
while 1:
  line=file.readline()
  if not line: break
  print line,
```

line の後に「,」を付けると print 文の改行が抑止される.

b) ファイルをまとめて一気に読む.

```
file=open(’ID.py’, ’r’)
for line in file.readlines():
   print line,
```

★ ファイルに書く

a) ファイル書き込み:

```
fn=open(fname,"w")
fn.write(data)
fn.close()
```

b) 日本語 euc コード書き込み:

```
fname=’tmp.txt’
fn=open(fname,’w’)
for i in range(3):
  data=u’医大太郎\n’
  data1=data.encode(’euc_jp’)
  fn.write(data1)
fn.close()
```

こんな書き方もできる．

```
fname='tmp.txt'
fn=open(fname,'w')
for i in range(3):
  fn.write(
    u' 医大太郎\n'.encode('euc_jp'))
fn.close()
```

★ 関数の代入と間接呼び出し

```
def echo(var):
  print var
x=echo
x(' 医大太郎')
```

★ コマンドラインの取り扱い

命令の引数はよく利用する．

```
#!/usr/bin/env python
import sys
for item in sys.argv:
   print item
```

★ 文字列の数値計算 (電卓)

```
z=eval("2+3+4")
print z
9
```

```
from math import *
z=eval('sin(1.0)')
print z
0.841470984808
```

★ 文字列やファイルスクリプトの実行

a) メモリーにスクリプトを作成

```
cmd='''
for i in range(3):
  print ' あいうえお'
'''
exec cmd
```

とすると,「あいうえお」が 3 回端末表示する．

b) ファイルにスクリプトを作成

```
vi a.txt
for i in range(3):
  print ' あいうえお'
```

とソースファイルを作成する．

python を対話モードで駆動し
execfile('a.txt')
とすると,「あいうえお」が 3 回端末表示する．

c) バッチファイルを対話形式で運用

実は最もデバグに効化的な方法を紹介する．

```
vi Ved3.py
最後の行のイベント待ち受け行
mainloop() をコメント行にする．
#c.mainloop()

execfile('Ved3.py')
とすると，対話形式で実行でき，
デバグがしやすくなる．
```

[3]

オブジェクト指向の考え方

従来のプログラム書法は動詞である命令と目的語が分離されており,「何をどうするか」という考えが中心であったが，オブジェクト指向のプログラミングでは動詞部分がメソッドに置き換えられて,「何がどうなるか」という形式に変更されたと思えばよい．目的語を必要とする他動詞中心の英語的発想ではなく，自動詞中心の日本語的発想，もしくは文章の最後に動詞が列ぶ日本語文法だといえる．

オブジェクト指向とは整数の'+' である足し算記号が，リストに対して項目の追加演算子になり，また文字列の連結子になる．変数がそのような演算情報を組み込んだ存在だと思えばよい．

また，われわれプログラムの利用者は，言語学者ではないのだから，オブジェクト指向を大上段に構えて，始めから種々のメソッドを構築する必要はない．その場面に出会った時，問題解決のための方法を考え，継承を利用してオブジェクトの幅を広げていくだけで十分である．

[3] イベントは Tkinter などの GUI プログラムで使用されている．

GUI Tkinter の使い方

なぜ Tkinter か？

Tkinter は Tc/Tk の GUI 部分（Tk）を Python で動作するモジュールである．Tcl/Tk のプログラミングではラベル ，ボタン，スライダーなど全ての GUI ウィジェットはグローバル変数で登録しなければ画像表示ができない欠点がある．そのため少し複雑な GUI 画面を作成すると，ウィジェット変数名の混乱に陥る．その欠点を克服する手段がクラスの概念で，オブジェクト指向でプログラミングができる．このインターフェースとなるモジュールが Tkinter である．また，Tcl/Tk 独特の癖である$記号を使わず，set 命令でなく，= 記号で代入操作ができるため，Tcl 言語に比べてプログラムが読みやすくなる．

実は，まだ Tkinter についての詳細なテキストがないが，Tcl/Tk の on line マニュアルと Tkinter の変換方式に慣れれば，書法は検討がつく．インターネットでも多くの資料が公開されている．また，今のところ，Linux 系の日本語処理はコピー，ペーストが不安定であるが，Windows では CTRL+C，CTRL+V の組合せで快適にコピー，ペーストができる．労力を惜しまなければ自作のエディターも作成できる．もし，Tcl/Tk のプリグラミング経験が無ければ単純な画像 GUI が作成できる程度までは習得してほしい．

Tkinter の書法

ラベルとボタンを表示し，ボタンクリックでプログラムが終了するだけのプログラムを Tcl/Tk と Python とで比較する．まず背景色を赤色にする Frame をつけ，その輪郭を 10 ポイントと指定する．その中にプログラム終了ボタンを配置する．また，終了ボタンをクリックすると端末に

図 2 ボタン
10 ポイントの赤枠の中に終了ボタンを表示している．ボタンクリックで端末に終了と表示し，プログラムは終了する．

終了と表示してプログラムが終了する．単にボタンだけなら，ここまで冗長にする必要はないが，クラス設計に Frame コンテナーは重要な役割りを担うため，あえて，このような例を選んだ．仕上りの GUI を図 2 に示し，プログラムを以下に示す．

Tcl/Tk プログラム：

```
#!/usr/bin/env wish
frame .f -bg "red" -bd 10
button .b -text "終了" -command goout
pack .b -in .f
pack .f -fill both -expand 1
proc goout {} {
  puts "終了"
  exit
}
```

以下は同じ内容を Python で記述したプログラムである．Tcl/Tk をできるだけ忠実に python に変更したから，ウィジェット名はグローバルのままメインプログラムに残る．

Python Tkinter プログラム：

```
#!/usr/bin/env python
# -*- coding: euc-jp -*-
import Tkinter as tk
def goout():
  print u'終了'
  exit()
root=tk.Tk()
f=tk.Frame(root,bg='red',bd=10)
b=tk.Button(f, text=u'終了',
  command=goout)
b.pack()
f.pack(fill='both',expand=1)
root.mainloop()
```

となる．

プログラム解説：

最初の 2 行は従来通りで，Windows であれば
-*- coding: cp932 -*-
とする．次の import 文は Tkinter のモジュールの読み込み命令であり，接頭語 tk を付けて，ラベル名の衝突を防ぐ．root は基本ウィジェットで Tcl/Tk の「.」に相当するものである．Tcl/Tk でボタン作成は button .b としているが，Python では b=tk.Button(root, ...) と表示する．root は親ウィジェットの名前で，どこに表示するかを指定している．Tcl/Tk でのオプションは-パラメータ 値となっているが，Python では「=」等号が使用できる．また，テキストの u はユニコードを指定するものであって，Windows でも必要である．pack 命令はウィジェットのメソッドとして登録されている．ボタンクリック処理関数 goout() はその呼び出しより前に書かねばならない．Tcl/Tk ではプログラム終了時に自動的にイベントループが作動するが，Python では明示的に mainloop メソッドを駆動しなければならない．Windows でこの行が無ければ，プログラムは瞬時に終了して GUI 画像は表示さない．

クラスで GUI を作成する．

クラスの概念で上記のプログラムを書き直すと，メインプログラムでのウィジェット変数はたった一つになる．そのプログラムを以下に示す．ボタン処理も継承できるようにファイル名を B.py とクラス名と同じにした．

File name: B.py

```
#!/usr/bin/env python
# -*- coding: euc-jp -*-
import Tkinter as tk
class B(tk.Frame):
  # クラス初期設定
  def __init__(self, master=None):
    tk.Frame.__init__(self, master)
  # クラスウィジェット作成
    f=tk.Frame(master,bg='red',bd=10)
    f.pack()
    b=tk.Button(f,text=u'終了',
      command=self.goout)
    b.pack()
  def goout(self):
    print u'終了'
    exit()
# メインプログラム
if __name__ == '__main__':
  g=B()
  g.mainloop()
```

確かにメインプログラムから見えるウィジェット変数は g 一つである．Frame の f，Button の b，さらに，ボタン処理関数 goout も class の中の局所変数になっている．

プログラム解説

初めての GIU クラスの説明だから，詳細に述べる．import 文まで含めて始めの 3 行は前例と同じである．次に class 定義を行う．クラス B は tk.Frame を利用するから括弧内に記述する．tk.Frame は tk.Button を包括するからこれ以上書く必要はない．クラスの初期設定は__init__関数に記述する．その中身は，まずウィジェットコンテナーである tk.Frame の初期設定，つまり，tk.Frame.__init__と Frame 初期設定を行う．続いて，tk.Frame を設定するが，親ウィジェットに master を選ぶ．親ウィジェットの選択に master，self，なし（デフォルト）の 3 種がプログラムを通過するが，必ず，master を選ぶ．この意味は None であって，root と同じ意味である．

さて，GUI 本体であるが，前述のプログラムと殆んど変わらない．ただ，command=self.goout のみが変更されている．self が接頭語になっている変数はクラス内グローバル変数だと思ってよい．この場合は goout メソッドを示している．goout

関数は前述のプログラムと同じだが，class 内に閉じ込められている．f，b のウィジェット変数は，外部から引用される場合には self を付けるが，局所変数にとどめるだけならその必要はない．

クラス文の書き方をまとめると，始めに Frame を初期設定する．次いで，GUI 本体ウィジェットと配置を記述する．必要ならメソッド関数を記述する．これだけである．

メインプログラムをみると，g=B() となって，インスタンス g を作成している．g はクラス呼び出しにより，self の部分が b に名前が変更される．GUI の表示はクラス初期設定の pack でなされるから，mainloop を走らせて待機状態にすればよい．プログラム終了はボタンクリックで関数 goout が動作する．

プログラムの実行順は始めに import 文を読み，次いで class 文を発見し，サブルーチンアドレスとして記憶する．その後，if 文でメインプログラムを知る．そこで，g=B() を見つけ，クラス B の処理を実行する．クラス内では__init__関数の処理を行う．つまり Frame の初期設定，Frame の表示，Button の Frame への張り付けを行い，Button command の登録をして，class B の処理を終えて，メインプログラムに戻る．メインプログラムではクラス B のインスタンスとしてそのポインターを g に記録する．そしてクラス B の既存メソッドの一つである mainloop を回してイベントの発生を待つ．

以上が B.py の実行順であるが，プログラムの途中に print 文で，メッセージを表示する命令を挿入すると，プログラムの流れがよくわかる．繰り返しになるが，class 文はコンパイルされて実行命令がメモリー配置される宣言文ではなく，実行されるサブルーチンであることに注意をしてもらいたい．

B.py の継承

ボタンクリックで，GUI 本体はそのままにして，メソッドだけを変更したり，新たなメソッドを追加したい場合がよくある．そこで B.py では「終了」とだけ端末に表示したが,「extend 終了」とメソッドの多態性を利用する継承プログラムを作成する．

File name: BE.py

```
#!/usr/bin/env python
# -*- coding: euc-jp -*-
from B import *
class BE(B):
  # メソッド再設定
  def goout(self):
    print u'extend 終了'
    exit()
# メインプログラム
if  __name__ == '__main__':
  g=BE()
  g.mainloop()
```

プログラム解説

import 文は継承元の拡張子「.py」を除いたファイル名を書く．Tkinter は継承元に記述されているため，不要．新しいクラス名 BE を明記し，括弧のなかに継承するクラス名 B を記述する．これで B の全てが継承される．

メソッドは新しく変更するから，それを示すために，同じメソッド関数名 goout を書き直す．本例は，簡単に「extend 終了」として，継承元と異なる表示にしただけである．

メインプログラムでは新たなクラス名を引用する．ただ，これだけで以前のプログラムを変更して，新たなプログラムに仕上るから，継承とは便利なツールである．

GUI 継承の落し穴

前に示した B.py の GUI を継承して，新たなクラスを作成してみよう．単純に赤色のフレー

ムの回りに緑のフレームを作成するだけである．簡単に以下のプログラムが考えられる．ファイル名を BE2.py として，

File name: BE2.py

```
#!/usr/bin/env python
# -*- coding: euc-jp -*-
from B import *
class BE(B):
  # クラス初期設定
  def __init__(self, master=None):
    B.__init__(self, master)
    f=tk.Frame(master,bg='green',
      bd=10)
    f.pack()
    l=B(f)
    l.pack()
  # メソッド初期設定
  def goout(self):
    print u'extend 終了'
    exit()
# メインプログラム
if  __name__ == '__main__':
  g=BE()
  g.mainloop()
```

仕上りを図 2a に示す．何と，GUI が 2 個立てに出現する．しかも，上の図は継承元の B であって，終了ボタンをクリックすると「extend 終了」と多態性が働いたメソッドになり，下の緑の枠で囲まれた新しい GUI は元の「終了」とだけ返る．

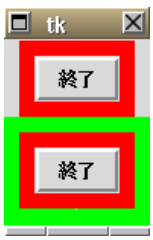

図 2a ボタンの継承
10 ポイントの赤枠の周りに 10 ポイントの緑の枠を表示する継承プログラム BE2.py の結果．本文参照のこと．

よく考えると先ほどの継承プログラム BE.py は class 文を記述しただけであり，GUI は古いまま，新しいメソッドに置き換わった．しかし BE2.py は，まず B.__init__でその継承作業をして，次に緑のフレームを作成し，その中に継承元の GUI をはめこんだ．しかも，それは古いメソッドポインターも変更されることなく同時に引用されたと考えられる．

ここから得られる教訓は，メソッドの継承はよいが，安易には GUI の継承はできないということである．

観光バス乗客窓の開閉コントローラクラスの効化的な使用例

例えば観光バスには乗客用に左右それぞれ 6 個のパワーウインドウが装備されているとする．これらの窓の開閉を運転席からのコントローラで開閉する場合，制御パネルは左右にそれぞれ 6 個のボタンを備えることになる（図 3 参照）．

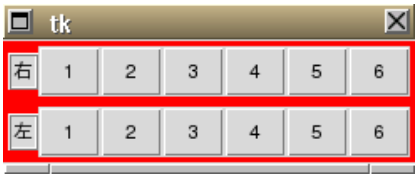

図 3 バス パワーウインドウコントローラー
ラベル名が入力できる Frame クラスを作成し，上下 2 段構成で表示している．ボタンをクリックすると，左右の何番目か端末に表示される．

これを従来通りのプログラムを行うとラベルを含めて 14 個のウィジェット変数が必要になる．しかし，クラスで設計すると 30 行程度でプログラムが仕上り，外部から見えるウィジェット変数は一つもない．プログラムを以下に示す．

File name: Bus6.py

```
#!/usr/bin/env python
# -*- coding: euc-jp -*-
import Tkinter as tk
# クラス作成
class Frame(tk.Frame):
```

```
  # 定型的クラス設定ルーチンを記述
  # 引数があれば self. 変数を作成
  def __init__(self, var,
    master=None):
    self.var=var
    tk.Frame.__init__(self, master)
  # クラス画面作成
    f=tk.Frame(master,bd=3,bg='red')
    f.pack()
    # Label
    l = tk.Label(f, text=self.var,
      relief=tk.RIDGE, bd=2)
    l.pack(side="left")
    for i in range(1,7):
      b=tk.Button(f, text=i,
        command=self.make_it(
          self.var, i))
      b.pack(side="left")
  # ボタンメソッド
  def make_it(self,side, i):
     def cmds():
       print u'%s %s 閉じる' % (
         side,i)
     return cmds

if __name__ == '__main__':
  fr=Frame(u"右")
  fl=Frame(u"左")
  fl.mainloop()
```

クラス設計では，新たに左右の区別を示すパラメータを受け入れる必要がある.. ボタンの割り込み番号は必要だが，ボタンウィジェット名は引用されることはないから不用である．仕上り図は図 3 である．

ボタンクリックを実行すると，

```
>>> 右 1 閉じる
左 2 閉じる
右 3 閉じる
左 4 閉じる
```

と端末表示される．

端末表示の代わりに本物の制御装置を作動させるインターフェースを作れば実用になる．

プログラム解説：

始めの 3 行は定型どおり．

class Frame の継承は tk.Frame だけでよく，tk.Label や tk.Button は自動的に組み込まれる．__init__関数の引数 var は左右の区別を行うラベルを受け入れる．メソッド関数で引用されるから self.var としてクラス内グローバル変数を作成し，コピーする．続いて Frame の初期設定を行う．親ウィジェットの None 指定は root 指定と同じ効果である．__init__関数からここまでの一連の書式は定型的な書式であるから，いつでも使えるようにメモしておくとよい．特に引数を伴ったインスタンス設計での self の使い方に注意してほしい．

ラベル名を l として引数 self.var を text に指定し，pack している．pack の親元は必ず f とする．Frame コンテナーを作らずにデフォルトで直接ウィジェットを張り付ける例が多数紹介されているが，複雑なウィジェット配置になると思いもしない所に pack されることもあるから，必ずしも行儀がよい方法とはいえない．明示的に Frame を作成し，そこにウィジェットを張り付けることが大切である．ラベル text オプションは self.var として，クラス内グローバル変数を引用している．

ボタンは 6 個必要であるから，range を利用し，for loop で作成する．ボタン名は全て b であるが，重複しても今後外部から引用されることがないから問題は生じない．ボタン命令 make_it は 6 個作られる．

次に，ボタンクリックで端末に番号を表示させるボタンメソッドを記述している．実は，このインスタンス作成時にプログラムが実行される．つまり，ボタン命令 make_it も実行されて端末にボタンラベル名が表示されてしまう．その誤動作を回避するために，make_it 関数の中で更に，cmd 関数を作成し，make_it 関数の return 命令

で cmd 関数を実行させた．引数 side と i は局所変数のスコープの性質を利用して上位の関数データを引用している．本来のボタン command オプションは実行関数名だけを入力しなければならない（「()」をつけてはならない）のに，引数を利用ために作ったトリックである．

Frame を新しく特化した Frame に変えたのだからもう従来の Frame の概念はない．また，l や b などのウィジェット名もメインプログラムから見えることはない．

Bus6.py の継承

練習のために Bus6.py のメソッドの継承プログラム Bus6E.py を作成する．例によって，「extend」を追加出力させる．

File name:　Bus6E.py

```
#!/usr/bin/env python
# -*- coding: euc-jp -*-
from Bus6 import *
# クラス作成
class FFrame(Frame):
  # ボタンメソッド
  def make_it(self,side, i):
     def cmds():
       print u'%s %s extend 閉じる' \
         % (side,i)
     return cmds
if __name__ == '__main__':
  fr=FFrame(u"右")
  fl=FFrame(u"左")
  fl.mainloop()
```

プログラム解説

ほとんど解説の必要がない．新しく出た項目は，print 文の行末の\は次行への継続符である．

クラス設計の雛型

GUI デザインでフォントの字体，大きさ，前景やや背景の色を変更すると見栄えも変わる．その基本命令は config と cget であり，単純なラベルウィジェットでこれらのメソッドプログラムを紹介する．対話形式で駆動すれば容易に変更できる．調整できればオプションとして組み入れればよい．雛型は Frame をもちいて bd=5，bg='green' で枠を明確に表示した．メインプログラムではクラスを 2 回引用したために外枠に段差がついた，そのことからも，独立してインスタンスが作成されていることがわかる．仕上りを図 4 に示す．

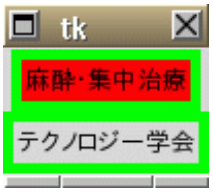

図 4 単純なラベルウィジェットクラス
フレームの中にラベルをパック配置している．上段のラベルは config メソッドで背景を赤色に変換している．

File name:　Prototype.py

```
#!/usr/bin/env python
# -*- coding: euc-jp -*-
import Tkinter as tk
class Frame(tk.Frame):
  # クラス初期設定
  # 引数があれば self. 変数を作る．
  def __init__(self, var,
    master=None):
    self.var=var
    tk.Frame.__init__(self, master)
  # クラスウィジェット作成
    f=tk.Frame(master,bd=5,
      bg='green')
    f.pack()
    self.l = tk.Label(f,
      text=self.var)
    self.l.pack(side="top")
  # 基本メソッド
  def cget(self,data):
    return self.l.cget(data)
  def config(self,pos,data):
    self.l[pos]=data
# メインプログラム
if __name__ == '__main__':
  l1 = Frame(u'麻酔・集中治療')
  l2 = Frame(u'テクノロジー学会')
```

```
l1.config('bg','red')
print l1.cget('text')
l2.mainloop()
```

プログラム解説：

__init__関数でラベル表示される引数をグローバル変数に記憶させる．フレームウィジェットを作成する．

明示的にフレーム f を作成し，属性変更を行っている．ラベルウィジェットを作成し，フレームにパックさせている．特定ののラベルインスタンスを認識させるために self を付けている．

メソッドの cget も同様に，どのインスタンスが必要であるかを指定するために self を付ける．config の書式は「bg='red'」の書式ではbg がわからないとエラーが発生するためにオプション名も文字列に変更して，上記の記述法を行う．

メインプログラムでは l1 と l2 のインスタンスを作成しパックしている．クラス名が Frame であるが，もはやフレーム機能はない．ラベルが中に入っているフレームである．config メソッドで l1 インスタンスの背景を赤色に変更し，また l1 の text 文を cget で端末に出力している．

この雛型は冗長だという意見があるかもしれないが，フレームや init の省略はいつでもできるから，まずはこの雛型をコピーしてから目的とする GUI に少しづつ変更していくことを強く進める．

テキストウィジェット

テキストウィジェットは行，列，フォント設定など多くのオプションが指定でき最も複雑なウィジェットの一つである．さらに，立て，横のスライダーを必要とする．従って，どのような class を作成するのが良いかを考えよう．まずは，クラスを使わずに直接表示する方法である．

フレームを f，テキスト画面を t，X 軸スクロールバーを sx，Y 軸スクロールバーを sy にしてフレーム f に張り付けている．フレームは 3 ピクセルで赤色で明示的に示した．長い一本のメインプログラムになる．プログラムを以下に示し，GUI の仕上りを図 5 に示す．

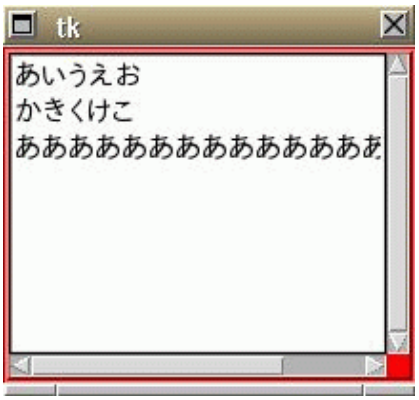

図 5 スクロールバー付きテキストウィンドウ
フレームの中にグリッド配置でテキスト，x 軸スクロール y 軸スクロールを設定している．

1) クラスを使わないスクロールドテキスト

スクロールバーのプログラムは Tcl/Tk の例題を参考にして Python に変換するのがよい．

```
#!/usr/bin/env python
# -*- coding: euc-jp -*-
import Tkinter as tk
root=tk.Tk()
f=tk.Frame(root, bg="red",
  relief="groove",borderwidth=3)
# テキスト画面作成
t=tk.Text(f,width=24, height=8,
  wrap=tk.NONE, setgrid=tk.TRUE,
  font=('FixedSys', 14))
t.grid(column=0, row=0, sticky='nsew')
# X 軸 スクロールバー作成
sx=tk.Scrollbar(f,orient='horizontal',
   command=t.xview, width=10)
t["xscrollcommand"]=sx.set
sx.grid(column=0, row=1, sticky='ew')
# Y 軸 スクロールバー作成
```

```
sy=tk.Scrollbar(f,orient='vertical',
  command=t.yview, width=10)
t["yscrollcommand"]=sy.set
sy.grid(column=1, row=0, sticky='ns')
f.pack(fill=tk.BOTH,expand=1)
# 画面を拡大するとテキストだけが拡大する
f.grid_columnconfigure(0,weight=1)
f.grid_rowconfigure(0,weight=1)
f.mainloop()
```

プログラム解説：

Frame の中に grid 配置でテキスト，x 軸スクロール y 軸スクロールを設定している．grid 命令は pack 命令と同じく，ウィジェットメソッドに組み入れられている．スクロールバーの設定と接続法は考えても思い付かないから例を真似するとよい．テキスト名，スクロール名がグローバルであるために，ウィジェット名の衝突に注意しなくてはならない．

上記だけのプログラムだと，テキスト画面の書き込みはできるが，プログラム終了と同時にデータは失われる．実際は，新規，読み取り，書き込みなどのメニューバーを作成してテキストデータとリンクするのであるが，その基本命令であるテキストデータの書き込み，読み取り命令を以下に示す．対話形式で Python を走らせて，各自試みてもらいたい．テキストウィジェットの命令は大変多く，これ以外の命令は割愛する．

```
書き込み：
t.insert(tk.END,'abc')
改行書き込み：
t.insert(tk.END,'\n')
日本語書き込み：
t.insert(1.0,u' 日本語 abc\n')
読み取り：
print t.get(1.0,tk.END)
行は 1 行目から始まり，
列は 0 列から始まる．
削除：
t.delete(2.0,2.2)
2 行目 0 列から 2 文字を削除
```

2) クラスを用いたスクロールドテキスト

スクロールドテキストは結構ややこしいプログラムになるから class でまとめる効果は大きい．クラスを使わない方法のプログラムを，前述のクラス雛型に準じて，また，テキスト，水平スクロール，垂直スクロールバーのウィジェット名はそのままにして，変更を加えた．

本質的な話しではないが，プログラム作成にあたってネスティングがひとつでも増えると，コーディングを失敗する確率が高くなる．筆者のコーディング方法は，まず，雛型をコピーし，def init 以下，すなわち GUI 本体を消す．次いで，クラスを使わない方法で作成したプログラムをコピーし，インデントの調整を行う．root で都合の悪いところを修正し，クラス内局所変数で画像が表示されるところまで努力する．それが済めば，何がクラス内グローバル変数かをよく考えて，変数に self を付ける．この例ではテキスト変数 t 以外に必要な変数はない．メソッドのコーディングは正しく GUI が表示されてからの話しである．

File name: TS.py

```
#!/usr/bin/env python
# -*- coding: euc-jp -*-
import Tkinter as tk
class TS(tk.Frame):
  # クラス初期設定
  def __init__(self, master=None):
    tk.Frame.__init__(self, master)
  # クラスウィジェット作成
    f=tk.Frame(master, bg="red",
      relief="groove",borderwidth=3)
    # Text 作成
    self.t=tk.Text(f,width=24,
      height=8, wrap=tk.NONE,
      setgrid=tk.TRUE,
      font=('FixedSys', 14))
    self.t.grid(column=0, row=0,
      sticky='nsew')
    # X 軸 スクロールバー作成
```

```
    sx=tk.Scrollbar(f,
       orient='horizontal',
       command=self.t.xview,
       width=10)
    self.t["xscrollcommand"]=sx.set
    sx.grid(column=0, row=1,
      sticky='ew')
    # Y 軸 スクロールバー作成
    sy=tk.Scrollbar(f,
      orient='vertical',
      command=self.t.yview,
      width=10)
    self.t["yscrollcommand"]=sy.set
    sy.grid(column=1, row=0,
      sticky='ns')
    f.grid_columnconfigure(0,
      weight=1)
    f.grid_rowconfigure(0,
      weight=1)
    f.pack(fill=tk.BOTH,expand=1)
# メインプログラム
if __name__ == '__main__':
  t=TS()
  t.t.insert('1.0',u'あいうえお\n')
  print t.t.cget('bg')
  t.t['bg']="green"
  t.t.delete('1.0','1.1')
  t.mainloop()
```

プログラム解説：

grid によるウィジェット配置はクラスでも可能である．クラス名を TS として tk.Frame を継承するだけで tk.Text や tk.Scrollbar が取り込まれる．テキストウィジェットのみが他の関数で引用されるから self を付ける必要があるが，スクロールバーに関しては引用されないから不要である．

メインプログラムでは cget，config の代わりの辞書設定，insert，delete の例を示した．テキストウィジェットのみが self で公開されているから．呼び出しは t.t. と t を重ねればよい．

スクロールウィジェットがクラス化されたから，これでいくつでもテキスト画面を作成することができる．例に示したテキスト画面では立て 8 行，横 24 文字の小さな画面であるが，コーナーをドラグすると，いくらでも広がるようにプログラミングしている．

フォント設定は各システムで異なり，難しい問題であるが，例に示されている「FixedSys」は Windows も含めて True Type で使用できるようである．サイズは 14 としているが，任意のサイズが指定できる．他に「Helvetica，Times」などを指定すると日本語では明朝体表示になる．さらに，

```
from tkFont import *
```

と tkFont をプログラムに導入すると，

```
font1=Font(family="Helvetica",
weight=NORMAL, size=18)
t.configure(font=font1)
```

という使い方ができ，

```
font1.configure(size=24)
font1.configure(weight=BOLD)
```

と記述できる．

スクロールドテキストの継承

前述の tkFont を import し，フォントを変更する継承プログラム TSE.py を作成する．

File name: TSE.py

```
#!/usr/bin/env python
# -*- coding: euc-jp -*-
from TS import *
from tkFont import *
class TSE(TS):
  # クラス初期設定
  def __init__(self, master=None):
    TS.__init__(self, master)
    font1=Font(family="Helvetica",
      weight=NORMAL, size=8)
    self.t.configure(font=font1)
# メインプログラム
if __name__ == '__main__':
  t=TSE()
  t.t.insert('1.0',
```

```
    u' テクノロジー学会\n')
t.mainloop()
```

プログラム解説

tkFont はクラス属性でないのか上記の方法以外に import できない．つまり，名前空間はないようである．TS の継承クラス名を TSE にした．クラス内では self.t と指定すると configure 関数が通過する．

このプログラムの流れは、メインプログラムで TSE クラスを駆動する．TSE の初期設定で継承元の TS を初期設定する．そこでテキスト画面が表れ，次いで，font1 設定，self.t.configure を実行し，メインプログラムに返る．最後に，メインプログラムの t.insert を実行し，mainloop に入る．

以前 GUI 本体の継承は無理と述べたが，今回の TSE は継承が成功している．その理由は GUI 本体そのものの変更ではなく，単に GUI 属性の追加，変更であるためである．つまり，pack 命令の必要な変更はダメで，メソッドを含むそれ以外の GUI 属性の変更は継承すると考えればよい．

次項ではテキストウィジェットのデータを読み書きできるプルダウンメニューを述べる．

プルダウンメニュー

プルダウンメニューは種々のオプションがあり，相当複雑なプログラムになる．またどの程度までクラス内に閉じ込めて汎用性を保つかという問題もある．とりあえず従来通りの方法を以下に示す (fm.py)．

1) クラスを使わないプルダウンメニュー

メニュー命令はいろいろなボタン操作で種々のコマンドを呼び出し，また継承を利用することが多い．

プログラム解説

プルダウンメニューは Menubutton が分かりやすいので，それを利用する．上記の例はフレー

File name: fm.py

```
#!/usr/bin/env python
# -*- coding: euc-jp -*-
import Tkinter as tk
#ボタンクリック処理
def callback():
  print 'exit'
  exit()
# メインプログラム
root=tk.Tk()
f=tk.Frame(root,bg='red',bd=3)
f.pack(anchor='nw')
file=tk.Menubutton(f, text='File',
  relief=tk.RAISED)
file.pack()
file.m =tk.Menu(file, tearoff=0 )
file['menu']=file.m
file.m.add_separator()
file.m.add_command(label='Quit',
  command=callback)
f.mainloop()
```

ムに「File」と表示されたプルダウンメニューを作成し，クリックするとセパレータと「Qiit」と表示するボタンが表れる.「Quit」をクリックすると callback 関数が呼び出され，端末に「exit」と記述してプログラムが終了する．

プルダウンメニューのメインが file ボタンで，各メニューが file.m になり，それに add_命令を加えていると読めば意味がわかる．Tcl/Tk では Menubutton で-menu file.m が認められるが，Tkinter では file.m が存在しないとエラーが発生する．そのためにそのパラメータ無しで file を作成し，その後，file["menu"]=file.m と記述する．

メニュー全体が Frame コンテナーに乗っていることの確認に，bg='red',bd=3 で見えるようにしたが，bd=0 にすれば簡単に消せる．

2) クラスを用いたプルダウンメニュー

前述のプログラムはTcl/Tkプログラムに準じて作成したので，長い一連のプログラムになり，見通しが悪い．クラスで記述すると，内容はより複雑になるが，メインプログラムからはクラス内はブラックボックスであるため，見通しは大変よくなる．また，callback関数の多態性も利用できる．FMはクラス名であるために，前述のfm.pyをFM.pyにプログラム名を変更してクラスで書き直す．

File name: FM.py

```
#!/usr/bin/env python
# -*- coding: euc-jp -*-
import Tkinter as tk
class FM(tk.Frame):
  # クラス初期設定
  # 引数があればself.変数を作る.
  def __init__(self, master=None):
    tk.Frame.__init__(self, master)
  # クラスウィジェット作成
    f=tk.Frame(master,bg='red',bd=3)
    f.pack(anchor='nw')
    file=tk.Menubutton(f,
      text='File', relief=tk.RAISED)
    file.pack()
    file.m =tk.Menu(file, tearoff=0)
    file['menu']=file.m
    file.m.add_separator()
    file.m.add_command(label='Quit',
      command=self.callback)
# メソッド
  def callback(self):
    print 'exit'
    exit()
# メインプログラム
if __name__ == '__main__':
  f=FM()
  master=None
  g=tk.Button(master,text=u'終了',
    command=exit)
  g.pack(side='left')
  f.mainloop()
```

プログラム解説：

雛型に準じて，先の例を書いただけである．ただし，tk.Frameの親ウィジェットをrootでなくmasterに変更している．それ以降の各ウィジェットにselfはない．考えれば，同じクラスを数多く作る必要がないためである．しかし，ボタンメソッドは継承に必要であるから，selfを付けている．なお，クラス名のFMはFile Menuのイニシャルを選んだ．

3) 実用になるプルダウンメニュー

ファイル選択ダイアログにtkFileDialogがあり，その動作を対話モードで調べると．

```
>>> import tkFileDialog as D
>>> dir(D)
['Dialog', 'Directory', 'Open',
'SaveAs', '_Dialog', 'askdirectory',
'askopenfile', 'askopenfilename',
'askopenfilenames', 'askopenfiles',
'asksaveasfile', 'asksaveasfilename']
```

などがメソッドとして登録されている．そこでaskopenfilename と検討をつけて，

```
>>> f=D.askopenfilename
>>> f=D.askopenfilename()
>>> print f
/home/tanaka/a1.py
```

と思う通りに選択したファイル名がfに記録された．この結果を考慮して同じファイル名でプルダウンメニューを作成する．

File name: FM.py

```
#!/usr/bin/env python
# -*- coding: euc-jp -*-
import Tkinter as tk
import tkFileDialog as D
# グローバル変数
fname='noname.txt'
# ファイルメニュー
class FM(tk.Frame):
  # クラス初期設定
  # 引数があればself.変数を作る.
  def __init__(self, master=None):
    tk.Frame.__init__(self, master)
  # クラスウィジェット作成
    self.f=tk.Frame(master,bg='red',
      bd=3)
```

```
    self.f.pack(anchor='nw')
    file=tk.Menubutton(self.f,
      text='File', relief=tk.RAISED)
    file.pack(side='left')
    file.m =tk.Menu(file, tearoff=0 )
    file['menu']=file.m
    file.m.add_command(label='New',
      command=self.new_f)
    file.m.add_command(label='Open',
      command=self.open_f)
    file.m.add_command(label='Save',
      command=self.save_f)
    file.m.add_command(
      label='Save As',
      command=self.saveas_f)
    file.m.add_separator()
    file.m.add_command(label='Quit',
      command=self.exit_f)
# メソッド
  def new_f(self):
    fname='noname.txt'
    print 'FM',fname
  def open_f(self):
    global fname
    fname = D.askopenfilename(
      event=None)
    print 'FM',fname
  def save_f(self):
    print 'FM',fname
  def saveas_f(self):
    fname = D.asksaveasfilename(
      event=None)
    print 'FM',fname
  def exit_f(self):
    print 'exit_f'
    exit()
# メインプログラム
if __name__ == '__main__':
  f=FM()
  f.mainloop()
```

プログラム解説：

メニューフレームは将来の機能追加のために公開にした．また，左から右へとボタンが並ぶように pack した．ファイルメニューは New，Open，Save，Save As，Quit の順に作成している．メニュー選択をクリックすると外部関数に作業が移動する方式にしている．オープンするファイル名はメインプログラムで fname とし，新規 (New) は'noname.txt' と初期設定する．このプログラムはプロトタイプなので，単純に端末にファイル名を表示するだけであるが，実際はテキストウィジェットの画面をクリアーする作業が続く．保存 (Save や Save As) ならテキストウィジェットのデータを書き込む作業になる．当然終了なら，その前に注意喚起ダイアログを引き出す必要があろう．次項では，前述のスクロールドテキストとメニュー画面を含めてエディタープログラムを作成するが，その前に FM.py のメソッドの継承を行う FME.py を作成して，継承の仕方を明らかにする．

FM.py の継承プログラム FME.py

メニューの基本構成は祖のままに，メニュー選択による callback 関数だけを変更してプログラムの柔軟性を上げることはオブジェクト指向の最も得意とするところである．FM.py をスーパクラスとして Edit メニューの追加と，File メニューのメソッドだけを変更する FMJ.py を作成する．単純な例題であるから，単にメニュークリックで端末出力の行頭に「extend 」と打ち出すだけのプログラムである．

File name: FME.py

```
#!/usr/bin/env python
# -*- coding: euc-jp -*-
from FM import *
# ファイルメニュー 継承
class FME(FM):
# メソッド
  def new_f(self):
    fname='noname.txt'
    print 'extend FM',fname
  def open_f(self):
    global fname
    fname = D.askopenfilename(
      event=None)
    print 'extend FM',fname
  def save_f(self):
```

```
    print 'extend FM',fname
  def saveas_f(self):
    fname = D.asksaveasfilename(
      event=None)
    print 'extend FM',fname
  def exit_f(self):
    print 'extend exit_f'
    exit()
# メインプログラム
if __name__ == '__main__':
  f=FME()
  f.mainloop()
```

プログラム解説

継承元のプログラムを from 命令で import する．File メニューのメソッド関数は全て書き直おすだけである．多態性を利用して確認に必要な「extend」だけを追加した．メインプログラムのクラス名は FME に書き直し，後はそのままである．

テキストエディター作成

Tkinter のテキストウィジェットはそれだけで十分な編集機能が備えている．従って，テキストウィジェットのデータの入出力をメニューボタンで操作できればエディターとしての基本機能は備わることになる．そこで，これまで作成した TS クラス，FM クラスを継承するエディター Ed.py を製作する．

作業としては，二つのリソースを継承し，一つの GUI を作成すること，FM クラスのメソッドを変更し，テキストウィジェットとのデータの入出力が対応できるメソッドを製作することになるが，よく考えると FM.py の継承版 FME.py のコピーファイルを Ed.py に名前を変えて，その class 名を Ed に変更するだけでよい．終了メッセージダイアログを import しておく．また，新しい GUI 構築は継承できないので，メインプログラムで TS を呼び出す必要がある．仕上り GUI を図 6 に示す．

まず，骨格となる基本プログラムを示し，次にデータ授受のためのメソッド部分については，順次明らかにしていく．

図 6 簡易エディター Ed.py
メニュー FM，スクロールドテキスト TS を継承した．

File name: Ed.py

```
#!/usr/bin/env python
# -*- coding: euc-jp -*-
from FM import *
from TS import *
import tkMessageBox as M
# エディター
class Ed(FM):
# ファイルメニュー 継承
# メソッド
  def new_f(self):
    fname='noname.txt'
    print 'extend FM',fname
  def open_f(self):
    global fname
    fname = D.askopenfilename(
      event=None)
    print 'extend FM',fname
  def save_f(self):
    print 'extend FM',fname
  def saveas_f(self):
    fname = D.asksaveasfilename(
      event=None)
    print 'extend FM',fname
  def exit_f(self):
    print 'extend exit_f'
    exit()
```

```
# メインプログラム
if __name__ == '__main__':
  master=None
  f=Ed()
  t=TS()
  t.t.insert('1.0',u' あいうえお\n')
  f.mainloop()
```

画面表示して，画面の各コーナーをドラグしてもテキスト画面だけが拡張してメニューは左上端に固定されたままであることを確認する．

それではメニューメソッドの継承プログラムを class Ed 初期設定の下に継ぎ足すことにする．最初は new_f 関数である．

new_f 関数は，テキスト内の内容を全て消し，記録ファイル名を「nonmae.txt」にする．プログラムは

```
  def new_f(self):
    fname='noname.txt'
    t.t.delete(1.0,'end')
```

となる．Ed.py を実行させ，適当に画面に文字を入力し，New を指定して，画面の「あいうえお」が消えれば成功である．

open_f 関数は既存のファイルを読み込むモードであるから，読み取りダイアログを開け，指定ファイルを読み取り，テキスト画面に書き込む．既存のデータの後に書き込むことにした．最初からファイルだけを書き込む場合は一旦，New を実行させ，続いて Open を実行する．

```
  def open_f(self):
    global fname
    fname = D.askopenfilename(
      event=None)
    fn=open(fname,"r")
    t.t.delete(1.0,tk.END)
    for line in fn.readlines():
      t.t.insert(tk.END,line[:-1]. \
        decode('euc_jp')+"\n")
    fn.close()
```

となる．

save_f 関数は，テキストデータ全てを fname で指定したファイルに書き込む操作である．

```
  def save_f(self):
    data=t.t.get(1.0,tk.END-1)
    data1=data.encode('euc_jp')
    fn=open(fname,"w")
    fn.write(data1)
    fn.close
```

となる．

saveas_f 関数は，始めに書き込みファイルダイアログを開き，ファイル名を指定し，そこに書き込む操作である．

```
  def saveas_f(self):
    global fname
    fname = D.asksaveasfilename(
      event=None)
    data=t.t.get(1.0,tk.END)
    data1=data.encode('euc_jp')
    fn=open(fname,"w")
    fn.write(data1)
    fn.close
```

となる．

exit_f 関数はエディターを終了すればよいが，注意喚起をした方が親切である．ダイアログの呼び出しは，tkMessageBox モジュールが簡単である．python 対話モードで調べると，

```
>>> import tkMessageBox as M
>>> dir(M)
['askokcancel', 'askquestion',
'askretrycancel', 'askyesno',
'showerror', 'showinfo',
'showwarning']
>>>
```

となっている．askokcancel を選択すると，'OK' なら True，'Cancel' なら False が返る．これで exit_f 関数が出来る．

```
  def exit_f(self):
  # メッセージボックス
    ans=M.askokcancel(u'メッセージ',
      u'終わりますよ．')
    if ans==True:
      exit()
    else:
      return
```

となる．

以上で簡易エディターEd.pyのプログラムは完成であるが，Windowsのためのencode，decodeオプションは'shift_jis'である．カット/ペーストを行うselection機能はLinuxではマウス位置が大変クリティカルで使いにくく，更に他のディスプレー画面の日本語文字が通過しないなどの欠点がある．しかし，同じテキスト画面内であれば，Ctrl+C，Ctrl+X，Ctrl+Vキーで快適に操作できる．デフォルト画面を小さくした理由はいくつもの簡易エディターをディスプレーに表示させるためである．表示画面の邪魔にならないことが重要であろう．また，画面のコーナーをドラグすれば，いくらでも大きく拡張できる．

世の中にviを始めとして立派なエディターがあるのに，なぜこのような簡易エディターを作成したのかという質問に対して，汎用エディターでは出来ない機能，例えば，2行にまたがる検索/置換，上付き，下付き文字の検索/置換などに役立てることができる．これらの作業はTexソースの作成時に必要で，私的な編集メニューで利用している．また，Pythonの継承能力を確かめる意味もある．分散プログラミングと継承の概念があればこそ，この程度の行数で実用可能なエディターが作成できることを理解してもらいたい．

画像ウィジェット

今日では立派な画像ソフトが数多く公開されているが，出来上がった画像ファイル（jpg，png，gif など）からその X-Y 座標を数値として読み取るソフトは筆者は知らない．例えば心電図のグラフを数値として読み取ることができれば，心電図解析のための強力なツールになる．画像処理とはそのような可能性を持つプログラムである．

Tcl/Tk の拡張版である Tkinter では bitmap，gif ファイルしか対応できていない．しかし，ImageTk を import することにより，jpg，png 画像をキャンバスに表示することができる．その基本となるプログラムと結果を以下に示す．

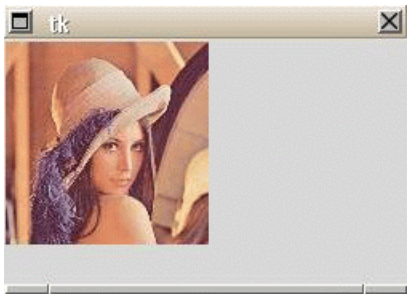

図 7 キャンバス画像基本表示
ImageTk.Photoimage で jpg 画像をキャンバスに表示している．

```
#! /usr/bin/env python
# -*- coding: euc-jp -*-
import Tkinter as tk
import Image as I
import ImageTk as Itk
c = tk.Canvas(width=300, height=200)
p=Itk.PhotoImage(file='lena.jpg')
c.create_image(0,0,anchor=tk.NW,
  image=p)
c.pack()
c.mainloop()
```

プログラム解説：

画像キャンバス c を作成し，ファイル「lena.gif」はプログラムと同じディレクトリにあるとして，Itk.PhotoImage で Tkinter で表示できる画像に変換する．そしてキャンバス c の X-Y 座標の原点に画像の左上端を合わせて表示している．Image モジュールは Python Imaging Library(PIL) で公開されている．また ImageTk は PIL と Tkinter との変換モジュールで公開されている．

練習を兼ねて，前述のプログラムをクラスでプログラミングをしよう．例により Frame にキャンバスを張り付ける．似たような構造はボタンウィジェットであったことを思い出してほしい．

```
#! /usr/bin/env python
# -*- coding: euc-jp -*-
import Tkinter as tk
import Image as I
import ImageTk as Itk
class C(tk.Frame):
  # クラス初期設定
  def __init__(self, master=None):
    tk.Frame.__init__(self, master)
  # クラスウィジェット作成
    f=tk.Frame(master,bg='red',bd=3)
    f.pack()
    self.c=tk.Canvas(f,width=250,
      height=150)
    self.c.pack()
# メインプログラム
if __name__ == '__main__':
  c=C()
  p=Itk.PhotoImage(file='lena.jpg')
  c.c.create_image(0, 0,
    anchor=tk.NW, image=p)
  c.mainloop()
```

プログラム解説：

無事に赤枠の中にキャンバス，画像が表示されれば正解である．何がクラスの中に入り，self をどこに付けるか，メインプログラムとクラスの仕分けが重要である．

1) クラスを使ったスクロールドキャンバス

テキストと同様にキャンパスもスクロールできなければ大きな画像は表示できない．方法は，今までに作成したファイルメニューを継承し，テキ

ストウィジェットの代わりにキャンバスウィジェットを利用すれば画像表示は可能である．問題は画像ファイルの X-Y 座標から色データの抽出であるが，Photoimage の get，put 命令で色データの抽出，書き込みができる．まずはスクロールドキャンバスの作成である．TS.py を変更すればよく，ファイル名は，CA.py とする．

File name: CA.py

```
#!/usr/bin/env python
# -*- coding: euc-jp -*-
import Tkinter as tk
import Image as I
import ImageTk as Itk
class CA(tk.Frame):
  # クラス初期設定
  def __init__(self, master=None):
    tk.Frame.__init__(self, master)
  # クラスウィジェット作成
    f=tk.Frame(master, bg="red",
      relief="groove",borderwidth=3)
    # キャンバス作成
    self.c=tk.Canvas(f,width=250,
      height=150,
      scrollregion=(0,0,400,400))
    self.c.grid(column=0, row=0,
      sticky='nsew')
    # X 軸 スクロールバー作成
    sx=tk.Scrollbar(f,
       orient='horizontal',
       command=self.c.xview,width=10)
    self.c["xscrollcommand"]=sx.set
    sx.grid(column=0, row=1,
      sticky='ew')
    # Y 軸 スクロールバー作成
    sy=tk.Scrollbar(f,
      orient='vertical',
      command=self.c.yview, width=10)
    self.c["yscrollcommand"]=sy.set
    sy.grid(column=1, row=0,
      sticky='ns')
    f.grid_columnconfigure(0,weight=1)
    f.grid_rowconfigure(0,weight=1)
    f.pack(fill=tk.BOTH,expand=1)
  # 基本メソッド
  def cget(self,data):
    return self.c.cget(data)
  def config(self,pos,data):
    self.c[pos]=data
  def insert(self,pos,data):
    self.c.insert(pos,data)
  def get(self,pos1,pos2):
    return self.c.get(pos1,pos2)
  def delete(self,pos1,pos2):
    return self.c.delete(pos1,pos2)
# メインプログラム
if __name__ == '__main__':
  c=CA()
  p=Itk.PhotoImage(file='lena.jpg')
  c.c.create_image(0, 0,
    anchor=tk.NW, image=p)
  c.c.create_line(0,0,60,30,
    tags="t1")
  c.mainloop()
```

となる．上記プログラムの仕上りを図 8 に示す．メインプログラムに画像と直線を表示する命令を書き込んでいるから，スクロール機能が確かめられる．

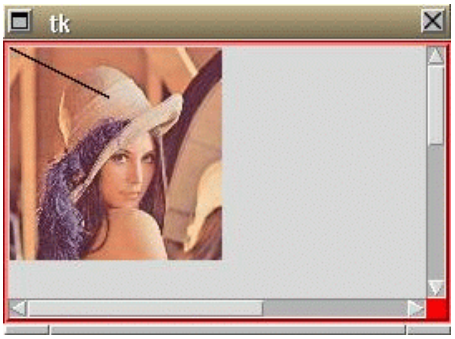

図 8 スクロールドキャンバス CA.py
スクロールドキャンバスは TS.py を変更しただけで作成できる．

2) スクロールドキャンバスとメニューの結合

キャンパスには新規画像を作成する機能，画像を読み取る機能，記録する機能が必要である．そこで，メニュー FM.py を import し，Ed.py と同様の画像エディター Ied1.py を作成する．実際は Ed.py をコピーし，メソッドの変更を加えるだけである．Ied1 の 1 は改定番号で，変更を加えるごとに番号を上げ，最終版は Ied.py の名前に落ち着くことにする．

File name: Ied1.py

```
#!/usr/bin/env python
# -*- coding: euc-jp -*-
from FM import *
from CA import *
import tkMessageBox as M
# 図形エディター
class Ied(FM):
# ファイルメニュー 継承
# メソッド
  def new_f(self):
    fname='noname.txt'
    t.t.delete(1.0,'end')
  def open_f(self):
    global fname
    fname = D.askopenfilename(
      event=None)
    fn=open(fname,"r")
    t.t.delete(1.0,tk.END)
    for line in fn.readlines():
      t.t.insert(tk.END,line[:-1]. \
        decode('euc_jp')+"\n")
    fn.close()
  def save_f(self):
    data=t.t.get(1.0,tk.END-1)
    data1=data.encode('euc_jp')
    fn=open(fname,"w")
    fn.write(data1)
    fn.close
  def saveas_f(self):
    global fname
    fname = D.asksaveasfilename(
      event=None)
    data=t.t.get(1.0,tk.END)
    data1=data.encode('euc_jp')
    fn=open(fname,"w")
    fn.write(data1)
    fn.close
  def exit_f(self):
  # メッセージボックス
    ans=M.askokcancel(u'メッセージ',
      u'終わりますよ．')
    if ans==True:
      exit()
    else:
      return
# メインプログラム
if __name__ == '__main__':
  f=Ied()
  c=CA()
  p=Itk.PhotoImage(file='lena.jpg')
  c.c.create_image(0, 0, anchor=tk.NW,
    image=p)
  c.c.create_line(0,0,60,30,tags="t1")
  f.mainloop()
```

とりあえず，Ied1.py はグラフィクスに関係する import 文の追加，class 文を変更，そして，メインプログラムを変更しただけである．メソッドは変更を加えていないから，テキストエディター Ed.py の最終版のままであり，その参考にしてもらいたい．

さっそく新規画像 (new_f) メソッドの作成であるが，デフォルトの名前設定と，キャンバスに表示されている画像部品の全てを消し去る仕事を行う．find_all 関数は表示されている部品番号を返すからそれを利用する．delete 関数は部品の消去である．プログラムは

```
  def new_f(self):
    fname='noname.txt'
    for i in c.c.find_all() :
      c.c.delete(i)
```

となる．

画像入力 (open_f) メソッドは現在のキャンパスに画像を追加する機能である．どの位置に画像を追加させるかは次の課題として，とりあえず (0,0) に表示させる．プログラムは

```
  def open_f(self):
    global fname
    fname = D.askopenfilename(
      event=None)
    self.p=Itk.PhotoImage(file=fname)
    c.c.create_image(0,0,
      anchor=tk.NW,
      image=self.p, tags='latest')
```

となる．部品番号は自動的に配付されるが，最新という意味で「latest」とタグを付けている．Itk.PhotoImage 変換したイメージ p には self を付けなければ create_image が引用できない．

画像記録 (save_f) メソッドは Tkinter では

Postscript(PS) ファイルしかサポートされていない．そのためにプログラムは

```
  def save_f(self):
    c.c.postscript(file='a.ps')
```

となる．PIL で ps ファイルより別の画像に変換することは可能だが，現バージョン (PIL 1.16) では画像が縮小劣化するので，これ以上は深入りしない．

名前を付けての保存 (saveas_f) メソッドもダイアログを表示してファイル名を定めるだけである．プログラムは

```
  def saveas_f(self):
    global fname
    fname = D.asksaveasfilename(
      event=None)
    c.c.postscript(file=fname)
```

となる．fname はグローバル変数であるから self の必要はない．

終了 (exit_f) メソッドは変更がない．ここまでの変更をまとめて Ied2.py とすると，プログラムは

File name: Ied2.py

```
#!/usr/bin/env python
# -*- coding: euc-jp -*-
from FM import *
from CA import *
import tkMessageBox as M
# 図形エディター
class Ied(FM):
# ファイルメニュー 継承
# メソッド
  def new_f(self):
    fname='noname.txt'
    for i in c.c.find_all() :
      c.c.delete(i)
  def open_f(self):
    global fname
    fname = D.askopenfilename(
      event=None)
    self.p=Itk.PhotoImage(file=fname)
    c.c.create_image(0,0,
      anchor=tk.NW,
      image=self.p, tags='latest')
  def save_f(self):
    c.c.postscript(file='a.ps')
  def saveas_f(self):
    global fname
    fname = D.asksaveasfilename(
      event=None)
    c.c.postscript(file=fname)
  def exit_f(self):
  # メッセージボックス
    ans=M.askokcancel(u'メッセージ',
      u'終わりますよ．')
    if ans==True:
      exit()
    else:
      return
# メインプログラム
if __name__ == '__main__':
  master=None
  f1=tk.Frame(master,bg='green',bd=3)
  f1.pack(anchor='nw')
  f=Iedd(f1)
  c=CA()
  c.c.create_line(0,0,60,30,tags="t1")
  p=tk.PhotoImage(file='lena.gif')
  c.c.create_image(50,0,
    anchor=tk.NW,image=p)
  f.mainloop()
```

となる．

Tcl/Tk の知識で Tkinter でのキャンバスウィジェット命令の検討はつくが，正確ではない．課題を解決する前に，どの様なキャンバスウィジェットメソッドがサポートされているか，正確に知る必要がある．その方法は，対話もードで python を呼び出し，

```
>>> from FM import *
>>> from CA import *
>>> c=CA()
>>> dir(c.c)
['_Misc__winfo_getint',
..... 省略 .....
ew_moveto', 'yview_scroll']
>>>
```

とすると，莫大な数のメソッドが端末に表示さ

れる．また PIL の正式情報は，
http://www.pythonware.com/library
/index.htm
に公開されている．

画像の移動

読み取った画像を (0,0) に張り付けるのでは役に立たないので，マウスで画像を選択し，任意の部位に移動するルーチンを付加する．そのためにはマウスの位置情報を取得するバインド機能が必要になる．

さっそく最も簡単なバインドプログラムを以下に示す．キャンバスを作成し，そこにボタン 1 を押してドラグすると x-y 座標が端末に表示されるだけのプログラムである．

File name:　pr_xy.py

```
#!/usr/bin/env python
# -*- coding: euc-jp -*-
import Tkinter as tk
# B1 Motion バインド
def b1motion(event):
  print 'x = ',event.x, \
    'y = ', event.y
# メインプログラム
c = tk.Canvas(width=300, height=200)
c.pack()
c.bind('<B1-Motion>',b1motion)
c.mainloop()
```

プログラム解説

バインド関数は b1motion で，引数 event を浮け付ける．event は構造体になっており，カーソルの X-Y 座標は event.x，event.y の変数名になっており、それらを表示している．キャンバスインスタンス c には bind メソッドがあり，イベント条件とイベント処理関数を登録する．

次は実際に，サークルを作成し，それにマウスを当ててドラグするとサークルが移動するプログラムを作成する．

File name:　mv_circle.py

```
#!/usr/bin/env python
# -*- coding: euc-jp -*-
import Tkinter as tk
# マウスバインド処理
# タグ 'a1' 移動処理
def mv(event):
  data=c.coords('a1')
  x1,y1,x2,y2=data
  oldx=(x1+x2)/2.0
  oldy= (y1+y2)/2.0
  dx=event.x-oldx; dy=event.y-oldy
  c.move('a1',dx,dy)
# メインプログラム
c = tk.Canvas(width=300, height=200)
c.pack()
r1=c.create_oval(10, 10, 30, 30,
  fill='red', tag='a1')
#c.bind('<B1-Motion>',mv)
#c.bind('<Button-1>',mv)
c.tag_bind('a1','<Button-1>',mv)
c.tag_bind('a1','<B1-Motion>',
  mv, '+')
c.mainloop()
```

プログラム解説

キャンバスに直径 20 のサークルを (20,20) の位置に作成し，タグを'a1' と文字列で指定する．マウス左クリックで'<Button-1>'，ドラグで'<B1-Motion>' イベントが発生し，共に mv 関数を呼ぶ．

mv 関数の move メソッドはサークルを dx，dy だけ移動する命令を行うが，event で示される現在のマウスの位置，そして coords で示されるサークルの位置情報から計算する．oldx，oldy はサークルの中点座標になる．

バインドプログラムは引数が (event) に限られるため汎用性が望めず，残念ながら mv 関数はメインプログラムと切り離して記述はできない．コメントマーク (#) が付いている c.bind 命令ではキャンバスのどこでもクリックすれば mv 関数が作動する．一方の c.tag_bind では，タグで示めされるウィジェットにマウスを移動しなけれ

ば mv 関数が作動しない．

もう一つの例として，いくつかのサークルがキャンバスに有り，その一つを左ボタンで指定し，ドラグで移動させるプログラムを作成する．仕上りを図 9 に示す．

File name: mv_circles.py

```
#!/usr/bin/env python
# -*- coding: euc-jp -*-
import Tkinter as tk
# マウスバインド処理
def mv(event):
  try:
    id=c.find('closest',
      event.x,event.y)
    tag,now=c.gettags(id)
    data=c.coords(tag)
    x1,y1,x2,y2=data
    oldx=(x1+x2)/2.0
    oldy= (y1+y2)/2.0
    dx=event.x-oldx; dy=event.y-oldy
    c.move(tag,dx,dy)
  finally:
    return
# メインプログラム
c = tk.Canvas(width=300, height=200)
c.pack()
x=10; y=30
for i in range(3):
  t='a'+str(i)
  c.create_oval(x, x, y, y,
    fill='red', tag=t)
  x=x+20; y=x+20
c.bind('<B1-Motion>',mv)
c.bind('<Button-1>',mv)
c.mainloop()
```

プログラム解説

メインプログラムで range(3) を利用してサークルを 3 個作成する．サークルのタグ名は'a' を接頭語にして range の数字を利用している．サークルの配置は x，y にそれぞれ 20 を加算して斜め下に並べた．

マウスバインドの mv 関数は find 命令でマウス座標に最も近い部品 ID を得る．gettags で ID のタグ名を得る．coords でその部品の位置を知り，dx,dy を計算して move 命令で部品をマウス位置に移動させている．

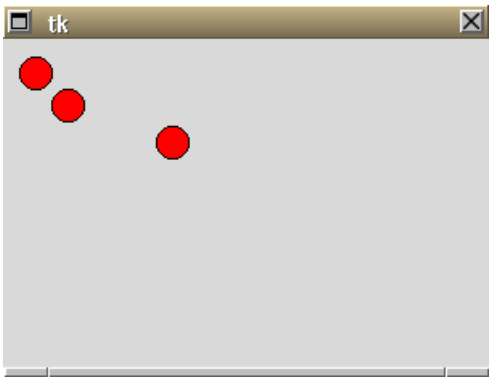

図 9 サークルの移動
サークルを 3 個キャンバス上に作成し，3 番目のサークルをマウスドラグで移動した．

もし，マウス位置に部品が見当たらない場合はエラーが発生する．その場合，何も動作をせずにイベントを強制終了させるために try/finally 例外処理を行った．bind 命令と tag_bind 命令との使い分けは，特定のタグが明らかであれば tag_bind 命令，そうでなければウィジェット名を find 命令で探し，bind 命令を実行する．

バインド処理と部品移動の理解が深まったところで本論に戻り，画像ファイルの移動機能および部品どうしの重なりを変更する機能を付加したプログラムを最終版 Ied.py として，プログラムと仕上りを以下に示す．

File name: Ied.py

```
#!/usr/bin/env python
# -*- coding: euc-jp -*-
from FM import *
from CA import *
import tkMessageBox as M
# 図形エディター
class Ied(FM):
# ファイルメニュー 継承
# メソッド
  def new_f(self):
```

```
    fname='noname.txt'
    for i in c.c.find_all() :
      c.c.delete(i)
  def open_f(self):
    global fname
    fname = D.askopenfilename(
      event=None)
    self.p=Itk.PhotoImage(file=fname)
    c.c.create_image(0,0,
      anchor=tk.NW,
      image=self.p, tags='latest')
  def save_f(self):
    c.c.postscript(file='a.ps')
  def saveas_f(self):
    global fname
    fname = D.asksaveasfilename(
      event=None)
    c.c.postscript(file=fname)
  def exit_f(self):
  # メッセージボックス
    ans=M.askokcancel(u'メッセージ',
      u'終わりますよ．')
    if ans==True:
      exit()
    else:
      return
# メインプログラム
if __name__ == '__main__':
  # マウスバインド
  # 部品移動
  def mv(event):
    try:
      id=c.c.find('closest',event.x,
        event.y)
      idn=id[0]
      c.c.tag_raise(idn)
      data=c.c.coords(idn)
      x1=data[0]
      y1=data[1]
      dx=event.x-x1; dy=event.y-y1
      c.c.move(idn,dx,dy)
    finally:
      return
  # 部品重なり：下の部品を上へ
  def up(event):
    try:
      id=c.c.find('closest', event.x,
        event.y)
      idn=id[0]
      c.c.tag_raise(idn)
    finally:
      return
  # キャンバスメインプログラム
  f=Ied()
  c=CA()
  c.c.create_line(0,0,60,30,tags="t1")
  p=Itk.PhotoImage(file='lena.jpg')
  c.c.create_image(50,0,
    anchor=tk.NW,
    image=p, tag='abc')
  c.c.bind('<B1-Motion>',mv)
  c.c.bind('<Button-1>',mv)
  c.c.bind('<Button-3>',up)
  f.mainloop()
```

プログラム解説

前述のごとく，イベント処理プログラムは import で読み込むことができないためにメインプログラム領域に書き込む．左クリックおよび左ドラグは mv 関数で対応し，右クリックは指定された部品を画面前方に移動する．これらのキャンバスバインドは c.c.bind として mainloopu() のすぐ上に記述している．

上記プログラムではキャンバスでマウスクリック，ドラグ操作で必ずバインドイベントが動作する．この動作を停止するには

```
c.c.bind('<B1-Motion>','-',mv)
c.c.bind('<Button-1>','-',mv)
c.c.bind('<Button-3>','-',up)
```

とイベントと実行関数との間に'-' を挿入すればよい．再度，別の実行関数をバインドに登録することができる．

mv 関数は左クリックを行ったとき，部品が発見できなければエラーでプログラムが停止する．それを防止する目的で try:を使用した．また，タグの取扱いも面倒なので，部品 ID を利用した．直線などの coors は x-y 座標の左上，右下を示す 4 要素リスト情報が得られるが，画像イメージは左上端だけの 2 要素リストで返る．そのため，左上端の x 座標を x1，y 座標を y1 として，

マウス移動を左上端に合わすプログラムに変更した．また，部品が他の部品の下に存在すると正確な位置調整ができないため，移動する部品は前面に位置するように c.c.tag_raise 関数を使用した．

up 関数は mv 関数の上の部分だけであるから，理解できると思う．仕上りを図 10 に示す．

トップレベル作成法

前述の例では使用しなかったが，ダイアログと同じように root ウィンドウとは別にトップレベルでウィンドウを作成することができる．プログラムの雛型は，

```
# トップレベル
def top():
  t1=tk.Toplevel()
  t1.title('Toplevel')
  w = tk.Canvas(t1,width=150,
    height=100,bg='white')
  w.pack()
  w.create_line(0,0,100,100)
```

とすればよい．ボタン機能などを用いて top 関数を呼び出せば，幅 150，高さ 100 のキャンバスに (0,0,100,100) の直線が表示される．

ボタン機能の追加は FM.py の Frame を横並びにすること，および Frame 名を公開することでできる．変更点は，

```
  # クラスウィジェット作成
    self.f=tk.Frame(master,bg='red',
      bd=3)
    self.f.pack(anchor='nw')
    file=tk.Menubutton(self.f,
      text='File',relief=tk.RAISED)
    file.pack(side=tk.LEFT)

# Ied.py メインプログラム追加
# キャンバスメインプログラム
f=Ied()
b=tk.Button(f.f,text='Top',
  command=top)
b.pack(side=tk.LEFT)
c=CA()
```

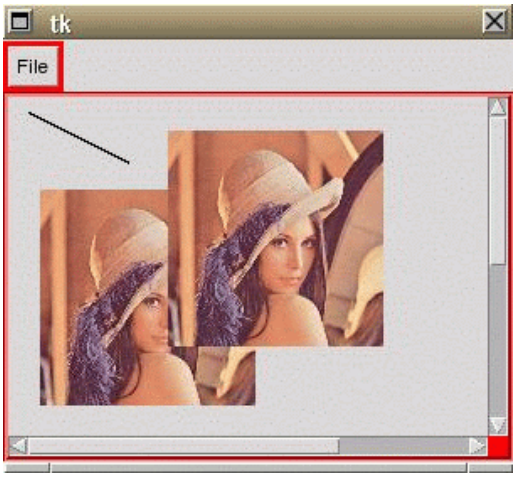

図 10 Ied.py による画像入力と部品移動
直線と'lena.gif' はプログラム駆動時に表示されているもので，新たに'lena.gif' を加えた．それぞれの部品はマウスドラグで移動できる．

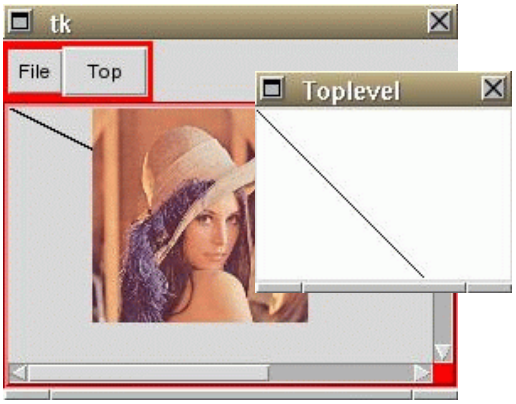

図 11 グラフィクエディター Ied.py
メニューボタンの横並びにトップレベル呼び出しボタンを追加している．

となる．つまり，赤で示す Frame 名 f に self を付け，メニューボタンの親ウィジェットを self.f にする．これでフレーム名 f は公開される．メニューボタン名 file のパックは side=tk.LEFT を指定するだけである．Top と表示されているボタンは FM.py で公開されたフレームを使用するから，親ウィジェットに f.f が指定でき，横並びに pack できる．仕上りを図 11 に示す．

以上で，Tkinter，クラスを用いた GUI 基本

ソフトの作成を終える．テキストエディター，グラフィクエディター共にもっと機能を増やすことができるが，一般に公開されている機能まで近づけることに意味はないと思うし，またプログラムの見通しが悪くなり，容易に変更できなくなる．この程度のプログラムを基本ソフトとして常時たづさえており，特別な処理プログラムが必要なときに，これらのプログラムをベースにして，開発を進めるのがよいと思う．

心電図ベクトル波形作成システム

心電図ベクトル波形作成システムとは心電図12誘導で測定した検査結果を画像スキャナーで読み取り，その画像よりベクトル心電図波形をつくり出すプログラムである．画像読み取り作業は市販のプリンタースキャナーシステムを利用することにして，1) 得られた画像をトリミングする，2) 心電図波形だけを抽出する，3) その波形よりX-Y座標を検出する，そして最後に4) ベクトル心電図を合成する作業が必要になる．これらの作業を編集メニューに組み入れるか，それともファンクションボタンを横に列べるのが良いかなどの使い勝ては後で判断することにして，まずはファイルメニューの右横にボタンを配置し，下に画像編集画面を配置するだけのプログラムだけを考えて，プログラム名をVed1.pyとする．FM.py のフレーム f は前項で self を付けて横並び公開を行った．

ファンクションボタンの設定

Ied.py のコピーファイルを Ved1.py として，バインドルーチンを削除した骨格プログラムを以下に示す．

File name: Ved1.py

```
#!/usr/bin/env python
# -*- coding: euc-jp -*-
from FM import *
from CA import *
import tkMessageBox as M
class Ved(FM):
  pass
# キャンバスメインプログラム
if __name__ == '__main__':
  master=None
  f=Ved()
  b1=tk.Button(f.f,text=u'終了１',
    command=exit)
  b1.pack(side='left')
  b2=tk.Button(f.f,text=u'終了２',
    command=exit)
  b2.pack(side='left')
  b3=tk.Button(f.f,text=u'終了３',
    command=exit)
  b3.pack(side='left')
  c=CA()
  c.c.create_line(0,0,60,30,
    tags="t1")
  p=tk.PhotoImage(file='lena.gif')
  c.c.create_image(50,0,
    anchor=tk.NW,
    image=p, tag='abc')
  f.mainloop()
```

となり，結果を図12に示す．

図 12 ベクトル心電図処理画面 Ved1.py
メニューボタンのフレームは FM.py のフレームを横並びに指定，また，self を付けて公開している．

上記のプログラムはオリジナルの FM.py のフレームを横並びに設定することこと，フレーム名を公開することにより容易に作成できる．またそれを継承して新たな Ved クラスを作成した．どの様なメソッドが必要になるのか，不定であるため，とりあえず，pass を入力しておき，Python プログラム書法を守ることにした．もしフレームの枠が邪魔ということであれば，bd=0 とすることで隠すことができる．また，改良予定のファンクション機能のためにボタンを3個並べている．

キャンバスサイズの変更

いよいよ心電図画像ファイルの貼り付けにな

る．新規作成は編集画面の部品削除であるから問題はない．画像読み取りも読み取りダイアログと編集画面の貼り付けは問題がない．しかし，画像データがデフォルト設定より大きい場合にはスライダーが対応できなくなる．

PhotoImag を p とすると，入力画面の縦，横のサイズを求めるには，p.width()，p.height() で得られる． またキャンバスのサイズ変更は，c.c['width']=x，c.c['height']=y で変更できるから，前述プログラムで pass と記述した部分に新しく継承メソッドを書き直すと，

```
class Ved(FM):
# ファイルメニュー 継承
# メソッド
  def new_f(self):
    self.fname='noname.txt'
    for i in c.c.find_all() :
      c.c.delete(i)
  def open_f(self):
    for i in c.c.find_all() :
      c.c.delete(i)
    self.fname = D.askopenfilename(
      event=None)
    self.p=Itk.PhotoImage(
      file=self.fname)
    c.c.create_image(0,0,
      anchor=tk.NW,
      image=self.p, tags='latest')
    #画像サイズ
    x=self.p.width()
    y=self.p.height()
    #キャンバスサイズ変更
    c.c['width']=x
    c.c['height']=y
```

となる．

画像トリミング

終了 1 ボタンの表示ををトリムに変更して，トリミング枠を表示させる．マウス左ボタンのドラグでトリミング枠が確定し，フラッシュさせると，見やすくなる．ドリミングの実行は終了 2 ボタンで行い，自動的にトリミングを受けた画像サイズに変更させる．

この作業は伸縮自在の四角形ラバーアングル作成と呼ばれている Tcl/Tk のテクニックの一つで，まずは独立したプログラムを紹介し，次に Ved2.py に導入する．

```
#!/usr/bin/env python
# -*- coding: euc-jp -*-
import Tkinter as tk
# 四角形を作る.
def r_create(event):
  global sx,sy
  c.create_rectangle(event.x,
    event.y,event.x,event.y,
    outline='black',width=2,
    tags='rubbershape')
  sx=event.x; sy=event.y
# 四角形をドラグする.
def r_drag(event):
  global sx,sy,x2,y2
  data=c.coords('rubbershape')
  x1,y1,x2,y2=data
  x2=event.x; y2=event.y
  if (x2>sx) and (y2>sy):
    c.coords('rubbershape',sx,y1,
      x2,y2)
  else:
    c.coords('rubbershape',x2,y2,
      sx,sy)
# 四角形を消す
def r_end(event):
  global sx,sy,x2,y2
  print sx,sy,x2,y2
  c.delete('rubbershape')
# メインプログラム
c = tk.Canvas(width=300, height=200)
c.pack()
c.bind('<Button-1>', r_create)
c.bind('<B1-Motion>', r_drag)
c.bind('<ButtonRelease-1>', r_end)
c.mainloop()
```

プログラム解説

始めの 3 行は定型通り．バインド関数 r_create は左ボタンを押した時点で線幅 2 の四角形を作成，タグ名を'rubbershape' とする．バインド関数 r_drag は'rubbershape' の左上端，右下端の X-Y 座標を得，ドラグしている間，その四角形の coords を変化させる．四角形が伸

縮しているように見えるのはバインド割り込みで何度も書き直しているためである．バインド関数 r_end は左ボタンを放すことにより実行し，四角形の座標を端末表示し，'rubbershape' の四角形をキャンバスより削除する．本例は四角形がキャンバスに表示されるだけだから図示しない．読者はプログラムを作成して確かめてほしい．

ラバーアングルの作成法が理解できたところで Ved1.py のコピーを Ved2.py としてプログラム修整を行う．

new_f メソッドおよび，open_f メソッドは先ほど示した．トリミング開始ボタンは，左ボタンドラグで黒色ラバーアングルを表示させる．その関係するプログラムは全てメインプログラムのバインド領域に書き込む．したがって，Ved2.py のメインプログラムは，

```
# メインプログラム
if __name__ == '__main__':
  master=None
  f=Ved()
  # マウスバインド
  # 四角形を作る.
  def r_create(event):
    global sx,sy
  # find_withtag
    c.c.delete('rubbershape')
    c.c.create_rectangle(event.x,
      event.y,event.x, event.y,
      outline='black',
      width=2, tags='rubbershape')
    sx=event.x; sy=event.y
  # 四角形をドラグする.
  def r_drag(event):
    global sx,sy,x2,y2
    data=c.c.coords('rubbershape')
    x1,y1,x2,y2=data
    x2=event.x; y2=event.y
    if (x2>sx) and (y2>sy):
      c.c.coords('rubbershape',
        sx,y1,x2,y2)
    else:
      c.c.coords('rubbershape',
        x2,y2,sx,sy)
  # トリミング開始
  def trim_on():
    c.c.bind('<Button-1>', r_create)
    c.c.bind('<B1-Motion>', r_drag)
  # トリミング終了
  def trim_off():
    im=I.open(f.fname)
    x1,y1,x2,y2=c.c.coords(
      'rubbershape')
    im2=im.crop((int(x1),int(y1),
      int(x2),int(y2)))
    im2.show()
  b1=tk.Button(f.f,text=u'トリム開始',
    command=trim_on)
  b1.pack(side='left')
  b2=tk.Button(f.f,text=u'トリム終了',
    command=trim_off)
  b2.pack(side='left')
  b3=tk.Button(f.f,text=u'終了３',
    command=exit)
  b3.pack(side='left')
  c=CA()
  f.mainloop()
```

となる．

トリミング終了ボタンはプログラムが正しく動作するかを確かめるために，一次的に PIL の show() 命令を使用した．仕上りを図 13 に示す．

図 13 ラバーアングルでトリム画像を抽出 Ved2.py
File メニューの横のトリム開始で編集画面にラバーアングルを作成し，トリム終了でトップレベルで選択領域の画面が表示される．

プログラム解説

前に述べたラバーアングルの雛型プログラ

ムをトリム開始ボタン command にコピーし，c.create が c.c.create になっただけで，説明の必要はないだろう．

トリミング領域は改めて coords で得ることにした．その理由は右から左にドラグしたときの四角形の指定が逆になっているため，再度確かなデータを得ることにした．また，図形ファイルの表示が (0,0) から始まっているから，PIL の im 波形と同じ座標になり，容易に crop ができる．

トリミングが成功し，show() 命令で画像が表示できれば，トリム終了命令の第 2 段階に移る．crop で作成したトリム画像を'tmp.jpg' に保存し，再度キャンバスに表示する．PIL でのイメージファイル呼び出しが f.fname で，tmp.gif のファイル呼び出しが fname のままになっている理由は前者はクラスで self.f と指定したファイル名であり，後者は trim_off 関数の局所変数を使用して区別する．また，イメージ画像 p はグローバル宣言しなければ表示されない．常に画像表示ポインターは self またはグローバル宣言が必要である．trim_off 関数の完成したプログラムを以下に示す．

```
# トリミング終了
def trim_off():
  global p
  global maxx,maxy
  x1,y1,x2,y2=c.c.coords(
    'rubbershape')
  im=I.open(f.fname)
  im2=im.crop((int(x1),int(y1),
    int(x2),int(y2)))
  im3=im2.convert('RGB')
  im3.save('tmp.gif')
  for i in c.c.find_all() :
    c.c.delete(i)
  fname='tmp.gif'
  p=Itk.PhotoImage(file=fname)
  c.c.create_image(0, 0,
    anchor=tk.NW, image=p)
  maxx=p.width()
  maxy=p.height()
  c.c['width']=maxx
  c.c['height']=maxy
  c.c.bind('<Button-1>','-',
    r_create)
  c.c.bind('<B1-Motion>','-',
    r_drag)
```

となる．トリミング画像は im2，それを'RGB' にモード変換し，im3 を tmp.jpg ファイルに保存する．再度キャンバス内をクリアーし，保存ファイルを読み込む．これらの作業は Tkinter だけでは処理できず，PIL の助けを借りている．トリミング後は，最大画像で表示できるようにキャンバスの縦，横のサイズを変更している．またこの縦，横の値はカーソルや，スキャン表示などに利用するため，グローバル変数 maxx，maxy とした．最後にバインドイベントをクリアーする．同様の変更は open_f でも行った．そのようにしてできあがった GUI を図 14 に示す．

図 14 トリミング後の心電図解析画面
トリミング終了後直接キャンバスに張り付けている．

心電図波形の基線設定

図 15 に示す心電図は上段より第 I 誘導，第 II 誘導，第 III 誘導であり，それらの基線 (X 軸) を指定しなければ，数値としての波形座標が計算できない．そこで，本例では目視により 3 チャネルの基線を設定することにして，カーソル表

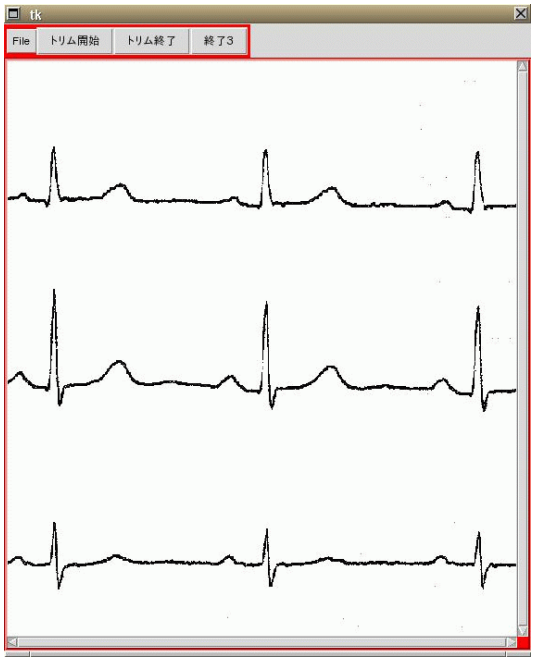

図 15 心電図解析画面
スキャナーで取り込んだ心電図チャート．
上段より I，II，III 誘導を示す．

示，移動法，位置決定のプログラミングを考える．Ved2.py を Ved3.py にコピーしてプログラムを改良する．

メインプログラムの終了 3 ボタンの位置に追加のカーソル表示ボタン，また，基線 1，基線 II，基線 III の位置決定ボタンを作成する．

```
  # カーソル移動
  def cur1_move(event):
    global old_cur1y
    c.c.move('cur1',0,
      event.y-old_cur1y)
    old_cur1y=event.y
  # カーソル作成
  def base():
    global old_cur1y
    global maxx
    old_cur1y=5
    c.c.create_rectangle(0,0,10,10,
      fill='green', tag='cur1')
    c.c.create_line(0,5,maxx,5,
      fill='green',
        width=2, tag='cur1')
    c.c.bind('<Button-1>',
      cur1_move)
    c.c.bind('<B1-Motion>',
      cur1_move)
  # 基線記録
  def baseI():
    global old_cur1y,baseI_y
    baseI_y=old_cur1y
  def baseII():
    global old_cur1y,baseII_y
    baseII_y=old_cur1y
  def baseIII():
    global old_cur1y,baseIII_y
    baseIII_y=old_cur1y
  # キャンバスメインプログラム
  b1=tk.Button(f.f,text=u'トリム開始',
    command=trim_on)
  b1.pack(side='left')
  b2=tk.Button(f.f,text=u'トリム終了',
    command=trim_off)
  b2.pack(side='left')
  b3=tk.Button(f.f,text=u'カーソル',
    command=base)
  b3.pack(side='left')
  b3=tk.Button(f.f,text=u'基線 I',
    command=baseI)
  b3.pack(side='left')
  b3=tk.Button(f.f,text=u'基線 II',
    command=baseII)
  b3.pack(side='left')
  b3=tk.Button(f.f,text=u'基線 III',
    command=baseIII)
  b3.pack(side='left')
  b3=tk.Button(f.f,text=u'終了３',
    command=exit)
  b3.pack(side='left')
  c=CA()
  f.mainloop()
```

プログラム解説

一気にファンクションボタンが増えたが複雑なことはない．メインプログラムから説明すると，「カーソル」ボタンは base を呼び，カーソルを表示する．「基線 I」は baseI を呼び，第 I 誘導の基線の Y 座標を記憶，以下「基線 III」まで同じである．ボタンウィジェット名が全て b3 となっているが，個別ボタンとして引用しなければ問題ない．

カーソルを作成する base 関数は，心電図チャート幅 maxx をグローバルで呼ぶ．また始めのカーソル Y 座標 old_cur1y を 5 に指定する．次いで（0,0,10,10）の位置に緑の四角形を作成，タグを’cur1’ とする．また，（0,5）から，(maxx,5) まで緑の幅 2 の直線を作成，タグを同じ’cur1’ として四角形とグループ化した．そして，マウスボタン 1 のドラグでカーソルの移動関数 cur1_move を呼び出している．マウスバインドは常に最終命令に対応するから，重複して指定しても問題ないことはバスの乗客窓コントローラで実証すみである．

実際にカーソルを移動させる関数は cur1_move 関数である．move 命令は dx,dy，すなわち x,y 方向の変化分だけを移動させるために初期値が必要である．そのため old_cur1y をグローバル変数にして，Y 軸の初期値を得た．x 軸方向にカーソルは移動しないので，dx 方向の変化は 0 である．dy は現在のカーソル位置，event.y との差を指定し，移動タグは’cur1’ と指定しているから，四角形も直線も同じ量だけ移動する．移動後に，old_cur1y の値を現在値に更新している．

I，II，III 誘導の各基線の Y 値は baseI，baseII，baseIII 関数で求める．単に，old_cur1y の値を記録しているだけだから簡単である．新しく追加したボタンとトリミングされた心電図波形を図 16 に示す．

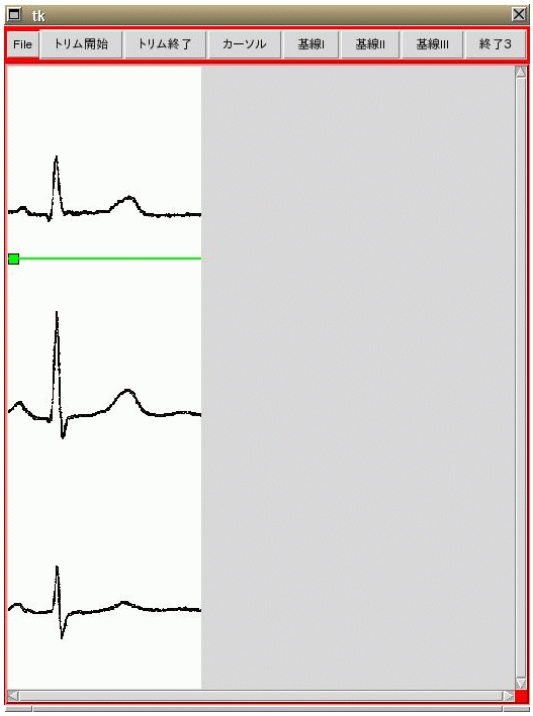

図 16 心電図解析画面 Ved3.py
カーソル表示，基線 I，基線 II，基線 III 設定ボタンを追加．心電図は 1 心拍にトリミングを行った．上段より I，II，III 誘導を示す．

心電図波形の座標抽出

記録用紙に描かれた心電図は一見画像が美しく見えていても，詳細に分析すると汚れやが数多く有り，そのグラフから X-Y 座標を自動的に読み取ることは難しい．

筆者の行った方法はカラーのチャートを濃淡のある白黒チャートに変換し，灰色になったグラフの目盛を灰色の閾値を移動することにより，出来るだけ灰色目盛を白色に変換し，心電図波形のみを浮き立つように調整した．使用したソフトは xv と gimp である．

Tkinter の PhotoImage は get 命令でピクセル単位で色を取得し，put 命令で色の指定ができる．今回は PIL の PhotoImage を利用しているため，画像データを p とし，その X-Y 座標の色は，

```
data=p._PhotoImage__photo.get(x,y)
```

で得られ，data には RGB の文字列リストが記録される．これを利用して黒点の数を調べる．

次にトレースの方法であるが，X 軸方向にトレースすることにして，Y 軸方向には 3 本の心電図波形が存在する．単純に Y 軸を 3 分割して，そ

れぞれの分画でこの Y 軸の高さの間であればこの誘導の黒点だと if 文で検出することにした．そして，それぞれの分画で，黒点に相当するピクセルの平均値を計算し，その平均値を心電図の Y 座標と判断する．このようにして作成したのが trace 関数で，以下に示す

```
# トレース
def trace():
  global AA,BB,CC
  # トップレベル作成
  t1=tk.Toplevel()
  t1.title('Toplevel')
  w = tk.Canvas(t1,width=maxx,
    height=maxy,bg='white')
  w.pack()
  # チャネル下限設定
  sep=maxy/3
  sep2=2*sep
  # キャンバスにカーソル作成
  c.c.create_line(1,0,1,maxy,
    fill='green',tag='cursor')
  curx=0
  AA=[]; BB=[]; CC=[]
  for i in range(maxx):
    AA.append(0); BB.append(0)
    CC.append(0)
  # x 軸スキャン
  for i in range(maxx):
    # カーソル移動
    curdx=i-curx
    c.c.move('cursor',curdx,0)
    curx=i
    c.c.update()
    # 黒点 y 座標検出
    A=[]; B=[]; C=[]
    c.c.coords('cursor',i,0,i,maxy)
    for j in range(maxy):
      data=p._PhotoImage__photo.get(
        i, j)
      r=string.split(data,' ')
      x=int(r[0])
      if x< 50:
        if j<sep :
          A.append(j)
        elif j<sep2 :
          B.append(j)
        else:
          C.append(j)
    # AA チャネルの平均
    An=len(A)
    sum=0
    if An!=0 :
      for k in range(An):
        sum=sum+A[k]
      AA[i]=sum/An
    # BB チャネルの平均
    Bn=len(B)
    sum=0
    if Bn!=0 :
      for k in range(Bn):
        sum=sum+B[k]
      BB[i]=sum/Bn
    # CC チャネルの平均
    Cn=len(C)
    sum=0
    if Cn!=0 :
      for k in range(Cn):
        sum=sum+C[k]
      CC[i]=sum/Cn
  if i>0:
    w.create_line(i-1,AA[i-1],i,
      AA[i], width=2)
    w.create_line(i-1,BB[i-1],i,
      BB[i], width=2)
    w.create_line(i-1,CC[i-1],i,
      CC[i], width=2)
```

プログラム解説：

トレース結果を視覚に訴えるためにトップレベルのウィンドウを作成し，そこに結果を表示することにした．また，キャンバスには Y 軸方向のカーソルを表示し，処理中の X 座標を示している．AA，BB，CC の各リストは I，II，III 誘導の Y 座標が記録される．このデータは後の解析に利用されるためグローバル宣言をしておく．Y 軸分画値は sep と sep2 とし，とりあえず AA，BB,CC の各データ領域には 0 を記入した．

X 軸のスキャンであるが，i を変数として最大値 maxx までの for 文でループを作る．カーソル移動を行い，黒点の Y 座標を収納する Y 軸のスキャンは j を変数として最大値 maxy までの for 文でループを作る．リスト変数 A，B，C を

作成する．もしピクセルの値が 50 以下であれば黒点とみなして，Y 座標の高さに従って，A，B，C のどれかに Y 値を記録する．変数 j のループが終了した所で，A，B，C の平均値を計算し，AA，BB，CC の Y 座標として格納し，トップレベルのキャンバスに直線で検出した心電図座標を描く．if 文はもったいないように思うが，そのための処理時間は 1msec 以下であることと，0 以下のリスト値は無いために付けた．およそ 1 秒でグラフは完成する．

trace 関数のファンクションボタンを新しく作成する必要がある．フレーム名を f1 として次の段を作成し，その中にトレースボタンを収納した．メインプログラムの追加文は以下のようになる．また string モジュールを利用するためにプログラムの始めに import 文を追加した．

```
b3=tk.Button(f.f,text=u'終了３',
  command=exit)
b3.pack(side='left')
f1=tk.Frame(master,bg='red',bd=3)
f1.pack(anchor='nw')
b3=tk.Button(f1,text=u'トレース',
  command=trace)
b3.pack(side='left')
c=CA()
f.mainloop()
```

結果を図 17 に示す．新しいフレームとトレースボタンが追加され，III 誘導の基線が表示され，トリミング画像の横にトップレベルでの心電図折れ線座標が表示されている．よく観察すると心電図のピークが幾分低くなっているが，黒点の平均値を採用した影響が表れている．

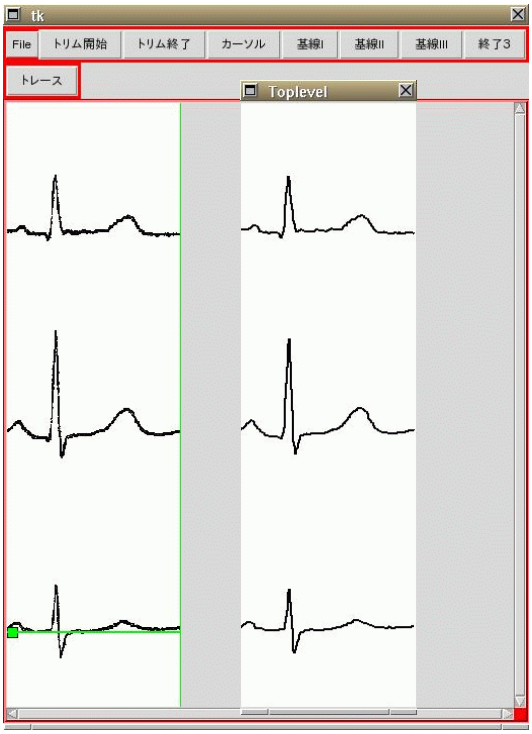

図 17 心電図波形のトレース
トレースボタン用に新しくフレームを作成．トレース結果を新しくトップレベルに表示している．

画像イメージの色操作

キャンバス上の多くのウィジェットはピクセル単位で色操作はできないが画像イメージだけはそれが可能であり，グラデーションのついた画像を作成することができる．以下のプログラムは簡単な get，put のピクセル操作であり，ぜひ，対話形式で Python を駆動し，動作を確認してほしい．

```
from Tkinter import *
c = Canvas(width=128, height=128)
p=PhotoImage(file='lena.gif')
c.create_image(0,0,anchor=NW,image=p)
c.pack()
# 赤の四角を作る．
p.put("#ff0000", to=(0,0,30,30))
# (29,0) の色を見る．
p.get(29,0)
u'255 0 0'  <- が返る．
p.get(30,0) <- が返る．
u'201 128 102'
```

プログラム解説：

キャンバス c を作成し，gif 形式画像を PhotoImage を作り，それをキャンバスに貼り付ける．次いで，座標 (0,0) から (29,29) までの

ピクセルを赤色に変換する．これは画像データの変換であって，キャンバスではない．キャンバスは自動的に新しい画像データに更新する．

色データの指定はRGBの順で，16進数字を#を付けて表示する．RGBデータを独自のr，g，bなどの変数にする場合は，

```
color='#%02x%02x%02x' % (r,g,b)
p.put(color, to=(0,0,30,30))
```

とすればよい．またr，g，bの数値を得るにはカラーコードがユニコード文字列で返るために

```
import string
data=p.get(29,0)
r=string.split(data,' ')
int(r[0])
255    <- と返る.
```

とすれば，後の計算ができるようになる．結果を図18に示す．

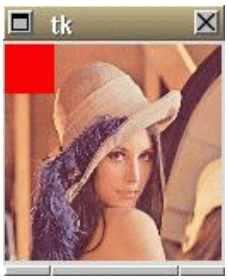

図18 get, put メソッド
PhotoImageはpicel単位で色情報を指定し，また読み取りができる．変更するのは画像データであって，キャンバスデータではない．キャンパス画像は自動的に更新される．

上記の例では画像がGIFファイルであるためにTkinter付属のPhotoImageメソッドを使用したが，PILの場合は，

```
data=p._PhotoImage__photo.get(x,y)
r=string.split(data,' ')
```

とすればよい．繰り返しになるが，これらの色データ処理は画像データに対して行われるものであって，キャンバスに表示されている画像ではない．

第II誘導心電図波形の合成

心電図の話しはPythonプログラミングに直接関係するものではないが，コンピュータを利用して科学すると言う意味で話題性があると思う．

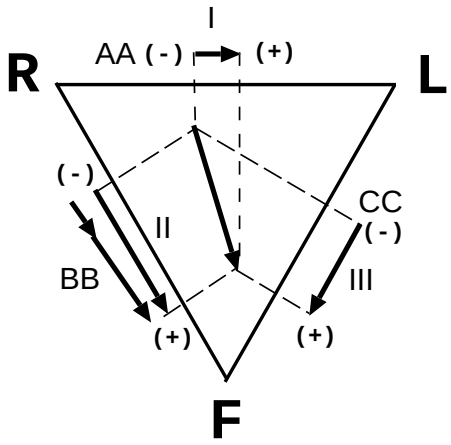

図19 アイントーベンの正三角形
右手，左手，左足に電極を貼り，体表心電図を計測すると，それらの電位は正三角形のベクトルに投影できる．

心電図を発見したアイントーベンは右手，左手，左足に電極を装着し，各電極間の電位計測を行ったところ，図19に示すように正三角形に投影できることを発見した．本来なら3点の計測であるからその起電力となる心電図は$f(x,y,z)$となるはずである．ところが2次平面の正三角形に投影できたことはヒトの体形に運が良かったとしか言いようのない出き事である．従って体表心電図の電気軸という概念が生まれた，また，このことから第II誘導は図に示すように第I誘導と第III誘導の絶対値の和で表される．したがって，この第II誘導心電図波形を先ほどトレースした心電図座標をもちいて再合成しようというのが次に示す関数secondである．

Ved3.pyをコピーしてVed4.pyを作成する．新たに関数secondを作動するボタンの作成，secondも結果をトップレベルで表示することに

して，第 I 誘導，第 II 誘導の絶対値は AA，CC より基線の Y 座標値を引けばよい．プログラムは，

```
# 第 II 誘導合成
# 第 II 誘導は第 I 誘導と
# 第 III 誘導の絶対値の和
def second():
  global AA,BB,CC,SdII
  global dI,dII,dIII,SdII
  global baseI_y,baseII_y,baseIII_y
  # キャンバスにカーソル作成
  c.c.delete('cursor')
  c.c.create_line(1,0,1,maxy,
    fill='green',tag='cursor')
  curx=0
  t2=tk.Toplevel()
  t2.title('II Image')
  w2 = tk.Canvas(t2,width=250,
    height=250,bg='white')
  w2.pack()
  dI=[]; dII=[]; dIII=[]; SdII=[]
  for i in range(maxx):
    # カーソル移動
    curdx=i-curx
    c.c.move('cursor',curdx,0)
    curx=i
    c.after(1)
    c.c.update()
    dI.append(baseI_y-AA[i])
    dII.append(baseI_y-BB[i])
    dIII.append(baseI_y-CC[i])
    SdII.append(-(dI[i]+dIII[i])
      -250)
    if i>0:
      w2.create_line(i-1,SdII[i-1],
        i,SdII[i],width=2)
```

第 II 誘導合成ボタンは，c=CA() の前に，

```
b3=tk.Button(f1,text=u'II 誘導合成',
  command=second)
b3.pack(side='left')
c=CA()
f.mainloop()
```

とすればよい．結果を図 20 に示す．第 II 誘導にかなり近似した波形が算出できている．

本プログラムの最終段階であるベクトル心電図の表示に進む．Ved4.py をコピーし Ved5.py にする．ベクトル心電図は 3 波形のうち 2 波形

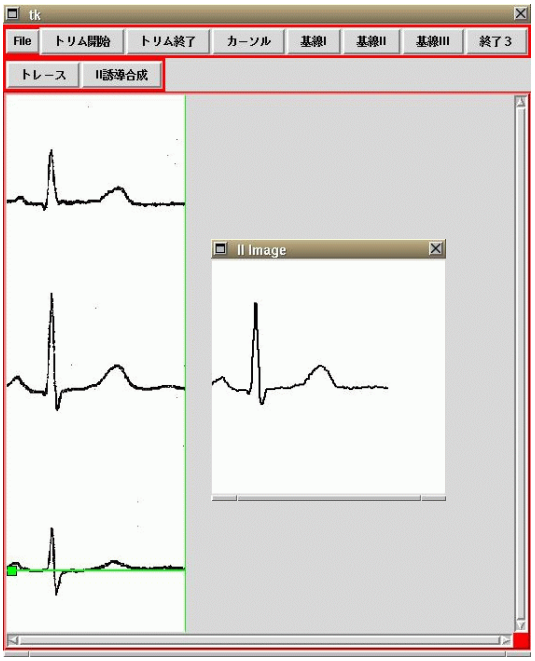

図 20 第 II 誘導の合成
第 I および第 III 誘導の絶対値を加算することにより第 II 誘導の波形が得られる．

の X，Y 方向のベクトルを抽出すれば良いが，その頂点のリサージュを描くことにする．前項で計算した絶対値 dI および dII をもちいて，以下に示す関数 axis を作成した．

```
# 電気軸
def axis():
  global dI,dII,dIII,SdII
  global dx,dy
  # キャンバスにカーソル作成
  c.c.delete('cursor')
  c.c.create_line(1,0,1,maxy,
    fill='green',tag='cursor')
  curx=0
  t3=tk.Toplevel()
  t3.title('Axis')
  w3 = tk.Canvas(t3,width=250,
    height=250,bg='white')
  w3.pack()
  dx=[]; dy=[]
  for i in range(maxx):
    # カーソル移動
    curdx=i-curx
    c.c.move('cursor',curdx,0)
```

```
        curx=i
        c.after(100)
        c.c.update()
        dx.append((dI[i]+0.5*dII[i])
          +150)
        dy.append((dII[i]*0.866)+250)
        if i>0:
          w3.create_line(dx[i-1],
            dy[i-1], dx[i],dy[i],
            width=2)
```

電気軸算出ボタンは以下のように追加する．

```
b3=tk.Button(f1,text=u'電気軸',
  command=axis)
b3.pack(side='left')
c=CA()
f.mainloop()
```

結果を図 21 に示す．以上が画像波形から X-Y 座標を取得する手段である．

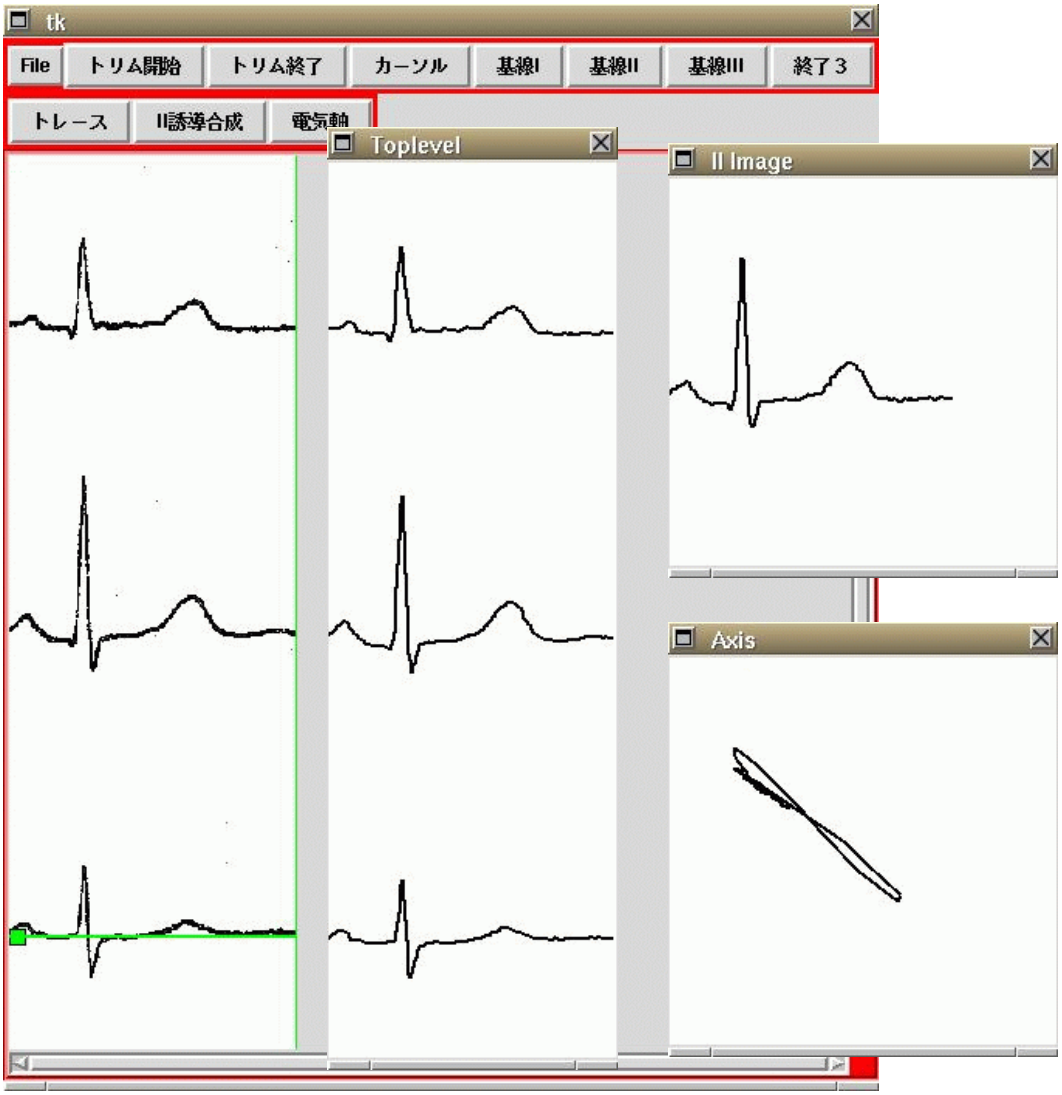

図 21 トレース，第 II 誘導の合成，電気軸リサージュ
心電図計測チャートの波形計測より，ベクトル心電図まで合成した．

ランチャー

ランチャー（launcher）とは発射装置という意味であるが，特にコンピュータソフトではディスプレーの隅に小さなボタンを並べて，日頃使用する電卓や電話番号帳などのソフトを起動させるボタン群を示す（図 22）．

通常の GUI プログラムはウィンドマネージャに依存，マウスクリックにより任意の位置に表示されるが，Python Tkinter の wm_命令を利用すると表示場所が指定できる．

ランチャープログラム (launcher.py)：

```
#!/usr/bin/env python
# -*- coding: euc-jp -*-
import Tkinter as tk
import os
def vi():
  os.system('kterm -e vi vicmds &')
def calc():
   os.system('calc.py &')
def phone():
  os.system('phone.py &')
# ランチャーメインプログラム
root=tk.Tk()
root.title('launcher')
root.wm_geometry('+0+0')
f=tk.Frame(root, bd=2,bg='red')
f.pack(side='left')
b = tk.Button(f,text='vicmds',
  command=vi)
b.pack(side='left')
b = tk.Button(f,text='calc',
  command=calc)
b.pack(side='left')
b = tk.Button(f,text='phone',
  command=phone)
b.pack(side='left')
f.mainloop()
```

プログラム解説：

GUI は 3 個のボタンを並べただけであるが，新しい命令に wm_geometry で Display の左上端に GUI を表示させている．この命令の駆動には Tk() クラスインスタンスである root を作成している．次に，ランチャーでは一般のシェル命令を Python 内より駆動させるため，os.system を利用した．エディター vi は端末の上で実行するプログラムだから，始めに Kterm(日本語端末) を表示させ，それに vi を駆動させる．vi 駆動時に特定のファイル vicmds を表示させるので -e オプションを使用している．また，&を用いて新たなセッションを作成し，多重処理を行わせている．

図 22 ランチャーボタンボックス
Display の左上端に表示され，左より vi 命令，電卓，電話帳がクリックで駆動する．

電卓プログラム (calc.py)：

電卓ソフトは通常の電卓の形式では一行しか表示されないため，結果を出力すると入力した内容が消されてしまう欠点がある．そのため，Entry ウィジェットを 2 段にして，入力用と出力用に分離している．また，行内の編集もでき，数学関数も組み入れている．

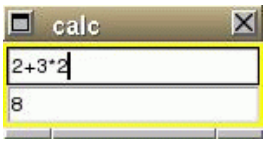

図 23 電卓 GUI
上段に式を入力し，下段に結果を示す．式は確認やカーソルを移動して編集ができる．

calc.py

```
#!/usr/bin/env python
# -*- coding: euc-jp -*-
import Tkinter as tk
import os
from math import *
def cmd(event): # エンターコールバック
   data=eval(e1.get())
   e2.delete(0,'end')
   e2.insert(tk.END,data)
# 電卓メインプログラム
```

```
root=tk.Tk()
root.title('calc')
f=tk.Frame(root, bd=2,bg='yellow')
f.pack()
e1 = tk.Entry(f)
e1.pack()
# エンターバインド
e1.bind("<Return>",cmd)
e2 = tk.Entry(f)
e2.pack()
f.mainloop()
```

プログラム解説：

Text ウィジェットは複数行の文字入出力を行うが，Entry は 1 行入力・出力表示のテキストウィジェットである．メソッドの insert や delete 機能はテキストウィジェットと同じ動作をする．入力終了は改行キー <Return> イベントを利用し，cmd 関数を呼び出す．そこでは，e1 エントリーのデータを読み取り，式を評価し，e2 の表示している内容を消去してから計算結果を表示させている．

電話帳プログラム (phone.py)：

電話帳はデータベースになり得るから pickle か shelve のよい例と思ったが，データ入力の問題，同姓の場合の処理法，ソーティングの方法などを考えると解決しなければならない多くの問題を抱えることになる．

一方データの再利用の利便性考えると，text データに優るファイル形式はない．それであれば，grep で検索，vi による編集，uniq，sort，cut，paste などの unix 本来の命令を利用することができる．したがって，電話帳の GUI は 1 行エントリーとスクロールドテキストで構成することにした．

phone.py

```
#!/usr/bin/env python
# -*- coding: euc-jp -*-
import Tkinter as tk
import commands
from TS import *
from math import *
def cmd(event):
   global data,dbta,a
   data=e1.get()
   dbta='grep '+data+
     ' ./nenga/n04.txt'
   a=commands.getstatusoutput(
     dbta.encode('euc_jp'))
   t.t.delete(1.0,tk.END)
   t.t.insert(
     1.0,a[1].decode('euc_jp'))
# 電話帳メインプログラム
root=tk.Tk()
root.title('phone')
f=tk.Frame()
f.pack()
l1=tk.Label(f,text=u'名前入力')
l1.pack(side='left')
e1 = tk.Entry(f)
e1.pack(side='left')
e1.bind("<Return>",cmd)
t=TS()
```

図 24 電話帳 GUI
上段に名前入力欄，下段はスクロールドテキストクラス TS() を利用している．

プログラム解説：

メインプログラムを見ると，フレームを作成し，その中にラベルとエントリーを横向けに配置，それとは別に，スクロールドテキスト TS() を立てに配置している．エントリーの改行イベントで cmd 関数が駆動する．

cmd 関数では grep で検索を行う．Python 内より unix 命令を呼び出して結果を取得する方法は，

commands モジュールの getstatusoutput 命令を使う．それは 1 つの引数で文字列を要求する．したがって，grep そのものを文字列に変換する．電話帳ファイルは’./nenga/n04.tx’であり，euc コードが検索文字列に入るため，encode させる．

検索結果は変数 a に euc コードで記録される．その文字列をスクロールドテキストに表示させる．まずテキスト内容を削除し，文字列を decode して insert 命令を使用する．

以上，ランチャーを含めて 4 個の短いプログラムを紹介したが，30 行足らずのプログラムでこれほどの性能を引き出すプログラムは他に類を見ない．「長いプログラムは寿命が短く，短いプログラムは寿命が永い」というのが筆者の経験法則である．

終りに際して

筆者は日立製作所のわずか 8K byte しかメモリーを装備していない HITAC-10 による Assembler，Fortran の時代，次いで C 言語による X11R4 の時代，そして現在へと CPU プログラミングを行ってきたが，最も効率のよい言語は Tcl/Tk と思っていたし，今でもそのように思っている．しかし，プログラムが少々長くなると，変数扱いが突然難しくなり，数日も達つと何の変数であたったかも忘れ，プログラムの再利用や改良も困難になる．さらに，Tcl/Tk は jpg や png の画像が処理できないなどの欠点を克服するために Python+PIL の組合せを利用するに至った．

Python を使用しての感想は，クラスの作成，そしてメソッドの継承の素晴らしさに感動した．長い一連のプログラムではなく，短いプログラムを重ねていく作業で，システムとして成長していくことが理解できた．変数の問題も局所関数として遮蔽することができ，また別の関数で引用されることがなければ同じ変数を多用することもできる．これらの性質を利用することにより，初めて自作のプログラムがリソースとして再利用できる楽しみを知った次第である．

オブジェクト指向として C++や C#などの言語，JAVA などを否定するわけではないが，今までの経験から Python は開発効率のよい一級のプログラム言語だと思う．

本稿は Python について基本的な言語解説書ではない．それについては多くの解説書が出版されているので参考にしてほしい．むしろ，一通りの知識を知った上で，どのようにプログラミングに利用していくかという点を強調してきたつもりである．また，本項で紹介したプログラムは文献 4 に公開しているから自由にコピーして次のステップへと進んでほしい．最後に応用編として，心電図チャートよりベクトル心電図のリサージュ波形の構築に至った経緯は 2009 年日本麻酔・集中治療テクノロジー学会で Tcl/Tk を用いて発表したプログラムの焼き直しであることを付け加える．

参考書など

1．アラン・ゴールド著，松葉素子訳：Python で学ぶプログラム作法．ピアソン・エデュケーション．2001．
2．Mark Lutz，David Ascher 著，世紀太 章訳：始めての Python． O’ Reilly，2000．
3．久野 晴：入門 tcl/tk． アスキー出版局．1997．
4．http://kansai.anesth.or.jp/gijutu/python/howto-python.php

索 引