Python3 入門

Kivy による GUI アプリケーション開発,
サウンド入出力,
ウェブスクレイピング

第 0.84.1 版

Copyright © 2017, Katsunori Nakamura

中村勝則

2017 年 11 月 8 日

目 次

1　はじめに

Python はオランダのプログラマであるグイド・ヴァンロッサム（Guido van Rossum）によって 1991 年に開発されたプログラミング言語であり，言語処理系は基本的にインタプリタである．Python は多目的の高水準言語であり，言語そのものの習得とアプリケーション開発に要する労力が比較的に少ないとされる．しかし，実用的なアプリケーションを開発するために必要とされる多くの機能が提供されており，この言語の有用性の評価が高まっている．

Python の言語処理系（インタプリタ）は多くのオペレーティングシステム（OS）に向けて用意されている．Python で記述されたプログラムはインタプリタ上で実行されるために，C や Java で記述されたプログラムを実行する場合と比較すると実行の速度は遅いが，C 言語など他の言語で開発されたプログラムとの連携が容易である．このため，Python に組み込んで使用するための各種の高速なプログラムが**モジュール**という形で多数提供されており，それらを利用することができる．Python 用のモジュールとして利用できるものは幅広く，言語そのものの習得や運用の簡便性と相俟って，情報工学や情報科学とは縁の遠い分野の利用者に対してもアプリケーション開発の敷居を下げている．

本書で前提とする Python の版は 3.6 である．

1.1　Python でできること

Python は汎用のプログラミング言語である．また，世界中の開発コミュニティから多数のモジュールが提供されている．このため，Python が利用できる分野は広く，特に科学，工学系分野のためのモジュールが豊富である．それらモジュールを利用することで，情報処理技術を専門としない領域の利用者も手軽に独自の情報処理を実現することができる．特に次に挙げるような処理を実現する上で有用なモジュールが揃っている．

- 機械学習
- ニューラルネット
- データサイエンス
- Web アプリケーション開発

これに加えて，グラフィックス（GUI 含む）を作成するためのモジュールも豊富であり，デスクトップ PC や携帯情報端末のためのアプリケーションプログラムの開発においても簡便な手段を与える．

1.2　本書の内容

本書は Python でアプリケーションプログラムを開発するために必要な最小限の事柄について述べる．特に，Python 以外の言語でプログラミングやスクリプティングの基本を学んだ人が読者として望ましい．本書の内容を概略として列挙すると次のようなものとなる．

- Python 処理系の導入の方法
  インタプリタとモジュールのインストール方法など
- Python の言語としての基本事項
  文法など
- 実用的なアプリケーションを作成するために必要なこと
  通信やマルチスレッド，マルチプロセスプログラミングに関することなど
- GUI アプリケーションを作成するために必要なこと
  GUI ライブラリの基本的な扱い方など
- サウンドの基本的な扱い
  WAV ファイルの入出力や音声のサンプリングと再生

1.3 処理系の導入（インストール）と起動の方法

Python の処理系（インタプリタ）は，インターネットサイト https://www.python.org/ から入手することができる．Microsoft 社の Windows や Apple 社の Mac [1] 用のインストーラが用意されており，それらをダウンロードしてインストールする [2] ことで Python 処理系が使用可能となる．Linux をはじめとする UNIX 系 OS においては，OS のパッケージ管理機能を介して Python をインストールできる場合が多く，あるいは先のサイトから Python のソースコードを入手して処理系をビルドすることもできる．

基本的に Python は OS のコマンドを投入することで起動する．Windows ではコマンドプロンプトウィンドウから，Linux や Mac ではターミナルウィンドウから `python` コマンド（あるいは `py` コマンド）を投入 [3] することで次の例のように Python インタプリタが起動する．

例. Windows のコマンドプロンプトから Python を起動する例

```
C:￥Users￥katsu> py   Enter        ← py コマンドの投入
Python 3.6.1 (v3.6.1:69c0db5, Mar 21 2017, 18:41:36) [MSC v.1900 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>        ← Python コマンドの入力待ち
```

この例の最後の行で `>>>` と表示されているが，これに続いて Python の文を入力することができる．処理系を終了するには `exit()` と入力する．

例. Python の終了

```
>>> exit()   Enter      ← Python 終了
C:￥Users￥katsu>          ← Windows のコマンドプロンプトに戻る
```

1.4 使用する GUI ライブラリ

グラフィカルユーザインターフェース（GUI）を備えたアプリケーションプログラムを構築するには GUI を実現するためのプログラムライブラリを入手して用いる必要がある．GUI のためのライブラリには様々なものがあり，Python 処理系に標準的に提供されている Tkinter をはじめ，GNOME Foundation が開発した GTK+，Qt Development Frameworks が開発した Qt，Julian Smart 氏（英）が開発した wxWidgets，Kivy Organization が開発した Kivy など多くのものが存在している．具体的には，それら GUI のライブラリを Python から使用するために必要となるモジュールを導入し，Python のプログラムから GUI ライブラリを呼び出すための API を介して GUI を構築する．

本書では，比較的新しく開発され，携帯情報端末（スマートフォンやタブレットなど）でも動作する GUI アプリケーションを構築することができる Kivy を取り上げる．

1.5 Python に関する詳しい情報

本書で説明していない情報に関しては，参考となる情報源（インターネットサイトなど）を巻末の付録に挙げる．

2 Python の基礎

他のプログラミング言語の場合と同様に，Python でも**変数**，**制御構造**，**入出力**の扱いを基本とする．Python の文はインタプリタのプロンプトに直接与えることもできるが，通常はプログラムはテキストファイルの形で作成してそれを Python 処理系に与えて実行する．

---

[1] Apple 社の Macintosh 用の OS の呼称は「Mac OS X」,「OS X」,「macOS」と変化している．本書ではこの内「OS X」,「macOS」を暗に前提としている．

[2] 巻末付録「A.2 Python のインストール作業の例」でインストール方法を概略的に紹介している．

[3] Apple 社 の Mac では Python 2.7 が基本的にインストールされており，`python` コマンドを投入するとこの版が起動する．本書で前提とする Python3 をインストールして起動する場合は `python3` コマンドを投入する．詳細に関しては利用する環境の導入状態を調べること．

2.1 テキストファイルに作成したプログラムの実行

例えば次のようなプログラムが test01.py というファイル名のテキストデータとして作成されていたとする．

プログラム：test01.py

```
1 # coding: utf-8
2
3 print( "私の名前はPythonです．" )
```

これを実行するには OS のターミナルウィンドウから，

`python test01.py` Enter

と入力するか，あるいは，

`py test01.py` Enter

と入力する．するとターミナルウィンドウに「私の名前は Python です.」と表示される．

このプログラム例 test01.py の中に print で始まる行がある．これは **print 関数** というもので端末画面への出力の際に使用する関数であり，今後頻繁に使用する．

＜ print 関数＞
書き方：　print(　出力内容　)

注意）　print は関数なので [4] 出力内容を必ず括弧 '(...)' で括る必要がある．

2.1.1 プログラム中に記述するコメント

「#」で始まる行は**コメント**であり，Python の文とは解釈されずに無視される．ただし，先のプログラムの冒頭にある

```
# coding: utf-8
```

は，プログラムを記述するための文字コードを指定するものであり，コメント行が意味を成す特別な例である．文字コードとしては utf-8 が一般的であるが，この他にも表 1 に示すような文字コードを使用してプログラムを記述することができる．

表 1: 指定できる文字コード

文字コード	coding:に指定するもの	文字コード	coding:に指定するもの
UTF-8	utf-8	EUC	euc-jp
シフト JIS	shift-jis, cp932*	JIS	iso2022-jp

* **マイクロソフト標準キャラクタセット**に基づく指定

coding:に指定するものは大文字／小文字の区別はなく，ハイフン '-' とアンダースコア '_' のどちらを使用してもよい．

2.1.2 プログラムのインデント

Python ではプログラムを記述する際のインデント（先頭の空白）に特別な意味があり，不必要なインデントをしてはならない．例えば次のような複数の行からなるプログラム test02.py は正常に動作するが，不必要なインデントを施したプログラム test02-2.py は実行時にエラーとなる．

[4] Python2 までは print は文であった事情から括弧は必要なかったが，Python3 からは仕様が変わり，print は関数となった．このため括弧が必須となる．

正しいプログラム：test02.py

```
1 # coding: utf-8
2
3 print( "1.私の名前はPythonです. " )
4 print( "2.私の名前はPythonです. " )
5 print( "3.私の名前はPythonです. " )
```

間違ったプログラム：test02-2.py

```
1 # coding: utf-8
2
3 print( "1.私の名前はPythonです. " )
4     print( "2.私の名前はPythonです. " )
5 print( "3.私の名前はPythonです. " )
```

test02-2.py を実行すると次のようになる.

```
File "test02-2.py", line 4
  print( "2. 私の名前は Python です. " )
  ^
IndentationError:  unexpected indent
```

インデントが持つ意味については「2.4 制御構造」のところで解説する.

2.2 変数とデータの型

変数はデータを保持するものであり，プログラミング言語の最も基本的な要素である．また，変数に格納する値には数値（整数，浮動小数点数）や文字コード列といった様々な**型**がある．C 言語や Java では，使用する変数はその使用に先立って明に宣言する必要がある．また，変数を宣言する際に格納するデータの型を指定しなければならず，宣言した型以外のデータをその変数に格納することはできない．この様子を指して「C 言語や Java は**強い型付け**の言語処理系である」と表現する．これに対して Python は**動的な型付け**の言語処理系であり，1 つの変数に格納できる値の型に制約は無い．すなわち，ある変数に値を設定した後で，別の型の値でその変数の内容を上書きすることができる．また Python では，変数の使用に先立って変数の確保を明に宣言する必要は無い．例えば次のような例について考える.

```
>>> x = 2   Enter        変数 x に整数の 2 を格納
>>> y = 3   Enter        変数 x に整数の 3 を格納
>>> z = x + y   Enter    x と y の値を加算したものを変数 z に格納
>>> print(z)   Enter     変数 z の内容を出力する処理
5                        出力された内容
```

これは Python の処理系を起動して変数 x,y に整数の値を設定し，それらの加算結果を表示している例である．変数への値の設定はイコール記号 '=' を使う．引き続いて次のように Python の文を与える.

```
>>> x = "2"   Enter      変数 x に文字記号"2"を格納
>>> y = "3"   Enter      変数 y に文字記号"3"を格納
>>> z = x + y   Enter    x と y の値を連結したものを変数 z に格納
>>> print(z)   Enter     変数 z の内容を出力する処理
23                       出力された内容
```

この例は x,y に格納された文字記号を，加算ではなく連結するものであり，同じ変数に異なる型の値が上書きされていることがわかる.

2.2.1 数値

Python では整数，浮動小数点数，複素数を数として扱う．また，数に対する算術演算の表記を表 2 に示す.

表 2: 算術演算

表記	意味	表記	意味
$a+b$	a と b の加算	$a-b$	減算
$a*b$	a と b の乗算	a/b	除算
$a//b$	除算（小数点以下切り捨て）	$a\%b$	剰余（a を b で割った余り）
$a**b$	冪乗（a の b 乗　a^b）		

この他にも再帰代入の演算子が使える．例えば，

```
a += b    a = a + b と同等
a -= b    a = a - b と同等
a *= b    a = a * b と同等
a /= b    a = a / b と同等
a %= b    a = a % b と同等
a **= b   a = a ** b と同等
```

といった演算である．

整数としては長い桁の値が扱える．(次の例)

例. 2^{1000} を計算する

```
>>> 2**1000   Enter
```

```
10715086071862673209484250490600018105614048117055336074437503883703510511249361224931983788156958581275946729175531468251871452856923140435984577574698574803934567774824230985421074605062371141877954182153046474983581941267398767559165543946077062914571196477686542167660429831652624386837205668069376
```

浮動小数点数は IEEE-754 で定められる倍精度（double）で表現される．

補足） Python インタプリタのプロンプトに対して直接に計算式や値を入力して Enter を押すと，その式の計算結果や値がそのまま出力される．

例. $1.00000000001^{100000000000}$ を求める

```
>>> 1.00000000001 ** 100000000000   Enter
2.71828205335711
```

※ 高い精度の（仮数部の桁数の大きい）浮動小数点数を扱う方法については後の「2.2.1.2 多倍長精度の浮動小数点数の扱い」を参照のこと．

【基数の指定】 2 進数，8 進数，16 進数

数を表現する際に表 3 に示すような接頭辞を付けることで 2 進法,8 進法, 16 進法の表現ができる．

表 3: 基数（n 進）の接頭辞

接頭辞	表現	例	
0b	2 進法	0b1011011	$(=91_{10})$
0o	8 進法	0o1234567	$(=342391_{10})$
0x	16 進法	0x7b4f	$(=31567_{10})$

【複素数】

Python では複素数が扱える．この際，虚数単位は j で現す．

例. $(1+i)(1-i)$ の計算を Python で実行する

```
>>> (1+1j)*(1-1j)   Enter
(2+0j)
```

このように虚部（j の係数）を明に記述する必要がある．すなわち，虚部が 1 であっても 1 を明に記述しなければならない．また計算の結果，虚部が 0 になっても 0j と表記される．

複素数の**実部**と**虚部**は，プロパティ real と imag に保持されており，取り出すことができる．また，複素数に対して conjugate メソッドを使用すると共役複素数が得られる．

例. 共役複素数を求める

```
>>> c = 1-3j   Enter      ←複素数の生成
>>> c.conjugate()   Enter      ←共役複素数の算出
(1+3j)      ←共役複素数が得られている
```

2.2.1.1 数学関数

次のようにして math モジュールを読み込むことで，各種の数学関数が使用できる．

```
>>> import math   Enter
```

各種関数は math クラスのメソッドとして呼び出す．例えば正弦関数の値を求めるには次のようにする，

```
>>> math.sin(math.pi/2)   Enter        ← sin(π/2) の算出
1.0            ←計算結果
```

表 4 に math クラスで使える数学関数の一部を挙げる．

表 4: 使用できる数学関数の一部

関数	説明	関数	説明
sqrt(x)	x の平方根　$\sqrt{x}$	pow(x,y)	x の y 乗　x^y
exp(x)	x の指数関数　e^x	log(x,y)	x の対数関数　$\log_y x$　※
log2(x)	x の対数関数　$\log_2 x$	log10(x)	x の対数関数　$\log_{10} x$
sin(x)	x の正弦関数　$\sin(x)$	cos(x)	x の余弦関数　$\cos(x)$
tan(x)	x の正接関数　$\tan(x)$	asin(x)	x の逆正弦関数　$\sin^{-1}(x)$
acos(x)	x の逆余弦関数　$\cos^{-1}(x)$	atan(x)	x の逆正接関数　$\tan^{-1}(x)$
sinh(x)	x の双曲線正弦関数　$\sinh(x)$	cosh(x)	x の双曲線余弦関数　$\cosh(x)$
tanh(x)	x の双曲線正接関数　$\tanh(x)$	asinh(x)	x の逆双曲線正弦関数　$\sinh^{-1}(x)$
acosh(x)	x の逆双曲線余弦関数　$\cosh^{-1}(x)$	atanh(x)	x の逆双曲線正接関数　$\tanh^{-1}(x)$
pi	円周率　π	e	ネイピア数　e

※ y を省略した場合は自然対数

2.2.1.2 多倍長精度の浮動小数点数の扱い

http://mpmath.org/ で公開されている mpmath パッケージ[5] を使用すると，IEEE-754 の倍精度浮動小数点数の精度に制限されない**任意の精度**による浮動小数点数の演算が可能となる．mpmath を使用するには必要なモジュールを次のようにして読み込んでおく．

```
from mpmath import mp
```

mpmath パッケージには math モジュールで提供されているものと類似の関数が多数提供されており，表 4 のような関数に接頭辞 'mp.' を付けることでその関数の値を求めることができることが多い．ただし関数名などの違いには注意すること．mpmath の使用方法の詳細に関しては先のサイトを参照のこと．

[5]導入方法に関しては巻末付録「A.3 PIP によるモジュール管理」を参照のこと．

■ **演算精度の設定**（仮数部の桁数の設定）

mp モジュールの dps プロパティに整数値を設定することで，計算精度（仮数部の桁数）を設定することができる．

例. 円周率を 200 桁求める

```
>>> from mpmath import mp   Enter      ← mpmath パッケージの読込み
>>> mp.dps = 200   Enter              ←演算精度の設定
>>> print( mp.pi )   Enter            ←円周率の算出
3.14159265358979323846264338327950288419716939937510582097494459230781640628620899862803
48253421170679821480865132823066470938446095505822317253594081284811174502841027019385
21105559644622948954930382           ←計算結果
```

注）mpmath パッケージの関数が返す値の型は，通常の浮動小数点数 float とは異なる型（mpmath.ctx_mp_python.mpf）であることに注意すること．

■ **値の設定**

mpmath のオブジェクトに値を設定するには mpf メソッドを使用する．

例. 値の設定

```
>>> from mpmath import mp   Enter      ← mpmath パッケージの読込み
>>> mp.dps = 72   Enter               ←演算精度の設定
>>> a = mp.mpf('0.0000000001')   Enter     ← a に 10^-10 の値を設定
>>> b = mp.mpf('1.0e10')   Enter           ← b に 10^10 の値を設定
>>> a*b   Enter                       ←積の算出
mpf('1.0')                           ←計算結果
```

mpf メソッドの引数には様々な型のオブジェクト（int，float，str）を与えることができる．特に仮数部の桁の長い浮動小数点数を与える場合は，文字列（str）として与えるすることができる．文字列として浮動小数点数を記述する場合は先の例の様に**指数表記**が使える．

mpmath オブジェクト a の値を int，float，str などの型の値に変換するには次のようにする．

整数に変換	:	int(a)
浮動小数点数に変換	:	float(a)
文字列に変換	:	str(a)

2.2.1.3　乱数の生成

乱数を生成するには random モジュールを使用する方法がある．このモジュールを使用するには次のようにする．

```
import random
```

この後，randrange メソッドを使用して乱数を得る．randrange に 2 つの引数を与えて実行すると，**第 1 引数以上, 第 2 引数未満**の整数の乱数が得られる．

例. 整数の乱数の発生

```
>>> import random   Enter      ←モジュールの読み込み
>>> random.randrange(0,100)   Enter     ← 0 以上 100 未満の整数の乱数の発生
16          ←得られた乱数
```

randrange はメルセンヌ・ツイスタ [6] を用いたものであり，乱数生成が確定的である．従って暗号学的な処理においてこの方法を使用してはならない．より安全な乱数生成には secrets モジュールを使用するべきである．secrets モジュールを使用するためには次のようにする．

[6]乱数生成器の 1 つ．参考文献：M.Matsumoto, T.Nishimura, "Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator", ACM Trans. on Modeling and Computer Simulation Vol.8, No.1, January pp.3-30 (1998)

```
import secrets
```

この後，randbelow メソッドを使用して乱数を得る．このメソッドの引数に 1 つの正の整数与えると，**0 以上, 与えた引数未満**の整数の乱数を生成する．

例. 整数の乱数の発生

```
>>> import secrets   Enter      ←モジュールの読み込み
>>> secrets.randbelow(100)   Enter      ← 0 以上 100 未満の整数の乱数の発生
 52          ←得られた乱数
```

2.2.2 文字列

文字列はダブルクオート「”」もしくはシングルクオート「’」で括ったデータである．(どちらを使っても良い[7])

文字列の例. ’Python’，”パイソン”

Python では C 言語と同様に，文字列中には特殊な機能を持った記号（エスケープシーケンス）を含めることができる．エスケープ記号は「￥」である．

【エスケープシーケンス】

一般的な意味では「文字列」は可読な記号列のことを指すが，記号列の中に特別な働きを持った，ある種の「制御記号」を含めると，その文字列を端末画面などに表示する際の制御ができる．例えば,「￥n」というエスケープシーケンスを含んだ文字列

”一行目の内容です．￥n 二行目の内容です．”

を端末画面に表示してみる．

```
>>> a = ”一行目の内容です．￥n 二行目の内容です．”   Enter
>>> print( a )   Enter
一行目の内容です.
二行目の内容です.
```

このように,「￥n」の位置で文字列の表示が改行されていることがわかる．この例における print は端末装置に出力する（画面に表示する）ための機能であるが，詳しくは以後の章で解説する．

表示に使用する端末装置によっては，ベルを鳴らすことも可能である．例えば次のように「￥a」というエスケープシーケンスを含む文字列を print する．

```
>>> a = "Ring the bell.￥a"   Enter
>>> print( a )   Enter
Ring the bell.
```

この表示と同時にベル音が鳴る．(装置によって音は違う．音が鳴らない端末もある)

エスケープシーケンスには様々なものがあり，代表的なものを表 5 に挙げる．

■ raw 文字列（raw string）について

文字列中のエスケープシーケンスを無視して ’￥’ を単なる文字として扱うには **raw 文字列**とすると良い．通常の文字列の先頭に ’r’ もしくは ’R’ を付けるとその文字列に含まれる記号は全て「そのままの記号」として扱われる．

[7] Python 処理系はシングルクオートを付けて文字列を表示する．

表 5: 代表的なエスケープシーケンス

ESC	機能	ESC	機能
¥n	改行	¥t	タブ
¥r	行頭にカーソルを復帰	¥v	垂直タブ
¥f	フォームフィード	¥b	バックスペース
¥a	ベル	¥¥	'¥' そのもの
¥"	ダブルクオート文字	¥'	シングルクオート文字

例. raw 文字列

```
>>> s = r'abc¥ndef'  Enter        ← raw 文字列の生成
>>> print(s)  Enter              ←出力してみる
abc¥ndef                ←'¥n' で改行されず，そのままの文字として扱われている
```

2.2.2.1　文字列の分解と合成

文字列の指定した部分を取り出すには '[...]' で添字（**スライス**）を付ける．例えば，'abcdef' という文字列があった場合，'abcdef'[2] は 'c' である．このように位置を意味する添字を付けることで，文字列の部分を取り出すことができる．先頭の文字の添字は 0 である．(開始は「0 番目」)

＜部分文字列の取り出し＞

書き方 (1)：　文字列 $[n_1 : n_2]$

「文字列」の n_1 番目から $n_2 - 1$ 番目までの部分を取り出す．

書き方 (2)：　文字列 $[n]$

「文字列」の n 番目の 1 文字を取り出す．

※ 添字（スライス）の更に高度な応用については「2.3.5 添字（スライス）の高度な応用」を参照のこと．

例. 部分文字列の取得

```
>>> s = 'abcdefg'  Enter      ←文字列の生成
>>> s[2:5]  Enter             2 番目から 5-1 番目までの部分文字列の取得
'cde'        ←取り出した部分文字列
```

例. 1 文字の取り出し

```
>>> s = '美味しい食べ物'  Enter      ←文字列の生成
>>> s[1]  Enter              1 番目の文字の取得
'味'         ←取り出した部分文字列
```

文字列の連結には，'+' による加算表現が使える．

例. 文字列の連結 (1)

```
>>> 'abc' + 'def'  Enter        ←加算記号で連結
'abcdef'                        ←連結結果
```

更に，再帰代入 '+=' を使って次々と累積的に連結してゆくこともできる．

例. 文字列の連結 (2)

```
>>> s = 'abc'  Enter      ←文字列の生成
>>> s += 'def'  Enter      ←'def' を追加
>>> s += 'ghi'  Enter      ←'ghi' を追加
>>> s  Enter      ←内容確認
'abcdefghi'        ←最終的な内容
```

積の演算 '*' を使うと文字列の繰り返しを生成できる.

例. 文字列の繰り返し

```
>>> 'abc' * 10  Enter              ← 'abc' を 10 回繰り返す
'abcabcabcabcabcabcabcabcabcabc'        ←生成結果
```

split メソッドを使用すると，特定の文字（列）を区切りとして文字列を分解することができる.

例. 文字列の分解

```
>>> s = 'ab,cd,ef,gh'  Enter      ←文字列の生成
>>> s.split(',')  Enter        ←コンマ ',' を境（区切り）にして文字列を分解
['ab', 'cd', 'ef', 'gh']        ←分解されたリスト
```

分解されたものが**リスト**の形で得られている．リストに関しては「2.3.1 リスト」のところで説明する.

文字列の連結 (3)

join メソッドを使用すると，リストの要素を全て連結することができる.（split メソッドの逆の処理）
ただしその場合のリストの要素は全て文字列型でなければならない.

例. リストの連結（split の逆）

```
>>> lst = ['a','b','c','d']  Enter      ←リストの生成
>>> ':'.join(lst)  Enter        ←コロン ':' を境（区切り）にして文字列を連結
'a:b:c:d'        ←リストを連結した文字列
```

join メソッドは区切り文字となる文字列型データに対して実行し，その引数に連結対象要素をもつリストを与える.

2.2.2.2 文字列の置換

文字列中の特定の部分を置き換えるには replace メソッドを使用する.

＜文字列の置換＞
書き方： 文字列.replace(**対象部分, 置換後の文字列**)
「文字列」の中の「対象部分」を探し，それを「置換後の文字列」に置き換えたものを返す．元の文字列は変更されない.

例. 文字列の置換

```
>>> a = 'abcdefgabcdefgabcdefg'  Enter      ←元の文字列
>>> a.replace('bcd','BCD')  Enter      ← 'bcd' を 'BCD' に置き換える
'aBCDefgaBCDefgaBCDefg'                ←置換結果
```

2.2.2.3 文字コード

1 文字（文字列ではない）の**文字コード**を取得するには ord 関数を使用する．また逆に，文字コードを表す整数値から文字を取得するには chr 関数を使用する.

例. 文字コードの取得／指定した文字コードの文字

```
>>> ord('a')  Enter      ←'a'（半角文字）の文字コードの取得
97      ←得られた文字コード（ASCII コード）
>>> ord('あ')  Enter      ←'あ'（全角文字）の文字コードの取得
12354      ←得られた文字コード（utf-8 の場合）
>>> chr(97)  Enter      ←文字コード 97（10 進数）に対応する文字の取得
'a'      ←得られた文字
>>> chr(12354)  Enter      ←文字コード 12354（10 進数）に対応する文字（utf-8）の取得
'あ'      ←得られた文字
```

2.2.3 真理値

真理値は真か偽かを現す値で，True, False の 2 値から成る．例えば，両辺の値が等しいかどうかを検査する記号 '==' があり，`1 == 2` という式は誤りなのでこの式の値は False となる．(次の例を参照)

例.

```
>>> p = 1==2  Enter      1==2 の検査結果を変数 p に与えている．
>>> print( p )  Enter      変数 p の内容を表示する．
False      変数 p の内容が False（偽）であることがわかる．
```

このように，値の比較などを検査する式は真理値を与えるものであり,「2.4 制御構造」のところで条件判定の方法として解説する．

2.2.4 型の変換

代表的な値の型として数値，文字列，真理値があるが，これら異なる型の間でデータを変換するには，

型 (値)

とする．型の変換処理の例を表 6 に挙げる．

表 6: 型の変換の例

変換元のデータ	整数へ	浮動小数点数へ	文字列へ	真理値へ
整数 浮動小数点数	- int(3.1) → 3	float(3) → 3.0 -	str(3) → '3' str(3.1) → '3.1'	bool(3) → True bool(3.1) → True 0 や 0.0 は False となる
文字列	int('2') → 2	float('2.1') → 2.1	-	bool('a') → True bool('True') → True bool('False') → True (注意!) 空文字列 ' ' は False となる
真理値	int(True) → 1 int(False) → 0	float(True) → 1.0 float(False) → 0.0	str(True) → 'True' str(False) → 'False'	- -

2.3 データ構造

2.3.1 リスト

複数のデータの並びは**リスト**で表す．例えば，0〜9 の数を順番に並べたものはリストとして

```
[0,1,2,3,4,5,6,7,8,9]
```

と表すことができる．このようにリストは '[' と ']' 括ったデータ構造である．リストの要素としては任意の型のデータを並べることができる．例えば，

```
['nakamura',51,'tanaka',22,'hashimoto',14]
```
のように，型の異なるデータが要素として混在するリストも作成可能である．また，リストを要素として持つリストも作成可能であり，例えば，
```
['a',['b','c'],'d']
```
といったリストも作成することができる．

■ リストの要素へのアクセス

リストの要素を取り出すには，取り出す要素の位置を示すインデックスを '[...]' で括って指定する．これは次の例を見ると理解できる．

```
>>> lst = [2,4,6,8,10]  Enter      リストの生成
>>> lst[1]  Enter                  1 番目の要素を指定
4                                  対象の要素が表示されている
```

このようにリストの要素は **0 番目から始まる位置**（インデックス）を指定してアクセスすることができる．
リストの中の，インデックスで指定した特定の範囲（部分リスト）を取り出すこともできる．

例. 部分リストの取り出し

```
>>> lst = [2,4,6,8,10]  Enter      リストの生成
>>> lst[1:4]  Enter                部分リスト（1〜4-1 番目）の取り出し
[4, 6, 8]                          部分リストが得られている
```

この例のように，コロン ':' でインデックスの範囲を指定するが，[n1:n2] と指定した場合は，n1〜(n2-1) の範囲の部分を示していることに注意しなければならない．

リストというデータ構造を用いることで，複数の要素を持つ複雑な構造を 1 つのオブジェクトとして扱うことができる．この意味で，リストは複雑な情報処理を実現するに当たって非常に有用なデータ構造であると言える．

■ リストの編集

リストは要素の追加や削除を始めとする編集ができるデータ構造である．リストを編集するための基本的な機能について解説する．

● 要素の追加

リストに対して要素を末尾に追加するには append メソッドを使用する．

例. リストへの要素の追加

```
>>> lst = ['x','y']  Enter      リストの作成
>>> lst.append('z')  Enter      要素の追加
>>> lst  Enter
['x', 'y', 'z']                 要素が追加されている
```

これは，リスト `lst` の末尾に要素`'z'` を追加している例である．

> **＜オブジェクトに対するメソッドの適用＞**
> Python ではオブジェクト指向の考え方に則って，
> **作用対象オブジェクト. メソッド**
> というドット '.' を用いた書き方で，メソッドをオブジェクトに対して適用する．

先の例ではリスト `lst` に対して，メソッド `append('z')` を適用している．

＜リストへの要素の追加＞
書き方：　対象リスト.append(追加する要素)
「対象リスト」そのものが変更される..

● 要素の挿入
リストの指定した位置に要素を挿入するには insert メソッドを使用する.

例. リストの指定した位置に要素を挿入

```
>>> lst = ['a','c','d']   Enter      リストの作成
>>> lst.insert(1,'b')   Enter        要素の挿入
>>> lst   Enter
['a', 'b', 'c', 'd']                 要素が挿入されている
```

これは，リスト `lst` の 1 番目に要素'b' を挿入している例である.

● リストの連結 (1)
演算子 '+' によって複数のリストを連結し，結果を新しいリストとして作成することができる.

例. リストの連結

```
>>> lst1 = ['a','b']   Enter          リストの作成 (1)
>>> lst2 = ['c','d']   Enter          リストの作成 (2)
>>> lst3 = ['e','f']   Enter          リストの作成 (3)
>>> lst4 = lst1 + lst2 + lst3   Enter  リストの連結
>>> lst4   Enter
['a', 'b', 'c', 'd', 'e', 'f']       3 つのリストが連結されている
```

'+' によるリストの連結処理において重要なことに，連結結果が**新たなリストとして生成されている**ということがある．すなわち，この例においてリスト `lst1,lst2,lst3` の内容は処理の前後で変更はなく，連結結果のリストが新たなリスト `lst4` として生成されている.

● リストの連結 (2)
'+' によるリストの連結処理とは別に，与えられたリストを編集する形でリストを連結することも可能である.

＜ extend メソッドによるリストの連結＞
書き方：　対象リスト.extend(追加リスト)
対象リストの末尾に追加リストを連結する．結果として対象リストそのものが拡張される.

例. リストの拡張

```
>>> lst = ['a','b','c']   Enter       リストの作成
>>> lst.extend(['d','e','f'])   Enter  リストの拡張
>>> lst   Enter          ←内容の確認
['a', 'b', 'c', 'd', 'e', 'f']   Enter  リストが拡張されている
```

● リストの要素の削除
リストの特定のインデックスの要素を削除するには del 文を使用する.

＜ del 文によるリストの要素の削除＞
書き方：　del　削除対象のオブジェクト
　削除対象のオブジェクトを削除する．

リストの 1 つの特定要素を削除するには次の例のようにする．

```
>>> lst = [2,4,6,8,10]  Enter      リストの作成
>>> del lst[1]  Enter      リストの 1 番目の要素の削除
>>> lst  Enter
[2, 6, 8, 10]      リストの要素が削除されている
```

同様の方法で，リストの中の指定した部分を削除することもできる．(次の例)

```
>>> lst = [2,4,6,8,10]  Enter      リストの作成
>>> del lst[1:4]  Enter      リストの 1〜(4-1) 番目の要素の削除
>>> lst  Enter
[2, 10]      指定した範囲の要素が削除されている
```

　リストから指定した値の要素を取り除くには remove メソッドを使用する．

＜ remove メソッドによる要素の削除＞
書き方：　削除対象リスト.remove(値)
　削除対象リストの中の指定した値の要素を削除する．削除対象の値の要素が複数存在する場合は，最初に見つかった要素を 1 つ削除する．

例. remove メソッドによる要素の削除

```
>>> lst = ['a','b','c','b','a']  Enter      リストの作成
>>> lst.remove('b')  Enter          'b' という値の要素を削除
>>> lst  Enter
['a', 'c', 'b', 'a']                最初の'b' が削除されている
```

　リストの特定のインデックスの要素を取り出してから削除するメソッド pop がある．

＜ pop メソッドによる要素の取り出しと削除＞
書き方：　対象リスト.pop(インデックス)
　対象リストの中の指定したインデックスの要素を取り出して削除する．このメソッドを実行すると，削除した要素を返す．また pop の引数を省略すると，末尾の要素が対象となる．

pop メソッドを用いると，スタックやキューといったデータ構造が簡単に実現できる．

例. pop メソッドによるスタック

```
>>> lst = []  Enter          空のリストを作成
>>> lst.append('a')  Enter    lst の末尾に要素'a' を追加
>>> lst.append('b')  Enter    lst の末尾に要素'b' を追加
>>> lst.append('c')  Enter    lst の末尾に要素'c' を追加
>>> lst  Enter                lst の内容を確認する
['a', 'b', 'c']               要素が蓄積されていることがわかる
>>> lst.pop()  Enter         lst の末尾の要素を取り出す
'c'                           lst の末尾の要素が得られた
>>> lst.pop()  Enter         lst の末尾の要素を取り出す
'b'                           lst の末尾の要素が得られた
>>> lst.pop()  Enter         lst の末尾の要素を取り出す
'a'                           lst の末尾の要素が得られた
>>> lst  Enter               lst のの内容を確認
[]                            空になっている
```

参考. 上の例で lst.pop(0) として実行するとキューが実現できる.

● リストの検査

リストに対する各種の検査方法について説明する.

・要素の存在検査

リストの中に，指定した要素が存在するかどうかを検査する in 演算子がある.

> **＜ in 演算子によるメンバシップ検査＞**
> **書き方： 要素 in リスト**
> リストの中に要素があれば True，なければ False を返す．要素が含まれないことを検査するには not in と記述する.

例. in によるメンバシップ検査

```
>>> lst = ['a','b','c','d']  Enter    リストの作成
>>> 'c' in lst  Enter                 'c' が lst に含まれるか
True                                  含まれる
>>> 'e' in lst  Enter                 'e' が lst に含まれるか
False                                 含まれない
>>> 'e' not in lst  Enter             'e' が lst に含まれないか
True                                  含まれない
```

・要素の位置の特定

リストの中に，指定した要素が存在する位置を求めるには index メソッドを用いる.

> **＜ index メソッドによる要素の探索＞**
> **書き方： 対象リスト.index(要素)**
> 対象リストの中の要素の位置（インデックス）を求める．対象リストの中に要素が含まれていなければエラーとなる.

例. index メソッドによる要素の探索

```
>>> lst = ['a','b','c','d']    Enter      リストの作成
>>> lst.index('c')    Enter               'c' の位置を求める
2                                         2 番目に存在する
```

探索する要素が対象リストに含まれていない場合，このメソッドの実行は次の例のようにエラーとなる．

例. 要素が含まれていない場合に起こるエラー

```
>>> lst.index('e')    Enter               'e' の位置を求める
Traceback (most recent call last):        以下，エラーメッセージ
  File "<stdin>", line 1, in <module>
ValueError:  'e' is not in list
```

これは ValueError という種類のエラーである．
Python では，エラーが発生する可能性のある処理を行う場合は，適切に**例外処理**の対策をしておくべきである．例えば次のようなサンプルプログラム test03.py のような形にすると良い．

プログラム：test03.py

```
1  # coding: utf-8
2
3  lst = ['a','b','c','d']
4
5  try:
6      n = lst.index('e')
7      print(n,"番目にあります．")
8  except ValueError:
9      print("要素が見つかりません…")
```

プログラムの 5 行目にある try は，エラー（例外）を発生する可能性のある処理を指定するための文である．すなわち 6,7 行目で例外が発生した場合は except 以下のプログラムに実行が移る．（処理系の動作は中断しない[8]）

2.3.1.1　例外処理

＜例外処理のための try と except ＞

書き方：　**try:**
（例外を起こす可能性のある処理）
except　エラー（例外）の種類:
（例外を起こした場合に実行する処理）
except:
（上記以外の例外を起こした場合に実行する処理）
finally:
（全ての except の処理の後に共通して実行する処理）

except は複数記述することができる．

先のプログラム test03.py を実行すると次のような結果となる．

```
py test03.py    Enter      OS のコマンドからプログラムを実行
要素が見つかりません…      結果の表示
```

● 要素の数のカウント

リストは同じ要素を複数持つことができるが，指定した要素（値）がリストにいくつ含まれるかをカウントするメソッド count がある．

[8]更に「4.6 エラーと例外の処理」では，システムが生成する例外やエラーに関するメッセージを文字列型データとして取得する方法を紹介する．

＜count メソッドによる要素のカウント＞
書き方：　対象リスト.count(要素)
　対象リストの中の要素の個数を求める．

例. count メソッドによる要素のカウント

```
>>> lst = ['a','b','a','c','a','d']  Enter      リストの作成
>>> lst.count('a')  Enter            'a' の個数を求める
3                                    3 個ある
>>> lst.count('e')  Enter            'e' の個数を求める
0                                    0 個（含まれない）
```

リストの長さ（全要素の個数）を求めるには len 関数を使用する．

＜リストの長さ＞
書き方：　len(対象リスト)
　対象リストの長さ（全要素の個数）を求める．

例. len 関数によるリストの長さの取得

```
>>> lst = [1,2,3,4,5]  Enter      リストの作成
>>> len(lst)  Enter               長さの算出
5                                  5 個の要素を持つ
```

● 要素の並べ替え
リストの要素を整列するには sort メソッドを使用する．

例. リストの要素の整列

```
>>> lst = [8,3,2,7,5,6,9,1,4]  Enter      リストの作成
>>> lst.sort()  Enter                      整列の実行
>>> lst  Enter                             内容確認
[1, 2, 3, 4, 5, 6, 7, 8, 9]                整列されている
```

リストの要素の並び順を逆にするには reverse メソッドを使用する．

例.（つづき）要素の順序の逆転

```
>>> lst.reverse()  Enter          逆転の実行
>>> lst  Enter                    内容確認
[9, 8, 7, 6, 5, 4, 3, 2, 1]       逆転されている
```

● リストの複製
既存のリストの複製には copy メソッドを使用する．リストなどのデータ構造に対するメソッドは，対象のデータそのものを改変するものが多い．そのようなメソッドの実行において，元のデータの内容を変更したくない場合は，元のデータの複製を作成し，その複製に対してメソッドを実行するのが良い．
　このメソッドは copy モジュールに含まれるものであり，copy メソッドは copy クラスの**クラスメソッド**（後述）である．従って，このメソッドを使用するには，予め copy モジュールをインポートしておく必要がある．

例. リストの複製

```
>>> import copy  Enter                    copy モジュールをインポート
>>> lst1 = [8,3,2,7,5,6,9,1,4]  Enter     リストの作成
>>> lst2 = copy.copy(lst1)  Enter         lst1 の複製を lst2 として作成
>>> lst2.sort()  Enter                    lst2 を整列
>>> lst2  Enter                           lst2 の内容を確認
[1, 2, 3, 4, 5, 6, 7, 8, 9]               整列されている
>>> lst1  Enter                           lst1 の内容を確認
[8, 3, 2, 7, 5, 6, 9, 1, 4]               元のままのリストである
```

2.3.2 タプル

タプルはリストとよく似ており，'(' と ')' で括った構造である．

タプルの例.

```
(0,1,2,3,4,5,6,7,8,9)
('nakamura',51,'tanaka',22,'hashimoto',14)
('a',('b','c'),'d')
```

次のように，リストとタプルは互いに要素として混在させることができる．

例.

```
['a',('b','c'),'d']
('a',['b','c'],'d')
```

【リストとタプルの違い】

リストとタプルは表記に用いる括弧の記号が異なるだけではない．リストは先に解説したように，要素の追加や削除など自在に編集ができるが，タプルは 1 度生成した後は変更ができない．例えば Python では変数への値の代入をする際に，

```
x,y = 2,3
```

という並列的な表記が許されている．実はこの例の両辺はタプルになっており，

```
(x,y) = (2,3)
```

という処理を行ったことと等しい．これは 1 つの例にすぎないが，データ構造の変更を伴わない処理の例であり，リストよりも若干ではあるが高速に処理ができるタプルの性質を利用している例であると言える．
この例の操作をした直後に x,y それぞれの変数の値を確認すると，

```
>>> x  Enter
2
>>> y  Enter
3
```

と設定されていることが確認できる．

リストに対する文やメソッドで，データ構造を変更しないものは概ねタプルに対しても使うことができる．

2.3.3 セット

セットは '{' と '}' で括ったデータ構造であり，集合論で扱う集合に近い性質を持っている．例えば，

```
>>> s = {4,1,3,2,1,3,4,2}  Enter
```

としてセットを生成した後に内容を確認すると，

```
>>> s   Enter
{1, 2, 3, 4}
```

となっていることがわかる．すなわち，セットでは要素の重複が許されず，要素に順序の概念がない．

● セットの生成

先の例のように要素の列を直接 '{' と '}' で括って記述してもよい．ただし，空セット（空集合）を生成する際は {} という記述をせずに

```
st = set()
```

のように set クラスのコンソトラクタを使用する．これは {} という表記が，後で述べる辞書型データの空辞書と区別できない事情があるためであり，このように明に set コンストラクタで生成することになる．

● 要素の追加と削除

・要素の追加

セットに要素を追加するには add メソッドを使用する．

例. セットへの要素の追加

```
>>> st = set()   Enter      空セットの生成
>>> st.add('a')   Enter      要素'a' の追加
>>> st   Enter      内容確認
{'a'}               要素が追加されている
```

・要素の削除

セットの要素を削除するには discard メソッドを使用する．例えばセット `st` から要素 `'a'` を削除するには

```
st.discard('a')
```

とする．リストの場合と同様に remove メソッドを使用することもできるが，削除対象の要素がセットの中にない場合にエラー KeyError が発生するので discard メソッドを使用する方が良い．

・全要素の削除

セットの全ての要素を削除して空セットにするには clear メソッドを使用する．例えばセット `st` の全ての要素を削除するには

```
st.clear()
```

とする．

● 集合論の操作

セットに対する操作と集合論における操作との対応を表 7 に挙げる．

● セットの複製

リストの場合と同様に，copy モジュールをインポートすることで copy クラスのクラスメソッド copy が使用できる．

■ frozenset

セットとよく似た frozenset というデータ構造もある．frozenset のオブジェクトは生成後に変更ができないが，ハッシュ化されるため，高速な処理に向いている．

表 7: Python のセットに対する操作と集合論の操作の対応

Python での表記	集合論における操作	戻り値のタイプ	意味
X in S	$X \in S$	真理値	要素 X は集合 S の要素である
X not in S	$X \notin S$	真理値	要素 X は集合 S の要素でない
S1.issubset(S2)	$S1 \subseteq S2$	真理値	集合 S1 は集合 S2 の部分集合である
S1.issuperset(S2)	$S2 \subseteq S1$	真理値	集合 S2 は集合 S1 の部分集合である
S1.intersection(S2)	$S1 \cap S2$	セット	集合 S1 と集合 S2 の共通集合を生成する
S1.intersection_update(S2)	$S1 \cap S2$	セット	集合 S1 と集合 S2 の共通集合を S1 の内容として更新する
S1.union(S2)	$S1 \cup S2$	セット	集合 S1 と集合 S2 の和集合を生成する
S1.update(S2)	$S1 \cup S2$	セット	集合 S1 と集合 S2 の和集合を S1 の内容として更新する
S1.difference(S2)	$S1 - S2$	セット	集合 S1-S2（集合の差）を生成する
S1.symmetric_difference(S2)	$S1 \cup S2 - S1 \cap S2$	セット	集合 S1 と S2 の共通しない要素の集合を生成する

2.3.4 辞書型

辞書型オブジェクトはキーと値のペアを記録するもので，文字通り辞書のような働きをする．辞書型オブジェクトの生成は

｛キー 1:値 1，キー 2:値 2，...｝

とする．例えば

```
>>> dic = {'apple':' りんご','orange':' みかん','lemon':' レモン'}
```

とすると 3 つの英単語と和訳を保持する辞書 dic ができる．この後，

```
>>> dic['apple']
```

として値を確認すると，

' りんご'

と表示される．

辞書型のオブジェクトはキーの値でハッシュ化されており，探索が高速である．

キーと値のペアを新たに辞書に追加するには，

```
dic['banana'] = ' バナナ'
```

などとする．

辞書型オブジェクトに登録されていないキーを参照しようとすると次の例のようにエラー KeyError が発生する．

```
>>> dic['grape']   Enter      ←存在しないキー 'grape' を参照すると…
Traceback (most recent call last):     ←エラーメッセージが表示される
  File "<stdin>", line 1, in <module>
KeyError:  'grape'
```

辞書型オブジェクトにアクセスする際は適切に例外処理[9] しておくか，キーが登録されているかを検査する必要がある．辞書にキーが登録されているかを検査するには 'in' を用いる．

例. キーの存在検査

```
>>> 'grape' in dic   Enter      ←辞書 dic にキー 'grape' が存在するか検査
False      ←存在しない．
```

[9]「2.3.1.1 リスト」の「例外処理のための try と except」参照のこと．

このように，**キー in 辞書型オブジェクト** という式を記述すると，キーが存在すれば True が，存在しなければ False が得られる．

空の辞書を生成するには

```
dic = dict()
```

とする．また，既存の辞書を空にするには clear メソッドを使用する．

辞書の中の特定の要素を削除するには del 文が使用できる．

例. 辞書の要素の削除

```
del dic['banana']
```

辞書型オブジェクトから全てのキーを取り出すには，次のように keys メソッドと list 関数を使用する．

例.

```
>>> dic={'apple':' りんご','orange':' みかん','lemon':' レモン'} Enter ←辞書型オブジェクト生成
>>> k = dic.keys()  Enter          ←全てのキーの取り出し
>>> k  Enter                       ←内容確認
dict_keys(['apple', 'orange', 'lemon'])        ←結果表示
>>> list(k)  Enter                 ←リストにする
['apple', 'orange', 'lemon']        ←全キーのリスト
```

このように keys メソッドにより dict_keys(...) というオブジェクトが生成され，更にそれを list 関数でリストにしている．もちろん

```
list(dic.keys())
```

のようにしても良い．

同様に，辞書型オブジェクトに対して values メソッドを使用して，全ての値のリストを生成することもできる．

2.3.5 添字（スライス）の高度な応用

リストやタプルあるいは文字列といったデータ構造では，添字 '[...]' を付けることで部分列を取り出すことができるが，ここでは添字の少し高度な応用例を紹介する．

添字の値の省略

先頭あるいは最終位置の添字は省略することができる．例えば，

```
>>> a = 'abcdefghijklmnopqrstuvwxyz'  Enter
```

として変数 a にアルファベット小文字の列を設定しておくと，a[0:26] は文字列全体を示すが，このとき先頭の 0 や 26 は省略することができる．(次の例参照)

```
>>> a[:26]  Enter                  ←先頭位置の 0 を省略
>>> 'abcdefghijklmnopqrstuvwxyz'   ←文字列全体
>>> >>> a[0:]  Enter                ←最終位置の 26 を省略
>>> 'abcdefghijklmnopqrstuvwxyz'   ←文字列全体
>>> >>> a[:]  Enter                 ←両方省略
>>> 'abcdefghijklmnopqrstuvwxyz'   ←文字列全体
```

逆順の要素指定

添字に負の数を指定すると，末尾から逆の順に要素を参照することができる．例えば上の例において，a[-1] とすると，末尾の要素を参照したことになる．(次の例参照)

```
>>> a[-1]  Enter      ←末尾の要素を参照
'z'      ←末尾の要素
>>> a[-2]  Enter      ←末尾から 2 番目の要素を参照
'y'      ←末尾から 2 番目の要素
```

不連続な部分の取り出し

添字は　**[開始位置：終了位置：増分]**　のように増分を指定することができ，不連続な「飛び飛びの」部分を取り出すこともできる.（次の例参照）

```
>>> a[::2]  Enter        ←偶数番目の取り出し
'acegikmoqsuwy'          ←先頭から 1 つ飛びの要素
>>> a[1::2]  Enter       ←奇数番目の取り出し
'bdfhjlnprtvxz'          ← 1 番目から 1 つ飛びの要素
```

更に，増分には負（マイナス）の値を指定することもできる.（次の例参照）

```
>>> a[::-1]  Enter                ←逆順に取り出す
'zyxwvutsrqponmlkjihgfedcba'  ←結果的に反転したことになる
```

2.4　制御構造

2.4.1　繰り返し (1)：　for

処理の繰り返しを実現する文の 1 つに for がある.

＜ for による繰り返し＞
書き方：　for　変数　in　データ構造:
（変数を参照した処理）

「変数を参照した処理」は繰り返しの対象となるプログラムの部分である．この部分は for の記述開始位置よりも右にインデント（字下げ）される必要がある．またこの部分には複数の行を記述することができるが，その場合は同じ深さのインデントを施さなければならない.

for 文は繰り返しの度に「データ構造」から順番に要素を取り出してそれを「変数」に与える.

繰り返しに使用するデータ構造にはリストの他，タプルや様々なもの（順序を持つ要素群から構成されるデータ構造）が指定できる.

for 文を用いたサンプルプログラム test04-1.py を次に示す.

プログラム：test04-1.py

```
1  # coding: utf-8
2
3  # 例1：リストの全ての要素を表示する
4  for word in ['book','orange','man','bird']:
5      print( word )
6
7  # 例2：繰り返し対象が複数の行の場合
8  for x in [1,3,5,7]:
9      print("x=",x)
10     print("x^3=",x**3)
11     print("x^10=",x**10)
12     print("x^70=",x**70)
13
14 # 例3：上記と同様の処理
15 for x in range(1,9,2):
16     print("x=",x)
```

```
17      print("x^3=",x**3)
18      print("x^10=",x**10)
19      print("x^70=",x**70)
```

このプログラムの例 1 の部分（4〜5 行目）は与えられたリストの要素を先頭から 1 つづつ変数 word にセットしてそれを出力するものである．この例 1 の部分の実行により出力結果は

```
book
orange
man
bird
```

となる．

例 2 の部分（8〜12 行目）は 1 から 7 までの奇数に対して 3 乗，10 乗，70 乗を計算して出力するものである．このように同じインデント（字下げ）を持つ複数の行を繰り返しの対象とする [10] ことができる．この例 2 の部分の実行により出力結果は

```
x= 1
x^3= 1
x^10= 1
x^70= 1
x= 3
x^3= 27
x^10= 59049
x^70= 2503155504993241601315571986085849
x= 5
x^3= 125
x^10= 9765625
x^70= 847032947254300339068322500679641962051391601562 5
x= 7
x^3= 343
x^10= 282475249
x^70= 14350360160986843428560307635667107174007738373924606663924 9
```

となる．

for 文では，与えられたデータ構造の要素を順番に取り出す形で繰り返し処理を実現するが，長大な数列の各項に対して処理を繰り返す場合（例えば 1〜1 億までの繰り返し）には直接にデータ列を書き綴る方法は適切ではない．繰り返し処理に指定する数列の代わりに range 関数を使用することができる．

＜ range 関数＞

書き方 1： **range**(n)

意味： 0〜$(n-1)$ の整数列

書き方 2： **range**(n_1,n_2)

意味： n_1〜(n_2-1) の整数列

書き方 3： **range**(n_1,n_2,n_3)

意味： n_1〜(n_2-1) の範囲の公差 n_3 の整数列

書き方 1 の例. `range(10)` `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]` と同等

書き方 2 の例. `range(5,10)` `[5, 6, 7, 8, 9]` と同等

書き方 3 の例. `range(1,10,2)` `[1, 3, 5, 7, 9]` と同等

この range 関数を応用したのが先のプログラム test04-1.py の例 3（15〜19 行目）の部分である．

[10] C や Java をはじめとする多くの言語では，ソースプログラムのインデント（字下げ）にはプログラムとしての意味はない．制御対象のプログラムをインデントによって指定する言語は Python を含め少数派である．

for 文における else

for 文の記述に else を付けると，終了処理を実現できる．

＜ else を用いた for 文の終了処理＞
書き方：　for　変数　in　データ構造:
（変数を参照した処理）
else:
（終了処理）
for による繰り返しが終了した直後に 1 度だけ「終了処理」を実行する．

2.4.1.1　for を使ったリストの生成（リストの内包表記）

for の別の使い方として，リストデータの生成がある．次の例について考える．

リスト生成の例

```
>>> [ x**2 for x in range(10) ]  Enter      ←リストの生成
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]      ←得られたリスト
```

このようにして，データのリストを生成できる．

＜ for を使ったリストの生成＞：リストの内包表記
書き方：
[　**繰り返し変数**を使った要素の表現　for　**繰り返し変数**　in　**データ構造**　]

他の例. 文字列を構成する文字を要素とするリスト

```
>>> [c for c in 'abcde']  Enter      ←リストの生成
['a', 'b', 'c', 'd', 'e']      ←得られたリスト
```

このように「for 〜 in･･･」で順番に要素を取り出すことができるデータ構造に対して同様の処理が適用可能であり，リストを生成できる．

2.4.2　繰り返し (2)：　while

条件判定 [11] に基いて処理を繰り返すための while 文がある．

＜ while による繰り返し＞
書き方 1：　while　条件:
（繰り返し対象の処理）
「条件」を満たす間「繰り返し対象の処理」を繰り返す.「繰り返し対象の処理」は while の記述開始位置よりも右にインデント（字下げ）される必要がある.（for 文の場合と同様）
書き方 2：　while　条件:
（繰り返し対象の処理）
else:
（条件が不成立の場合の処理）
書き方 1 の場合の動作に加えて，条件が不成立になった場合に 1 度だけ「条件が不成立の場合の処理」を実行して while 文を終了する．

[11] 条件の記述に関しては「2.4.4.1 条件式」にまとめている．

while 文を用いたサンプルプログラム test04-2.py を次に示す.

プログラム：test04-2.py

```
1 # coding: utf-8
2
3 n = 0
4 while n < 10:
5     print(n)
6     n += 2
7 else:
8     print('end')
```

このプログラムの実行により出力結果は

```
0
2
4
6
8
end
```

となる.

このサンプルの 6 行目の += は**代入演算子**（再帰代入）[12] である．すなわちこれは,

```
n = n + 2
```

と記述したものと同等である.

2.4.3 繰り返しの中断とスキップ

for や while による処理の繰り返しは break で中断して抜け出すことができる．また，繰り返し対象の部分で continue を使うと，繰り返し処理を次の回にスキップできる.（C や Java と同じ）

[12] C や Java のそれと同じ働きを持つ.

2.4.4　条件分岐

条件判定により処理を選択するには if 文を使用[13] する.

＜ if 文による条件分岐＞

書き方 1：　if　条件:

（対象の処理）

「条件」が成立したときに「対象の処理」を実行する.「対象の処理」は if の記述開始位置よりも右にインデント（字下げ）される必要がある.（for 文の場合と同様）

書き方 2：　if　条件:

（対象の処理）

else:

（条件が不成立の場合の処理）

書き方 1 の場合の動作に加えて，条件が不成立になった場合に「条件が不成立の場合の処理」を実行する.

書き方 3：　if　条件 1:

（条件 1 を満たした場合の処理）

elif　条件 2:

（条件 2 を満たした場合の処理）

⋮

else:

（条件が不成立の場合の処理）

複数の条件分岐を実現する場合にこのように記述する.

2.4.4.1　条件式

条件として記述できるものは表 8 のような**比較演算子**を用いた式や，それらを**論理演算子**（表 9）で結合（装飾）した式である.

表 8: 比較演算子

比較演算子を用いた条件式	説　　明
a == b	a と b が等しい場合に True，それ以外の場合は False.
a != b	a と b が異なる場合に True，等しい場合は False.
a ＞ b	a が b より大きい場合に True，それ以外の場合は False.
a ＞= b	a が b 以上の場合に True，それ以外の場合は False.
a ＜ b	a が b より小さい場合に True，それ以外の場合は False.
a ＜= b	a が b 以下の場合に True，それ以外の場合は False.

表 9: 論理演算子

論理演算子を用いた条件式	説　　明
P and Q	P と Q が共に True の場合に True，それ以外の場合は False.
P or Q	P と Q の少なくとも 1 つが True の場合に True，それ以外の場合は False.
not P	P が False の場合に True，それ以外の場合は False.

条件式もそれ自体が真理値の値（True か False）を返す.

[13] Python には C や Java のような switch 文はない.

※ 「if〜else…」の構文は 3 項演算子として記述することも可能である．これに関しては
「2.8.4 3 項演算子としての if〜else…」のところで解説する．

2.5 入出力

コンピュータのプログラムは各種の装置（デバイス）から入力を受け取って情報処理を行い，処理結果を各種の装置に出力する．入出力のための代表的な装置としては，

- **ディスプレイ**
- **キーボード**
- **ファイル（ディスク）**

が挙げられる[14]．特にディスプレイとキーボードは常に使用可能なデバイスであることが前提とされている．このため，ディスプレイとキーボードはそれぞれ**標準出力**，**標準入力**と呼ばれている．

ここでは，これら 3 種類のデバイスに対する入出力の方法について説明する．

2.5.1 標準出力

通常の場合，標準出力はディスプレイを示している．これまでの解説にも頻繁に使用してきた print 関数は標準出力に対して出力するものであり，引数として与えた値を順番に標準出力に出力する．print 関数は任意の個数の引数を取る．

2.5.1.1 出力データの書式設定

表示桁数や表示する順番などの書式設定を施して出力をする際には，出力対象のデータ列（値の並び）を書式編集して一旦文字列にしてから出力する．

書式設定の方法 (1)　format メソッド

文字列に対する format メソッドを用いて書式編集をすることができる．具体的には {} を含む文字列に対して format メソッドを適用すると，format の引数に与えた値が {} の部分に埋め込まれる．

例 1.

```
>>> s = '{}, {}, {}'.format('one','two','three')  Enter    ←書式編集
>>> print(s)  Enter    ←編集結果の確認
one, two, three    ←編集されて出力された結果
```

{} の中に埋め込みの順番を表すインデックス（整数）を与えると，format の引数を埋め込む順番の制御ができる．

例 2. 埋め込み位置の制御

```
>>> s = '{2}, {1}, {0}'.format('one','two','three')  Enter    ←書式編集
>>> print(s)  Enter    ←編集結果の確認
three, two, one    ←編集されて出力された結果
```

{} の中には更に，埋め込むデータのタイプと桁数（長さ）を指定することができる．

例 3. 埋め込むデータの型と桁数の指定（文字列型）

```
>>> s = '|{2:7s}|{1:7s}|{0:7s}|'.format('one','two','three')  Enter    ←書式編集
>>> print(s)  Enter    ←編集結果の確認
|three  |two    |one    |    ←編集されて出力された結果
```

この例では，'7s' の記述によってそれぞれ 7 桁の文字列になっている．

[14] この他にも重要なものとして，印刷装置（プリンタ）やネットワークインターフェース（NIC）などがある．

例 4. 埋め込むデータの型と桁数の指定（整数型）

```
>>> s = '|{2:7d}|{1:7d}|{0:7d}|'.format(1,2,3)  Enter    ←書式編集
>>> print(s)  Enter    ←編集結果の確認
|      3|      2|      1|    ←編集されて出力された結果
```

この例では，'7d' の記述によってそれぞれ 7 桁の整数になっている．

例 5. 埋め込むデータの型と桁数の指定（小数点数）

```
>>> s = '|{2:8.2f}|{1:8.2f}|{0:8.2f}|'.format(1.2,2.3,3.4)  Enter    ←書式編集
>>> print(s)  Enter    ←編集結果の確認
|    3.40|    2.30|    1.20|    ←編集されて出力された結果
```

この例では，'8.2f' の記述によってそれぞれ 2 桁の小数部を持つ 7 桁の小数点数になっている．

書式設定の方法 (2)　'%' を用いた方法

これは C や Java における書式編集に似た方法である．先に説明した format メソッドによる方法と考え方が似ており，文字列中の'%' に続く表記の場所に値を埋め込む方法である．

例 1. 文字列データの埋め込み (1)

```
>>> s = '|%7s|%7s|%7s|' % ('one','two','three')  Enter    ←書式編集
>>> print(s)  Enter    ←編集結果の確認
|    one|    two|  three|    ←編集されて出力された結果
```

この例では，'%7s' の部分に後方のタプルの要素の値が 7 桁の文字列として埋め込まれている．基本的には右寄せの配置となるが，次の例のように'%' の後の数を負の値にすると左寄せの配置となる．

例 2. 文字列データの埋め込み (2)

```
>>> s = '|%-7s|%-7s|%-7s|' % ('one','two','three')  Enter    ←書式編集
>>> print(s)  Enter    ←編集結果の確認
|one    |two    |three  |    ←編集されて出力された結果
```

このように'%' の後に表示桁数とデータタイプを指定する．データタイプとしては

d：　整数,　　f：　小数点数

が指定できる．

【sys モジュールを用いた出力】

print 関数による方法とは別に，**sys モジュール**を用いて標準出力に出力することもできる．

＜ sys.stdout ＞

オブジェクト sys.stdout は標準出力を示すオブジェクトである．このオブジェクトに対して write などのメソッドを使用して出力処理ができる．

標準出力への出力：　sys.stdout.write(文字列オブジェクト)

標準出力に対して文字列を出力する．出力処理が正常に終了すると，出力したバイト数が返される．

sys の使用に先立って，sys モジュールを読み込んでおく必要がある．

print 関数で出力すると行末で改行されるが，この方法による出力では自動的に改行処理はされない．(次の例を参照)

例. sys.stdout に対する出力

```
>>> import sys  Enter      ← sys モジュールの読込み
>>> n = sys.stdout.write('abcd')  Enter      ←出力処理の実行
abcd>>>        ←出力結果
```

出力後は改行されずにプロンプト'>>>' が直後に表示されている.

```
>>> n  Enter      ←出力バイト数の確認
4      ← 4 バイト出力されたことが確認できる
```

表示の最後に改行するには，次のように文字列の最後に'￥n' を付ける.

```
>>> n = sys.stdout.write('abcd￥n')  Enter      ←出力の最後で改行
abcd      ←表示の最後に改行サされている
>>> n  Enter      ←出力バイト数の確認
5      ← 5 バイト出力されたことが確認できる（改行コードも含む）
>>> n = sys.stdout.write('ab￥tcd￥n')  Enter      ←タブも表示
ab        cd      ←タブの表示と改行処理がされている
>>> n  Enter      ←出力バイト数の確認
6      ← 6 バイト出力されたことが確認できる（タブ，改行コードも含む）
```

2.5.2 標準入力

通常の場合，標準入力はキーボードを示しており，ユーザからのキーボード入力を取得することができる.

【input 関数による入力の取得】

input 関数を呼び出すと，標準入力から 1 行分の入力を読み取って，それを文字列として返す. [15]

＜ input 関数による入力の取得＞
書き方： input(プロンプト文字列)

input 関数を呼び出すと,「プロンプト文字列」を標準出力に表示して入力を待つ. 1 行分の入力と改行入力（Enter）により，その 1 行の内容が文字列として返される.

input 関数を用いたサンプルプログラム test05-1.py を次に示す.

プログラム： test05-1.py

```
1 # coding: utf-8
2
3 x = input('入力x> ')
4 y = input('入力y> ')
5 print( x+y )
```

このプログラムを実行すると，

入力 x>

と表示され，プログラムは入力を待つ. 続けて実行した様子を次に示す.

実行例.

```
入力 x> 2  Enter      ←「2」と入力
入力 y> 3  Enter      ←「3」と入力
23         ←出力
```

[15] getpass モジュールを使用すると，パスワード用の秘匿入力ができる. その場合は,「from getpass import getpass」としてモジュールを読み込んで，getpass(**プロンプト文字列**) として入力を取得する.

x+y の計算結果が ’23’ として表示されている．これは入力された値が文字列型であることによる．
次にサンプルプログラム test05-2.py について考える．

プログラム：test05-2.py

```
1 # coding: utf-8
2
3 x = int( input('入力x> ') )
4 y = int( input('入力y> ') )
5 print( x+y )
```

これは，input 関数の戻り値を int 関数によって整数型に変換している例である．このプログラムを実行した例を次に示す．

実行例.

入力 x> 2 Enter ←「2」と入力
入力 y> 3 Enter ←「3」と入力
5 ←出力

x+y の計算結果が 5 として表示されている．

【sys モジュールを用いた入力】

input 関数による方法とは別に，**sys モジュール**を用いて標準入力から入力することもできる．

＜ sys.stdin ＞

オブジェクト sys.stdin は標準入力を示すオブジェクトである．このオブジェクトに対して read や readline などのメソッドを使用して入力を取得することができる．

標準入力からの入力： sys.stdin.readline()

標準入力から 1 行分の入力を取得する．入力処理が正常に終了すると，取得したデータを文字列型のデータとして返す．

sys の使用に先立って，sys モジュールを読み込んでおく必要がある．

input 関数で入力する場合と異なり，この方法による入力では行末の改行コードも取得したデータに含まれる．（次の例を参照）

例. sys.stdin からの入力

```
>>> import sys  Enter   ← sys モジュールの読込み
>>> s = sys.stdin.readline()  Enter   ←入力を開始
 abcde  Enter   ← 1 行分のデータを入力
>>> s  Enter   ←戻り値の確認
 'abcde￥n'   ←行末の改行コードも含まれている
```

2.5.3 ファイルからの入力

ファイルからデータを読み込むには**ファイルオブジェクト**を使用する．ファイルオブジェクトはディスク上のファイルを示すものでる．すなわち，ファイルからの入力に先立って open 関数を使用してファイルを開き，そのファイルに対応するファイルオブジェクトを生成しておく．以後はそのファイルオブジェクトからファイルのデータ（中身）を取得することになる．

＜ファイルのオープン＞

open 関数を使用してファイルを開く

書き方：　open(パス, モード)

「パス」は開く対象のファイルのパスを表す文字列型オブジェクトである.「モード」はファイルの開き方に関する設定であり，入力用か出力用か，あるいはテキスト形式かバイナリ形式かの指定をするための文字列型オブジェクトである．ファイルのオープンが成功すると，そのファイルのファイルオブジェクトを返す.

モード：

r: 読取り用（入力用）にファイルを開く　　**w**: 書き込み用（出力）にファイルを開く

a: 追記用（出力用）にファイルを開く　　**r+**: 入出力両用にファイルを開く，

ファイルは通常はテキスト形式として開かれるが，上記モードに **b** を書き加えるとバイナリ形式の扱いとなる.

テキスト形式でファイルを開く場合は，対象のファイルの文字コードに注意する必要がある．Python の処理系が文字コードとして shift_jis の扱いを前提としている場合，utf-8 など他の文字コードのファイルを読み込むとエラーが発生する．従って，ファイルをテキスト形式で開く場合は，次の例のように encoding を指定して，読み込むファイルの文字コードを指定しておくべきである.

例. utf-8 のテキストファイルを開く場合

```
open(ファイル名,'r',encoding='utf-8')
```

Python で扱える文字コードは「2.1.1 プログラム中に記述するコメント」で説明した表 1 のものを指定する.

ファイルからのデータ入力の例：

テキストファイルから 1 行づつデータを取り出すプログラム test06-1.py を例示する.

プログラム：test06-1.py

```
1  # coding: utf-8
2
3  f = open('test06-1.txt','r')
4
5  while True:
6      s = f.readline()
7      if s:
8          print(s)
9      else:
10         break
11
12 f.close()
```

基本的な考え方：

このプログラムはテキストファイル test06-1.txt を開き，それをファイルオブジェクト f としている．このファイルオブジェクトに対して readline [16] メソッドを使用してデータを 1 行づつ取り出して変数 s に与え，それを print 関数で表示している．ファイルの内容を全て読み終わると，次回 readline 実行時にデータが得られない [17] ので，break により while を終了する.

テキストファイル：test06-1.txt

```
1 1行目
2 2行目
3 3行目
```

[16] readline メソッドは，開かれているファイルのモードによって返す値の型が異なり，テキストモードで開いているときは文字列型で，バイナリモードで開いているときはバイト列で返す.

[17] 変数 s に None（ヌルオブジェクト）がセットされる.

プログラム test06-1.py を実行すると次のように表示される．

1 行目

2 行目

3 行目

この実行例では余分に改行された形で表示されている．これは，readline が改行コードも含めて取得するため．変数 s にセットされる文字列オブジェクトの末尾にも改行コードが含まれるからである．

文字列オブジェクトの行末の改行コードを削除するには rstrip メソッドを使用する．

＜改行コードの削除＞
書き方：　対象文字列.rstrip()
「対象文字列」の末尾にある改行コードを削除する．

rstrip メソッドを用いた形に修正したプログラム test06-2.py を示す．

プログラム：test06-2.py

```
1  # coding: utf-8
2
3  f = open('test06-1.txt','r')
4
5  while True:
6      s = f.readline().rstrip()
7      if s:
8          print(s)
9      else:
10         break
11
12 f.close()
```

このプログラムを実行すると次のように表示される．

1 行目
2 行目
3 行目

ファイルからの読込みが終われば，そのファイルを閉じる．

＜ファイルのクローズ＞
書き方：　ファイルオブジェクト.close()
開かれている「ファイルオブジェクト」を閉じる．

テキスト読込みに伴う文字コードの不具合について：

テキスト形式の入力データの文字コードによっては，入力処理において問題を起こす場合がある．(次の実行例)

1 陦檎岼
2 陦檎岼
3 陦檎岼

ファイルのデータをテキスト形式として扱うと，文字コードに応じた処理が自動的に施されるが，それが適切に働かない場合にはこのような現象[18] が起こることがある．

解決策の 1 つとして，テキストファイルの読み込みにおいてもバイナリ形式でファイルを扱うということが挙げられ

[18] いわゆる「文字化け」

る．バイナリ形式としてファイルからデータを入力すると，それらはデータ型とは無関係な**バイト列**とみなされる．（「2.5.3.1 バイト列の扱い」参照）
バイト列として得られたデータを，プログラム側で明に各種のデータ型の値に変換することでより安全な処理が実現できる．すなわち，バイト列として得られたデータを，正しい文字コードの扱いを指定した上で文字列型のオブジェクトに変換するということが可能になる．

次のプログラム test06-3.py について考える．

プログラム：test06-3.py

```
 1 # coding: utf-8
 2
 3 f = open('test06-1.txt','rb')
 4
 5 while True:
 6     s = f.readline().rstrip().decode('utf-8')
 7     if s:
 8         print(s)
 9     else:
10         break
11
12 f.close()
```

このプログラムの 6 行目の部分に見られる decode メソッドは，バイト列のデータに対するメソッドであり，引数に指定した文字コードに変換して，結果を文字列型のオブジェクトとして返す．
このような形でファイルからデータを入力すると，他バイト系の文字も正しく処理される．

2.5.3.1 バイト列の扱い

バイト列は文字列型や数値型とは異なるオブジェクトであり，ファイル入出力や通信をする場合に用いられることが多いオブジェクトである．例えば文字列型オブジェクトを用いると，多バイト系文字列も次の例のように正しく表示することができる．

```
>>> a = '日本語'  Enter      ←日本語文字列の作成
>>> a  Enter                 ←内容の確認
>>> '日本語'                 ←正しく表示されている
```

これは Python 処理系が多バイト系文字列データの文字コード体系を正しく解釈して処理しているからであるが，この文字列はバイト列としては，

```
e6 97 a5 e6 9c ac e8 aa 9e          （16 進数表現）
```

というデータの列であり，ファイルとして保存する場合もこのような値の並びとして記録されている．
すなわち，Python を始めとする処理系は日本語など多バイト文字をディスプレイに表示する際，正しい記号として表示するように制御をしている．
Python では，通信やファイル入出力において各種のオブジェクトの内容をやりとりする際に，一旦バイト列のデータに変換する．上の例で扱った日本語の文字列型オブジェクトもバイト列に変換することができる．（次の例参照）

```
>>> b = a.encode('utf-8')  Enter    ←バイト列に変換
>>> b  Enter                        ←内容の確認
b'¥xe6¥x97¥xa5¥xe6¥x9c¥xac¥xe8¥xaa¥x9e'      ←バイト列データ
```

このように encode メソッドを使用することで，文字型オブジェクトをバイト列に変換することができる．

＜バイト列⇔多バイト系文字列の変換＞
バイト列⇒文字列の変換　対象バイト列.decode(文字コード)　戻り値は文字列型オブジェクト
文字列⇒バイト列の変換　対象文字列.encode(文字コード)　戻り値はバイト列

2.5.3.2　ファイルの内容を一度で読み込む方法

開いたファイルに対して，read メソッドを実行すると，ファイルの内容を全て読み込むことができる．

例. ファイルの内容を全て読み込む

```
f = open(ファイル名,'r',encoding='utf-8')
text = f.read()
f.close()
```

ただし，ファイルのデータサイズが大きい場合は注意が必要である．

ファイルを開くための別の方法を「2.5.5.4 パスを扱うための更に別の方法」で説明する．

2.5.4　ファイルへの出力

開かれたファイルに対してデータを出力することができる．ファイルのオープンとクローズについては「2.5.3 ファイルからの入力」のところで説明したとおりであり，ここではファイルオブジェクトに対する出力について解説する．

ファイルオブジェクトに対する出力には write メソッドを使用する．

＜ write メソッド＞
書き方：　ファイルオブジェクト.write(データ)
　「ファイルオブジェクト」に対して「データ」を出力する.「データ」に与えるオブジェクトの型は，ファイルオブジェクトのモードによる．すなわち，ファイルオブジェクトがテキストモードのときは文字列型で，バイナリモードの場合はバイト列で与える．

2.5.5　パス（ファイル，ディレクトリ）の扱い

ここではパス（ファイル，ディレクトリ）に対する各種の操作について説明する．パスに対する操作をするには os モジュールを使用するので，次のようにして読み込んでおく．

```
import os
```

2.5.5.1　カレントディレクトリに関する操作

相対パスを指定してファイルの入出力を行う場合は**カレントディレクトリ**を基準とする．Python のプログラム実行時には，処理系を起動した際のディレクトリがカレントディレクトリとなるが，プログラムの実行時にこれを変更することができる．

【カレントディレクトリの取得】
os モジュールの getcwd メソッドを使用する．

実行例.

```
>>> os.getcwd()   Enter      ←カレントディレクトリを調べる
'C:￥￥Users￥￥katsu      ←カレントディレクトリ
```

結果は文字列の形式で得られる．これは Windows における実行例であり,「￥」はエスケープされて「￥￥」となる．

【カレントディレクトリの変更】
os モジュールの chdir メソッドを使用する．

実行例.

```
>>> os.chdir('..')  Enter    ←カレントディレクトリを変更（1 つ上へ）
>>> os.getcwd()  Enter    ←カレントディレクトリを調べる
'C:￥￥Users'    ←カレントディレクトリが変更されている
```

2.5.5.2 ディレクトリ内容の一覧

os モジュールの listdir メソッドを使用する.

実行例.

```
>>> os.listdir()  Enter    ←内容リストの取得
['file1.txt','a.exe',  …  ]    ←実行結果
```

結果はリストの形式で得られる．listdir メソッドの引数にパスを文字列の形式で指定すると，そのパスのディレクトリ配下の内容のリストが得られる.

os モジュールにはこの他にも様々なメソッドが用意されている．詳しくは Python のインターネットサイト（巻末付録 A.1）参照のこと.

2.5.5.3 ファイル，ディレクトリの削除

ファイルを削除するには remove メソッドを使用して,

os.remove(**削除対象のファイルのパス**)

とする．また空ディレクトリを削除するには rmdir メソッドを使用して,

os.rmdir(**削除対象のディレクトリのパス**)

とする．削除対象のディレクトリの配下にはファイルやディレクトリがあってはならない.

2.5.5.4 パスを扱うための更に別の方法：pathlib

Python 3.4 から標準ライブラリとして導入された pathlib は，ファイルとディレクトリを扱うための別の方法を提供する．このライブラリを使用するには次のようにして必要なパッケージを読み込む.

```
from pathlib import Path
```

■ パスオブジェクトの生成

ファイルシステムのパスを表す Path オブジェクトを生成する．コンストラクタの引数にはパスを文字列型で与える.

例. /Users/katsu/test01.txt を表すパスオブジェクト

```
p = Path('/Users/katsu/test01.txt')
```

この例はファイルへのパス /Users/katsu/test01.txt を表すオブジェクト [19] を p として生成するものである.

■ ファイルのオープン

パスオブジェクトに対して open メソッドを使用してファイルを開くことができる．このとき open メソッドの引数に**モード**や**文字コード**を与える．例えば，パスオブジェクト p を文字コードが utf-8 のテキスト形式として読取り用に開くには次のようにする.

```
f = p.open('r',encoding='utf-8')
```

処理の結果，ファイルオブジェクト f が返される.

[19] 処理を実行する OS が Windows の場合，p は WindowsPath というタイプのオブジェクトとなる.

■ パスの存在の検査

パスオブジェクトに対して exists メソッドを使用することで，そのパスが存在するかどうかを調べることができる．（exists の引数は空にする）そのパスが存在する場合は True を，存在しない場合は False を返す．

■ ファイル／ディレクトリの検査

パスオブジェクト p がファイルかディレクトリかを調べるには，例えば次のようにして is_dir メソッドを使用する．

```
p.is_dir()
```

p がディレクトリの場合は True を，それ以外の場合は False を返す．

■ ディレクトリの要素を取得する

パスオブジェクト p がディレクトリの場合，glob メソッドを使用して，配下の要素（ファイル，サブディレクトリ）を取得することができる．glob メソッドの引数には**パターン**（ワイルドカードを含む）を文字列型で与える．例えば，

```
plst = list( p.glob('*') )
```

とすると，ディレクトリ p の配下の要素のリスト plst が得られる．

■ その他

パスオブジェクトに対して二項演算子 '/' を使用することで，パスの連結ができる．例えば，`p = Path('/Users')` に対して `p / 'katsu'` と記述すると，それは`'/Users/katsu'` を意味するパスオブジェクトとなる．

2.5.6 コマンド引数の取得

ソースプログラムを Python 処理系（インタプリタ）に与えて実行を開始する際，起動時に与えたコマンド引数を取得するには sys モジュールのプロパティargv を参照する．次のプログラム test17.py の実行を例にして説明する．

プログラム：test17.py

```
1 # coding: utf-8
2
3 # 必要なモジュールの読込み
4 import sys
5
6 # コマンド引数の取得
7 print('args>',sys.argv)
```

このプログラムは，起動時のコマンド引数の列をリストにして表示するものであり，実行すると次のように表示される．

```
py test17.py 1 2 3 a b  Enter     ← OS のコマンドラインから起動
args> ['test17.py', '1', '2', '3', 'a', 'b']   ←与えた引数が得られる
```

このように，ソースプログラム名から始まる引数が文字列のリストとして得られることがわかる．

2.6 関数の定義

関数とは，

関数名（引数列）

の形式のサブプログラムで，メインプログラムや他のサブプログラムから呼び出す形で実行する．関数は，処理結果を何らかのデータとして返す（戻り値を持つ）ものである．

＜関数定義の記述＞
書き方：　def　関数名（仮引数列）：
（処理内容）
return　戻り値
処理内容から return までの行は，def よりも右の位置に同一の深さのインデント（字下げ）を施して記述する．

関数定義の例. 加算する関数 kasan の定義（プログラム test08-1.py）

プログラム：test08-1.py

```
1  # coding: utf-8
2
3  # 加算する関数
4  def kasan(x,y):
5      z = x + y
6      return z
7
8  # メインルーチン
9  a = kasan(12,34)
10 print(a)
```

このプログラムを実行すると，

```
46
```

と表示される．

関数の仮引数について

関数はそれを呼び出したプログラムから仮引数に値を受け取って処理を行う．先のプログラム test08-1.py では関数 kasan は 2 つの仮引数を持ち，それらに受け取った値の加算を行っている．

仮引数の個数が予め決まっていない関数も定義できる．それを実現する場合は，関数定義の仮引数にアスタリスク「＊」で始まる名前を指定する．

個数未定の仮引数の例. kansu(*args)

この例のようにすると，仮引数 args がタプルとして機能し，関数内では args が受け取った値を要素として持つタプルとして扱える．これを応用したプログラムの例を test08-2.py に示す．

プログラム：test08-2.py

```
1  # coding: utf-8
2  # 個数未定の引数を取る関数
3  def argtest1( *a ):
4      n = len(a)
5      for m in range(n):
6          print( a[m] )
7      return n
8
9  # メインルーチン
10 a = argtest1('a',1,'b',2)
11 print('引数の個数',a)
```

これを実行すると次のような表示となる．

```
a
1
b
2
引数の個数 4
```

Python の関数では**キーワード引数**が使える，次のプログラム例 test08-3.py について考える．

プログラム：test08-3.py

```
1  # coding: utf-8
2
3  # キーワード引数を取る関数
4  def argtest2( **ka ):
5      print( '名前: ', ka['name'] )
6      print( '年齢: ', ka['age'] )
7      print( '国籍: ', ka['country'] )
8      n = len(ka)
9      return n
10
11 # メインルーチン
12 a = argtest2( name='tanaka', country='japan', age=41 )
13 print('引数の個数: ',a)
```

このようにアスタリスク 2 つ「**」で始まる仮引数を記述すると関数側でその仮引数は辞書型オブジェクトとして扱える．(関数側ではキーは文字列型)

このプログラムの 12 行目に

```
argtest2( name='tanaka', country='japan', age=41 )
```

という形の関数呼び出しがある．キーワード引数を用いると「キーワード=値」という形式で引数を関数に渡すことができる．このプログラムを実行すると次のような表示となる．

```
名前:  tanaka
年齢:  41
国籍:  japan
引数の個数:  3
```

関数定義の内外での変数の扱い

関数の内部で生成したオブジェクトは基本的にはその関数の**ローカル変数**であり，その関数の実行が終了した後は消滅する．関数の外部で生成された**大域変数**を関数内部で使用（参照，更新）するには，当該関数内で大域変数の使用を宣言する必要がある．具体的には関数定義の内部で，

global　**大域変数の名前**

と記述する．次に示すプログラム test08-4.py は，大域変数 gv の値が関数呼出しの前後でどのように変化するかを示す例である．

プログラム：test08-4.py

```
1  # coding: utf-8
2  # 変数のスコープのテスト
3
4  gv = '初期値です，'      # 大域変数
5
6  #--- 関数内部で大域変数を使用する例 ---
7  def scopetest1():
8      global gv        # 大域変数であることの宣言
9      print('scopetest1 の内部では:',gv)
10     gv = 'scopetest1が書き換えたものです．'
11
12 #--- 大域変数と同名の局所変数を使用する例 ---
```

```
13 def scopetest2():
14     gv = 'scopetest2の局所変数gvの値です. '
15     print(gv)
16
17 #--- メインルーチン ---
18 print('【大域変数gvの値】')
19 print('scopetest1 呼び出し前:',gv)
20 scopetest1()
21 print('scopetest1 呼び出し後:',gv)
22 scopetest2()
23 print('scopetest2 呼び出し後:',gv)
```

このプログラムを実行した例を次に示す.

【大域変数 gv の値】
scopetest1 呼び出し前:　初期値です,
scopetest1 の内部では:　初期値です,
scopetest1 呼び出し後:　scopetest1 が書き換えたものです.
scopetest2 の局所変数 gv の値です.
scopetest2 呼び出し後:　scopetest1 が書き換えたものです.

2.7　オブジェクト指向プログラミング

Python におけるオブジェクト指向の考え方も他の言語のそれと概ね同じである．ここでは，オブジェクト指向についての基本的な考え方の説明は割愛して，**クラス**と**メソッド**の定義の具体的な方法について説明する.

2.7.1　クラスの定義

クラスの定義は class で開始する.

＜ class の記述＞
書き方 1：　class　クラス名:
（定義の記述）
「定義の記述」は class よりも右の位置に同一の深さのインデント（字下げ）を施して記述する．別のクラスをスーパークラスとし，その拡張クラス（サブクラス）としてクラスを定義するには次のように記述する.
書き方 2：　class　クラス名 (スーパークラス):
（定義の記述）
「スーパークラス」はコンマで区切って複数記述することができる．すなわち Python では多重継承が可能である.

2.7.1.1　コンストラクタ

クラスのインスタンスを生成する際のコンストラクタは，クラスの定義内に次のように__init__ を記述する.（init の前後にアンダースコアを 2 つ記述する）

＜__init__の記述＞
書き方：　def　__init__(self, 仮引数):
（定義の記述）
仮引数は複数記述することができる．また仮引数は省略可能である.「定義の記述」は def よりも右の位置に同一の深さのインデント（字下げ）を施して記述する．self は生成するインスタンス自身を指しており，第 1 仮引数に記述する.

コンストラクタはクラスのインスタンスを生成する際に実行される処理である．インスタンスの生成は
インスタンス名 = クラス名 (引数)
とする.「引数」にはコンストラクタの __init__に記述した self より右の仮引数に与えるものを記述する.

2.7.1.2　メソッドの定義

クラスの定義内容としてメソッドの定義を記述する．これは関数の定義の記述とよく似ている.

＜メソッドの定義＞
書き方：　def　メソッド名 (self, 仮引数):
（定義の記述）
仮引数は複数記述することができる．また仮引数は省略可能である.「定義の記述」は def よりも右の位置に同一の深さのインデント（字下げ）を施して記述する．self は生成するインスタンス自身を指しており，第 1 仮引数に記述する.

クラスメソッドを定義するには，def の 1 行前に def と同じインデント位置に
@classmethod
というデコレータ [20] を記述する.

2.7.1.3　クラス変数

インスタンスの変数ではなく，クラスそのものに属する変数を**クラス変数**という．クラス変数の定義は，値の設定などオブジェクトの生成をクラスの定義の中に記述することで行う.

通常のオブジェクト指向の考え方では，クラスに属する**メンバ**と呼ばれるオブジェクトが定義され，そのクラスのインスタンスが個別に持つ属性の値を保持する．Python ではメンバの存在を明に宣言するのではなく，そのクラスのコンストラクタやメソッドの定義の中で，self の属性として値を与える（オブジェクトを生成する）ことで**インスタンス変数**として生成する

オブジェクト指向のプログラム例 test09-1.py を次に示す.

プログラム：test09-1.py

```
1  # coding: utf-8
2
3  # クラス定義
4  class Person:
5      # クラス変数
6      population = 7400000000
7      # コンストラクタ
8      def __init__(self,Name,Age,Gender,Country):
9          self.name = Name
10         self.age = Age
11         self.gender = Gender
12         self.country = Country
13     # クラスメソッド
```

[20] Java のアノテーションとよく似た働きをする記述である.

```
14      @classmethod
15      def belongTo(cls):
16          print(cls,'は人類に属しています. ')
17          return 'Human'
18      # メソッド
19      def getName(self):
20          print('名前は',self.name,'です. ')
21          return self.name
22      def getAge(self):
23          print('年齢は',self.age,'才です. ')
24          return self.age
25      def getGender(self):
26          print('性別は',self.gender,'性です. ')
27          return self.gender
28      def getCountry(self):
29          print(self.country,'から来ました. ')
30          return self.country
31
32  # メインルーチン
33  m1 = Person('太郎',39,'男','日本')
34  m2 = Person('アリス',28,'女','アメリカ')
35
36  m1.getName()
37  m1.getAge()
38  m1.getGender()
39  m1.getCountry()
40
41  m2.getName()
42  m2.getAge()
43  m2.getGender()
44  m2.getCountry()
45
46  Person.belongTo()
47  print('現在の人口は約',Person.population,'人です. ')
```

このプログラムを実行すると，次のように表示される.

```
名前は 太郎 です.
年齢は 39 才です.
性別は 男 性です.
日本 から来ました.
名前は アリス です.
年齢は 28 才です.
性別は 女 性です.
アメリカ から来ました.
<class '__main__.Person'> は人類に属しています.
現在の人口は約 7400000000 人です.
```

2.7.1.4 属性の調査

dir 関数を使用すると，オブジェクトの属性（プロパティ，メソッド）のリストを取得できる.

例. 文字列クラスの属性をリストとして取得する

```
>>> dir(str)   Enter          ← str クラスの調査
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__',
'__ge__', '__getattribute__', '__getitem__', '__getnewargs__', '__gt__', '__hash__', '__init__',
            (途中省略)
'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith',
'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
```

2.8 データ構造に則したプログラミング

Python にはデータ列の要素に対して一斉に処理を適用する機能が提供されている．例えば 2 倍した値を返す関数 dbl が次にように定義されているとする．

```
def dbl(n):return(2*n)
```

通常は下記のように，これを 1 つの値に対して実行する．

例.
```
>>> dbl(3)   Enter        ←関数の評価
6                    ←評価結果
```

この評価方法とは別に，map **関数**を使用することもできる．map 関数を使うと，リストの全ての要素に対して評価を適用することができる．このことを例を挙げて説明する．

例. map 関数による一斉評価（第一段階）
```
>>> res = map( dbl, [1,2,3] )   Enter
```

これにより，関数の評価をリストの全要素に対して一斉に行うことができる．ただしこの段階では「関数の一斉評価のための式」が res に生成されただけであり，res の内容を確認すると次のようになっている．

例. map 関数の実行結果
```
>>> res   Enter        ←内容の確認
<map object at 0x000002BE46760DD8>  ← map オブジェクトが格納されている
```

この res をリストとして評価すると，最終的な処理結果が得られる．

例. リストとしての処理結果
```
>>> list( res )   Enter        ←リストとして処理結果を求める
[2, 4, 6]        ←処理結果
```

当然であるが，次のようにして一回の処理で結果を得ることもできる．

例. リストとしての処理結果 (2)
```
>>> list( map( dbl, [1,2,3] ) )   Enter        ←同様の処理を一度で行う
[2, 4, 6]        ←処理結果
```

2.8.1 map 関数

< map 関数>

map(**関数名, 対象のデータ構造**)

「対象のデータ構造」の全ての要素に対して「関数名」で表される関数の評価を実行し，結果のリストを返す．

リストの全要素に対して同じ関数を適用する処理を実現する場合，for 文を用いた繰り返し処理の形で記述することもできるが，map 関数を用いた方が記述が簡潔になるだけでなく，実行にかかる時間も概ね短くなる．

【サンプルプログラム】 map 関数と for 文の実行時間の比較

map 関数と for 文の実行時間を比較するためのサンプルプログラム map01.py を次に示す．これは，奇数／偶数を判定する関数 EvenOrOdd を，乱数を要素として持つリストに対して一斉適用する例である．

プログラム：map01.py

```
# coding: utf-8
# 必要なモジュールの読込み
import time                         # 時間計測用
from random import randrange    # 乱数発生用

############################################################
#    偶数／奇数を判定する関数の定義                        #
############################################################
def EvenOrOdd(n):
    if n % 2 == 0:
        return( '偶' )
    else:
        return( '奇' )

############################################################
#    試み(1)                                               #
############################################################
# 乱数リストの生成(1)：  短いもの
lst = list( map( randrange, [100]*10 ) )

# 偶数／奇数の識別結果
lst2 = list(map(EvenOrOdd,lst))
print('10個の乱数の奇数／偶数の判定')
print(lst)
print(lst2,'\n')

############################################################
#    試み(2)                                               #
############################################################
# 乱数リストの生成(2)：  1,000,000個
print('1,000,000個の乱数の奇数／偶数の判定')
t1 = time.time()
lst = list( map( randrange, [100]*1000000 ) )
t = time.time() - t1
print('乱数生成に要した時間:',t,'秒')

# 速度検査(1)：forによる処理
lst2 = []
t1 = time.time()
for i in range(1000000):
    lst2.append( EvenOrOdd(lst[i]) )
t = time.time() - t1
print('forによる処理:',t,'秒')

# 速度検査(2)：mapによる処理
t1 = time.time()
lst2 = list(map( EvenOrOdd, lst ))  # mapによる処理
t = time.time() - t1
print('mapによる処理:',t,'秒')

# 判定結果の確認
#print(lst2)
```

このプログラムを実行した例を次に示す.

```
10個の乱数の奇数／偶数の判定
[92, 3, 7, 26, 11, 22, 76, 61, 79, 10]
['偶', '奇', '奇', '偶', '奇', '偶', '偶', '奇', '奇', '偶']

1,000,000個の乱数の奇数／偶数の判定
乱数生成に要した時間:  0.9250545501708984 秒
forによる処理:  0.32311391830444336 秒
mapによる処理:  0.15199923515319824 秒
```

map 関数による方が for 文による処理よりも実行時間が半分以下になっていることがわかる.

2.8.2 lambda と関数定義

引数として与えられた値（定義域）から別の値（値域）を算出して返すものを**関数**として扱うが，Python のプログラムの記述の中では**関数名のみ**を記述する場合もある．map 関数の中に記述する関数名もその 1 例であるが，この「関数名のみ」の表記は実体としては**関数オブジェクト**もしくは lambda である．

【実体としての関数】

map 関数の解説のところで示したサンプルプログラムで扱った偶数／奇数を判定する関数 EvenOrOdd について考える．この関数の定義を次のようにして Python インタプリタに与える．

例. Python インタプリタに関数定義を与える

```
>>> def EvenOrOdd(n):  Enter          ←定義の記述の開始
...     if n % 2 == 0:  Enter
...         return( '偶' )  Enter
...     else:  Enter
...         return( '奇' )  Enter
...   Enter        ←定義の記述の終了（改行のみ）
>>>              ← Python コマンドラインに戻る
```

この関数の評価を実行する際，次のようにして引数を括弧付きで与える．

```
>>> EvenOrOdd(13)   Enter          ←評価の実行
'奇'        ←評価結果
```

次に，'EvenOrOdd' という記号そのものには何が割当てられているのかを確認する．

```
>>> print( EvenOrOdd )   Enter          ←内容の確認
<function EvenOrOdd at 0x000001D863C63E18>     ←関数オブジェクト
```

このように，記号 'EvenOrOdd' には**関数オブジェクト**が割当てられていることがわかる．このオブジェクトは，データとして変数に割り当てることができるものであり，このような扱いができるオブジェクトは**第一級オブジェクト** [21]（first-class object）と呼ばれる．この例で扱っている関数オブジェクトも，他の変数に割り当てて使用すること（評価の実行）が可能である．(次の例を参照のこと)

例. 変数への関数オブジェクトの割当てと実行

```
>>> f = EvenOrOdd   Enter          ←別の変数 f に関数オブジェクトを複製
>>> f(13)   Enter                ←評価の実行
'奇'              ←値が得られている
```

【lambda】

関数の定義を lambda [22] 式として記述して取り扱うことができる．先に例示した「2 倍の値を返す関数」dbl を lambda で実装する例を示す．

例. 値を 2 倍する関数 dbl2 の実装例

```
>>> dbl2 = lambda n:2*n   Enter     ← lambda 式による実装
>>> dbl2(3)   Enter          ←評価の実行
6          ←結果が得られている
```

[21] 変数に値として割り当てることができ，プログラムの実行時にデータとして生成と処理ができるオブジェクトを指す．関数定義を第一級オブジェクトとして扱える言語処理系は少なく，一部のリスト処理系（LISP の各種実装）がそれを可能にしている．

[22] 数学の関数表記 $f(x)$ の f を実体として扱い，対象となるデータにそれを適用して値を算出する**ラムダ計算**が由来である．ラムダ計算に関しては提唱者 A. チャーチの著書 "The Calculi of Lambda Conversion", *Princeton University Press, 1941* を参照のこと．

この例の dbl2 の内容を確認すると次のようになる.

```
>>> dbl2   Enter                    ← dbl2 の内容確認
<function <lambda> at 0x00000210D6EB3E18>  ←結果
```

このように関数オブジェクト（lambda 式）が格納されていることがわかる.

＜ lambda の記述＞

記述 1)　lambda **仮引数**：**戻り値の式**

記述 2)　lambda **仮引数並び**：**戻り値の式**

「仮引数並び」はコンマ ',' で区切る.

複数の仮引数を取る関数 wa の記述例を次に示す.

例. 2 つの引数の和を求める関数 wa の実装

```
>>> wa = lambda a,b :  a+b   Enter     ← lambda 式を wa に割り当てる
>>> wa(2,3)   Enter          ←評価の実行
5           ←結果
```

また，次の例のように lambda 式を記号に割当てずに適用することもできる.

例. lambda 式を直接引数に適用する.

```
>>> (lambda a,b :  a+b)(2,3)   Enter     ← lambda 式の直接適用
5           ←結果
```

2.8.3　filter

リストの要素の内，指定した条件を満たす要素のみを取り出す関数に filter がある.

＜ filter 関数＞

filter(**条件判定用の関数名, 対象リスト**)

「対象リスト」の要素の内,「条件判定用の関数」の評価結果が真（True）となるものを抽出して返す.
第一引数には関数名の他，lambda 式も指定できる.

整数を要素として持つリストの中から偶数の要素のみを取り出す処理を例として示す.

例. 偶数の取り出し

```
>>> from random import randrange   Enter     ←乱数発生用のパッケージの読込み
>>> lst = list( map( randrange, [100]*17 ) )   Enter     ←乱数リスト（17 要素）の生成
>>> lst   Enter     ←内容確認
[15, 74, 64, 81, 58, 18, 70, 88, 45, 44, 3, 81, 36, 98, 57, 18, 27]     ←乱数リスト
>>> lst2 = list( filter( lambda n:n%2==0, lst ) )   Enter     ←偶数のみの抽出
>>> lst2   Enter     ←内容確認
[74, 64, 58, 18, 70, 88, 44, 36, 98, 18]     ←偶数のみのリスト
```

2.8.4　3 項演算子としての if〜else…

条件分岐のための「if〜else…」文に関しては「2.4.4 条件分岐」のところで解説したが，3 項演算子としての「if〜else…」の記述も可能である．

＜ if〜else… 演算子＞

値 1 if **条件式** else **値 2**

この文は演算の式であり，結果として値を返す．「条件式」が真の場合は「値 1」を，偽の場合は「値 2」を返す．

この式は，条件による値の評価の選択を lambda 式などの中で簡潔に記述するのに役立つ．

例. 偶数／奇数を判定する lambda 式

```
>>> evod = lambda n :’偶’ if n % 2 == 0 else ’奇’   Enter      ← 1 行で条件分岐を記述
>>> evod( 2 )   Enter      ← 2 は偶数か奇数か？
’偶’      ←処理結果
>>> evod( 3 )   Enter      ← 3 は偶数か奇数か？
’奇’      ←処理結果
```

3　Kivy による GUI アプリケーションの構築

Kivy は MIT ライセンスで配布されるライブラリモジュールであり，インターネットサイト https://kivy.org/から入手できる．インストール方法[23] から API の説明まで当該サイトで情報を入手することができる．本書では Kivy の基本的な使用方法について解説する．

3.1　Kivy の基本

Kivy によるアプリケーションプログラムは，App クラスのオブジェクトとして構築する．App クラスの使用に際して，下記のようにして必要なモジュールを読み込んでおく．

App クラスの読込み

```
from kivy.app import App
```

3.1.1　アプリケーションプログラムの実装

具体的には，App クラスかそれを継承する（拡張する）クラス（以後**「アプリケーションのクラス」**と呼ぶ）をプログラマが定義し，そのクラスのインスタンスを生成することでアプリケーション・プログラムが実装できる．アプリケーションのインスタンスに対して run メソッドを実行することでアプリケーションプログラムの動作が開始する．

run メソッドを呼び出すと，最初にアプリケーションのクラスのメソッド build が呼び出される．この build メソッドは，App クラスに定義されたメソッドであり，これをプログラマが上書き定義（**オーバーライド**）することで，アプリケーション起動時の処理を記述することができる．build メソッドで行うことは主に GUI の構築などである．

Kivy によるアプリケーションプログラム構築の素朴な例として，サンプルプログラム kivy01-1.py を次に示す．

プログラム：kivy01-1.py

```
1  # coding: utf-8
2
3  # 基本となるアプリケーションクラス
4  from kivy.app import App
5  # ラベルオブジェクト
6  from kivy.uix.label import Label
7
8  # アプリケーションのクラス
9  class kivy01(App):
10     def build(self):
11         self.lb1 = Label(text='This is a test of Kivy.')
12         return self.lb1
13
14 #---- メインルーチン -----
15 # アプリケーションのインスタンスを生成して起動
16 ap = kivy01()
17 ap.run()
```

この例では，App クラスを拡張した Kivy01 クラスとしてアプリケーションを構築している．Kivy01 クラスの中では build メソッドをオーバーライド定義しており，Label ウィジェット（文字などを表示するウィジェット）を生成している．

このプログラムを実行すると図 1 のようなウィンドウが表示される．

[23]巻末付録「B.1 Kivy のインストール作業の例」でインストール方法を概略的に紹介している．

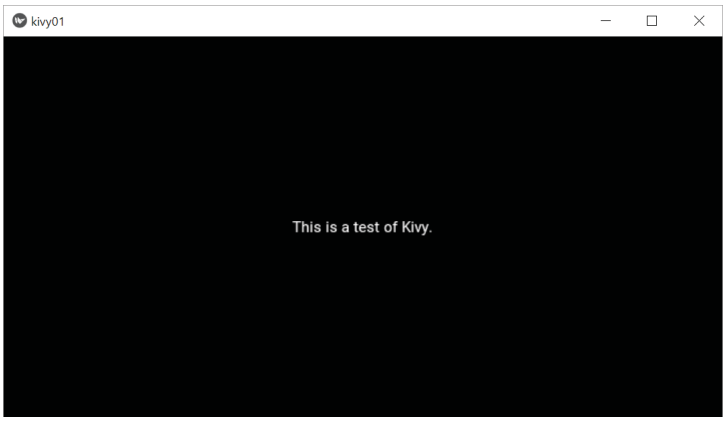


図 1: Kivy01-1.py の実行結果

3.1.2 GUI 構築の考え方

先のプログラム kivy01-1.py においては，GUI 要素として Label ウィジェットを生成しており，このウィジェットが build メソッドの戻り値となる．

Kivy では，GUI を階層構造として構築する．すなわち「親」のオブジェクトの配下に「子」のオブジェクト群が従属する形で GUI を構築する．Kivy には GUI の配置を制御する Layout や Screen といったクラスがあり，それらクラスのオブジェクト配下にウィジェットなどの要素を配置する形式で GUI を構築する．先のプログラム kivy01-1.py では，最も単純に GUI の導入説明をするために．Label オブジェクトが 1 つだけ存在するものとした．すなわち，この Label オブジェクトが GUI の最上位の「親オブジェクト」となっている．実際のアプリケーション構築においては，Layout や Screen のオブジェクトなど，要素の配置を制御するものを GUI の最上位オブジェクトとすることが一般的である．

Kivy における GUI のためのクラスは，大まかに**ウィジェット**，**レイアウト**，**スクリーン**の 3 つに分けて考えることができる．

3.1.2.1 Widget（ウィジェット）

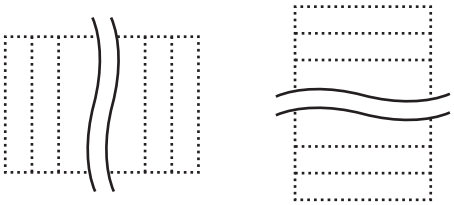
ボタンやラベル，チェックボックス，テキスト入力エリアといった基本的な要素である．代表的なウィジェットを表 10 に挙げる．

表 10: 代表的なウィジェット

クラス	機能
Label	ラベル（文字などを表示する）
Button	ボタン
TextInput	テキスト入力（フィールド／エリア）
CheckBox	チェックボックス
ProgressBar	進捗バー
Slider	スライダ
Switch	スイッチ
ToggleButton	トグルボタン
Image	画像表示
Video	動画表示

3.1.2.2 Layout（レイアウト）

レイアウトは GUI オブジェクトを配置するためのもので，一種の「コンテナ」（容器）と考えることができる．例えば BoxLayout を使用すると，その配下にウィジェットなどを水平あるいは垂直に配置する（図 2）ことができる．



(a) 水平の配置　　(b) 垂直の配置

図 2: BoxLayout

例えば，BoxLayout を入れ子の形で（階層的に）組み合わせると，図 3 のような GUI デザインを実現することができる．

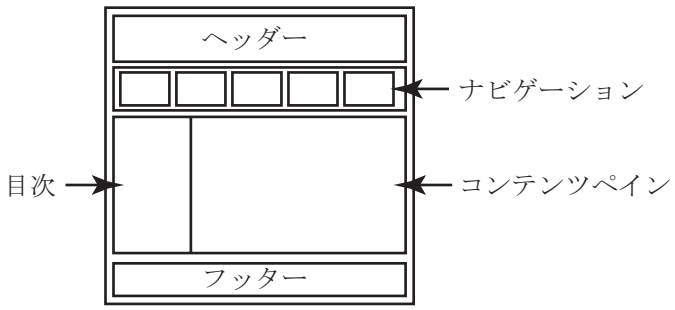


図 3: BoxLayout を組み合わせたデザインの例

利用できるレイアウトを表 11 に挙げる．

表 11: 利用できるレイアウト

クラス	機能
BoxLayout	水平／垂直のレイアウト
GridLayout	縦横（2 次元）のグリッド配置
StackLayout	水平あるいは垂直方向に順番に追加される配置
AnchorLayout	片寄せ，中揃え（均等配置）の固定位置
FloatLayout	直接位置指定（絶対，相対）
RelativeLayout	直接位置指定（画面の位置：絶対，相対）
PageLayout	複数ページの切り替え形式
ScatterLayout	移動，回転などを施すためのレイアウト

3.1.2.3　Screen（スクリーン）

スクリーンにはレイアウトやウィジェットを配置することができ，1 つのスクリーンは 1 つの操作パネルと見ることができる．更に複数のスクリーンをスクリーンマネジャ（Screen は ScreenManager）と呼ばれるオブジェクト配下に設置することができ，それらスクリーンを切り替えて表示することができる．

スクリーンマネジャを用いて構築された GUI は，いわゆるプレゼンテーションスライドのように動作し，各スクリーンを遷移して切り替えること（transition）で異なる複数のインターフェースを切り替えることができる．

Screen の扱いに関しては「3.9 GUI 構築の形式」のところで説明する．

3.1.3　ウィンドウの扱い

ウィンドウ（Window）は構築する GUI アプリケーション全体を表すオブジェクトであり，1 つのアプリケーションに 1 つのウィンドウオブジェクトが存在する．

ウィンドウオブジェクトを明に扱うには，次のようにして Window モジュールを読み込んでおく必要がある．

Window モジュールの読込み

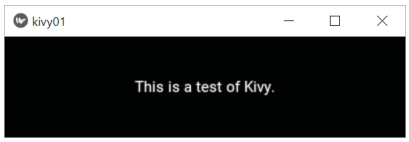
```
from kivy.core.window import Window
```

Window オブジェクトを用いると，例えばウィンドウサイズの調整などができる．先のプログラムを改変した kivy01-2.py を次に示す．

プログラム：kivy01-2.py

```
1  # coding: utf-8
2
3  # 基本となるアプリケーションクラス
4  from kivy.app import App
5  # ラベルオブジェクト
6  from kivy.uix.label import Label
7  # ウィンドウの操作に必要なもの
8  from kivy.core.window import Window
9
10 # アプリケーションのクラス
11 class kivy01(App):
12     def build(self):
13         self.lb1 = Label(text='This is a test of Kivy.')
14         return self.lb1
15
16 #---- メインルーチン -----
17 # ウィンドウサイズの設定（400×100）
18 Window.size = (400,100)
19 # アプリケーションのインスタンスを生成して起動
20 ap = kivy01()
21 ap.run()
```

このように Window オブジェクトの size プロパティを指定することで，アプリケーションのウィンドウサイズを変更することができる．（図 4 参照）



Window の size プロパティを 400 × 100 に設定している

図 4: Kivy01-2.py の実行結果

Window のプロパティとしてはこの他にも clearcolor もあり，これの値 [24] を設定することでウィンドウの色を変更することもできる．

3.2 基本的な GUI アプリケーション構築の方法

3.2.1 イベント処理（導入編）

GUI アプリケーションプログラムはユーザからの操作をはじめとする**イベント**の発生を受けて**イベントハンドラ**を起動する，いわゆる**イベント駆動型**のスタイルを基本とする．ここではサンプルプログラムの構築を通してイベント処理の基本的な実装方法について説明する．

[24] clearcolor の値は (R,G,B, α) のタプルで与える．

先に挙げたサンプルプログラムを更に改変して，ラベルオブジェクトに**タッチ**[25]が起こった際のイベント処理について説明する．

3.2.1.1 イベントハンドリング

ここでは，Label オブジェクトが touch_down（タッチの開始／ボタンの押下），touch_move（ドラッグ），touch_up（タッチの終了／ボタンを放す）といったイベントを受け付ける例を挙げて説明する．ウィジェットには，それらイベントを受け付けるためのメソッド（on_touch_down，on_touch_move，on_touch_up）が定義されており，プログラマはウィジェットの拡張クラスを定義して，それらメソッドをオーバーライドして，実際の処理を記述する．

プログラム：kivy01-3.py

```
1  # coding: utf-8
2
3  # 基本となるアプリケーションクラス
4  from kivy.app import App
5  # ラベルオブジェクト
6  from kivy.uix.label import Label
7  # ウィンドウの操作に必要なもの
8  from kivy.core.window import Window
9
10 # ラベルの拡張
11 class MyLabel(Label):
12     # タッチ開始（マウスボタンの押下）の場合の処理
13     def on_touch_down(self,touch):
14         print(self.events())
15         print('touch down: ',touch.spos)
16     # タッチの移動（ドラッグ）の場合の処理
17     def on_touch_move(self,touch):
18         print('touch move: ',touch.spos)
19     # タッチ終了（マウスボタンの開放）の場合の処理
20     def on_touch_up(self,touch):
21         print('touch up  : ',touch.spos)
22
23 # アプリケーションのクラス
24 class kivy01(App):
25     def build(self):
26         self.lb1 = MyLabel(text='This is a test of Kivy.')
27         return self.lb1
28
29 #---- メインルーチン -----
30 # ウィンドウサイズの設定
31 Window.size = (400,100)
32 # アプリケーションのインスタンスを生成して起動
33 ap = kivy01()
34 ap.run()
```

解説：

このプログラムでは，Label クラスを拡張した MyLabel クラスを定義し（11〜21 行目），そのクラスでイベントハンドリングをしている．このクラスではそれぞれのイベントに対応するメソッドの記述をしており，仮引数として記述された touch に，イベント発生時の各種情報を保持するオブジェクトが与えられる．

アプリケーション全体は，App クラスの拡張クラス Kivy01 クラスとして定義されている．(24〜27 行目)．14 行目にあるように，events メソッドを実行すると，そのオブジェクトが対象とするイベントの一覧情報が得られる．また，マウスやタッチデバイスのイベントが持つ spos プロバティには，イベントが発生した位置の情報が保持されている．

このプログラムを実行した際の標準出力の例を次に示す．

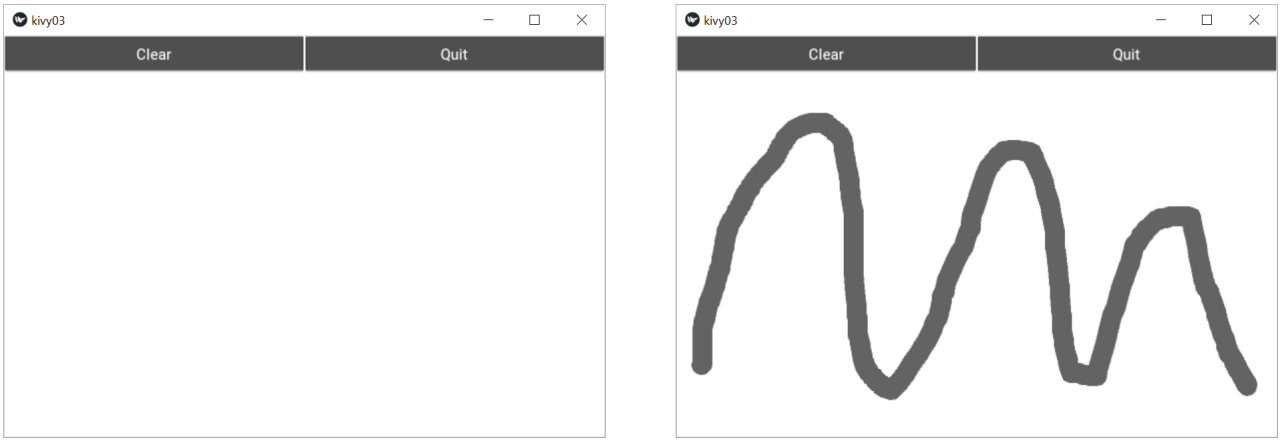
[25] パーソナルコンピュータの操作環境ではマウスのクリックがこれに相当する．Kivy はスマートフォンやタブレットコンピュータでの処理を前提にしており，タッチデバイスを基本にしたイベント処理となっている．

```
dict_keys([ón_ref_press, ón_touch_down, ón_touch_move, ón_touch_up])
touch down:  (0.0875, 0.45999999999999996)
touch move:  (0.0875, 0.48)
touch move:  (0.09, 0.48)
touch move:  (0.0925, 0.48)
touch move:  (0.095, 0.48)
touch move:  (0.0975, 0.48)
touch move:  (0.1, 0.48)
touch move:  (0.0975, 0.47)
touch up :  (0.0975, 0.47)
```

3.2.2 アプリケーション構築の例

ここでは，簡単な事例を挙げて，GUI アプリケーションプログラムの構築について説明する．

事例として示すアプリケーションは簡単な描画アプリケーションで，概観を図 5 に示す．このアプリケーションは，ウィンドウ下部の描画領域にタッチデバイスやマウスでドラッグした軌跡を描画する．またウィンドウ上部の「Clear」ボタンをタッチ（クリック）すると描画領域を消去し，「Quit」ボタンをタッチ（クリック）するとアプリケーションが終了する．



(a) 起動時　　(b) タッチデバイスやマウスで描画した例

図 5: kivy03.py の実行結果

この事例では，BoxLayout を使用してウィンドウ内のレイアウトを実現している．具体的には，水平方向の BoxLayout を用いて「Clear」と「Quit」の 2 つのボタンを配置し，そのレイアウトと描画領域を垂直方向の BoxLayout で配置している．BoxLayout を使用するには次のようにして必要なモジュールを読み込んでおく．

BoxLayout を使用するためのモジュールの読込み

```
from kivy.uix.boxlayout import BoxLayout
```

BoxLayout の配置方向（水平，垂直）の指定は，インスタンス生成時にコンストラクタの仮引数として**キーワード引数**「orientation=」を与える．この引数の値として'horizontal' を与えると水平方向，'vertical' を与えると垂直方向の配置となる．

ボタンウィジェット（Button）を使用するには次のようにして必要なモジュールを読み込んでおく．

Button を使用するためのモジュールの読込み

```
from kivy.uix.button import Button
```

ボタンウィジェットのトップに表示する文字列は，インスタンス生成時にコンストラクタの仮引数としてキーワード引数「text=」を与える．この引数の値として文字列を与えると，それがボタントップに表示される．

グラフィックスを描画するには，Wedget クラスを使用するのが一般的である．このクラスは多くのウィジェットの上位クラスであり，描画をするための canvas という要素を持つ．グラフィックスの描画はこの canvas に対して行う．Widget を使用するには次のようにして必要なモジュールを読み込んでおく．

Widget を使用するためのモジュールの読込み

```
from kivy.uix.widget import Widget
```

【サンプルプログラム】

今回の事例のプログラムを kivy03.py に挙げる．これを示しながら GUI アプリケーションプログラム構築の流れを説明する．

プログラム：kivy03.py

```
1  # coding: utf-8
2
3  #----- 必要なパッケージの読込み -----
4  import sys
5  from kivy.app import App
6  from kivy.uix.boxlayout import BoxLayout
7  from kivy.uix.button import Button
8  from kivy.uix.widget import Widget
9  from kivy.graphics import Color, Line
10 from kivy.core.window import Window
11
12 #----- 拡張クラスの定義 -----
13 # アプリケーションのクラス
14 class kivy03(App):
15     def build(self):
16         return root
17
18 # Clearボタンのクラス
19 class BtnClear(Button):
20     def on_release(self):
21         drawArea.canvas.clear()
22 # Quitボタンのクラス
23 class BtnQuit(Button):
24     def on_release(self):
25         sys.exit()
26
27 # 描画領域のクラス
28 class DrawArea(Widget):
29     def on_touch_down(self,t):
30         self.canvas.add( Color(0.4,0.4,0.4,1) ) # 描画色の設定
31         self.lineObject = Line( points=(t.x,t.y), width=10 )
32         self.canvas.add( self.lineObject )
33     def on_touch_move(self,t):
34         self.lineObject.points += (t.x,t.y)
35     def on_touch_up(self,t):
36         pass
37
38 #----- GUIの構築 -----
39 # ウィンドウの色とサイズ
40 Window.size = (600,400)
41 Window.clearcolor = (1,1,1,1)
42
```

```
43 # 最上位レイアウトの生成
44 root = BoxLayout(orientation='vertical')
45
46 # ボタンパネルの生成
47 btnpanel = BoxLayout(orientation='horizontal')
48 btnClear = BtnClear(text='Clear')    # Clearボタンの生成
49 btnQuit  = BtnQuit(text='Quit')      # Quitボタンの生成
50 btnpanel.add_widget(btnClear)        # Clearボタンの取り付け
51 btnpanel.add_widget(btnQuit)         # Quitボタンの取り付け
52 btnpanel.size_hint = ( 1.0 , 0.1 )  # ボタンパネルのサイズ調整
53 root.add_widget(btnpanel)   # ボタンパネルをメインウィジェットに取り付け
54
55 # 描画領域の生成
56 drawArea = DrawArea()
57 root.add_widget(drawArea)
58
59 #----- アプリケーションの実行 -----
60 ap = kivy03()
61 ap.run()
```

全体の概要：

このプログラムの 4〜10 行目で必要なモジュール群を読み込んでいる．必要となる各種のクラスの定義は 14〜36 行目に記述している．GUI を構成するための各種のインスタンスの生成は 44〜57 行目に記述しており，60〜61 行目にアプリケーションの実行を記述している．

BoxLayout へのウィジェットの登録

「Clear」「Quit」の 2 つのボタンは 48〜49 行目で生成しており，これらを add_widget メソッドを用いて水平配置の BoxLayout である btnpanel に登録している．同様の方法で，最上位の BoxLayout（垂直配置）である root に btnpane と描画領域のウィジェット drawArea を登録（53,57 行目）している．

※ 親ウィジェットに子ウィジェットを登録，削除する方については「3.2.4 ウィジェットの登録と削除」を参照のこと．

BoxLayout 配下に登録されたオブジェクトは均等な大きさで配置されるが，今回の事例では，ボタンの領域の高さを小さく，描画領域の高さを大きく取っている．このように，配置領域の大きさの配分を変えるには，BoxLayout の子の要素に対して size_hint プロバティを指定する．52 行目で実際にこれをしているが，本来均等となるサイズに対する比率を **(水平比率, 垂直比率)** の形で与える．

canvas に対する描画

Widget の canvas に対して描画するには Graphics クラスのオブジェクトを使用する．具体的には Graphics クラスのオブジェクトを canvas に対して add メソッドを用いて登録する．

canvas に対して登録できる Graphics オブジェクトには Line（折れ線），Rectangle（長方形），Ellipse（楕円）をはじめとする多くのものがある．また描画の色も Color オブジェクトを canvas に登録することで指定する．今回のプログラムでは 30 行目で Color オブジェクトを登録して描画色を指定している．また，31 行目で Line オブジェクトを登録し，34 行目でこれを更新することで描画している．

イベント処理と描画の流れ

今回のプログラムでは，描画領域のオブジェクト drawArea に対するイベント処理によって描画を実現している．具体的には DrawArea クラスの定義の中で，タッチが開始（マウスボタンのクリックが開始）したことを受けるイベントハンドラである on_touch_down メソッド，ドラッグしたことを受けるイベントハンドラである on_touch_move メソッド，タッチが終了（マウスボタンが開放）したことを受けるイベントハンドラである on_touch_up メソッドを記述することで描画処理を実現している．これらメソッドは 2 つの仮引数を取る．

まず，on_touch_down で描画の開始をするが，Color オブジェクトの登録による色の指定（30 行目）をして，Line オブジェクトを生成（31 行目）して登録（32 行目）している．このときはまだ Line オブジェクトは描画の開始点の座標のみを保持している．

次に，ドラッグが起こった際に on_touch_move で Line オブジェクトの座標を追加（34 行目）することで，実際の描画を行う．タッチやマウスの座標はメソッドの第 2 引数である t に与えられるオブジェクトに保持されている．このオブジェクトの x プロパティに x 座標の値が，y プロパティに y 座標の値がある．

canvas の消去

canvas オブジェクトに対して clear メソッドを実行することでそこに登録された Graphics オブジェクトを全て消去する．今回のプログラムでは，消去ボタンである btnClear オブジェクトのタッチ（クリック）を受けるイベント処理でこのメソッドを実行（21 行目）して描画を消去している．Button オブジェクトのタッチ（クリック）の開始と終了のイベントは，on_press，on_release メソッドで受ける．これらメソッドは 1 つの仮引数を取る．

アプリケーションの終了

今回のプログラムでは，終了ボタンである btnQuit オブジェクトに対するイベント処理でアプリケーションの終了を実現している．sys モジュールを読み込み，sys クラスのメソッド exit を呼び出すことでプログラムが終了する．

3.2.3 イベント処理（コールバックの登録による方法）

ウィジェットの拡張クラスを定義してイベントハンドリングのメソッドをオーバーライドする方法とは別に，bind メソッドを用いてイベントハンドリングする方法もある．

GUI のオブジェクトの生成後，そのオブジェクトに対して，

オブジェクト.bind(イベント=コールバック関数)

とすることで，bind メソッドの引数に指定したイベントが発生した際に，指定した**コールバック**関数を呼び出すことができる．この方法を採用することにより，イベント処理を目的とする拡張クラスの定義を省くことができる．

bind を用いてイベント処理を登録する形で先のプログラム kivy03.py を書き換えたプログラム kivy03-2.py を示す．

プログラム：kivy03-2.py

```
1  # coding: utf-8
2
3  #----- 必要なパッケージの読込み -----
4  import sys
5  from kivy.app import App
6  from kivy.uix.boxlayout import BoxLayout
7  from kivy.uix.button import Button
8  from kivy.uix.widget import Widget
9  from kivy.graphics import Color, Line
10 from kivy.core.window import Window
11
12 #----- 拡張クラスの定義 -----
13 # アプリケーションのクラス
14 class kivy03(App):
15     def build(self):
16         return root
17
18 #----- GUIの構築 -----
19 # ウィンドウの色とサイズ
20 Window.size = (600,400)
21 Window.clearcolor = (1,1,1,1)
22
23 # 最上位レイアウトの生成
24 root = BoxLayout(orientation='vertical')
25
26 # ボタンパネルの生成
27 btnpanel = BoxLayout(orientation='horizontal')
28 btnClear = Button(text='Clear') # Clearボタンの生成
29 btnQuit  = Button(text='Quit')      # Quitボタンの生成
```

```
30 btnpanel.add_widget(btnClear)        # Clearボタンの取り付け
31 btnpanel.add_widget(btnQuit)         # Quitボタンの取り付け
32 btnpanel.size_hint = ( 1.0 , 0.1 )  # ボタンパネルのサイズ調整
33 root.add_widget(btnpanel)   # ボタンパネルをメインウィジェットに取り付け
34
35 # 描画領域の生成
36 drawArea = Widget()
37 root.add_widget(drawArea)
38
39 #----- コールバックの定義と登録 -----
40 # 描画領域を消去する関数
41 def callback_Clear(self):
42     drawArea.canvas.clear()
43     drawArea.canvas.add( Color(0.4,0.4,0.4,1) )
44     drawArea.lineObject = Line(points=[],width=10)
45     drawArea.canvas.add( drawArea.lineObject )
46 btnClear.bind(on_release=callback_Clear)        # ボタンへの登録
47
48 # アプリケーションを終了する関数
49 def callback_Quit(self):
50     sys.exit()
51 btnQuit.bind(on_release=callback_Quit)      # ボタンへの登録
52
53 # 描画のための関数
54 def callback_drawStart(self,t):
55     self.canvas.add( Color(0.4,0.4,0.4,1) )
56     self.lineObject = Line( points=(t.x,t.y), width=10 )
57     self.canvas.add( self.lineObject )
58 def callback_drawMove(self,t):
59     self.lineObject.points += (t.x,t.y)
60 def callback_drawEnd(self,t):
61     pass
62 drawArea.bind(on_touch_down=callback_drawStart)
63 drawArea.bind(on_touch_move=callback_drawMove)
64 drawArea.bind(on_touch_up=callback_drawEnd)
65
66
67 #----- アプリケーションの実行 -----
68 ap = kivy03()
69 ap.run()
```

全体の概要：

GUI 構築の部分は概ね kivy03.py と同じであるが，イベント処理のためのコールバック関数の定義と各ウィジェットへの登録が，41〜64 行目に記述されている．

イベントハンドリングには，拡張クラスを定義してイベントハンドラをオーバーライドする方法と，bind メソッドによるコールバック関数の登録の 2 種類の方法があるが，プログラムの可読性などを考慮してどちらの方法を採用するかを検討するのが良い．

3.2.4 ウィジェットの登録と削除

ウィジェットは親の要素に子の要素を登録する方法で階層的関係を構築する．親のウィジェット wp に子のウィジェット wc を登録するには add_widget メソッドを用いて，

```
wp.add_widget(wc)
```

と実行する．また逆に wp から wc を削除するには remove_widget を用いて，

```
wp.remove_widget(wc)
```

と実行する．

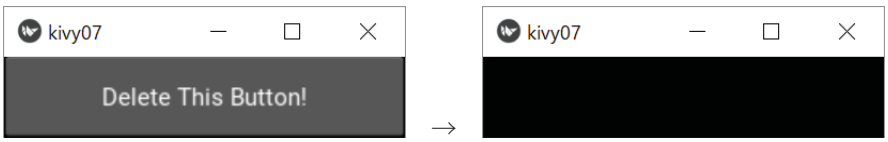
次のプログラム kivy07.py は，一旦登録したウィジェトを取り除く処理を示すものである．

プログラム：kivy07.py

```
# coding: utf-8

#----- 必要なパッケージの読込み -----
from kivy.app import App
from kivy.uix.anchorlayout import AnchorLayout
from kivy.uix.button import Button
from kivy.core.window import Window

#----- 最上位のウィジェット -----
root = AnchorLayout()

#----- ボタンの生成 -----
btn1 = Button(text='Delete This Button!')

# 最上位のウィジェットからボタンを取り除く処理（コールバック関数）
def delbtn(self):
    root.remove_widget(btn1)

# ボタンへのコールバックの登録
btn1.bind(on_release=delbtn)

#----- 最上位のウィジェットにボタンを登録 -----
root.add_widget(btn1)

#----- アプリケーションの実装 -----
class kivy07(App):
    def build(self):
        return root

Window.size=(250,50)
kivy07().run()
```

解説

このプログラムは最上位の AnchorLayout にボタン を 1 つ登録するものであり，そのボタンをクリック（タッチ）すると，そのボタン自身を取り除く．プログラムの実行例を図 6 に示す．



(a) ボタンをクリックすると　　(b) そのボタンが除去される

図 6: ウィジェット（ボタン）を取り除く処理

3.2.5　アプリケーションの開始と終了のハンドリング

Kivy アプリケーションの実行が開始する時と終了する時のハンドリングは，アプリケーションクラスの on_start メソッドと on_stop メソッドでそれぞれ行う．この様子を次のサンプルプログラム kivy08.py で確認できる．

プログラム：kivy08.py

```
# coding: utf-8
#----- 必要なパッケージの読込み -----
from kivy.app import App
from kivy.uix.label import Label
from kivy.core.window import Window

#----- 最上位のウィジェット -----
root = Label(text='This is a sample.')

#----- アプリケーションの実装 -----
class kivy08(App):
```

```
12
13      # アプリケーションのインスタンス生成
14      def build(self):
15          print('0) アプリケーションのインスタンスが生成されました.')
16          return root
17
18      # アプリケーション実行開始時
19      def on_start(self):
20          print('1) アプリケーションの実行が開始されました.')
21
22      # アプリケーション終了時
23      def on_stop(self):
24          print('2) アプリケーションを終了します.')
25
26  Window.size=(250,50)
27  kivy08().run()
```

このプログラムを実行すると図 7 のようなウィンドウが表示される.

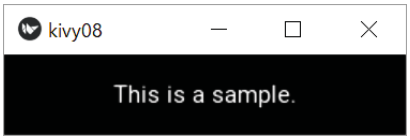


図 7: Kivy08.py のアプリケーションウィンドウ

アプリケーションの生成時，開始時，終了時にそれぞれメッセージを標準出力に出力する.（次の例参照）

```
0) アプリケーションのインスタンスが生成されました.      ←生成時
1) アプリケーションの実行が開始されました.              ←開始時
2) アプリケーションを終了します.                        ←終了時
```

3.3 各種ウィジェットの使い方

ここでは使用頻度の高い代表的なウィジェットの使い方について説明する．また，ここでは基本的な使用方法について説明する.

3.3.1 ラベル：Label

GUI に文字を表示する場合に標準的に用いられるのがラベル（Label）オブジェクトである．Label のインスタンスを生成する際，コンストラクタの引数に text=' 文字列' とすると，与えた文字列を表示するラベルが生成される．図 8 は `Label(text='This is a test for Label object.')` として生成したラベルを表示した例である.

図 8: Label オブジェクトの表示

ラベルに表示する文字のフォントを指定することができる．特に現在の版の Kivy（1.9.2）は，フォントを指定せずに日本語の文字列を表示することができないので，日本語文字列を表示するばあいにはフォント指定が必須である.

図 9 は Windows 環境で

```
Label( text='日本語を MS ゴシックで表示するテスト',
    font_name='C:￥Windows￥Fonts￥msgothic.ttc',
    font_size='24pt')
```

として，MS ゴシックフォントを使用して表示した例である．この書き方の中にある font_name はフォントのパスを，font_size はフォントのサイズを指定するものである．

図 9: Label オブジェクトの表示（MS ゴシックによる日本語表示）

使用するフォントによっては，Kivy 独自の**マークアップ**を使用して，強調，斜体などのスタイルを施すことができる．マークアップは与える文字列の中に直接記述できる．図 10 は，

```
Label( text='[u] 日本語を IPA 明朝で表示するテスト（下線付き）[/u]',
    markup=True, color=(0.6,1,1,1), font_name='C:￥Windows￥Fonts￥ipam.ttf',
    font_size='20pt')
```

として，IPA 明朝フォントを使用して下線（アンダーライン）を施した例である．
マークアップを使用する場合は markup=True を指定する．またこの例のようにフォントの色を指定する際は

```
color=(R,G,B, α)
```

と指定する．

マークアップは '[...]〜[/...]' で括るタグを使用する．代表的なマークアップには

［b］文字列［/b］　強調文字
［u］文字列［/u］　下線（アンダーライン）
［i］文字列［/i］　斜体（イタリック）

といったものがある．

図 10: Label オブジェクトの表示（IPA 明朝による日本語表示：下線付き）

Label のコンストラクタに与える代表的なキーワード引数：
text=' 文字列'
font_name=' フォントのパス'
font_size=' フォントサイズ'
color=［赤，緑，青，α（不透明度）］　全て 0〜1 の値
markup=True　マークアップを使用する場合

3.3.1.1　リソースへのフォントの登録

フォントファイルが収められているディレクトリのパスを Kivy のリソース（resource）に登録しておくと font_name の指定において，フォントのファイル名のみの記述で済む．フォントファイルのディレクトリをリソースに登録するには，resource_add_path メソッドを使用する．このメソッドを使用するためには次のようにして必要なモジュールを読み込んでおく．

```
from kivy.resources import resource_add_path
```

この後 resource_add_path メソッドを実行する．

＜フォントパスのリソースへの登録＞

resource_add_path(フォントディレクトリのパス)

例えば Windows 環境では，

```
resource_add_path('C:￥Windows￥Fonts')
```

などとする．

これに加えて DEFAULT_FONT の設定をしておくと，font_name の設定を省略した際のデフォルトフォントを指定できる．デフォルトフォントの設定には LabelBase クラスの register メソッドを使用する．これを行うには次のようにして必要なモジュールを読み込んでおく．

```
from kivy.core.text import LabelBase, DEFAULT_FONT
```

この後デフォルトフォントを設定する．

＜デフォルトフォントの設定＞

LabelBase.register(DEFAULT_FONT, フォントファイル名)

例えば Windows 環境で IPA ゴシックフォントをデフォルトフォントにするには，

```
LabelBase.register(DEFAULT_FONT,'ipag.ttf')
```

などとする．

3.3.2 ボタン：Button

Button はボタンを実現するクラスである．ボタントップに表示する文字列の扱いについては Label の場合とほぼ同じであるが，下記のようにボタンのタッチ（クリック）によるイベント処理ができる点が特徴である．

タッチ（クリック）のイベント：

on_press	タッチ（クリック）の開始
on_release	タッチ（クリック）の終了

これらのイベントハンドリングには引数が 1 つ（自オブジェクト：self）与えられる．
bind メソッドでコールバック関数を登録する場合は，

ボタンオブジェクト.bind(on_press=コールバック関数)
ボタンオブジェクト.bind(on_release=コールバック関数)

とする．コールバック関数の仮引数は 1 つである．

3.3.3 テキスト入力：TextInput

TextInput は文字列の入力，編集をする場合に使用する．フォントやフォントサイズの設定は Label の場合と同じである．

テキストの入力や変更の際に起こるイベント

TextInput オブジェクト内でテキストの新規入力や変更があった場合，それがイベントとして発生する．TextInput の拡張クラスを定義する際はそれを on_text メソットとしてハンドリングする．また TextInput オブジェクトに対して bind メソッドでコールバック関数を登録する際は

オブジェクト.bind(text=コールバック関数)

とする．

on_text メソッドでイベントハンドリングする際は仮引数を 3 つ取り，コールバック関数でハンドリングする際は仮引数を 2 つ取る．両方の場合において，第 1 引数には，そのオブジェクト自身（self）が与えられる．

入力されているテキストの文字列は，プロパティ text に保持されている．

3.3.4 チェックボックス：CheckBox

チェックボックスのイベントハンドリングは基本的にボタンと同様（on_press,on_release）である．チェックされているか否かはチェックボックスオブジェクトのプロパティactive に真理値（True/False）として与えられる．

3.3.5 進捗バー：ProgressBar

進捗バーは値の大きさを水平方向に可視化するものである．値のプロパティは value である．
可視化範囲の最大値はプロバティmax に設定する．

3.3.6 スライダ：Slider

スライダは縦あるいは横方向にスライドするウィジェットであり，視覚的に値を調整，入力する際に用いる．値のプロパティは value である．このクラスのインスタンスを生成する際，キーワード引数 orientation= を与えることで，縦横のスタイルを選択できる．この引数の値に 'horizontal' を与えると横方向，'vertical' を与えると縦方向になる．(デフォルトは水平)

スライダの値の変更に伴って起こるイベント

Slider ブジェクトを操作（値を変更）した場合，それがイベントとして発生する．Slider の拡張クラスを定義する際はそれを on_value メソッドとしてハンドリングする．また Slider オブジェクトに対して bind メソッドでコールバック関数を登録する際は

オブジェクト.bind(value=コールバック関数)

とする．

on_value メソッドでイベントハンドリングする際は仮引数を 3 つ取り，コールバック関数でハンドリングする際は仮引数を 2 つ取る．両方の場合において，第 1 引数には，そのオブジェクト自身（self）が与えられる．

3.3.7 スイッチ：Switch

スイッチは水平方向の「切り替えスイッチ」で ON/OFF の 2 つの状態を取り，それぞれの状態はプロパティ active に真理値（True/False）として与えられる．(デフォルトは OFF)

スイッチの切り替えに伴って起こるイベント

Switch ブジェクトを操作（値を変更）した場合，それがイベントとして発生する．Switch の拡張クラスを定義する際はそれを on_active メソッドとしてハンドリングする．また Switch オブジェクトに対して bind メソッドでコールバック関数を登録する際は

オブジェクト.bind(active=コールバック関数)

とする．

on_active メソッドでイベントハンドリングする際は仮引数を 3 つ取り，コールバック関数でハンドリングする際は仮引数を 2 つ取る．両方の場合において，第 1 引数には，そのオブジェクト自身（self）が与えられる．

3.3.8　トグルボタン：ToggleButton

トグルボタンは「ラジオボタン」として知られる GUI と同様の働きをする．すなわち，複数のボタンを 1 つの「グループ」としてまとめ，同一のグループ内のボタンの内，1 つだけが ON（チェック済みもしくはダウン）になるウィジェットである．イベントハンドリングは基本的に Button クラスと同様であるが，プロパティ state に押されていない状態を意味する 'normal' か，押されている状態を意味する 'down' が保持されている．

トグルボタンの例

次のように 3 つのトグルボタン tb1，tb2，tb3 を生成した例について考える．

```
tb1 = ToggleButton(group='person',text='nakamura',state='down')
tb2 = ToggleButton(group='person',text='tanaka')
tb3 = ToggleButton(group='person',text='itoh')
```

これらを BoxLayout で水平に配置した例が図 11 である．

図 11: トグルボタンの例

トグルボタン生成時のコンストラクタに，キーワード引数 grouup= を与えることで複数のトグルボタンをグループ化することができる．またグループ内の 1 つのトグルボタンに state='down' を指定することで，初期状態で押されているトグルボタンを決めることができる．

3.3.9　画像：Image

Image は画像を扱うためのウィジェットクラスである．そのインスタンスに画像ファイルを読み込むことで画像を配置することができる．インスタンス生成時のコンストラクタにキーワード引数 source= を指定することで画像ファイルを読み込む．

＜ Image オブジェクト＞
Image(source=' 画像ファイル名')
指定したファイルから画像を読み込む．

Image オブジェクトの texture_size プロパティに読み込んだ画像のピクセルサイズが保持されている．

画像の表示サイズに関しては，それを配置するレイアウトオブジェクトに制御を委ねるのが一般的である．次に示すサンプルプログラムは，スライダに連動して画像の表示サイズが変わるものである．

3.3.9.1　サンプルプログラム

スライダの値によって画像の表示サイズが変わるプログラム kivy02Earth.py を示す．拡大，縮小する画像を常に中央に表示するため，Image オブジェクトを AnchorLayout で配置している．

プログラム：kivy02Earth.py

```
1 # coding: utf-8
2
3 #----- モジュールの読込み -----
```

```
4  from kivy.app import App
5  from kivy.uix.boxlayout import BoxLayout
6  from kivy.uix.anchorlayout import AnchorLayout
7  from kivy.uix.image import Image
8  from kivy.uix.slider import Slider
9  from kivy.core.window import Window
10
11 #----- GUIとアプリケーションの定義 -----
12
13 # イベントのコールバック
14 def onValueChange(self,v):
15     im.size_hint = ( sl.value, sl.value )
16
17 # GUIの構築
18 root = BoxLayout(orientation='vertical')
19
20 # 画像とそのレイアウト
21 anchor = AnchorLayout()
22 im = Image(source='Earth.jpg')
23 anchor.add_widget(im)
24
25 # スライダ
26 sl = Slider(min=0.0,max=1.0,step=0.01)
27 sl.size_hint = ( 1.0 , 0.1 )
28 sl.value = 1.0
29 sl.bind(value=onValueChange)
30
31 root.add_widget(anchor)
32 root.add_widget(sl)
33
34
35 # アプリケーションのクラス
36 class kivy02(App):
37     def build(self):
38         return root
39
40 # ウィンドウの色とサイズ
41 Window.size = (500,550)
42
43 # アプリ起動
44 print(im.texture_size)  # 画像サイズの調査
45 ap = kivy02()
46 ap.run()
```

全体の概要：

最上位のレイアウト（垂直の BoxLayout）である root は，画像表示部とスライダを収めるものである．Image オブジェクト im は AnchorLayout オブジェクト anchor に収められており，その内部で im の size_hint を調整することで表示サイズを調節している．

このプログラムを実行した様子を図 12 に示す．

3.4 Canvas グラフィックス

ウィジェット（Widget）には canvas 要素があり，これに対してグラフィックスの描画ができる．先の「3.2.2 アプリケーション構築の例」でも少し説明した通り，canvas に対して色や図形などを追加することで描画ができる．

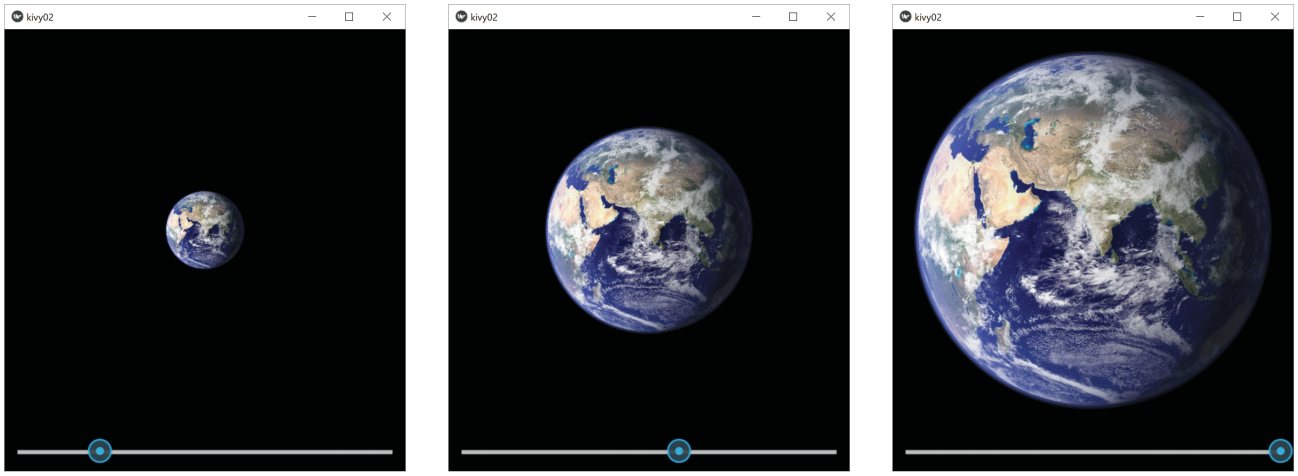
Widget を使用するには，必要なモジュールを，

```
from kivy.uix.widget import Widget
```

として読込み，これの canvas に対して Graphics を描画する．基本的な Graphics オブジェクトには `Color`,`Line`,`Rectangle`, `Ellipse` があり，それらを使用する際は `from kivy.graphics` からインポートする．

Graphics モジュールを読み込む例

```
from kivy.graphics import Color, Line, Rectangle, Ellipse
```



スライダに連動して画像の表示サイズが変わる

図 12: kivy02Earth.py の実行例

こうすることで，複数の Graphics のモジュールを読み込むことができる．

3.4.1 Graphics クラス

以下に紹介する Graphics オブジェクトを canvas に対して add メソッドで登録して描画する．
Kivy の座標系は多くの GUI ライブラリと異なり，左手系（図 13）である．

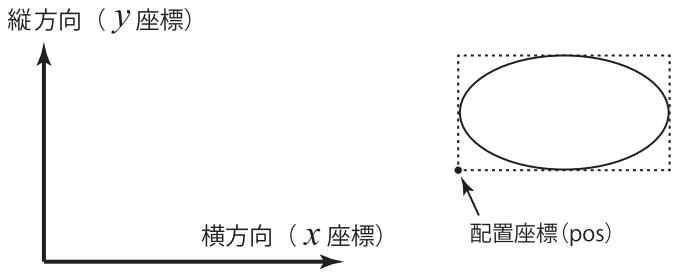


図 13: Kivy の座標系

また Kivy 以外の多くの GUI ライブラリはオブジェクトの配置を決める際，オブジェクトの左上の位置を指定するが，Kivy では Graphics オブジェクトの左下の位置を配置座標（pos の値）とする．

■ Color

描画色を指定するためのオブジェクトのクラスである．

コンストラクタ： Color(Red, Green, Blue, Alpha)
引数は全て 0〜1.0 の範囲の値である．Alpha は不透明度を指定するもので，1.0 を指定すると，完全に不透明になる．

■ Line

折れ線を描画するためのオブジェクトのクラスである．

コンストラクタ： Line(points=座標リスト, width=線幅)
canvas 上の座標，$(x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n)$ を結ぶ折れ線を，線の幅 width で描画する．描画座標リストは $[x_1, y_1, x_2, y_2, \cdots, x_n, y_n]$ とする．

■ Rectangle

長方形領域を描画するためのオブジェクトのクラスである．canvas 上に画像を表示する場合にも Rectangle を使用する．

コンストラクタ 1： Rectangle(pos=(描画位置の座標), size=(横幅, 高さ))
canvas 上の pos で指定した座標に，size で指定したサイズの長方形を描く．
コンストラクタ 1： Rectangle(pos=(描画位置の座標), size=(横幅, 高さ),
texture=テクスチャオブジェクト)
canvas 上の pos で指定した座標に，size で指定したサイズでテクスチャオブジェクトを描く．

テクスチャは canvas に画像を表示する際の標準的なオブジェクトであり，これの使用に際しては，下記のようにして必要なモジュールを読み込んでおく．

```
from kivy.graphics.texture import Texture
```

先に，「3.3.9 画像：Image」で画像を読み込んでほじするウィジェットである Image について説明したが，テクスチャオブジェクトはこの Image オブジェクトから texture プロパティとして取り出すことができる．

■ Ellipse

楕円を描画するためのオブジェクトのクラスである．

コンストラクタ： Ellipse(pos=(描画位置の座標), size=(横幅, 高さ))
canvas 上の pos で指定した座標に，size で指定したサイズの楕円を描く．

この他にも Bezier オブジェクトもあり，多角形や曲線を描くことができる，

3.4.2 サンプルプログラム

■ 正弦関数のプロット

canvas グラフィックスを使うと，簡単に数学関数の軌跡がプロットできる．

プログラム：kivy04-1.py

```
1  # coding: utf-8
2
3  #----- 必要なパッケージの読込み -----
4  import math
5  from kivy.app import App
6  from kivy.uix.widget import Widget
7  from kivy.graphics import Color, Rectangle
8  from kivy.core.window import Window
9
10 #----- アプリケーションの構築 -----
11 class kivy04(App):
12     def build(self):
13         return root
14
15 root = Widget()
```

```
16
17 root.canvas.add(    Color(1,0,0,1)  )
18 x = 0.0
19 while x < 6.28:
20     y = 100.0*math.sin(x)
21     root.canvas.add( Rectangle( pos=(32.0*x+5.0,y+105.0), size=(3,3) ) )
22     x += 0.005
23
24 # アプリの実行
25 Window.size = (210,210)
26 kivy04().run()
```

このプログラムの実行結果を図 14 に示す.

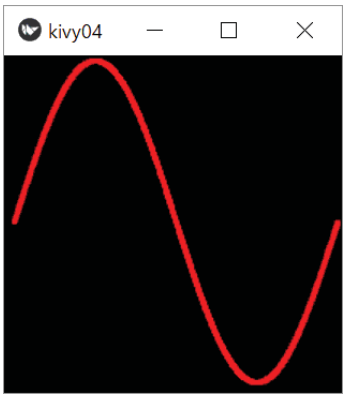


図 14: kivi04-1.py の実行結果

■ 各種図形，画像の表示

長方形，楕円，折れ線，テクスチャ画像を表示するプログラム kivy04-2.py を示す.

プログラム：kivy04-2.py

```
1  # coding: utf-8
2  #----- 必要なパッケージの読込み -----
3  from kivy.app import App
4  from kivy.uix.widget import Widget
5  from kivy.uix.image import Image
6  from kivy.graphics import Color, Line, Rectangle, Ellipse
7  from kivy.core.window import Window
8
9  #----- アプリケーションの構築 -----
10 class kivy04(App):
11     def build(self):
12         return root
13
14 # 画像の読込みとテクスチャの取り出し
15 im = Image(source='Earth.jpg')
16 tx = im.texture
17
18 root = Widget()
19 # 長方形の描画
20 root.canvas.add( Color(1,0,0,1) )
21 root.canvas.add( Rectangle(pos=(10,210),size=(350,100)) )
22 # 画像の描画
23 root.canvas.add( Color(1,1,1,1) )
24 root.canvas.add( Rectangle(pos=(0,0),texture=tx,size=(210,200)) )
25 # 楕円の描画
26 root.canvas.add( Color(0,1,0,1) )
27 root.canvas.add( Ellipse(pos=(200,10),size=(170,170)) )
28 # 折れ線の描画
29 root.canvas.add( Color(1,1,1,1) )
30 root.canvas.add( Line(width=10,points=[580,25,400,25,580,160,400,290,580,290]) )
31
32 # スクリーンショット
```

```
33 def save_shot(self,t):
34     # 方法-1
35     Window.screenshot(name='kivy04-2-1.png')
36     # 方法-2
37     self.export_to_png('kivy04-2-2.png')
38
39 root.bind(on_touch_up=save_shot)
40
41 # アプリの実行
42 Window.size = (600,330)
43 kivy04().run()
```

このプログラムの実行結果を図 15 に示す.

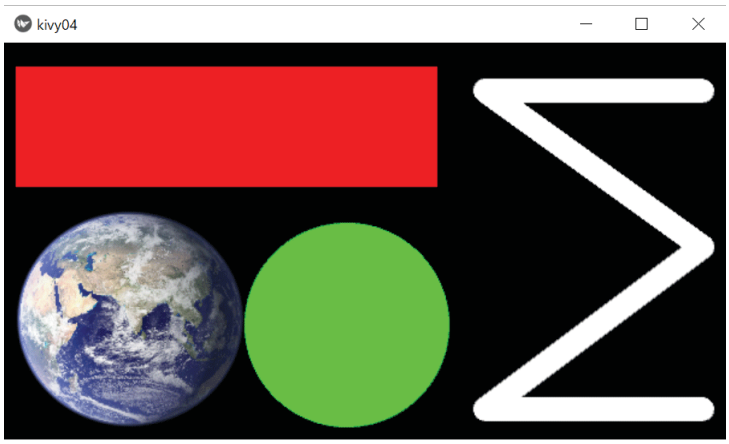


図 15: kivi04-2.py の実行結果

プログラムの 33〜37 行目で Window のスクリーンショットの画像を保存する機能を実装している.

```
Window.screenshot(name='kivy04-2-1.png')
```

としている部分がスクリーンショットのメソッドの 1 つであり，その時点のウィンドウの様子を png 形式の画像として保存する．また,

```
self.export_to_png('kivy04-2-2.png')
```

としている部分も同じ処理を実現するもので，これは Widget の表示内容を PNG 形式の画像として保存するものである.（Widget に対する export_to_png メソッド）

3.4.3 フレームバッファへの描画

Canvas への描画とは別に，**フレームバッファ**（FBO）[26] と呼ばれる 固定サイズの領域 への描画が可能である．フレームバッファからは**ピクセル値の取り出し**が可能である.

FBO を使用するには次のようにして必要なモジュールを読み込んでおく.

```
from kivy.graphics import Fbo
```

FBO の生成時には，描画サイズを指定する.

例. FBO の生成

```
f = Fbo( size=(400,300) )
```

これで画素サイズ 400 × 300 の FBO が f として生成される．FBO は描画対象の Widget の Canvas に登録しておく必要がある.

[26] OpenGL の描画フレーム

例. Widget オブジェクト root への FBO オブジェクト f の登録

```
root.canvas.add( f )
```

更にこの後，FBO の texture プロパティを与えた Rectangle を Canvas に描画することで実際に FBO の内容が表示される．

FBO への描画は Canvas への描画とほぼ同じ方法（add メソッド）が使用できる．

FBO を用いて，タッチした場所の画素の値（ピクセル値）を取得するプログラム kivy04-3.py を次に示す．

プログラム：kivy04-3.py

```
1  # coding: utf-8
2  #----- 必要なパッケージの読込みとモジュールの初期設定 -----
3  from kivy.config import Config
4  Config.set('graphics','resizable',False)    # ウィンドウリサイズの禁止
5  from kivy.app import App
6  from kivy.uix.widget import Widget
7  from kivy.graphics import Color, Rectangle, Fbo
8  from kivy.core.window import Window
9
10 #----- アプリケーションの構築 -----
11 class kivy04(App):
12     def build(self):
13         return root
14
15 # 最上位ウィジェットの生成
16 root = Widget()
17
18 # フレームバッファの生成とCanvasへの登録
19 fb = Fbo(size=(300, 150))
20 root.canvas.add( fb )
21
22 # フレームバッファへの描画
23 fb.add( Color(1, 0, 0, 1) )
24 fb.add( Rectangle(pos=(0,0),size=(100, 150)) )
25 fb.add( Color(0, 1, 0, 1) )
26 fb.add( Rectangle(pos=(100,0),size=(100, 150)) )
27 fb.add( Color(0, 0, 1, 1) )
28 fb.add( Rectangle(pos=(200,0),size=(200, 150)) )
29 fb.add( Color(1,1,1,1) )
30 fb.add( Rectangle(pos=(0,75),size=(300,75)) )
31 # フレームバッファの内容をCanvasに描画
32 root.canvas.add( Rectangle(size=(300, 150), texture=fb.texture) )
33
34 # タッチ位置の色の取得（コールバック関数）
35 def pickColor(self,t):
36     # ウィンドウサイズの取得
37     (w,h) = Window.size
38     # タッチ座標の取得
39     (x,y) = t.spos
40     # フレームバッファ上での座標
41     fbx = int(w*x);     fby = int(h*y)
42     # フレームバッファ上のピクセル値の取得
43     c = fb.get_pixel_color(fbx,fby)
44     print( '位置:\t',(fbx,fby),'\tピクセル:',c )
45
46 # コールバックの登録
47 root.bind(on_touch_up=pickColor)
48
49 # アプリの実行
50 Window.size = (300,150)
51 kivy04().run()
```

解説

19〜20 行目で FBO を生成して Widget の Canvas に登録している．23〜30 行目で FBO に対して描画し，それを 32 行目で Rectangle として Canvas に描画している．

アプリケーションのウィンドウ内をタッチ（クリック）するとコールバック関数 pickColor が呼び出され，その位置のピクセルを 43 行目で取得している．

3.4.3.1　ピクセル値の取り出し

FBO のピクセル値を取り出すには get_pixel_color メソッドを使用する．

書き方　FBO オブジェクト.get_pixel_color(**横位置, 縦位置**)

ピクセル値を取り出す横位置と縦位置は，画像左下を基準とするピクセル位置である．得られたピクセル値は

[**赤, 緑, 青, α**]

のリストであり，各要素は 0〜255 の整数値である．

プログラム kivy04-3.py を実行すると図 16 のようなウィンドウが表示される．

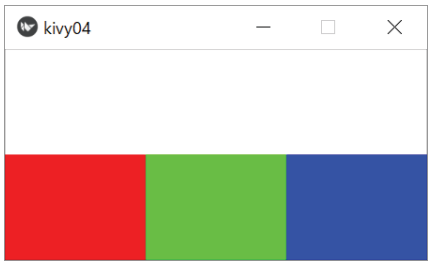


図 16: kivy04-3.py の実行結果

このウィンドウ内をタッチ（クリック）すると，次の例のように，その位置のピクセル値を表示する．

```
位置:  (143, 117)  ピクセル:  [255, 255, 255, 255]
位置:  (52, 39)    ピクセル:  [255, 0, 0, 255]
位置:  (134, 34)   ピクセル:  [0, 255, 0, 255]
位置:  (256, 24)   ピクセル:  [0, 0, 255, 255]
```

3.4.3.2　イベントから得られる座標位置

Kivy のウィジェット上のイベントから取得される座標位置は，ウィジェットサイズの縦横を共に 1.0 に正規化した位置であり，基準は左下である．このため，先のプログラム kivy04-3.py では，37〜41 行目にあるように FBO 上の座標位置を得るための変換処理をしている．

3.5　スクロールビュー（ScrollView）

大きなサイズのウィジェットやレイアウトを，それよりも小さなウィジェットやウィンドウの内部でスクロール表示する場合は**スクロールビュー**（ScrollView）を使用する．これは**スクロールバー**を装備した矩形領域であり，内部のオブジェクト（子要素）を縦横に並行移動して表示するものである．

スクロールビューの使用に際して，次のようにして必要なモジュールを読み込む．

```
from kivy.uix.scrollview import ScrollView
```

スクロールビューは次のようにしてインスタンスを生成し，基本的にはウィジェットの 1 つとして扱う．

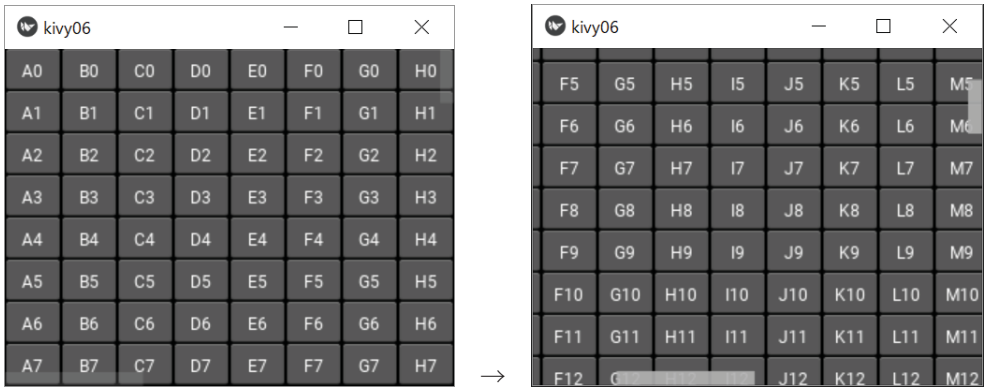
コンストラクタ：　`ScrollView()`

コンストラクタにキーワード引数「bar_width=**幅**」を与えることで，スクロールバーの幅を設定することができる．

ここではサンプルプログラムを示しながらスクロールビューの使用方法について説明する．

【サンプルプログラム】

図 17 のような，縦横にたくさんのボタンが配置されたウィジェットをスクロール表示するアプリケーションプログラム kivy06.py を考える．



(a) 多量のボタンが並んでいる　(b) マウスやタッチデバイスでスクロール

図 17: 縦横に並んだボタンパネルのスクロール

基本的な考え方：

スクロールビューよりも大きなサイズのウィジェットを，スクロールビューに子要素として登録する．次に示すプログラム kivy06.py では，多量のボタンを配置した巨大な BoxLayout を生成して，それをスクロールビューに登録している．

プログラム：kivy06.py

```
1  # coding: utf-8
2
3  ###### 必要なパッケージの読込み ##################################
4  from kivy.app import App
5  from kivy.uix.boxlayout import BoxLayout
6  from kivy.uix.button import Button
7  from kivy.uix.scrollview import ScrollView
8  from kivy.core.window import Window
9
10 ###### GUIの構築 ################################################
11 # ウィンドウのサイズ
12 Window.size = (320,240)
13
14 # 最上位ウィジェット（スクロールビュー）の生成
15 root = ScrollView( bar_width=10 )
16
17 # 大きいボックスレイアウトの作成
18 bx = BoxLayout( orientation='vertical',
19         size_hint=(None, None),     # サイズに関して親の制御を受けない設定
20         size=(1040,1500) )          # 1040×1500の固定サイズ
21
22 # A0～Z49 のボタンを生成（1300個）
23 al = [chr(i) for i in range(65, 65+26)]     # アルファベットのリスト
24 # 0～49行のボタン配列を生成
25 for m in range(50):
26     bx2 = BoxLayout( orientation='horizontal' )
27     # An～Znの横方向のボタン生成（26個）
28     for n in al:
29         # 40×30のサイズのボタンを生成
30         bx2.add_widget( Button(text=n+str(m), font_size=12,
31                     size_hint=(None, None), size=(40,30) ) )
32     bx.add_widget(bx2)
33 root.add_widget(bx)
34
```

```
35 ###### アプリケーションの構築 ####################################
36
37 # アプリケーションのクラス
38 class kivy06(App):
39     def build(self):
40         return root
41
42 # アプリケーションの実行
43 kivy06().run()
```

解説

18〜20 行目で生成した大きなサイズの BoxLayout である bx に多量のボタンを配置して，それを，15 行目で生成したスクロールビュー root に 33 行目で登録している．

25〜32 行目は多量のボタンを生成している部分である．An〜Zn（n は整数値）の 26 個のボタンを生成して，それらを水平方向（行）の BoxLayout である bx2 に登録して，それを垂直方向の BoxLayout である bx に次々と登録している．

3.5.1 ウィジェットのサイズ設定

通常の場合は，階層的に構築されたウィジェット群のサイズは自動的に調整される．これは，親ウィジェットに収まるように子ウィジェットのサイズを調整するという Kivy の機能によるものであるが，先のプログラム kivy06.py では，ウィジェットサイズの自動調整の機能に任せることなく，大きなサイズのウィジェット（1,040 × 1,500 のサイズの BoxLayout）を生成している．これは，ウィジェット生成時のコンストラクタにキーワード引数

size_hint=(None, None)

を与えることで可能となる．これと同時に，コンストラクタにキーワード引数

size=(**横幅, 高さ**)

を与えて，具体的なサイズを設定する．

3.6 ウィンドウサイズを固定（リサイズを禁止）する設定

ウィンドウのリサイズを禁止（ウィンドウサイズを固定）するには，Kivy モジュールを読み込む先頭の位置で，次のように設定する．

```
from kivy.config import Config
Config.set('graphics','resizable',False)
```

これは Kivy の他のモジュールの読み込みに先立って記述すること．

3.7 Kivy 言語による UI の構築

実用的な GUI アプリケーションを構築する場合，GUI の構築と編集の作業に多くの時間と労力を要する．この部分の作業を容易にするために，インターフェースを構築するための特別なフレームワークを使用することが，アプリケーション開発において一般的になって [27] きている．Python と Kivy によるアプリケーション開発においても，インターフェース構築に特化した言語 **Kivy 言語**（以下 Kv と略す）を使用することができ，開発効率を高めることができる．

Kv は Python とは異なる独自の言語であり，ここでは Kv 自体の基礎的な解説からはじめ，Kv で記述した GUI と Python で記述したプログラムとの相互関係について説明する．ただし本書は Kv の全般的なリファレンスではなく，あくまで導入的な内容に留める．

[27] JavaFX における FXML や，それらを基本とする統合開発環境はまさにその例である．

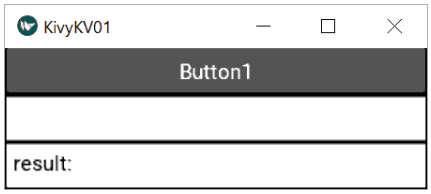
3.7.1 Kivy 言語の基礎

Kv はレイアウトやウィジェットの階層構造である**ウィジェットツリー**を宣言的に記述する言語である．Kv で記述されたウィジェットツリーは可視性が高く，GUI の構造全体の把握が容易になる．

3.7.1.1 サンプルプログラムを用いた説明

まずは Kv を使用せずに GUI を構築したアプリケーションを示す．そして，同じ機能を持つアプリケーションの GUI を Kv で記述する例を示す．

サンプルとして示すアプリケーションは図 18 に示すようなものである．これは，1 段目にボタン，2 段目と 3 段目にテキスト入力を備えたもので，それらを BoxLayout で配置している．



1 段目がボタン，2 段目と 3 段目がテキスト入力

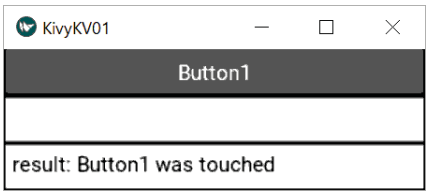
図 18: サンプルプログラムの実行例（起動時）

このアプリケーションの動作は次のようなものである．

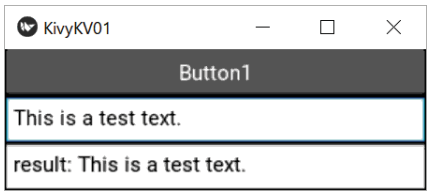
(a) ボタンをタッチ（クリック）すると，3 段目にその旨のメッセージが表示される．

(b) 2 段目にテキスト入力すると，3 段目にその旨のメッセージが表示される．

それらの様子を図 19 に示す．



(a) ボタンをクリックしたときの反応



(b) 2 段目にテキストを入力したときの反応

図 19: サンプルプログラムの実行例（動作）

このアプリケーションを Kv を使用せずに構築したものがプログラム kivyKV01-1.py である．

プログラム：kivyKV01-1.py

```
1  # coding: utf-8
2
3  #----- 関連モジュールの読込み -----
4  from kivy.app import App
5  from kivy.uix.boxlayout import BoxLayout
6  from kivy.uix.button import Button
7  from kivy.uix.textinput import TextInput
8  from kivy.core.window import Window
9
10 #----- GUIの構築 -----
11 # コールバック関数
12 def funcButton(self):
13     tx2.text = 'result: Button1 was touched'
14 def funcTextInput(self,v):
```

```
15      tx2.text = 'result: ' + self.text
16
17  # GUI
18  root = BoxLayout(orientation='vertical')
19  bt1 = Button(text='Button1')
20  bt1.bind(on_release=funcButton)
21  tx1 = TextInput()
22  tx1.bind(text=funcTextInput)
23  tx2 = TextInput()
24  root.add_widget(bt1)
25  root.add_widget(tx1)
26  root.add_widget(tx2)
27
28  #----- アプリケーション本体 -----
29  class KivyKV01App(App):
30      def build(self):
31          return root
32
33  # アプリの実行
34  Window.size = (300,100)
35  KivyKV01App().run()
```

次に，このプログラムのインターフェース部を Kv で，その他を Python で記述することを考える．

ところで，このアプリケーションの GUI の構成を階層的に示すと図 20 のようになる．

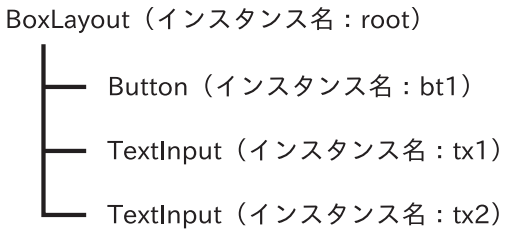


図 20: サンプルプログラムのウィジェットツリー

Kv で GUI を記述するとこの階層構造をそのまま反映した形となる．その Kv ファイル kivyKV01-1.py を示す．

Kv ファイル：kivyKV01.kv

```
1   <RootW>:
2       BoxLayout:
3           orientation: 'vertical'
4           Button:
5               id: bt1
6               text: 'Button1'
7               on_release: root.funcButton(tx2,bt1.text)
8           TextInput:
9               id: tx1
10              on_text:    root.funcTextInput(tx2,tx1.text)
11          TextInput:
12              id: tx2
```

このように，インスタンス名，プロパティ，イベントなどを含め，GUI の階層構造が簡潔に記述できる．この Kv ファイルを読み込んで GUI を実装するアプリケーション（Python プログラム）を kivyKV01.py に示す．

プログラム：kivyKV01.py

```
1   # coding: utf-8
2
3   #----- 関連モジュールの読込み -----
4   from kivy.app import App
```

```
 5 from kivy.uix.boxlayout import BoxLayout
 6 from kivy.core.window import Window
 7
 8 #----- GUIの構築 -----
 9 class RootW(BoxLayout):
10     # コールバック関数
11     def funcButton(self,tx2,t):
12         tx2.text = 'result: Button1 was touched'
13     def funcTextInput(self,tx2,t):
14         tx2.text = 'result: ' + t
15
16 #----- アプリケーション本体 -----
17 class KivyKV01App(App):
18     def build(self):
19         return RootW()
20
21 # アプリの実行
22 Window.size = (300,100)
23 KivyKV01App().run()
```

解説：

Kv ファイルの 1,2 行目が，Python プログラムの 9 行目のクラス定義に対応している．Python プログラムでは，GUI の最上位オブジェクトのクラス宣言と，コールバック関数の定義のみを記述しており，GUI の階層構造は全て Kv ファイル内に記述している．

Kv ファイルにおけるクラスの定義

'＜... ＞' の記述は Kv における規則の記述である．この記述を応用することで，Python プログラム側のクラス定義に対応させることができる．

Kv ファイルにおける id

Kv ファイル内では，ウィジェットなどに識別名を与えるために 'id:' を記述する．(id により与えられた識別名は，厳密にはインスタンス名ではない)

Kv ファイルにおけるイベントハンドリング

'イベント名:' に続いて呼び出すコールバック関数を記述する．このとき，Python プログラムの中のどこに記述された関数（メソッド）かを明示するために root という指定をしている．この root は Kv において記述対象のツリーの最上位を意味するものであり，Python 側プログラムでは，これを用いたクラス RootW （Python 側の 9 行目）が対応する．すなわち，`root.funcButton(tx2,bt1.text)` は，RootW クラスのメソッド funcButton を呼び出すことになる．

Kv 側からの関数（メソッド）呼び出しにおいては自由に引数を与えることができ，それを受ける Python 側の関数（メソッド）の仮引数も対応する形に記述する．ただし Python 側の仮引数には，第 1 引数として，それを呼び出したオブジェクト自身（self）を受け取る仮引数を記述する必要があり，結果として，引数の数が 1 つ多い記述となる．

3.7.1.2 Python プログラムと Kv ファイルの対応

先の例では，アプリケーションのクラス名が KivyKV01App なので，それに対応する Kv ファイルの名前は KivyKV01.kv とする．すなわち，アプリケーションのクラスの名前は，

'任意の名前 App'

と末尾に'App' を付ける．そして対応する Kv ファイルの名前には，App の前の部分に拡張子 '.kv' を付けたものとする．こうすることで，アプリケーションの起動時に自動的に対応が取られて Kv ファイルが読み込まれる．

Builder クラスを用いた Kv ファイルの読込み

Builder クラスの load_file メソッドを使用することで，先に説明した名前の制限にとらわれることなく，Python 側，Kv 側共に自由にファイル名を付けることもできる．Builder クラスを使用するためには次のようにして必要なモジュー

ルを読み込んでおく．

```
from kivy.lang.builder import Builder
```

この後，Python 側プログラムの冒頭で，

```
Builder.load_file( 'Kv ファイル名' )
```

とすることで，指定したファイルから Kv の記述を読み込むことができる．

3.8 時間によるイベント

Kivy には Clock モジュールがあり，ユーザからの入力以外に時間によるイベントハンドリングが可能である．すなわち，設定された時間が経過したことをイベントとしてハンドリングすることができ，いわゆる**タイマー**の動作を実現することができる．Clock モジュールを使用するには次のようにして必要なものを読み込んでおく．

```
from kivy.clock import Clock
```

3.8.1 時間イベントのスケジュール

ClockEvent として時間イベントを生成することで，指定した時間が経過した時点でコールバック関数を起動することができる．

＜コールバック関数のスケジュール＞
一度だけ： Clock.schedule_once (**コールバック関数, 経過時間**)
繰り返し： Clock.schedule_interval (**コールバック関数, 経過時間**)
この結果，ClockEvent オブジェクトが返される．経過時間の単位は「秒」であり，浮動小数点数で表現する．コールバック関数は仮引数を 1 つ取る形で定義しておく．コールバック関数には起動時に経過時間が引数として渡される．コールバック関数は戻り値として True を返すように記述するが，(False を返すと schedule_interval によってスケジュールされた時間イベントがキャンセルされる)

schedule_interval によってコールバック関数の繰り返し起動がスケジュールされた場合，得られた ClockEvent オブジェクトに対して cancel メソッドを使用することで，スケジュールを解除（キャンセル）することができる．また，

```
Clock.unschedule(スケジュールされた ClockEvent)
```

とすることでもキャンセルできる．

時間イベントの実装例は「3.9.4 スワイプ」のところで紹介する．

3.9 GUI 構築の形式

Kivy は通常の PC だけでなく，スマートフォンやタブレット PC といった携帯情報端末のためのアプリケーション開発を視野に入れているため，独特の UI デザインを提供する．例えば，複数のウィンドウを同時に表示する形式ではなく，1 つのウィンドウ内で，UI をまとめた**スクリーン**を切り替える形式などが特徴的である．
ここでは，実用的な UI インターフェースを構築するためのいくつかの形式について説明する．

3.9.1 スクリーンの扱い： Screen と ScreenManager

Kivy では，UI の 1 つのまとまりを Screen として扱い，それらをスライドのようにして切り替えることが可能である．Screen オブジェクトには各種のレイアウトオブジェクトを登録することができ，各々のレイアウトにはこれまで説明した方法で UI を構築する．複数作成した Screen オブジェクトは 1 つの ScreenMananger オブジェクトに登録し

て管理する，ScreenMananger に登録されたスクリーンは transition によって切り替えることができる．

ScreenManager，Screen を使用するには，次のようにして必要なモジュールを読み込んでおく．

```
from kivy.uix.screenmanager import ScreenManager, Screen
```

■ ScreenManager

ScreenMananger オブジェクトは複数の Screen を登録管理するもので，次のようにして生成する．

```
sm = ScreenManager()
```

この例では，生成された ScreenManager オブジェクトが sm に保持されている．

■ Screen

Screen オブジェクトは，先の ScreenManager に登録して使用する．このオブジェクトが 1 つの UI スクリーンとなり，更にここにレイアウトオブジェクトなどを登録する．Screen オブジェクトは次のようにして生成する．

```
sc1 = Screen(name=識別名)
```

「識別名」は Screen を識別するためのもので文字列として与える．生成した Screen オブジェクトは add_widget メソッドで ScreenManager オブジェクトに登録する．

ScreenManager オブジェクトのプロパティ current に Screen の識別名を与えることで表示する Screen を選択できる．また，transition プロパティの設定により，Screen が切り替わる様子をを制御できる．

Screen の遷移の効果は，いわゆる**スライドイン**であるが，その他の効果も設定できる．

参考）遷移の効果

Kivy の ScreenManager には図 12 のような様々な transition が用意されている．これらは ScreenManager オブジェクト生成時に設定するが，詳しくは巻末付録に挙げている Kivy のサイトを参照のこと．

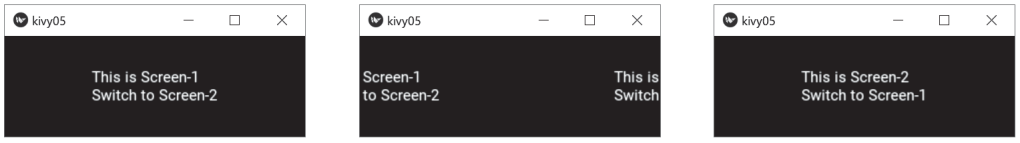
表 12: 遷移効果の効果

遷移	効果
SlideTransition	縦／横方向のスライディング（デフォルト）
SwapTransition	iOS のスワップに似た切り替え
FadeTransition	フェードイン／アウトによる切り替え
WipeTransition	ワイプによる切り替え
FallOutTransition	消え行くような切り替え
RiseInTransition	透明から不透明に変化するような切り替え
NoTransition	瞬時の切り替え
CardTransition	上に重ねるような切り替え

次に，図 21 のように，2 つのスクリーンを切り替え表示するプログラム kivy05-1.py を挙げて Screen の扱いについて説明する．

プログラム：kivy05-1.py

```
1 # coding: utf-8
2 #----- 必要なパッケージの読込み -----
3 from kivy.app import App
4 from kivy.uix.screenmanager import ScreenManager, Screen
5 from kivy.uix.anchorlayout import AnchorLayout
6 from kivy.uix.label import Label
7 from kivy.core.window import Window
```



スクリーンをタッチすると，　→　スライディングが起こり，　→　次のスクリーンに切り替わる

図 21: スクリーンを切り替えるアプリケーションの実行例

```
 8
 9  #----- アプリケーションクラスの定義 -----
10  class kivy05(App):
11      def build(self):
12          return root
13
14  #----- GUIの構築 -----
15  # ウィンドウのサイズ
16  Window.size = (300,100)
17
18  # スクリーンマネジャ
19  root = ScreenManager()
20  # スクリーン
21  sc1 = Screen(name='screen_1')
22  sc2 = Screen(name='screen_2')
23  # レイアウト
24  al1 = AnchorLayout()
25  al2 = AnchorLayout()
26  # ラベル
27  l1 = Label(text='This is Screen-1\nSwitch to Screen-2')
28  l2 = Label(text='This is Screen-2\nSwitch to Screen-1')
29  # 組み立て
30  al1.add_widget(l1)
31  al2.add_widget(l2)
32  sc1.add_widget(al1)
33  sc2.add_widget(al2)
34  root.add_widget(sc1)
35  root.add_widget(sc2)
36  # 最初に表示されるスクリーン
37  root.current = 'screen_1'
38
39  #----- スクリーン遷移の処理 -----
40  # コールバック関数
41  def callbk1(self,t):    # screen-1からscreen-2へ
42      root.transition.direction = 'left'
43      root.transition.duration = 3    # ゆっくり
44      root.current = 'screen_2'
45  def callbk2(self,t):    # screen-2からscreen-1へ
46      root.transition.direction = 'right'
47      root.transition.duration = 0.4  # デフォルト
48      root.current = 'screen_1'
49  # ラベルオブジェクトに登録
50  l1.bind(on_touch_up=callbk1)
51  l2.bind(on_touch_up=callbk2)
52
53  #----- アプリケーションの実行 -----
54  ap = kivy05()
55  ap.run()
```

解説

19〜35行目でスクリーンとUIを構築している．41〜48行目でスクリーンを切り替えるためのコールバック関数を定義して，Labelオブジェクトに登録している．この例でわかるように，ScreenManagerのプロパティ transition.direction で遷移の方向を，transition.duration で遷移にかかる時間（秒）を設定する．

ScreenManagerとは別に，より簡単にスワイプを実現する方法を「3.9.4 スワイプ」のところで解説する．

3.9.2 アクションバー： ActionBar

一般的な GUI アプリケーションで採用されているプルダウンメニューに近い機能を Kivy では**アクションバー**（ActionBar）という形で実現する．ActionBar を構築するために必要なクラスは次に挙げる 5 つのものである．

ActionBar，ActionView，ActionPrevious，ActionGroup，ActionButton

これらのクラスを使用するために，必要なモジュールを次のようにして読み込む．

```
from kivy.uix.actionbar import ActionBar, ActionView, ActionPrevious, ¥
                               ActionGroup, ActionButton
```

ここでは，よく知られた GUI における「メニューバー」「メニュー」「メニュー項目」の構成に対応させる形で ActionBar の構築について説明する．

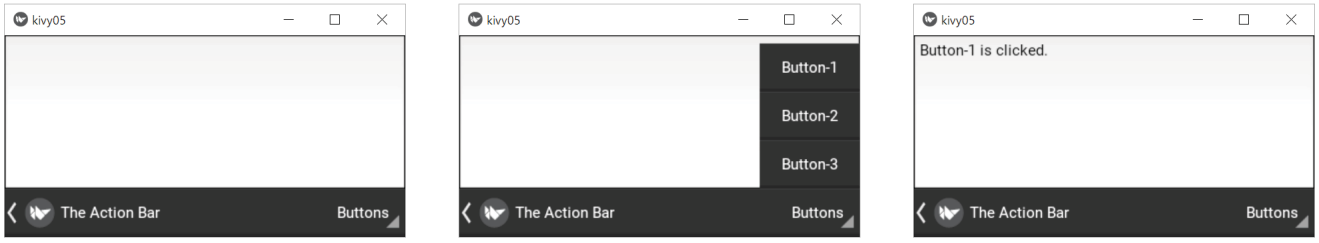
一般的に「メニューバー」と呼ばれるものは，例えば「ファイル」「編集」…といった「メニュー」を配置しており，それら各メニューをクリックするとプルダウンメニューが表示されて，その中に「新規」「開く」「保存」「閉じる」…といった「メニュー項目」が並んでいる．この場合の「メニューバー」は Kivy の ActionBar に相当する．ActionBar にはまず ActionView を登録して，それに対して ActionPrevious, ActionGroup を登録する．この ActionGroup が一般的な「メニュー」に相当する．あとは ActionGroup に対していわゆる「メニュー項目」に相当する ActionButton を必要なだけ登録する．

ActionBar を構築する様子を階層的に示すと次のようになる．

【ActionBar の構築】

```
ActionBar                （「メニューバー」に相当）
  ┣- ActionView
      ┣- ActionPrevious      （タイトルの表示など）
      ┣- ActionGroup         （「メニュー」に相当）
          ┣- ActionButton    （「メニュー項目」に相当）
          ┣- ActionButton    （「メニュー項目」に相当）
          ┇
```

図 22 のような ActionBar を実装するアプリケーションを例に挙げて構築方法について説明する．



ButtonGroup（右下）をクリックすると， → ActionButton が表示される． → 選択によるコールバック処理

図 22: ActionBar の実装例

これを実装したプログラムを kivy05-2.py に示す．

プログラム：kivy05-2.py

```
1 # coding: utf-8
2 #----- 必要なパッケージの読込み -----
3 from kivy.app import App
4 from kivy.uix.actionbar import ActionBar, ActionView, ActionPrevious, \
5                                 ActionGroup, ActionButton
```

```
 6 from kivy.uix.boxlayout import BoxLayout
 7 from kivy.uix.anchorlayout import AnchorLayout
 8 from kivy.uix.textinput import TextInput
 9 from kivy.core.window import Window
10
11 #----- アプリケーションクラスの定義 -----
12 class kivy05(App):
13     def build(self):
14         return root
15
16 #----- GUIの構築 -----
17 # ウィンドウのサイズ
18 Window.size = (400,200)
19
20 # 最上位のレイアウト
21 root = BoxLayout(orientation='vertical')
22
23 # テキストフィールド
24 txt = TextInput()
25
26 # アクションバー
27 acBar = ActionBar()
28 acView = ActionView()
29 acPrev = ActionPrevious(title='The Action Bar')
30 acGrp = ActionGroup(text='Buttons',mode='spinner')
31 acBtn1 = ActionButton(text='Button-1')
32 acBtn2 = ActionButton(text='Button-2')
33 acBtn3 = ActionButton(text='Button-3')
34
35 # 組み立て
36 root.add_widget(txt)
37 acGrp.add_widget(acBtn1)
38 acGrp.add_widget(acBtn2)
39 acGrp.add_widget(acBtn3)
40 acView.add_widget(acPrev)
41 acView.add_widget(acGrp)
42 acBar.add_widget(acView)
43 root.add_widget(acBar)
44
45 # コールバック関数
46 def clback1(self):
47     txt.text = 'Button-1 is clicked.'
48 def clback2(self):
49     txt.text = 'Button-2 is clicked.'
50 def clback3(self):
51     txt.text = 'Button-3 is clicked.'
52
53 acBtn1.bind(on_release=clback1)
54 acBtn2.bind(on_release=clback2)
55 acBtn3.bind(on_release=clback3)
56
57 #----- アプリケーションの実行 -----
58 ap = kivy05()
59 ap.run()
```

解説

27〜33 行目で ActionBar 構築に必要なオブジェクトを生成している．ActionGroup（いわゆるメニュー）を生成する際に，キーワード引数 'text=' を与えることでグループ名を設定することができ，これが ActionBar 上に表示される．また，キーワード引数 mode='spinner' を与えると，ActionGroup クリック時に ActionButton （いわゆるメニュー項目）が「立ち上がるメニュー」として表示される．

3.9.3 タブパネル： TabbedPanel

TabbedPanel を使用すると，いわゆる切り替えタブが実現できる．TabbedPanel には TabbedPanelItem オブジェクトを必要な数だけ登録する．各 TabbedPanelItem オブジェクトが個々のタブパネルであり，これにレイアウトオブ

ジェクトを配置することができる．

タブパネルを構築するには，次のようにして必要なモジュールを読み込んでおく．

```
from kivy.uix.tabbedpanel import TabbedPanel, TabbedPanelItem
```

ここでは，図 23 のように，2 つのタブを切り替えるプログラムを例に挙げる．

タブの切り替え表示

図 23: TabbedPanel の実装例

実装したプログラムを kivy05-3.py に示す．

プログラム：kivy05-3.py

```
1  # coding: utf-8
2  #----- 必要なパッケージの読込み -----
3  from kivy.app import App
4  from kivy.uix.tabbedpanel import TabbedPanel, TabbedPanelItem
5  from kivy.uix.anchorlayout import AnchorLayout
6  from kivy.uix.label import Label
7  from kivy.core.window import Window
8
9  #----- アプリケーションクラスの定義 -----
10 class kivy05(App):
11     def build(self):
12         return root
13
14 # タブパネル
15 root = TabbedPanel()
16 root.do_default_tab = False
17 # タブ1
18 ti1 = TabbedPanelItem(text='Tab1')
19 lb1 = Label(text='Tab 1')
20 ti1.add_widget(lb1)
21 root.add_widget(ti1)
22 # タブ2
23 ti2 = TabbedPanelItem(text='Tab2')
24 lb2 = Label(text='Tab 2')
25 ti2.add_widget(lb2)
26 root.add_widget(ti2)
27
28 root.default_tab = ti1
29
30 #----- GUIの構築 -----
31 # ウィンドウのサイズ
32 Window.size = (300,150)
33 ap = kivy05()
34 ap.run()
```

解説

18,23 行目のように，TabbedItem のオブジェクト生成時にキーワード引数 'text=' を与えることで，タブの見出しを設定できる．

3.9.4 スワイプ： Carousel

Carousel による UI は ScreenManager による UI と似ているが，**スワイプ**機能が予め備わっており，ScreenManager の transition を使用するよりも UI の実装が容易であることが特徴である．

Carousel オブジェクトにはレイアウトをはじめとするウィジェットを登録するが，それらは登録した順に順序付けされる．そして，Carousel オブジェクトに対して，load_next ,load_previous といったメソッドを使用することで，登録されたウィジェット群を順番に（逆順に）「回転」させる [28] ことができる．

Carousel を使用したプログラム kivy05-4.py を次に示す．

プログラム：kivy05-4.py

```
1  # coding: utf-8
2  #----- 必要なパッケージの読込み -----
3  from kivy.app import App
4  from kivy.uix.carousel import Carousel
5  from kivy.uix.label import Label
6  from kivy.clock import Clock
7  from kivy.core.window import Window
8
9  #----- アプリケーションクラスの定義 -----
10 class kivy05(App):
11     def build(self):
12         return root
13
14 # Carouselの生成
15 root = Carousel(direction='right')
16 lb1 = Label(text='Slide - 1')
17 lb2 = Label(text='Slide - 2')
18 lb3 = Label(text='Slide - 3')
19 root.add_widget(lb1)
20 root.add_widget(lb2)
21 root.add_widget(lb3)
22 root.loop = True    # 循環の切り替え指定
23
24 #----- 自動スワイプ -----
25 # コールバック関数（引数にはdurationが与えられる）
26 def autoSwipe(dt):
27     print(dt)
28     root.load_next()
29     return True     # Falseを返すとスケジュールがキャンセルされる
30 # スケジュール
31 tm = Clock.schedule_interval(autoSwipe, 3)
32 print(type(tm))
33
34 # 停止用コールバック関数
35 def cancelSwipe(self,t):
36     print('自動スワイプがキャンセルされました．')
37     Clock.unschedule(tm)
38 #   tm.cancel()                          # キャンセル方法1
39 root.bind(on_touch_down=cancelSwipe)    # キャンセル方法2
40
41 #----- GUIの構築 -----
42 # ウィンドウのサイズ
43 Window.size = (300,150)
44 ap = kivy05()
45 ap.run()
```

解説

22 行目にあるように，Carousel オブジェクトの loop プロパティに True を設定すると，末尾のウィジェットの次は先頭のウィジェットに戻るように順序付けされ，ウィジェット群を順番に回転させることができる．

[28]"carousel"（英） は「回転木馬」という意味である．

4　実用的なアプリケーション開発に必要な事柄

4.1　日付と時間に関する処理

Python では日付や時刻，あるいは経過時間といった情報を扱うことができる．そのような処理をするためには次のようにして datetime モジュールを読み込んでおく必要がある．

```
from datetime import *
```

4.1.1　基本的な方法

datetime モジュールで扱える情報は年，月，日，時，分，秒，ミリ秒（あるいはマイクロ秒）の 7 つの要素[29] である．例えば現在時刻を取得するには now メソッドを使用する．

例. 現在時刻の取得

```
>>> datetime.now()  Enter                    ←現在時刻の取得
datetime.datetime(2017, 5, 5, 13, 10, 25, 653093)  ←処理結果
```

この例のように時間情報は datetime.datetime(...) という形式で扱われる．
また,「日付のみ」や「日付なし時刻」を取得することもできる．

例. 時刻情報の分解

```
>>> d = datetime.now()  Enter            ←現在時刻の取得
>>> d.date()  Enter                      ←日付のみの取り出し
datetime.date(2017, 5, 5)                ←処理結果
>>> d.time()  Enter                      ←日付なし時刻の取り出し
datetime.time(13, 23, 16, 396945)        ←処理結果
```

このように，dimedate オブジェクトに対して date メソッドや time メソッドを実行する．その結果，それぞれ
　　datetime.date(...),　　datetime.time(...)
という形式で結果が得られる．

now メソッドで現在時刻を取得することとは別に，指定した特定の日付，時間を構成することもできる．

例. 「1966 年 3 月 14 日」という日付と時刻を生成する

```
>>> datetime(1966,3,14)  Enter            ←日付情報の生成
datetime.datetime(1966, 3, 14, 0, 0)      ←得られたデータ
```

2 つの時刻の間の時間差を取得することもできる．

例. 1966 年 3 月 14 日 13 時 15 分から 2017 年 5 月 5 日 14 時 00 分までの経過時間

```
>>> d1 = datetime(1966,3,14,13,15)  Enter   ←日付情報 1966/03/14 13:15 の生成
>>> d2 = datetime(2017,5,5,14,0)  Enter     ←日付情報 2017/05/05 14:00 の生成
>>> d2 - d1  Enter                          ←経過時間の取得
datetime.timedelta(18680, 2700)             ←経過日数が「18680 日と 2700 秒」である
```

このように，自然な形の減算で時間差を取得することができ，結果は
　　datetime.timedelta(日数, 秒数)
という形式（timedelta）で得られる．

timedelta を用いると「〜日後の日付」や「〜日前の日付」を算出することもできる．計算方法は，datetime オブジェクトに対する timedelta オブジェクトの加算あるいは減算である．

[29] これらに加えて，多くのシステムではマイクロ秒まで扱える．

例. 1966 年 3 月 14 日 13 時 15 分から 18680 日と 2700 秒経過した日付と時刻

```
>>> d1 = datetime(1966,3,14,13,15)  Enter    ←日付情報 1966/03/14 13:15 の生成
>>> td = timedelta(18680, 2700)  Enter       ←経過した日数と秒数の生成
>>> d1 + td  Enter                           ←日付の取得
datetime.datetime(2017, 5, 5, 14, 0)         ←得られた日付
```

日付，時刻を書式整形して文字列で取得することもできる．

例. 日付，時刻の書式整形

```
>>> d1 = datetime(1966,3,14,13,15)  Enter    ←日付情報の生成
>>> str(d1)  Enter    ←文字列に変換
'1966-03-14 13:15:00'    ←得られた文字列
>>> d1.ctime()  Enter    ←文字列（UNIX 形式，C 言語）に変換
'Mon Mar 14 13:15:00 1966'    ←得られた文字列
```

このように str 関数の引数に datetime オブジェクトを与える．あるいは C 言語（UNIX）の形式で書式整形するには ctime メソッドを用いる．

datetime オブジェクトから各要素（年，月，日，時，分，秒，マイクロ秒）を取り出すための各種のプロパティがある．

例. datetime からの要素の取り出し

```
>>> d = datetime.now()  Enter    ←現在の日付，時刻の取得
>>> d  Enter    ←確認
datetime.datetime(2017, 5, 5, 14, 54, 23, 305326)  ←表示結果
>>> d.year  Enter    ←「年」の取得
2017                    ←表示結果（年）
>>> d.month  Enter    ←「月」の取得
5                       ←表示結果（月）
>>> d.day  Enter      ←「日」の取得
5                       ←表示結果（日）
>>> d.hour  Enter     ←「時」の取得
14                      ←表示結果（時）
>>> d.minute  Enter   ←「分」の取得
54                      ←表示結果（分）
>>> d.second  Enter   ←「秒」の取得
23                      ←表示結果（秒）
>>> d.microsecond  Enter    ←「マイクロ秒」の取得
305326                  ←表示結果（マイクロ秒）
```

4.1.2 time モジュールの利用

先に説明した datetime モジュールよりも更に基本的な機能を提供するのが time モジュールである．日付を含めた時刻情報の扱いには datetime モジュールを使用するのが良いが，プログラムの実行時間など，比較的短い時間範囲で時間を計測する場合には time モジュールを用いる方が良い．このモジュールは使用に先立って，次のようにしてシステムに読み込んでおく必要がある．

```
import time
```

4.1.2.1 時間の計測

プログラムの実行にかかった時間を計測するには，開始時点での時刻と終了時点での時刻をそれぞれ計測し，その差を取れば良い．今現在の時刻を秒単位で取得するには time のクラスメソッドである time を使用する．

現在時刻の取得： `time.time()`

2 を 100000 回掛け算することで 2^{100000} を求めるプログラム test10-1.py を示す．計算にかかった時間が表示される．

プログラム：test10-1.py

```
1  # coding: utf-8
2
3  # 必要なモジュールの読込み
4  import time
5
6  t1 = time.time()
7  n = 1
8  for i in range(100000):
9      n *= 2
10 t2 = time.time()
11
12 print(t2-t1,'(sec)')
13 #print(n)
```

このプログラムを実行すると，例えば

```
0.2474043369293213 (sec)
```

などと，実行に要した時間[30] が表示される．また，最終行のコメント '#' を外すと 2^{100000} の計算結果が表示される．

4.1.2.2 プログラムの実行待ち

time クラスの sleep メソッドを用いると，指定した時間の間プログラムを待機させることができる．待機させる時間は秒単位（小数点付きも可）で引数に与える．

例. プログラムの実行を 3 秒待機する．

```
time.sleep(3.0)
```

4.2 文字列検索と正規表現

re モジュールを用いることで，高度なパターン検索が実現できる．このモジュールは次のようにして読み込む．

```
import re
```

4.2.1 パターンの検索

re クラスのメソッド search メソッドを使用すると，文字列に含まれるパターンを検索することができる．

＜文字列の検索 (1) ＞
書き方 (1)：　re.search(検索キー, テキスト)
検索対象のテキスト中から指定した検索キー（パターン）が最初に現れる箇所を見つけ出す．
「検索キー」は **raw 文字列**で与える．
書き方 (2)：　コンパイル済み検索キー.search(テキスト)
「コンパイル済み検索キー」は re クラスの compile メソッドで生成された検索キーであり，これを用いることで検索処理が最適化される．
検索キーのコンパイル：　re.compile(検索キー)
検索処理が終わると，検索結果を保持する **match オブジェクト**が返される．

[30] 当然であるが，使用する計算機環境によって実行時間は異なる．

検索キーには正規表現を指定することができる．そのため，raw 文字列で与える必要がある．

検索の例.

テキスト 'My name is Taro. I am 19 years old.' の中から 'Taro' の位置を探す例を示す.

```
>>> import re  Enter      ←パッケージの読み込み
>>> txt = 'My name is Taro.  I am 19 years old.'  Enter      ←テキストの生成
>>> ptn = r'Taro'  Enter                          ←検索キーの生成
>>> res = re.search(ptn,txt)  Enter               ←検索の実行
>>> res.span()  Enter                             ←検出位置を調べる
(11, 15)        ← 11〜14 番目に'Taro' が存在することがわかる
```

この例のように，検索結果は res というオブジェクトとして得られており，それに対して span メソッドを実行することで，検出位置が得られる.

検索キーをコンパイルして同様の処理を行ったものが次の例である.

```
>>> p = re.compile(ptn)  Enter      ←検索キーのコンパイル
>>> res = p.search(txt)  Enter      ←検索の実行
>>> res.span()  Enter               ←検出位置を調べる
(11, 15)        ← 11〜14 番目に'Taro' が存在することがわかる
```

search メソッドは，テキストの中で最初に検索キーが現れる場所を探す．同じ検索キーがテキストの中に複数含まれる場合は finditer メソッドを使用することで全ての検出位置を取得することができる.

＜文字列の検索 (2) ＞

書き方 (1)：　re.finditer(検索キー, テキスト)

検索対象のテキスト中から指定した検索キー（パターン）が現れる箇所を全てを見つけ出す.
「検索キー」は **raw 文字列**で与える.

書き方 (2)：　コンパイル済み検索キー.finditer(テキスト)

「コンパイル済み検索キー」は re クラスの compile メソッドで生成された検索キーであり，これを用いることで検索処理が最適化される.

検索処理が終わると，検索結果を保持する **match オブジェクト**が返される.

次に，finditer メソッドを用いた検索の例を示す．test11.txt のようなテキストからキーワード「Python」を全て見つけ出す処理の例である.

検索対象のテキスト：test11.txt

```
1  （フリー百科事典「ウィキペディア」より）
2
3  コードを単純化して可読性を高め、読みやすく、また書きやすくしてプログラマの
4  作業性とコードの信頼性を高めることを重視してデザインされた、汎用の高水準言語
5  である。反面、実行速度はCなどの低級言語に比べて犠牲にされている。
6
7  核となる文法（シンタックス）および意味（セマンティクス）は必要最小限に抑え
8  られている。その反面、豊富で大規模な文書（document）や、さまざまな領域に
9  対応する大規模な標準ライブラリやサードパーティ製のライブラリが提供されている。
10 またPythonは多くのハードウェアとOS（プラットフォーム）に対応しており、複数の
11 プログラミングパラダイムに対応している。Pythonはオブジェクト指向、命令型、
12 手続き型、関数型などの形式でプログラムを書くことができる。動的型付け言語であり、
13 参照カウントベースの自動メモリ管理（ガベージコレクタ）を持つ。
14
15 これらの特性により、PythonはWebアプリケーションやデスクトップアプリケーション
16 などの開発はもとより、システム用の記述（script）や、各種の自動処理、理工学や
```

```
統計・解析など、幅広い領域における支持を得る、有力なプログラム言語となった。
プログラミング作業が容易で能率的であることは、ソフトウェア企業にとっては
投入人員の節約、開発時間の短縮、ひいてはコスト削減に有益であることから、
産業分野でも広く利用されている。Googleなど主要言語に採用している企業も多い。
```

このテキストの中から検索キー 'Python' を全て探し出すプログラムを test11.py に示す.

プログラム：test11.py

```
1  # coding: utf-8
2
3  # 必要なモジュールの読込み
4  import re
5
6  # ファイルの内容を一度で読み込む
7  f = open('test11.txt','r',encoding='utf-8')
8  text = f.read()
9  f.close()
10
11 # 検索処理
12 ptn = r'Python'      # 検索キー
13 p = re.compile(ptn) # 検索キーのコンパイル
14 res = p.finditer(text)
15
16 # 処理結果の報告
17 print('【検索キー"Python"の検索結果】')
18 print('------------------------------')
19 n = 1
20 for i in res:
21     print(n,'番目の検出位置: ',i.span())
22     n += 1
23 print('------------------------------')
24 print(n-1,'個検出しました.')
```

テキスト中には 'Python' という語は複数あり，得られる match オブジェクトも検索が該当した回数分のデータを含んでいる．そこで，20〜22 行目のように for 文で 1 要素づつ取り出している．取り出された要素に対して span メソッドを実行することで，テキスト中に見つかった検索キーワードの位置を取得することができる.

このプログラムを実行した結果を次に示す.

```
【検索キー"Python"の検索結果】
------------------------------
1 番目の検出位置:  (256, 262)
2 番目の検出位置:  (319, 325)
3 番目の検出位置:  (424, 430)
------------------------------
3 個検出しました.
```

4.2.1.1 正規表現を用いた検索

検索キーとして，固定された文字列のみを用いるのではなく，指定した条件に一致する部分をテキストの中から見つけ出す場合に**正規表現**が非常に有用である．例えばアルファベットのみから成る文字列を見つけ出す場合について考える.

「アルファベット文字列」を意味するパターンを正規表現で表すと '[a-zA-Z]+' となる.（詳しくは後述する）先のプログラムの 12 行目を

```
ptn = r'[a-zA-Z]+'
```

21 行目を

```
print(n,'番目の検出位置:¥t',i.span(),'¥t',i.group())
```

と書き換えて（タイトル表示部分も変えて）実行すると，次のような結果となる.

```
【アルファベット文字列の検索結果】
------------------------------
1 番目の検出位置:  (110, 111) C
2 番目の検出位置:  (193, 201) document
3 番目の検出位置:  (256, 262) Python
                :
          （途中省略）
                :
8 番目の検出位置:  (479, 485) script
9 番目の検出位置:  (631, 637) Google
------------------------------
9 個検出しました.
```

このように柔軟に検索パターンを構成する場合に正規表現が有用である．検索パターンが具体的にどのような文字列に一致したかを調べるには，メソッド group を使用する．

この例では '[a-zA-Z]+' で半角アルファベットを表したが，正規表現では '[...]' で文字の範囲を表す．例えば '[a-z]' とするとこれは「アルファベット小文字」を意味する．また，'[A-Z]' では「アルファベット大文字」を意味する．さらに '+' は「それらが 1 文字以上続く列」を意味する．従って，

'[a-zA-Z]+'

は，

「アルファベット小文字もしくは大文字」かつ「それらが 1 文字以上続く列」

を意味し,「アルファベットの文字列」を意味するパターンとなる．同様に「数字のみの列」を意味する正規表現は '[0-9]+' となる．

本書では正規表現の全てについての解説はしないが，特に使用頻度が高いと考えられるパターン表現について説明する．

正規表現で構成する文字列のパターン

基本的には「1 文字分のパターン」と「繰り返し」を連接したものである．先に説明した'[...]' が「1 文字分のパターン」，'+' が「繰り返し」を意味する.「1 文字分のパターン」の基本的なものを表 13 に挙げる．

表 13: 正規表現のパターン（一部）

パターン	意味
[...]	文字の範囲をハイフン '-' でつなげる.
.（ドット）	任意の 1 文字
¥d	数字（ [0-9] と同じ）
¥D	数字以外
¥s	空白文字（タブも含む）
¥S	空白文字以外
¥w	半角英数字（ [0-9a-zA-Z] と同じ）
¥W	半角英数字以外

「繰り返し」の基本的なものを表 14 に挙げる．

4.2.1.2 正規表現を用いたパターンマッチ

パターンマッチによって，テキストデータから必要な部分を抽出することができる．パターンマッチには match メソッドを用いる．

表 14: 繰り返しの表記（一部）

表記	意味	表記	意味
+	1 回以上	*	0 回以上
?	0 回かもしくは 1 回	{m,n}	m 回以上 n 回以下（回数が大きい方を優先）
		{m,n}?	m 回以上 n 回以下（回数が小さい方を優先）

例. テキスト 'uyhtgfdres98234yhnbgtrf' の中から数字の部分のみ取り出す

```
>>> txt = 'uyhtgfdres98234yhnbgtrf'  Enter        ←テキストの生成
>>> ptn = r'[a-zA-Z]+([0-9]+)[a-zA-Z]*'  Enter  ←パターンの生成
>>> res = re.match(ptn,txt)  Enter              ←パターンマッチの実行
>>> res.group(0)  Enter            ←マッチしたか確認
'uyhtgfdres98234yhnbgtrf'        ←マッチしている（テキスト全体）
>>> res.group(1)  Enter  ←パターン ([0-9]+) が一致した部分の取り出し
'98234'          ←数字のみが得られている
```

この例では，検索パターンが　'[a-zA-Z]+([0-9]+)[a-zA-Z]*'　として与えられている．パターンの中に括弧 '(...)' の部分があるが，抽出したい部分はこのように括弧で括る．match メソッドは，テキストとパターンを一致させるように動作する．すなわちテキストの先頭からパターンを一致させるように試みる．

パターンの中には抽出したい括弧付きの部分を複数記述することができ，match が成功すると，得られた match オブジェクトから group メソッドに順番の引数を与えて取り出すことができる．ここに示した例では括弧付きの部分は 1 つであるので，

```
res.group(1)
```

で抽出した部分を得ることができる．

4.3　マルチスレッドとマルチプロセス

4.3.1　マルチスレッド

スレッド（thread）の考え方に基いて複数のプログラムを同時に並行して実行する[31] ことができる．Python では threading モジュールを使用することで，関数やメソッドをメインプログラムから独立したスレッドとして実行できる．このモジュールを使用するには次のようにしてモジュールを読み込む．

```
import threading
```

＜スレッド管理のための基本的なメソッド＞

● **スレッドの生成**

threading.Thread(target=関数（メソッド）の名前, args=(引数の並び,))

このメソッドの実行によりスレッドが生成され，それが**スレッドオブジェクト**として返される．

● **スレッドの実行**

スレッドオブジェクト.start()

start メソッドの実行により，スレッドの実行が開始する．

● **スレッドの終了の待ち受け**

スレッドオブジェクト.join()

スレッドが終了するのを待ち受ける．

[31]マルチスレッドプログラミングにおいては，実際には「同時に並行」して実行されず，Python インタプリタの実行時間を各スレッドに分割配分する．後述のマルチプロセスプログラミングでは，CPU やコアの数によって「同時に並行」して実行されることがある．

2 つの関数を別々のスレッドとして実行するプログラム test12-1.py を示す.

プログラム：test12-1.py

```
1  # coding: utf-8
2
3  # 必要なモジュールの読込み
4  import threading
5  import time
6
7  # 第1スレッド
8  def th_1(name):
9      for i in range(8):
10         print(name)
11         time.sleep(1)
12
13 # 第2スレッド
14 def th_2(name):
15     for i in range(3):
16         print('\t',name)
17         time.sleep(2)
18
19 # スレッドの生成
20 t1 = threading.Thread(target=th_1, args=('Thread-1',))
21 t2 = threading.Thread(target=th_2, args=('Thread-2',))
22 # スレッドの開始
23 t1.start()
24 t2.start()
25
26 # スレッドの終了待ち
27 t2.join()
28 print('\tThread-2 ended')
29 t1.join()
30 print('Thread-1 ended')
```

解説

このプログラムでは関数 th_1 と th_2 を別々のスレッドとして実行する．20〜21 行目で引数を付けてスレッドを生成しているが，スレッドとなる関数に渡す引数の列の最後にコンマ ',' を付けることが必要である．

23,24 行目でスレッドを開始し，27,29 行目でそれらの終了を待ち受けている．

このプログラムを実行した様子を次に示す.

```
Thread-1
        Thread-2
Thread-1
        Thread-2
Thread-1
Thread-1
        Thread-2
Thread-1
Thread-1
        Thread-2 ended
Thread-1
Thread-1
Thread-1 ended
```

2 つのスレッドが同時に独立して動作していることがわかる.

4.3.2 マルチプロセス

マルチプロセスプログラミングでは，プログラムの実行単位（関数の実行など）を，独立したプロセスとして実行する．このため，複数の CPU やコアを搭載した計算機環境においては，OS の働きによって各プロセスの実行が複数の CPU やコアに適切に割り当てられる．Python には，マルチプロセスプログラミングのためのモジュール concurrent.futures が標準的に提供（Python3.2 から）されている．このモジュールを使用するには次のようにして必要なモジュールを

読み込む．

```
from concurrent import futures
```

4.3.2.1 ProcessPoolExecutor

concurrent.futures には，マルチプロセスの形で関数を実行するための ProcessPoolExecutor というクラスが用意されており，このクラスのインスタンスが複数のプロセスの実行を管理する．具体的には，

```
futures.ProcessPoolExecutor(max_workers=プロセスの最大数)
```

と記述する．例えば，プロセスの最大数を 4 としてインスタンスを生成する場合は次のように記述する．

```
exe = futures.ProcessPoolExecutor(max_workers=4)
```

この結果，exe というインスタンスが生成され，これに対して submit メソッドを実行することで独立したプロセスを実行することができる．例えば，ProcessPoolExecutor オブジェクト exe によって関数 f を実行するには次のように記述する．

```
exe.submit(f,'arg')
```

これによって，関数　f('arg')　が独立したプロセスとして実行される．

submit メソッドによって起動した関数の実行が終了して ProcessPoolExecutor オブジェクトが不要になった際は shutdown メソッド実行する．

例. ProcessPoolExecutor オブジェクト exe の終了処理

```
exe.shutdown()
```

参考. concurrent.futures モジュールには，ProcessPoolExecutor とは別に ThreadPoolExecutor クラスも提供されており，ProcessPoolExecutor クラスとほぼ同じ使用方法でマルチスレッド実行も実現している．

4.3.3 マルチスレッドとマルチプロセスの実行時間の比較

ここでは，サンプルプログラム mproc01.py を使って，マルチスレッドとマルチプロセスの間の実行時間の差異を調べる方法を例示する．

プログラム中には関数 calcf が定義されており，これは円軌道のシミュレーションを実行する（円を表現する差分方程式の数値解を求める）もの [32] である．これを各種の異なる方法で実行する．

プログラム：mproc01.py

```
 1 # coding: utf-8
 2
 3 # 必要なモジュールの読込み
 4 import time
 5 import threading
 6 from concurrent import futures
 7
 8 # 対象の関数：円軌道の精密シミュレーション
 9 def calcf(name):
10     dt = 0.0000001
11     x = 1.0;     y = 0.0
12     t1 = time.time()
13     for i in range(62831853):
14         x -= y * dt;     y += x * dt
15     t = time.time() - t1
16     print(name,'- time:',t)
17
18 ######################################################################
```

[32] 詳しくは「6 外部プログラムとの連携」の章にある，円の微分方程式を差分化する方法を参照のこと．

```
19 #    マルチスレッドとマルチプロセスによる実行時間の比較実験        #
20 ######################################################################
21 if __name__ == '__main__':
22
23 #    threadingモジュールによる実験
24 #    print('--- By threading ---')
25 #    p1 = threading.Thread(target=calcf,args=('p1',))
26 #    p2 = threading.Thread(target=calcf,args=('p2',))
27 #    p3 = threading.Thread(target=calcf,args=('p3',))
28 #    p1.start()
29 #    p2.start()
30 #    p3.start()
31 #    p1.join()
32 #    p2.join()
33 #    p3.join()
34
35 #    concurrent.futuresモジュールによるマルチプロセスの実験
36 #    print('--- By concurrent.futures : [ProcessPoolExecutor] ---')
37 #    exe = futures.ProcessPoolExecutor(max_workers=4)
38 #    f1 = exe.submit(calcf,'p1')
39 #    f2 = exe.submit(calcf,'p2')
40 #    f3 = exe.submit(calcf,'p3')
41 #    f4 = exe.submit(calcf,'p4')
42 #    exe.shutdown()
43
44 #    concurrent.futuresモジュールによるマルチスレッドの実験
45 #    print('--- By concurrent.futures : [ThreadPoolExecutor] ---')
46 #    exe = futures.ThreadPoolExecutor(max_workers=4)
47 #    f1 = exe.submit(calcf,'p1')
48 #    f2 = exe.submit(calcf,'p2')
49 #    f3 = exe.submit(calcf,'p3')
50 #    f4 = exe.submit(calcf,'p4')
51 #    exe.shutdown()
52
53     pass
```

このプログラムの 23 行目以降のコメントを適切に外して，

1) threading モジュールによる実行

2) concurrent.futures モジュールの ProcessPoolExecutor クラスによる実行

3) concurrent.futures モジュールの ThreadPoolExecutor クラスによる実行

の 3 つの実行形態を比較することができる．1) と 3) の形態では，CPU やコアの数によらず関数 caclf の実行時間が伸びてゆく様子が確認できる．2) の形態では，CPU やコアの数が実行時間に影響を及ぼす（実行時間が小さくなる）ことが確認できる．

4.4 モジュールの作成による分割プログラミング

実用的なアプリケーションプログラムの開発においては，多くの関数やクラスを定義する．また，システムの機能を細分化して別々のソースプログラムとしてアプリケーションを作り上げることが一般的である．更に，汎用性の高い関数やクラスは，別のアプリケーションを開発する際にも再利用できることが望ましい．

これまで，様々なモジュールの利用方法について説明したが，それらは多くの開発者（サードパーティー）が構築したプログラムであり，汎用性の高さゆえに多くの開発者に向けて公開され利用されている．再利用が望まれる関数やクラスはこのように**モジュール**や**パッケージ**という形で用意しておくが，ここでは，独自のモジュールやパッケージを作成する方法について説明する．

4.4.1　モジュール

4.4.1.1　単体のソースファイルとしてのモジュール

最も簡単な方法として，1 つのソースファイルに関数やクラスの定義を記述するという方法があるが，これに関してサンプルを示しながら説明する．

次のようなプログラム（モジュールファイル）MyModule.py を用意する．

プログラム：MyModule.py

```
1  # coding: utf-8
2
3  # 加算関数
4  def kasan(x,y):
5      print('module:',__name__)
6      return( x + y )
7
8  # 乗算関数
9  def jouzan(x,y):
10     print('module:',__name__)
11     return( x * y )
```

この場合のモジュールの名前はそのファイル名（拡張子は除く）であり，このモジュールには 2 つの関数 kasan と jouzan が定義されている．このモジュールは他のプログラムや Python インタプリタに読み込んで使用することができる．（下記参照）

例. Python インタプリタからモジュール MyModule.py を利用する例

```
>>> from MyModule import kasan, jouzan  Enter    ←読み込み
>>> kasan(2,3)  Enter    ← kasan の実行
module:  MyModule        ←モジュール名の表示
5                        ←処理結果
>>> jouzan(3,4)  Enter    ← jouzan の実行
module:  MyModule        ←モジュール名の表示
12                       ←処理結果
>>> print(__name__)  Enter    ←モジュール名の表示（メイン）
__main__            ←モジュール名の表示
```

この例にあるように，

from　モジュール名　import　使用する関数（クラス）名

としてモジュールを読み込む．

Python では大域変数（グローバル変数）__name__が定義されており，現在実行されているモジュール名がそこに保持されている．メインプログラムのモジュール名は '__main__' である．（詳しくは後の「4.4.2.1 モジュールの実行」を参照のこと）

モジュールを使用した後は，モジュールファイルと同じディレクトリ内にサブディレクトリ__pycache__ が作成される．

規模の大きなプログラムを開発する際は，複数のサブディレクトリと複数のファイルから構成される**パッケージ**の形にするのが一般的である．

4.4.2　パッケージ（ディレクトリとして構成するモジュール）

複数のソースファイルから構成されるモジュールを**パッケージ**という．1 つのパッケージは 1 つのディレクトリとして作成し，そのディレクトリ内にサブディレクトリ __init__.py を作成しておく．（空で良い）

パッケージ作成の例

次のようなディレクトリ構成のパッケージについて考える．

```
./MyPackage              （ディレクトリ）
    |
    +-  __init__.py      （ディレクトリ）
    |
    +-  kasan.py         （テキスト形式のソースプログラム）
    |
    +-  jouzan.py        （テキスト形式のソースプログラム）
```

この場合のモジュール名は，パッケージのファイル（フォルダ）を含む最上位のディレクトリ名である．この例では MyPackage がモジュール名となる．

次に kasan.py と jouzan.py の内容を示す．

プログラム：kasan.py

```
1  # coding: utf-8
2
3  # 加算関数
4  def kasan(x,y):
5      print('module:',__name__)
6      return( x + y )
7
8  # テスト実行用メイン部
9  if __name__ == '__main__':
10     print('テスト実行：kasan(2,3)=',kasan(2,3))
```

プログラム：jouzan.py

```
1  # coding: utf-8
2
3  # 乗算関数
4  def jouzan(x,y):
5      print('module:',__name__)
6      return( x * y )
7
8  # テスト実行用メイン部
9  if __name__ == '__main__':
10     print('テスト実行：jouzan(3,4)=',jouzan(3,4))
```

このパッケージを読み込んで使用する例を次に示す．

```
>>> from MyPackage.kasan import kasan   Enter      ←パッケージの読込み (1)
>>> from MyPackage.jouzan import jouzan   Enter    ←パッケージの読込み (2)
>>> kasan(2,3)   Enter              ← kasan の実行
module:  MyPackage.kasan           ←モジュール名の表示
5                                  ←処理結果
>>> jouzan(3,4)   Enter            ← jouzan の実行
module:  MyPackage.jouzan          ←モジュール名の表示
12                                 ←処理結果
```

この例にあるように，

from　モジュール名. サブモジュール名　import　使用する関数（クラス）名

としてモジュールを読み込む．ここで言う**サブモジュール**は，パッケージのディレクトリ配下にあるソースファイル（拡張子は除く）のことである．

この例の 2 つのソースプログラムの 8〜10 行目には**テスト実行用メイン部**というものが記述されているが，これについては次に説明する．

4.4.2.1 モジュールの実行

モジュールはメインプログラムや他のプログラムから呼び出して使用するものであるが，開発中にそのモジュールをテストするために，テスト実行用のメインプログラムを別のソースファイルとして作成するのは煩わし場合がある．この問題を軽減するために，モジュールのファイルの中に，テスト実行用の**メイン部**を記述しておいて，直接そのモジュールを Python インタプリタで実行するのが良い．例えば先のパッケージの例において，次のようにして OS のコマンドシェルからモジュールを直接実行することができる．

```
py MyPackage/kasan.py  Enter    ← OS のシェルからモジュールを直接実行
module:  __main__               ←メインプログラムとして実行されていることがわかる
テスト実行：kasan(2,3)= 5       ←実行結果
```

Python のプログラミングにおいては，このような形（下記のような形）で明に**メイン部**を記述するスタイルが一般的である．

Python のプログラミングスタイル

```
if __name__ == '__main__':
        :
    （メインプログラム）
        :
```

4.5 ファイル内でのランダムアクセス

通常の場合，ファイル入出力は**順次アクセス**（Sequential Access）と呼ばれる手法で行われる．すなわち，読込みの際は先頭からファイル末尾（EOF: End Of File）に向けて順番に読み取られ，書き込みの際はその時点でのファイル末尾にデータが追加される．これに対して，記録媒体の任意のバイト位置にアクセスする手法を**ランダムアクセス**（Random Access）という．具体的には，記録媒体の指定したバイト位置から内容を読み取ったり，指定したバイト位置にデータを書き込む手法を意味する．ここでは，ファイルに対するランダムアクセスのための基本的な事柄について説明する．

4.5.1 ファイルのアクセス位置の指定（ファイルのシーク）

通常の順次アクセスにおいては，現在開いているファイルのアクセス位置は「次にアクセスすべき位置」が自動的に取られるが，seek メソッドを使用すると，指定した任意のバイト位置にアクセス位置を移動することができる．seek メソッドの基本的な使い方を表 15 に示す．

表 15: ファイル f に対する seek メソッドの基本的な使い方

書き方	説明
f.seek(n)	ファイルの先頭から n バイト目にアクセス位置を移動する．
f.seek(n,0)	同上
f.seek(n,1)	現在のアクセス位置から n バイト目にアクセス位置を移動する．
f.seek(n,2)	ファイルの末尾から n バイト目にアクセス位置を移動する．

※ seek メソッド実行後はシーク結果のバイト位置の値を返す．

4.5.2 サンプルプログラム

ファイルへのランダムアクセスを行うサンプルプログラム fram01.py を示す．このプログラムは 10 バイトのレコードを 3 件持つファイルに対するランダムアクセスを行う．

プログラム：fram01.py

```
1  # coding: utf-8
2
3  ########################################################
4  # ファイル内容を表示する関数                           #
5  ########################################################
6  def dumpf(f):
7      for i in range(3):
8          f.seek(i*10)
9          rec = f.read(10).decode('utf-8')
10         print(rec)
11     print('')
12
13 ########################################################
14 # ファイルのランダムアクセスの試み                     #
15 ########################################################
16 # ファイルの作成（空ファイルの準備）
17 f = open('fram01.dat','wb')
18 f.close()
19
20 # 読み書き可能な形で再度開く
21 f = open('fram01.dat','rb+')
22
23 #===== '0000:*****' の形（長さ10バイト）で3件書き込み =====
24 for i in range(3):
25     f.write( '0000:*****'.encode('utf-8') )
26 # 内容確認
27 dumpf(f)
28
29 #===== 2レコード目に '0002:  two' を書き込み =====
30 f.seek(20)
31 f.write( '0002:  two'.encode('utf-8') )
32 # 内容確認
33 dumpf(f)
34
35 #===== 0レコード目に '000z: zero' を書き込み =====
36 f.seek(0)
37 f.write( '000z: zero'.encode('utf-8') )
38 # 内容確認
39 dumpf(f)
40
41 #===== 1レコード目に '0001:  one' を書き込み =====
42 f.seek(10)
43 f.write( '0001:  one'.encode('utf-8') )
44 # 内容確認
45 dumpf(f)
46
47 # ファイルサイズの取得
48 p = f.seek(0,2)
49 print('File size:',p,'bytes')
50
51 #=== 終了 ===
52 f.close()
```

解説：

17,18 行目で空のファイルを新規に作成して，21〜25 行目でファル内容を初期化している．この結果，ファイルの内容が '0000:*****' というレコードを 3 つ持つ形となる．

6〜11 行目はファイルの内容全体を表示するための関数（名前：dumpf）で，プログラム内で度々呼び出されている．

30〜45 行目ではファイル内のバイト位置を seek メソッドで直接指定してランダムアクセスしている．(順不同の書き込み)

このプログラムを実行した結果を次に示す．

```
0000:*****    ←ファイル内容の初期化の結果
0000:*****
0000:*****

0000:*****
0000:*****
0002:  two     ← 2 レコード目に書き込み

000z:  zero    ← 0 レコード目に書き込み
0000:*****
0002:  two

000z:  zero
0001:  one     ← 1 レコード目に書き込み
0002:  two

File size:  30 bytes
```

4.6 エラーと例外の処理

「2.3.1.1 例外処理」では，プログラムの実行中に発生するエラーや例外を扱うための方法を解説した．try〜except の例外処理を適切に記述することで，処理系を中断することなくプログラムの実行を続けることができるが，ここでは更に，エラーや例外の際に得られるメッセージを扱うための traceback モジュールを紹介する．このモジュールを使用するには次のようにしてモジュールを読み込む．

```
import traceback
```

この後，try〜except の例外処理において，format_exc 関数を呼び出すと，例外やエラーの発生においてシステムが生成するメッセージを文字列型データとして取得することができる．

format_exc を用いたサンプルプログラム error01.py を示す．

プログラム：error01.py

```
1  # coding: utf-8
2
3  # モジュールの読込み
4  import traceback
5
6  try:
7  #    エラーが発生する部分
8      a = 2 * b
9  except:
10 #    エラー処理の部分
11     print('*** 例外が発生しました ***')
12     er = traceback.format_exc()
13     print(er.rstrip())
14     print('**************************\n')
15
16 print('--- プログラム終了 ---')
17 print('処理系の動作は中断されていません．')
```

解説：

プログラムの 8 行目に，値が割当てられていない変数（記号）b が記述されている．通常ならばこの行はエラーとなり，処理系の実行は中断するが，例外処理が施されているので，11 行目以降に処理が移行する．12 行目に

```
er = traceback.format_exc()
```

という記述があり，これによりエラーメッセージが文字列型のデータとして変数 er に格納される．

このプログラムを実行した例を次に示す.

```
*** 例外が発生しました ***
Traceback (most recent call last):
  File "error01.py", line 8, in <module>
    a = 2 * b
NameError:  name 'b' is not defined
**************************

--- プログラム終了 ---
処理系の動作は中断されていません.
```

5　TCP/IP による通信

ここでは，TCP/IP による通信機能を提供するいくつかのモジュールを紹介して，それらの使用方法について基本的な部分を説明する．

5.1　socket モジュール

TCP/IP 通信は**サーバ**と**クライアント**の 2 者間の通信（図 24）を基本とする．

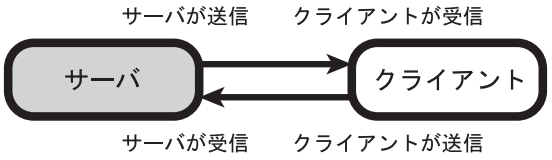


図 24: サーバとクライアントの通信

サーバ，クライアントはそれぞれ**ソケット**を用意しており，それを介して通信する．(図 25)

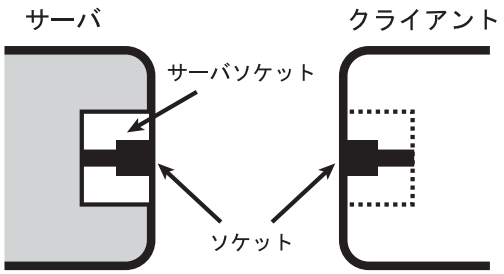


図 25: サーバ，クライアントそれぞれソケットを用意する

サーバはクライアントからの接続要求を受け付けるシステムであり，通常は接続待ちの状態で待機している．それに対してクライアントからの接続が要求され，接続処理が完了すると，両者の間で双方向の通信が可能となる．(図 26)

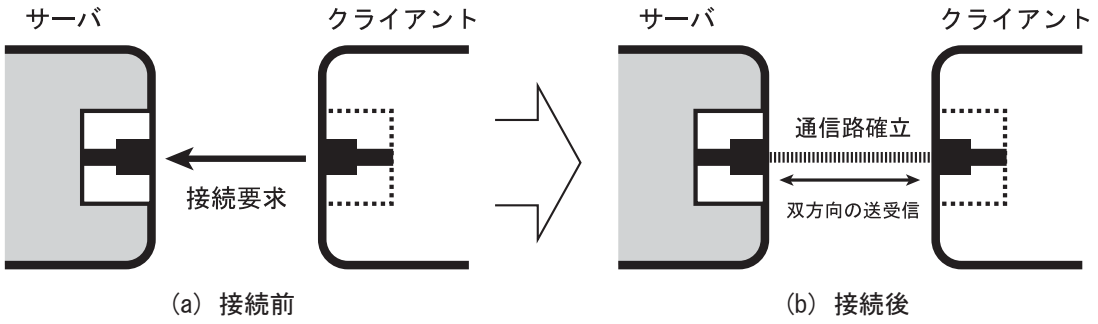


図 26: ソケットを介した接続

このような通信を実現するために Python には socket モジュールが用意されている．

socket モジュールは次のようにして読み込む．

```
import socket
```

5.1.1　ソケットの用意

ソケットは次のようにして生成する．

```
socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

この処理が正常に終了すると，ソケットオブジェクトが生成されて返される．ソケットを用意する方法はサーバ，クライアントの両方において同じである．

■ サーバ側プログラム

サーバ側ソケットには**ソケットオプション**[33] を設定する．これには次のように setsockopt メソッドを用いる．

ソケットオブジェクト`.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)`

次に，ソケットを **IP アドレス**と**ポート**にバインドする．これには次のようにして bind メソッドを用いる．

ソケットオブジェクト`.bind((`ホストアドレス`, `ポート`))`

ここまででサーバ側のソケットの準備がほぼ終わる．後は listen メソッドを使用してクライアントからの受信の準備をする．（下記参照）

ソケットオブジェクト`.listen()`

実際にクライアントからの接続要求を受信するには，次のように accept メソッドを使用する．

ソケットオブジェクト`.accept()`

この処理が完了すると，相手システム（クライアント）と通信するための新たなソケットオブジェクトとアドレスのタプルが返される．

サーバ側プログラムは，複数のクライアントからの接続要求を受け付けることができ，異なるクライアントから accept する度に，それぞれに対応するソケットオブジェクトが生成される．

■ クライアント側プログラム

ソケット生成後は，次のように connect メソッドを用いてサーバに接続を要求する．（その前に**タイムアウト**を設定しておくほうが良い）

ソケットオブジェクト`.settimeout(`秒数`)`
ソケットオブジェクト`.connect((`ホストアドレス`, `ポート`))`

5.1.2 送信と受信

ソケットを介して相手システムにメッセージを送信するには send メソッドを使用する．（下記参照）

ソケットオブジェクト`.send(`データ`)`

送信数データはバイトデータで，send メソッドの引数に与える．相手システムからのメッセージを受信するには，recv メソッドを使用する．（下記参照）

ソケットオブジェクト`.recv(`バッファサイズ[34] `)`

処理が正常に終わると，受信したメッセージがバイトデータとして返される．

通信が終了すると，次のように close メソッドを使用してソケットを終了しておく．

ソケットオブジェクト`.close()`

[33] 詳しい説明は TCP/IP の関連書籍や技術資料に譲る．
[34] 通常は 4096 にする．

5.1.3 サンプルプログラム

ここでは，サーバプログラムとクライアントプログラムが相互に接続してメッセージを交換するプログラムを示す．

サーバプログラム：test13-sv.py

```
# coding: utf-8

# 必要なモジュールの読込み
import socket

# 通信先（サーバ）の情報
host = '127.0.0.1'
port = 8001

# サービスの準備
sSock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sSock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
sSock.bind((host,port))
sSock.listen()

# 接続の受け付け
print('Waiting for connection...')
(cSock,cAddr) = sSock.accept()
print('Connection accepted!: ',cAddr)

# クライアントから受信
r = cSock.recv(4096)
print( 'クライアントから> ',r.decode('utf-8') )

# クライアントへ送信
msg = 'はい，届いていますよ．'.encode('utf-8')
cSock.send(msg)

# ソケットを終了する
cSock.close()
sSock.close()
```

クライアントプログラム：test13-cl.py

```
# coding: utf-8

# 必要なモジュールの読込み
import socket

# 通信先（サーバ）の情報
host = '127.0.0.1'
port = 8001

# 接続処理
cSock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
cSock.settimeout(3.0)
print('Connecting...')
cSock.connect((host, port))
print('Connection accepted!')

# サーバへ送信
msg = '受信できていますか？'.encode('utf-8')
cSock.send(msg)

# サーバから受信
r = cSock.recv(4096)
print( 'サーバから> ',r.decode('utf-8') )

# ソケットを終了する
cSock.close()
```

先にサーバプログラムを起動しておき，次にクライアントプログラムを起動すると両者が接続される．接続が確立

されると，クライアント側からサーバ側に対して

「受信できていますか？」

と送信される．サーバ側はこれを受信して標準出力に出力した後，クライアント側に対して

「はい，届いていますよ.」

と送信する．クライアント側はこれを受信して標準出力に出力する．実行例を次に示す.

サーバ側の実行例：

```
Waiting for connection...
Connection accepted!: ('127.0.0.1', 55735)
クライアントから> 受信できていますか？
```

クライアント側の実行例：

```
Connecting...
Connection accepted!
サーバから> はい，届いていますよ.
```

5.2 WWW コンテンツ解析

WWW は TCP/IP 通信の基礎の上に成り立っているサービスであり，Python と各種のモジュールを使用することで，WWW サーバへのリクエストの送信，コンテンツの取得，コンテンツの解析などが実現できる．ここでは，WWW コンテンツの取得と解析[35] に関する基礎的な事柄について説明する.

5.2.1 requests モジュール

requests モジュールは WWW サーバに対して各種のリクエストを送信したり，WWW サーバからコンテンツを取得するための基本的な機能を提供する．このモジュールはインターネットサイト http://docs.python-requests.org/ から入手できる．利用に先立って Python システム用にインストールしておく必要があるので，ソフトウェア管理ツール（Anaconda,pip など）[36] を使用してインストールする.

requests モジュールを Python で使用するには次のようにしてモジュールを読み込む.

```
import requests
```

【リクエストの送信に関するメソッド】

■ コンテンツの取得（GET リクエストの送信）

URL を指定してコンテンツを取得するには get メソッドを使用する.

書き方： `requests.get(コンテンツの URL)`

BASIC 認証を行ってコンテンツを取得するには次のようにする.

書き方： `r = requests.get(コンテンツの URL, auth=(ユーザ名, パスワード))`

■ フォームの送信（POST リクエストの送信）

フォームの内容[37] を WWW サーバに送信するには post メソッドを使用する．ことのきフォームの項目の「名前」

[35] これら一連の処理は**ウェブスクレイピング**と呼ばれている.
[36] 巻末付録「A.3 PIP によるモジュール管理」を参照のこと.
[37] < FORM >... </FORM >で記述された HTML コンテンツ.

と「値」の対応を Python の辞書型オブジェクト[38] にして与える．

書き方： `requests.post(コンテンツの URL, 辞書型オブジェクト)`

これら各メソッドが返すオブジェクトにリスエスト結果の情報が保持されている．

【取得したコンテンツに関するメソッド】

WWW サーバにリクエストを送信すると，それに対するサーバからの応答が返され，それら情報がメソッドの戻り値となる．以後これを**応答オブジェクト**と呼ぶ．

例. Wiki ペディアサイトの Python に関する記事の取得

```
r = requests.get('https://ja.wikipedia.org/wiki/Python')
```

この結果，r にサーバからの応答（WWW コンテンツを含む）が応答オブジェクトとして得られる．次に紹介する各種のメソッドを使用することで，取得した応答オブジェクトから様々な情報を取り出すことができる．

■ 取得したコンテンツ全体

WWW サーバから送られてきたコンテンツ全体は応答オブジェクトの text プロパティに保持されている．すなわち，

応答オブジェクト.text

とすることでコンテンツ全体を取得できる．先の例で得られた応答オブジェクトの text プロパティを表示すると次のようになる．

例. `r.text` の表示

```
<!DOCTYPE html>
<html class="client-nojs" lang="ja" dir="ltr">
<head>
<meta charset="UTF-8"/>
<title>Python - Wikipedia</title>
<script>document.documentElement.className = document.documentElement.className.
 replace( /(^|￥s)client-nojs(￥s|$)/, "$1client-js$2" );</script>
<script>(window.RLQ=window.RLQ||[]).push(function()mw.config.
 set("wgCanonicalNamespace":"","wgCanonicalSpecialPageName":false,
"wgNamespaceNumber":0,"wgPageName":"Python","wgTitle":"Python","wgCurRevisionId":64015453,
"wgRevisionId":64015453,"wgArticleId":993,"wgIsArticle":true,"wgIsRedirect":false,
"wgAction":"view","wgUserName":null,"wgUserGroups":["*"],"wgCategories":["プログラミング言語",
"オブジェクト指向言語","スクリプト言語","オープンソース","Python"],"wgBreakFrames":false,
"wgPageContentLanguage":"ja","wgPageContentModel":"wikitext","wgSeparatorTran
                        :
                  （以下省略）
                        :
```

■ 応答のステータスの取得

応答オブジェクトの status_code プロパティには，リクエスト送信に対する WWW サーバからの応答が保持されている．

例. `r.status_code` の表示

```
200
```

■ エンコーディング情報の取得

応答オブジェクトの encoding プロパティには，得られたコンテンツの文字コードに関する情報が保持されている．

[38] INPUT タグの 'NAME=' と 'VALUE=' に指定するものを辞書型オブジェクトにする．

例. r.encoding の表示

```
UTF-8
```

■ ヘッダー情報の取得

応答オブジェクトの headers プロパティには，得られたコンテンツのヘッダー情報（Python の辞書型オブジェクト）が保持されている．

例. r.headers の内容

```
{'Date':'Tue, 09 May 2017 06:22:40 GMT',
 'Content-Type':''text/html; charset=UTF-8',
 'Content-Length':'46335'.
 'Connection':'keep-alive',
 'Server':'mw1264.eqiad.wmnet',
 'Vary':'Accept-Encoding,Cookie,Authorization',
 'X-Powered-By':'HHVM/3.12.14',
 'Content-Encoding':'gzip',
          :
      （途中省略）
          :
 'Cache-Control':'private, s-maxage=0, max-age=0, must-revalidate',
 'Accept-Ranges':'bytes'}
```

■ Coocie 情報の取得

応答オブジェクトの cookies プロパティには，得られたコンテンツの Coocie 情報（Python の辞書型オブジェクト）が保持されている．

【Session オブジェクトに基づくアクセス】

WWW サーバとの間で情報をやり取りする際の**セッション情報**は，Session オブジェクトとして扱うことができ，Session オブジェクトに対してコンテンツの取得といったメソッドが使用できる．

例. Session オブジェクトに基づく処理

```
>>> s = requests.Session()  Enter     ← Session オブジェクトの生成
>>> r = s.get('https://ja.wikipedia.org/wiki/Python')  Enter     ←コンテンツの取得
```

この例は Session オブジェクトに基づいてコンテンツの取得を行うものであり，処理の結果として応答オブジェクト r が得られている．

本書での requests モジュールに関する解説は以上で終わるが，より多くの情報が配布元サイトや有志の開発者たちによって配信されているのでそれらを参照すること．

requests モジュールは WWW サーバとの通信における基本的な機能を提供するが，取得した WWW コンテンツの解析といったより高度な処理を行うには，更に別のモジュールを使用した方が良い．次に紹介するモジュールは，取得した WWW コンテンツ（より一般的に XML コンテンツ）の解析や，あるいはコンテンツの構築自体を可能とするものである．

5.2.2 Beautiful Soup モジュール

Beautiful Soup は HTML を含む XML 文書の解析や構築を可能とするモジュールである．このモジュールは

```
https://www.crummy.com/software/BeautifulSoup/
```

から入手することができるので，利用に際して Python システムにインストール[39] しておく．また利用するには，次

[39] Beautiful Soup モジュールは PIP で導入ができる．これに関しては巻末付録「A.3 PIP によるモジュール管理」を参照のこと．

のようにして Python に読み込む.

```
from bs4 import BeautifulSoup
```

以後は Beautiful Soup を BS と略す.

【BS におけるコンテンツの表現】

BS では，与えられた HTML（XML）コンテンツを独自の記憶構造 BeautifulSoup オブジェクト（以下 **BS オブジェクト**と略す）に変換して保持し，それに対して解析や編集の処理を行う.

次のような HTML コンテンツ test15.html がある場合を例に挙げて BS の使用方法を説明する.

HTML コンテンツ：test15.html

```
1  <!DOCTYPE html>
2  <html>
3
4  <head>
5  <meta charset="UTF-8">
6  <title>Pythonに関する情報 </title>
7  </head>
8
9  <body>
10 <h1>Pythonに関する情報 </h1>
11 <p>下記のリンクをクリックしてPythonに関する紹介を読んでください． </p>
12 <a href="https://ja.wikipedia.org/wiki/Python">ウィキペディアの記事へ</a>
13 <h2>Beautiful Soupに関する情報 </h2>
14 <p>下記のリンクをクリックしてBeautiful Soupのドキュメントを閲覧してください． </p>
15 <a href="https://www.crummy.com/software/BeautifulSoup/bs4/doc/">
16 Beautiful Soupのドキュメント </a>
17 </body>
18
19 </html>
```

この HTML コンテンツを読み込んで BS オブジェクトに変換する例を次に示す.

```
>>> from bs4 import BeautifulSoup  Enter        ←モジュールの読込み
>>> txt = open('test15.html','r',encoding='utf-8').read()  Enter  ← HTML ファイルの読込み
>>> sp = BeautifulSoup(txt,'html5lib')  Enter        ← BS オブジェクトの生成
```

この例では，ファイル test15.html の内容を txt に読込みし，それを BeautifulSoup のコンストラクタの引数に与えて BS オブジェクト sp を生成している．コンストラクタの 2 番目の引数にはコンテンツを解析するための**パーサ** [40] を指定する.

■ BS オブジェクトから文字列への変換（整形あり）

BS オブジェクトに対して prettify メソッドを実行すると，BS オブジェクトの内容を整形して文字列オブジェクトに変換する.

例. prettify メソッドによる変換

```
>>> print( sp.prettify() )  Enter      ←変換して表示
<!DOCTYPE html>
<html>
 <head>
  <meta charset="utf-8"/>
  <title>
   Python に関する情報
  </title>
 </head>
 <body>
  <h1>
   Python に関する情報
  </h1>
       :
```

[40] この例の html5lib 以外にも lxml などもある．それらはモジュールとしてインストールしておく.

```
        （途中省略）
            :
      <a href="https://www.crummy.com/software/BeautifulSoup/bs4/doc/">
        Beautiful Soup のドキュメント
      </a>
    </body>
  </html>
```

整形しないテキストは BS オブジェクトのプロパティ contents が保持している．（次の例参照）

BS オブジェクトの contents プロパティ

```
>>> print( sp.contents )
 ['html', <html><head>
<meta charset="utf-8"/>
<title>Python に関する情報</title>
</head>
          :
       （途中省略）
          :
</body></html>]
```

このようにリストの形式で保持されている．従って HTML のテキストは

BS オブジェクト.contents[1]

として取り出すことができる．

■ 指定したタグの検索

BS オブジェクトに対して指定したタグの検索を実行するには find_all メソッドを使用する．

例. ＜ a ＞タグを全て取り出す．

```
>>> sp.find_all('a')  Enter        ←検索実行
[<a href="https://ja.wikipedia.org/wiki/Python">ウィキペディアの記事へ</a>,
 <a href="https://www.crummy.com/software/BeautifulSoup/bs4/doc/"> Beautiful Soup のドキュメント</a>]
```

見つかったタグをリストにして返す．

■ コンテンツの階層構造

BS オブジェクトの構造は基本的にリストである．従って先に説明した contents プロパティに要素の添え字を付けて更に配下の contents プロパティを取り出すということを繰り返すことで全ての要素にアクセスできる．

contents に添え字を付けて取り出したものは，1 つのタグで記述された文であり，それが持つ name プロパティにはそのタグの名前が保持されている．

例. タグの取り出し

```
>>> sp.contents[1].contents[2].name  Enter    ← body タグの取り出し
'body'                    ←処理結果
```

本書での Beautiful Soup モジュールに関する解説は以上で終わるが，より多くの情報が配布元サイトや有志の開発者たちによって配信されているのでそれらを参照すること．

6　外部プログラムとの連携

外部のプログラムを Python プログラムから起動する方法について説明する．C 言語や Java といった処理系でプログラムを翻訳，実行する場合と比べると，インタプリタとしての Python 処理系ではプログラムの実行速度は非常に遅い．従って，大きな実行速度を要求する部分の計算処理は，より高速な外部プログラムに委ねるべきである．実際に，科学技術系の計算処理やニューラルネットワークのシミュレーション機能を Python 用に提供するパッケージ（モジュール）の多くは外部プログラムとの連携を応用している．

計算速度の問題に限らず，外部の有用なプログラムを呼び出して Python プログラムと連携させることは，高度な情報処理を実現するための有効な手段となる．

Python から外部プログラムを起動するには subprocess モジュールを使用する．このモジュールは次のようにして Python 処理系に読み込む．

```
import subprocess
```

6.1　外部プログラムを起動する方法

＜外部プログラムの起動＞
書き方：　subprocess.run(**コマンド文字列**, shell=True)
　オペレーティングシステム（OS）のコマンドシェルを起動して「コマンド文字列」で与えられたコマンドを実行する．run メソッドはコマンドの終了を待ち，CompletedProcess オブジェクトを返す．コマンド文字列は，コマンド名と引数（群）を空白文字で区切って並べたものである．

この方法はコマンドシェルを介するので，外部プログラム起動時に環境変数の設定などが有効である．ただし，シェルも 1 つのプロセスとして起動するので，シェルを介さずに外部プログラムを起動する場合に比べると，システムに対するプロセス管理の負荷が高くなる．シェルを介さずに外部プログラムを起動するには，run メソッドの引数にキーワード引数 shell=False を指定する．

例. Windows のコマンド 'dir' を発行する例.

```
>>> import subprocess  Enter        ←モジュールの読み込み
>>> subprocess.run('dir *.eps',shell=True)  Enter  ← dir コマンドでファイルの一覧表示
  Volume in drive C has no label.
  Volume Serial Number is 18EE-2F9E

  Directory of C:￥Users￥katsu￥TeX￥Python

2016/07/23 18:16 530,687 ClientServer.eps
2016/07/23 19:09 511,071 CSsocket0.eps
2016/07/23 19:22 546,418 CSsocket1.eps
2017/05/21 18:10 851,177 Earth_small.eps
2017/04/29 14:23 528,051 HBoxVBox.eps
               :
         （以下省略）
               :
```

これは，Windows のコマンドシェル（コマンドプロンプト）である cmd.exe を起動して，その内部コマンド[41] である dir を実行している例である．呼び出された外部プログラムは**サブプロセス**として実行される．

6.1.1　標準入出力の受け渡し

サブプロセス（外部プログラム）の標準入出力に対してデータを送受信する方法について説明する．独立したプログラム同士が通信するための代表的な方法に**ソケット**と**パイプ**があり，subprocess モジュールはパイプを介した標準

[41] cmd.exe によって解釈されて実行されるコマンド（cmd.exe 自体が持つ機能）であり，dir 自体は単独の実行形式（*.exe）プログラムではない．

入出力の送受信を実現するための簡便な方法を提供している．

＜サブプロセスとのパイプを介した通信＞

書き方： subprocess.Popen(**コマンド文字列**, shell=True,
stdin=subprocess.PIPE, stdout=subprocess.PIPE, stderr=subprocess.PIPE)

キーワード引数 stdin, stdout, stderr に subprocess.PIPE を与えることで，サブプロセスの標準入力，標準出力，標準エラー出力が Python プログラム側からアクセスできるようになる．

Popen メソッドを実行すると，戻り値として Popen オブジェクトが返され，これのプロパティ stdin, stdout, stderr に対して各種の入出力用メソッドを使用することで，サブプロセスに対して実際にデータを送受信することができる．

【サンプルプログラム】

ここでは，C 言語で作成したシミュレーションプログラムを Python から起動する例を挙げて説明する．

■ シミュレーション・プログラム

円の軌跡を描く次のようなダイナミクスをシミュレートするプログラムを考える．

$$\frac{dx}{dt} = -y, \quad \frac{dy}{dt} = x$$

このダイナミクスを離散化すると次のような式となる．

$$\Delta x = -y \cdot \Delta t, \quad \Delta y = x \cdot \Delta t$$

これを実現するプログラムは，

```
x = x - y*dt,   y = y + x*dt
```

となり，x,y に適当なる初期値を与えてこのプログラムを繰り返すとダイナミクスのシミュレーションが実現できる．これを C 言語で記述したものが sbpr01.c である．

このプログラムは $(x, y) = (10.0, 0.0)$ を初期値として時間の刻み幅 dt を 10^{-8} として円の軌跡をシミュレートする．軌跡が円周を一周するとシミュレーションが終了する．

外部プログラム：sbpr01.c

```
#include    <stdio.h>
#include    <time.h>

int main()
{
    double  x, y, dt;    /* 座標と微小時間 */
    long    c;           /* ループカウンタ */
    clock_t t1, t2;      /* 時間計測用変数 */

    /* シミュレーションの初期設定 */
    x = 10.0;   y = 0.0;
    dt = 0.00000001;     /* 時間間隔 */
    t1 = clock();   /* 開始時刻 */
    for ( c = 0L; c < 628318530L; c += 1L ) {
        y += x * dt;
        x -= y * dt;
        /* 中ヌキ出力 */
        if ( c % 200000L == 0 ) {
            printf("%16.12lf,%16.12lf\n",x,y);
        }
    }
    t2 = clock();   /* 終了時刻 */

    /* 計算時間の表示（標準エラー） */
```

```
25      fprintf(stderr,"%ld (msec)", t2-t1);
26  }
```

このプログラムを翻訳して実行すると，標準出力に次のような形で x, y の値を出力する．

```
10.000000000000, 0.000000100000
9.999979999707, 0.020000086666
9.999919999507, 0.039999993333
9.999819999640, 0.059999739999
9.999680000507, 0.079999246666
9.999500002667, 0.099998433337
                :
          （以下省略）
```

このプログラムからの出力をテキストデータとして保存して gnuplot [42] でプロットした例を図 27 に示す．

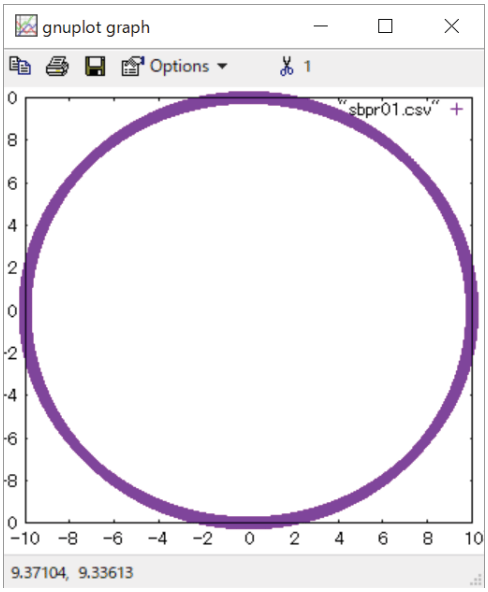


図 27: gnuplot によるプロット例

このシミュレーションは dt を非常に小さく取っており，ダイナミクスが終了するまで x, y の移動計算を 6 億回以上実行する．これは Python のプログラムとして実行するには時間的にも適切ではないため，C 言語で実装した．

次に，このシミュレーションプログラムを Python プログラムから呼び出して，データプロットのためのモジュール matplotlib [43] を使ってプロットする例を示す．

Python 側のプログラムを sbpr01.py に示す．

プログラム：sbpr01.py

```
1   # coding: utf-8
2   # モジュールの読み込み
3   import subprocess
4   import matplotlib.pyplot as plt
5
6   # サブプロセスの生成
7   pr = subprocess.Popen( 'sbpr01.exe', stdout=subprocess.PIPE, shell=True )
8
9   # サブプロセスの標準出力からのデータの受け取り
10  lx = [];    ly = []     # これらリストのデータを蓄積
11  buf = pr.stdout.readline().decode('utf-8').rstrip() # 初回読取り（1行）
12  while buf:
```

[42] オープンソースとして公開されているデータプロットツール．
[43] インターネットサイト https://matplotlib.org/ でモジュールとドキュメントが公開されている．

```
13      [bx,by] = buf.split(',')     # CSVの切り離し
14      lx.append( float(bx) ); ly.append( float(by) )  # データの蓄積
15      # 次回読取り（1行）
16      buf = pr.stdout.readline().decode('utf-8').rstrip()
17
18  # matplotlibによるプロット
19  plt.plot(lx, ly, 'o-', label="circle")
20  plt.xlabel("x")
21  plt.ylabel("y")
22  plt.legend(loc='best')
23  plt.show()
```

これはWindows環境における例である．先のシミュレーションプログラムはWindowsの実行形式ファイルsbpr01.exeとして作成されており，これとsbpr01.pyは同じディレクトリに配置されているものとする．

このプログラムを実行すると，シミュレーション結果が図28のようなプロットとして表示される．

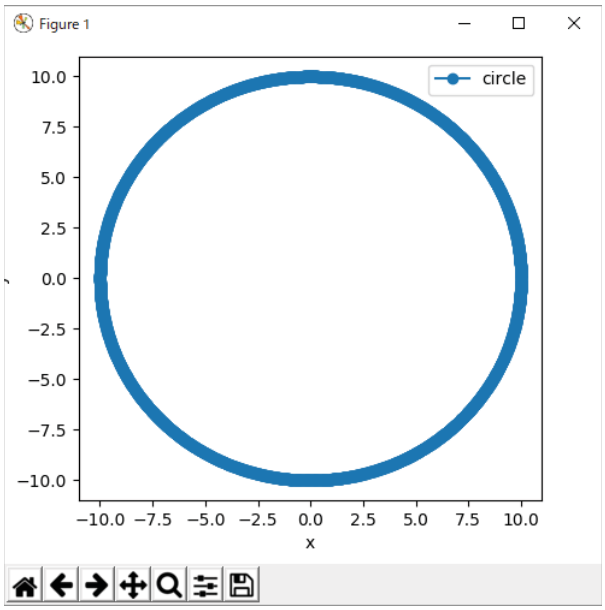


図 28: matplotlibを用いたプロット

外部プログラムとのデータの送受信を非同期に行う場合は，入出力の送受信を繰り返す処理を独立したスレッドで実行すると良い．

6.1.1.1 外部プログラムの標準入力のクローズ

外部プログラムの標準入力への送信を終了してクローズするにはcloseメソッドを用いる．具体的には，サブプロセスオブジェクトのstdinに対してcloseメソッドを実行する．

例. サブプロセスprのstdinを閉じる

```
pr.stdin.close()
```

多くのコマンドツールは，標準入力が閉じられると終了する．そのようなコマンドツールをPythonから起動した場合，この方法で当該サブプロセスを終了することができる．

6.1.2 非同期の入出力

先の例（sbpr01.py）では，サブプロセスからの出力を受け取る繰り返し処理が終了するまで他の処理はできない．サブプロセスとのパイプを介した入出力を非同期に実行する最も簡単な方法は，入出力の繰り返し処理を独立したス

レッドで実行すること[44]である．ここではサンプルプログラムを示しながらそのための方法を例示する．

【サンプルプログラム】

一定時間が経過する毎に経過時間を表示するプログラム（一種のタイマー）を C 言語で記述したものを sbpr02.c に示す．

外部プログラム：sbpr02.c

```
#include    <stdio.h>
#include    <time.h>

int main(ac,av)
int ac;
char **av;
{
    int     n, c = 1;
    clock_t d, t1, t2;  /* 時間計測用変数 */

    if ( ac < 3 ) {
        fprintf(stderr,"Usage: sbpr02 Duration(ms) Iteration...");
        return(-1);
    } else {
        sscanf(av[1],"%ld",&d);
        sscanf(av[2],"%d",&n);
    }

    t1 = clock();       /* 開始時刻 */
    printf("%ld\n",t1);
    while ( -1 ) {
        t2 = clock();   /* 現在時刻 */
        if ( t2 - t1 >= d ) {
            printf("%ld\n",t2);
            fflush(stdout); /* 出力バッファをフラッシュ */
            t1 = t2;
            c++;
            if ( c > n ) {
                break;
            }
        }
    }

    fprintf(stderr,"%s %s %s: finished.\n",av[0],av[1],av[2]);
}
```

このプログラムは，起動時の第 1 引数に経過時間 (ms) を，計測回数を第 2 引数に与えるものである．このプログラムを翻訳して実行した例を次に示す．

```
C:￥Users￥katsu￥Python> sbpr02 3000 2          ← 3 秒経過を 2 回通知する指定
0
3001
6001
sbpr02 3000 2:  finished.
```

これを外部プログラムとして同時に複数並行して起動するプログラムの例を sbpr02.py に示す．

プログラム：sbpr02.py

```
# coding: utf-8
# モジュールの読み込み
import subprocess
from threading import Thread

# サブプロセスの生成
pr1 = subprocess.Popen( 'sbpr02.exe 3000 3', stdout=subprocess.PIPE, shell=True )
pr2 = subprocess.Popen( 'sbpr02.exe 2250 4', stdout=subprocess.PIPE, shell=True )
```

[44] UNIX 系 OS の**デーモンプロセス**や Windows の**サービス**のように，入出力や通信の処理を受け付ける常駐型プログラムを Python で実現するには asyncio モジュールを使用するのが良い．ただし本書では解説しない．

```
 9
10 # pr1からの入力を受け取る関数
11 def Exe1():
12     while True:
13         buf = pr1.stdout.readline().decode('utf-8').rstrip()
14         if buf:
15             print('プロセス1:',buf)
16         else:
17             break
18
19 # pr2からの入力を受け取る関数
20 def Exe2():
21     while True:
22         buf = pr2.stdout.readline().decode('utf-8').rstrip()
23         if buf:
24             print('プロセス2:',buf)
25         else:
26             break
27
28 # サブプロセスの入力を受け付けるスレッド
29 th1 = Thread( target=Exe1, args=() )
30 th2 = Thread( target=Exe2, args=() )
31
32 # スレッドの起動
33 print('*** 実行開始 ***')
34 th1.start()
35 th2.start()
36
37 # スレッド終了の待ち受け
38 th1.join()
39 th2.join()
40 print('*** 実行終了 ***')
```

解説：

7,8 行目で外部プログラムを 2 つサブプロセスとして生成して，それぞれ pr1, pr2 としている．それらの標準出力をパイプ経由で取得する処理を，関数 Exe1, Exe2 として定義（11〜17 行目）している．それら関数を独立したスレッドとして生成（29,30 行目）して起動（34,35 行目）している．それらスレッドは同時に並行して別々に動作する．38,39 行目では，2 つのスレッドが終了するのを待ち受けている．スレッドの扱いに関しては「4.3 マルチスレッドプログラミング」を参照のこと．

このプログラムを実行した例を次に示す．

```
*** 実行開始 ***
プロセス 2:  0
プロセス 2:  2251
プロセス 1:  0
プロセス 1:  3000
プロセス 2:  4504
プロセス 1:  6000
プロセス 2:  6768
 sbpr02.exe 3000 3:  finished.
プロセス 1:  9000
プロセス 2:  9031
 sbpr02.exe 2250 4:  finished.
*** 実行終了 ***
```

6.1.3 外部プロセスとの同期（終了の待ち受け）

Popen メソッドで生成されたプロセスが終了するのを待ち受けるには wait メソッドを使用する．すなわち，Popen で生成されたプロセス pr の終了に同期して，終了するまで処理をブロックする（終了を待ち受ける）には，

```
pr.wait()
```

とする．

7　サウンドの入出力

ここでは基本的なサウンドデータの取り扱い方法について説明する．内容は

1. データ（ファイル）としてのサウンドの入出力
2. リアルタイムのサウンド入力と再生

の 2 つである．

7.1　基礎知識

この章の内容を理解するに当たり必須となる知識を次に挙げる．

量子化ビット数

音の最小単位を表現するビット数であり，ある瞬間の音の大きさを何ビットで表現するかを意味する．量子化ビット数が大きいほど取り扱う音の質（音質）が良い．一般的な音楽 CD などでは量子化ビット数は 16（2 バイト）である．

サンプリング周波数（サンプリングレート）

連続的に変化するアナログの音声を**サンプリング**によって離散化してデジタル情報として扱うが，1 秒間に音声を何回サンプリングするか（1 秒の音声を何個に分割するか）がサンプリング周波数である．この数値が大きいほど扱う音声の音質が良くなる．一般的な音楽 CD などではサンプリング周波数は 44.1KHz であり，1 秒の音声を 44,100 個に分割している．

チャネル数

同時に取り扱う音声の本数が**チャネル数**である．日常的に鑑賞するオーディオはアナログ／デジタルの違いに関わらず，左右の音声を別々に扱っており，左右それぞれのスピーカーから再生されることが一般的である．この場合は「左右合計で 2 チャネルの音声」を扱っていることになる．高級なオーディオセットでは 2 チャネル以上の音声の取り扱いが可能なものもある．通常の場合**ステレオ音声**は 2 チャネル，**モノラル音声**は 1 チャネルの扱いを意味する．

7.2　WAV 形式ファイルの入出力：wave モジュール

wave モジュールを使用することで，WAV 形式[45] のサウンドデータをファイルから読み込んだり，ファイルに書き出すこと[46] ができる．

WAV 形式ファイルの取り扱いも通常のファイルの場合と基本的には同じであり，

1. WAV 形式ファイルをオープンする
2. WAV 形式ファイルから内容を読込む（あるいは書き込む）
3. WAV 形式ファイルを閉じる

という流れとなる．

wave モジュールを使用するには次のようにしてモジュールを読み込む．

```
import wave
```

7.2.1　WAV 形式ファイルのオープンとクローズ

通常のファイルの取り扱いと同様に，WAV 形式ファイルを開く場合も「読込み」もしくは「書き込み」のモードを指定する．開く場合には wave のメソッド open を，閉じる場合は close を使用する．

[45] Microsoft 社と IBM 社が開発した音声フォーマットである．音声データの圧縮保存にも対応しているが，非圧縮の PCM データを扱うことが多い．

[46] wave モジュールはサウンドデータの解析などに使用するものであり，音声を録音・再生する機能はない．録音や再生には別のモジュール（PyAudio など）を使用する．

＜ WAV 形式ファイルのオープンとクローズ＞

書き方： wave.open(**ファイル名**, **モード**)

WAV 形式ファイルのファイル名を文字列で与え，モードには 'rb'（読込み」）か'wb'（書き込み）を指定する．open メソッドが正常に終了すると「読込み」の場合は Wave_read オブジェクトが,「書き込み」の場合は Wave_write オブジェクトが返され，それらに対して音声データの読み書きを行う．

読み書きの処理を終える際は close メソッドを Wave_read オブジェクトや Wave_write オブジェクトに対して実行する．

書き方： Wave_read/Wave_write **オブジェクト**.close()

7.2.1.1 WAV 形式データの各種属性について

WAV 形式ファイルを開いて生成した Wave_read オブジェクトから基本的な属性情報を取り出すには表 16 のようなメソッドを使用する．

表 16: Wave_read オブジェクトからの情報の取得

Wave_read wf に対するメソッド	得られる情報
`wf.getnchannels()`	チャネル数
`wf.getsampwidth()`	量子化ビット数をバイト表現にした値
`wf.getframerate()`	サンプリング周波数 (Hz)
`wf.getnframes()`	ファイルの総フレーム数

フレームについて

wave モジュールで WAV 形式データを読み込む場合，音声再生の単位となる**フレーム**の数を指定する．すなわち「n フレームを WAV 形式ファイルから読み込む」という形の扱いとなる．表 16 の中にある**総フレーム数**は当該ファイルを構成する全てのフレームの数である．

7.2.2 WAV 形式ファイルからの読込み

Wave_read オブジェクトからフレームデータを読み込むには readframes メソッドを使用する．

＜フレームデータの読込み＞

書き方： **Wave_read オブジェクト**.readframes(**フレーム数**)

引数には読み込むフレーム数を指定する．実際のファイルに指定したフレーム数がなければ存在するフレームのみの読込みとなる．このメソッドの実行により読み込まれた一連のフレームがバイト列として返される．

7.2.3 サンプルプログラム

Wave_read オブジェクトから各種の属性情報を取得するプログラム wave00.py を次に示す．

プログラム：wave00.py

```
1 # coding: utf-8
2
3 # モジュールの読込み
4 import wave
5
6 # WAV形式ファイルのオープン
7 wf = wave.open('sound01.wav','rb')
8 print( 'チャネル数:\t\t',wf.getnchannels() )
```

```
 9 print( '量子化ビット数:\t\t',8*wf.getsampwidth(), '\t(bit)' )
10 print( 'サンプリング周波数:\t',wf.getframerate(), '\t(Hz)' )
11 print( 'ファイルのフレーム数:\t',wf.getnframes() )
12
13 b = wf.readframes(1)
14 print( '1フレームのサイズ:\t',len(b), '\t(bytes)' )
15
16 wf.close()
```

このプログラムを実行した例を次に示す.

```
チャネル数:             1
量子化ビット数:         16     (bit)
サンプリング周波数:     22050   (Hz)
ファイルのフレーム数:   1817016
1フレームのサイズ:      2       (bytes)
```

7.2.4 読み込んだフレームデータの扱い

WAV 形式ファイルから取得したフレームデータは適切な型のデータに変換することで音声データ解析などに使用することができる．数値データ解析のためのモジュールである numpy を用いて音声データを解析用の配列データに変換する例をサンプルプログラム wave01.py に示す.

プログラム：wave01.py

```
 1 # coding: utf-8
 2
 3 # モジュールの読込み
 4 import wave
 5 import numpy
 6
 7 # WAV形式ファイルのオープン
 8 wf = wave.open('sound01.wav','rb')
 9
10 # 各種属性の取得
11 chanel = wf.getnchannels()
12 qbit = 8*wf.getsampwidth()
13 freq = wf.getframerate()
14 frames = wf.getnframes()
15
16 b = wf.readframes(frames)
17
18 data = numpy.frombuffer(b, dtype='int16')
19 print( 'データタイプ:', type(data))
20 print( 'データ数:',     len(data) )
21
22 ############################################################
23 #   音声の波形データが data に格納されています.           #
24 #   波形の振幅は  -32768  ～  32767  です.                 #
25 #   このデータを numpy をはじめとするパッケージで          #
26 #   解析処理することができます.                           #
27 ############################################################
28
29 wf.close()
```

このプログラムの 18 行目でフレームデータを numpy の ndarray オブジェクトに変換しており，解析が可能となる．ステレオ（2 チャンネル）音声の場合は

[左，右，左，右，…]

の順序で数値が格納されている.

※ numpy モジュールについての解説は他の情報源に譲る.

先のプログラム wave01.py を少し拡張した，WAV 形式データの波形をプロットするプログラム wave01-2.py を次に示す.

プログラム：wave01-2.py

```
1  # coding: utf-8
2
3  # モジュールの読込み
4  import wave
5  import numpy
6  import matplotlib.pyplot as plt
7
8  # WAV形式ファイルのオープン
9  wf = wave.open('aaa.wav','rb')
10
11 # 各種属性の取得
12 chanel = wf.getnchannels()
13 qbit = 8*wf.getsampwidth()
14 freq = wf.getframerate()
15 frames = wf.getnframes()
16
17 # データをnumpyの配列に変換
18 b = wf.readframes(frames)
19 d = numpy.frombuffer(b, dtype='int16')
20 d_left = d[0::2]         # 左音声
21 d_right = d[1::2]        # 右音声
22 t = [i/freq for i in range(len(d_left))]     # 時間軸の生成
23
24 # WAV形式ファイルのクローズ
25 wf.close()
26
27 # matplotlibによるプロット
28 (fig, ax) = plt.subplots(2, 1, figsize=(10,5))
29 plt.subplots_adjust(hspace=0.6)
30
31 ax[0].plot(t, d_left, linewidth=1)
32 ax[0].set_title('left')
33 ax[0].set_ylabel('level')
34 ax[0].set_xlabel('t (sec)')
35 ax[0].grid(True)
36
37 ax[1].plot(t, d_right, linewidth=1)
38 ax[1].set_title('right')
39 ax[1].set_ylabel('level')
40 ax[1].set_xlabel('t (sec)')
41 ax[1].grid(True)
42
43 plt.show()
```

解説

19 行目で WAV 形式のデータを numpy の配列データに変換し，更に 20-21 行名で左右のデータとして分離し，最終的に matplotlib モジュールによって可視化している．このプログラムを実行して表示されたグラフの例を図 29 に示す.

※ matplotlib モジュールについての解説は他の情報源に譲る.

7.2.5 WAV 形式データを出力する例

WAV 形式データを出力するには Wave_write オブジェクトに対して各種の属性情報を設定（表 17 参照）し，バイナリデータとして作成した波形データを出力する.

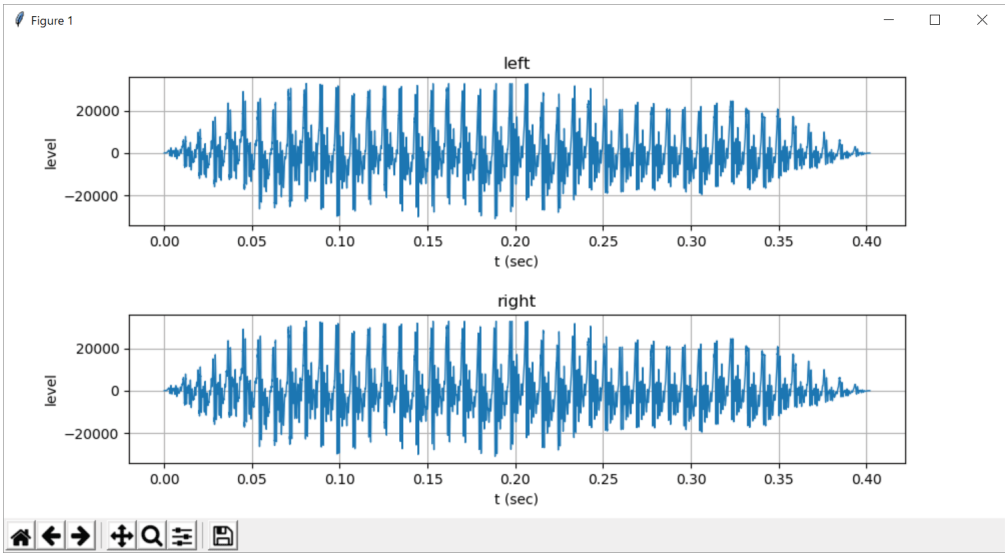


図 29: WAV 形式データのプロット

表 17: Wave_write オブジェクトへの属性情報の設定

Wave_write wf に対するメソッド	設定する属性情報
`wf.setnchannels(n)`	n=チャネル数
`wf.setsampwidth(n)`	n=量子化ビット数をバイト表現にした値
`wf.setframerate(f)`	f=サンプリング周波数 (Hz)

1KHz の正弦波形のデータを生成して，それを WAV 形式データとしてファイルに保存するプログラム wave02.py を次に示す.

プログラム：wave02.py

```
# coding: utf-8
# モジュールの読込み
import wave
import struct
from math import sin

##################################################
# 1KHzの正弦波サウンドを44.1KHzでサンプリング    #
# する際のsin関数の定義域の刻み幅                #
##################################################
dt = (3.14159265359*2)*1000 / 44100

##################################################
#    10秒間の正弦波データを生成                  #
##################################################
t = 0.0
dlist = []
for x in range(441000): # 10秒間のデータ
    y = int(32767.0 * sin(t))
    dlist.append(y)
    t += dt

# 数値のリストをバイナリデータに変換
data = struct.pack('h'*len(dlist), *dlist)
print('データサイズ:',len(data),'バイト')

##################################################
#    WAV形式で保存                               #
##################################################

```

```
31 # WAV形式ファイルの作成（オープン）
32 wf = wave.open('out01.wav','wb')
33
34 # 各種属性の設定
35 wf.setnchannels(1)       # チャネル数=1
36 wf.setsampwidth(2)       # 量子化ビット数=2バイト（16ビット）
37 wf.setframerate(44100)   # サンプリング周波数=44.1KHz
38
39 # ファイルへ出力
40 wf.writeframesraw(data)
41
42 wf.close()
```

解説

11〜21 行目の部分で正弦波形のデータをリスト dlist として作成している．それを 24 行目でバイナリデータ data に変換している．変換には struct モジュール [47] を使用している．

32 行目で WAV 形式ファイルを作成して，35〜37 行目の部分で WAV 形式として必要な各種の属性情報を設定している．実際の出力は 40 行目で行っており，バイト列をそのまま WAV 形式データとして出力するためのメソッド writeframesraw を使用している．WAV 形式データの出力にはこの他にも writeframes メソッドもある．writeframesraw と writeframes の違いは，出力するフレーム数の設定に関することであるが，詳しくは公式サイトのドキュメントを参照のこと．

7.2.6 サウンドのデータサイズに関する注意点

本書では解説を簡単にするために，WAV 形式ファイルの入出力を 1 度で行っている．すなわち，音声ファルの全フレームを一度に読み込んだり，全フレームを一度でファイルに書き込むという形を取った．実用的なオーディオアプリケーションでは扱う音声データが大きい（数十 MB〜数百 MB）場合が多く，全フレームを一度に読み書きするのはシステムの記憶資源の関係上，現実的でないことが多い．実際には使用するシステム資源から判断して「現実的な入出力の単位」のサイズを定めて，そのサイズの入出力を繰り返すという形でサウンドデータを取り扱うことが望ましい．これに関する具体的な方法については,「7.3.2 WAV 形式サウンドファイルの再生」,「7.3.3 音声入力デバイスからの入力」の所で実装例を挙げて示す．

[47] Python の標準ライブラリである．

7.3　サウンドの入力と再生：PyAudio モジュール

PyAudio モジュールを利用することで，リアルタイムの音声入出力ができる．先に説明した wave モジュールは WAV 形式の音声ファイルを取り扱うための基本的な機能を提供するものであり，これと PyAudio を併せて利用することでサウンドの入出力とファイル I/O が実現できるので，実用的なサウンドの処理が可能となる．

PyAudio は PortAudio ライブラリの Python バインディングである．PortAudio はクロスプラットフォーム用のオープンソースのサウンドライブラリであり，開発と保守は PortAudio community によって支えられている．PortAudio に関する情報はインターネットサイト http://www.portaudio.com/ を参照のこと．

PyAudio モジュールは PIP で導入することができる．これに関しては巻末付録「A.3 PIP によるモジュール管理」を参照のこと．また，PyAudio に関して本書で解説していない情報については PyAudio のドキュメントサイト https://people.csail.mit.edu/hubert/pyaudio/docs/ を参照のこと．

PyAudio モジュールを使用するには，次のようにしてモジュールを読み込んでおく．

```
import pyaudio
```

7.3.1　ストリームを介したサウンド入出力

PyAudio では，サウンドの入力源や再生のための出力先を**ストリーム**として扱う．すなわち，サウンドの入力はストリームからデータを取得する形で行い，サウンドの再生はストリームにデータを書き込む形で行う．

【PyAudio オブジェクト】

PyAudio を用いたサウンド入出力は **PyAudio オブジェクト**を基本とする．PyAudio の使用に際しては予め PyAudio オブジェクトを生成しておく．

＜ PyAudio オブジェクトの生成＞
書き方：　pyaudio.PyAudio()
　この結果 PyAudio オブジェクトが生成される．
実行例：　`p = pyaudio.PyAudio()`
　実行結果として PyAudio オブジェクト p が得られる．

PyAudio オブジェクトに対して open メソッドを使用することでストリームが得られる．open メソッドの引数にサウンドの取り扱いに関する属性情報などを与える．

＜ストリームの生成＞
書き方：　**PyAudio オブジェクト**.open(channels=**チャネル数**, rate=**サンプリング周波数**,
format=**量子化に関する情報**, input=**入力機能を使用するか**,
output=**出力機能を使用するか**)

　これを実行した結果ストリームが得られる．キーワード引数 input, output には真理値を指定する．input, output に同時に True を設定することができ，その場合は「入出力両用」のストリームが得られる．

キーワード引数 format
　キーワード引数 format には次のようにして生成した値を指定する．
　PyAudio オブジェクト.get_format_from_width(**バイト数**)
バイト数には量子化ビット数をバイト単位で表現した整数値を指定する．

ストリームの使用を終える場合は，ストリームのオブジェクトに対して stop_stream メソッドを実行し，更に close メソッドを実行する．また PyAudio の使用を終える場合は，PyAudio オブジェクトに対して terminate メソッドを実行する．

7.3.2 WAV 形式サウンドファイルの再生

WAV 形式ファイルから読み込んだフレームデータをシステムのサウンド出力で再生する方法についてプログラム例を挙げて説明する．まずは**ブロッキングモード**と呼ばれる素朴な形での再生（プログラム pyaudio01.py）について説明する．

プログラム：pyaudio01.py

```
1  # coding: utf-8
2  # モジュールの読込み
3  import wave
4  import pyaudio
5
6  ########################################################
7  # WAV形式サウンドファイルの読込み                      #
8  ########################################################
9  # ファルのオープン
10 wf = wave.open('sound01.wav', 'rb')
11 # 属性情報の取得
12 ch = wf.getnchannels()       # チャネル数
13 qb = wf.getsampwidth()       # 量子化ビット数（バイト数）
14 fq = wf.getframerate()       # サンプリング周波数
15 frames = wf.getnframes()     # ファイルの総フレーム数
16 # 内容の読込み
17 buf = wf.readframes(frames)
18 # ファイルのクローズ
19 wf.close()
20
21 ########################################################
22 # PyAudioによる再生                                    #
23 ########################################################
24 # PyAudioオブジェクトの生成
25 p = pyaudio.PyAudio()
26 # ストリームの生成
27 stm = p.open( format=p.get_format_from_width(qb),
28                 channels=ch, rate=fq, output=True )
29 # ストリームへのサウンドデータの出力
30 stm.write(buf)
31 # ストリームの終了処理
32 stm.stop_stream()
33 stm.close()
34 # PyAudioオブジェクトの廃棄
35 p.terminate()
```

解説

10〜19 行目で WAV 形式サウンドデータを取得して，バイトデータ buf として保持している．生成したストリーム stm に対して write メソッドを使用（30 行目）して buf の内容を書き込んでいる．

pyaudio01.py（ブロッキングモードの再生）では，サウンドの再生がメインプログラムのスレッドで実行されるので，サウンドの再生が終了するまで write メソッド部分でプログラムの流れがブロックされる．（待たされる）
実用的なアプリケーションでは，サウンド再生はメインプログラムとは別のスレッドで実行されるべきであり，サウンド再生中もメインプログラムの流れはブロックされるべきではない．次にサウンド再生を別のスレッドで実行する方法について説明する．

【コールバックモードによる再生の考え方】

ストリームへのサウンド出力処理には**コールバックモード**と呼ばれる形態がある．これは，WAV 形式ファイルからの

音声フレームの入力と，それをストリームに出力するサイクルを**コールバック関数**という形で定義しておき，PyAudio がこの関数を別スレッドで実行するものである．コールバック関数の呼び出しは，WAV 形式ファイルから読み込むべきフレームが無くなるまで自動的に繰り返される．

コールバック関数は次のような形で定義する．

＜コールバック関数の定義＞
書き方： def **関数名 (入力データ, フレーム数, 時間に関する情報, フラグ)：**
（WAV 形式ファイルからの「フレーム数」だけ読み込む処理）
return(**読み込んだバイトデータ**, pyaudio.paContinue)

このように定義した関数を PyAudio が自動的に繰り返し呼び出す．「入力データ」にはストリームからサウンドを入力する際に取得したバイトデータが与えられる．（再生処理の場合はこの仮引数は無視する）
「フレーム数」は一度に読み取るべきフレームの数であり，コールバック関数を呼び出す際に PyAudio がこの引数に適切な値を与える．「時間に関する情報」「フラグ」に関しては説明を省略する．（詳しくは PyAudio のドキュメントサイトを参照のこと）

この関数で実行する処理は，WAV 形式ファイルから readframes メソッドを使用して「フレーム数」だけ読込み，それと pyaudio.paContinue という値をタプルにして返す（return する）ことである．pyaudio.paContinue に関する説明は省略する．（詳しくは PyAudio のドキュメントサイトを参照のこと）

定義したコールバック関数は，ストリームを生成する段階で，open メソッドにキーワード引数

stream_callback=**関数名**

として与えることでストリームに登録する．ストリームに対して start_stream メソッドを使用することでサウンドの再生が開始する．

コールバック関数の呼び出しによる方法でサウンドを再生するプログラム pyaudio02.py を次に示す．

プログラム：pyaudio02.py

```
1  # coding: utf-8
2  # モジュールの読込み
3  import wave
4  import pyaudio
5
6  ########################################################
7  # WAV形式サウンドファイルの読込み                      #
8  ########################################################
9  # ファルのオープン
10 wf = wave.open('sound01.wav', 'rb')
11 # 属性情報の取得
12 ch = wf.getnchannels()       # チャネル数
13 qb = wf.getsampwidth()       # 量子化ビット数（バイト数）
14 fq = wf.getframerate()       # サンプリング周波数
15
16 ########################################################
17 # PyAudioによる再生                                    #
18 ########################################################
19
20 #---- 再生用コールバック関数（ここから） ------------
21 def sndplay(in_data, frame_count, time_info, status):
22     buf = wf.readframes(frame_count)
23     return( buf, pyaudio.paContinue )
24 #---- 再生用コールバック関数（ここまで） ------------
25
26 # PyAudioオブジェクトの生成
27 p = pyaudio.PyAudio()
28 # ストリームの生成
```

```
29 stm = p.open( format=p.get_format_from_width(qb),
30                  channels=ch, rate=fq, output=True,
31                  stream_callback=sndplay )
32
33 # コールバック関数による再生を開始
34 stm.start_stream()
35
36 # 待ちループ：「exit」と入力すれば終了処理に進む
37 while stm.is_active():
38     cmd = input('終了->exit: ')
39     if cmd == 'exit':
40         print('終了します…')
41         break
42
43 # ストリームの終了処理
44 stm.stop_stream()
45 stm.close()
46 # PyAudioオブジェクトの廃棄
47 p.terminate()
48
49 # WAVファイルのクローズ
50 wf.close()
```

解説

21〜23 行目がコールバック関数 sndplay の定義である．この関数は，入力元の WAV 形式データファイルに対応する Wave_read オブジェクトに readframes メソッドを実行して，音声データを取り出すものである．31 行目ではこの関数をストリームに登録している．34 行目でサウンド再生のコールバック処理が開始し，メインプログラムは更に下の行の実行に移る．

37〜41 行目はコマンド待ちのループであり，標準入力に対して exit と入力することで，44 行目以降の終了処理に進む．

7.3.2.1 サウンド再生の終了の検出

PyAudio のストリーム入力において，Wave_read オブジェクトからの読み込みが終了（ファイル末尾に到達）したことを検出するには，ストリームに対して is_active を実行する．この結果，読み込みが終了していれば False を，終了していなければ True を返す．これは，サウンド再生の終了の検出のための基本的な方法となる．先のプログラム pyaudio02.py の 37 行目でもこれを応用してサウンド再生の終了を検知している．

読み込みが一度終了した Wave_read オブジェクトに対して rewind メソッドを実行すると，サウンドの再生位置をファイルの先頭に戻すことができる．これを応用すると，サウンドの繰り返し再生が可能となる．具体的には再生が終了したストリームに対して stop_stream メソッドを実行し，再生対象の Wave_read オブジェクトに対して rewind を実行した後に再度ストリームに対して start_stream メソッドを実行する流れとなる．

7.3.3 音声入力デバイスからの入力

ここでは，システムに接続されたマイクやライン入力などの音声入力デバイスからリアルタイムにサウンドを入力する方法について説明する．

サウンドの入力に関しても，PyAudio オブジェクトから生成したストリームオブジェクトを介して処理をする．ストリームの生成に関してもサウンド再生の場合とよく似ており，異なるのは open メソッドの引数に

```
input=True
```

というキーワード引数を与えることと，1 度の入力で受け取るフレーム数を指定する

```
frames_per_buffer=フレーム数
```

というキーワード引数を与える点である．ストリームオブジェクトからサウンドを入力するには read メソッドを使用する．

＜サウンドの入力＞

書き方：　ストリームオブジェクト.read(**フレーム数**)

これを実行することで，引数に与えたフレーム数のサウンドを入力してバイナリデータとして返す.

素朴な**ブロッキングモード**によってサウンドを入力するプログラム pyaudio03.py を次に示す.

プログラム：pyaudio03.py

```
# coding: utf-8
# モジュールの読込み
import wave
import pyaudio

#########################################################
# PyAudioによる音声入力                                 #
#########################################################
# PyAudioオブジェクトの生成
p = pyaudio.PyAudio()
# ストリームの生成
stm = p.open( format=pyaudio.paInt16,
                channels=2, rate=44100,
                frames_per_buffer=1024, input=True )

#-----------------------------------------------
# ストリームからサウンドを取り込む回数の算定
# ・1回の読込みで1024フレームの入力
# ・毎秒44100回のサンプリング
# ・5秒間のサウンド採取
# ということはデータの読込み回数は次の通り
n = int( 44100 / 1024 * 5 )

# 5秒間の入力データをリストに蓄積
print('Recording started!')
dlist = []       # データリストの初期化
for i in range(n):
    buf = stm.read(1024)
    dlist.append(buf)
print('finished.')
# リストの要素を連結してバイナリデータ変換
data = b''
for i in range(n):
    data += dlist[i]
print('data size: ',len(data))

# ストリームの終了処理
stm.stop_stream()
stm.close()
# PyAudioオブジェクトの廃棄
p.terminate()

#########################################################
# WAV形式サウンドファイルへの書き込み                   #
#########################################################
# ファルのオープン
wf = wave.open('out02.wav', 'wb')
# 属性情報の取得
wf.setnchannels(2)       # チャネル数
wf.setsampwidth(2)       # 量子化ビット数（バイト数）
wf.setframerate(44100)   # サンプリング周波数

# 内容の書き込み
wf.writeframes(data)
# ファイルのクローズ
wf.close()
```

解説

これは 5 秒間のサウンド入力を受け付け，それを WAV 形式ファイルとして保存するものである．26〜29 行目の部分で入力したサウンドデータ（バイト列）を要素とするリスト dlist を作成し，32〜34 行目の部分でそのリストの要素を全て連結したバイトデータ data を作成している．

47 行目以降の部分では，バイトデータ data を WAV 形式ファイルとして保存している．

次にサウンド入力を別のスレッドで実行する方法について説明する．

コールバックモードでサウンドを入力すると，入力処理が別のスレッドで実行されるので，メインルーチンの処理の流れがブロックされない．コールバックモードのサウンド入力を行うプログラム pyaudio04.py を次に示す．

プログラム：pyaudio04.py

```
# coding: utf-8
# モジュールの読込み
import wave
import pyaudio

#########################################################
# 書き込み用WAV形式サウンドファイルの準備              #
#########################################################
# ファルのオープン
wf = wave.open('out03.wav', 'wb')
# 属性情報の取得
wf.setnchannels(2)      # チャネル数
wf.setsampwidth(2)      # 量子化ビット数（バイト数）
wf.setframerate(44100)  # サンプリング周波数

#########################################################
# PyAudioによる音声入力                                 #
#########################################################
#---- 入力用コールバック関数（ここから） ------------
def sndrec(in_data, frame_count, time_info, status):
    wf.writeframes(in_data)
    return( None, pyaudio.paContinue )
#---- 入力用コールバック関数（ここまで） ------------

# PyAudioオブジェクトの生成
p = pyaudio.PyAudio()
# ストリームの生成
stm = p.open( format=pyaudio.paInt16, channels=2,
                rate=44100, frames_per_buffer=1024,
                input=True, stream_callback=sndrec )

# コールバック関数による入力を開始
stm.start_stream()

# 待ちループ：「exit」と入力すれば終了処理に進む
while stm.is_active():
    cmd = input('終了->exit: ')
    if cmd == 'exit':
        print('終了します…')
        break

# ストリームの終了処理
stm.stop_stream()
stm.close()
# PyAudioオブジェクトの廃棄
p.terminate()

# WAV形式ファイルのクローズ
wf.close()
```

解説

コールバック関数の基本的な書き方は先に説明した通りであるが，このプログラムでは関数（sndrec）内の処理が

WAV 形式ファイルへの出力となっている．また return するタプルの最初の要素も今回は意味を持たないので None としている．

このプログラムではサウンド入力と同時に WAV 形式ファイルに保存しているが，フレームデータを累積するような処理にするなど，さまざまな応用が可能である．

8　免責事項

本書に掲載したプログラムリストは全て試作品であり，実用に向けた参考的資料である．また本書の使用に伴って発生した損害の一切の責任を筆者は負わない．

付録

A　Python に関する情報

A.1　Python のインターネットサイト

■ 英語サイト：　https://www.python.org/
■ 日本語サイト：　https://www.python.jp/

A.2　Python のインストール作業の例

Python3 のインストール方法の例を挙げる．

【Windows にインストールする作業の例】

先に挙げた Python のインターネットサイトから Python のインストールプログラム（インストーラ：図 30）を入手する．

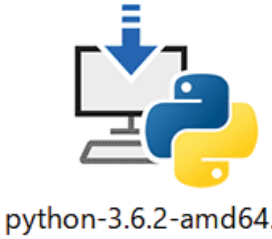


図 30: Windows 用インストーラ（64 ビット用）

インストーラのアイコンをダブルクリックしてインストールを実行する．図 31 は「Customize installation」を選んで詳細の設定を選択し，インストール対象のユーザは「all users」とした例である．各種モジュールを導入，保守するためのツールである「pip」のインストールを指定している．

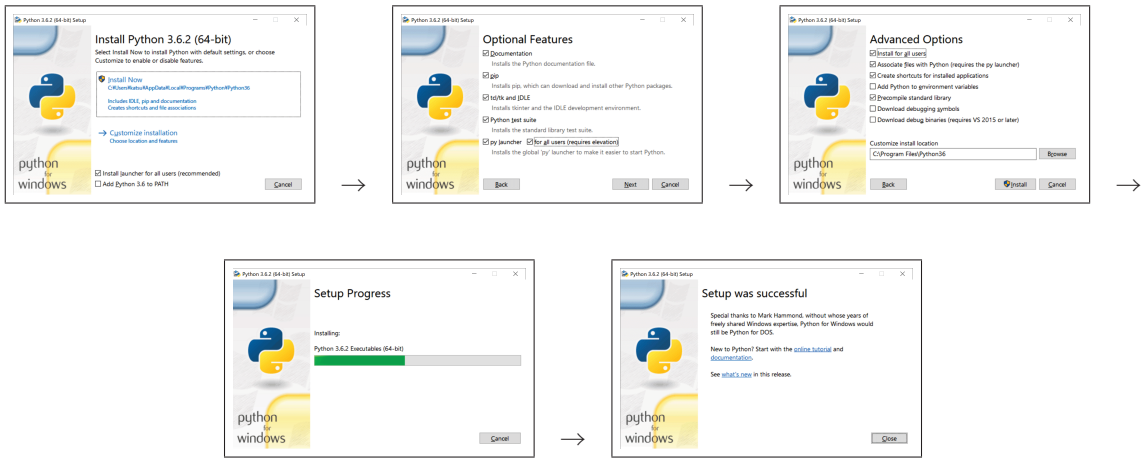


図 31: インストール処理の例（Windows）

【Mac にインストールする作業の例】

Apple 社の macOS には基本的に Python2.7 がインストールされているが，Python3 を使用するには，先に挙げた Python のインターネットサイトから Python3 のインストーラ（図 32）を入手してインストールする．

インストーラのアイコンをダブルクリックしてインストールを実行する．作業の流れの例を図 33 に示す．

図 32: Mac 用インストーラ

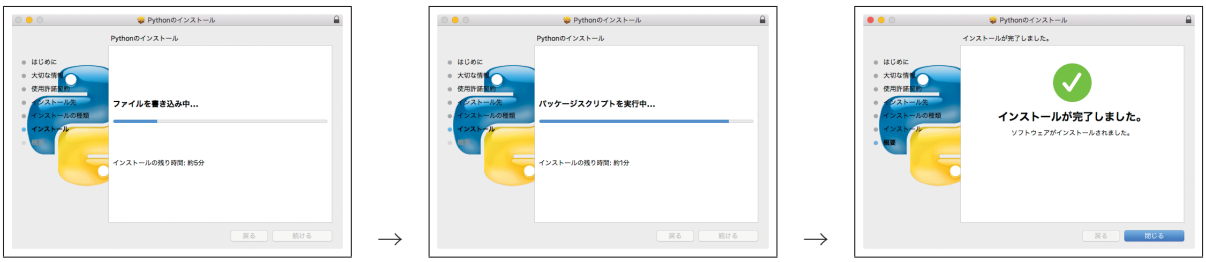


図 33: インストール処理の例（Mac）

A.3 PIP によるモジュール管理

Python3 と共に PIP をインストールすることができ，これを用いて各種のモジュールの導入と更新といった管理作業ができる．PIP は OS のコマンドシェル（Windows の場合はコマンドプロンプト）で pip コマンドを発行することで実行する．[48] pip コマンド実行時の引数やオプションの代表的なものを表 18 に挙げる．

表 18: pip コマンド起動時の引数とオプション（一部）

実行方法	動作
pip list	現在インストールされているパッケージ（モジュール）の一覧を表示する．
pip list –o	上記の内，より新しい版が入手可能なものを表示する．
pip search **キーワード**	「キーワード」を含む名前のモジュールで入手可能なものの一覧を表示する．
pip install **パッケージ名**	指定したパッケージ（モジュール）を新規にインストールする．
pip install -U **パッケージ名**	指定したパッケージ（モジュール）を更新する．
pip uninstall **パッケージ名**	指定したパッケージ（モジュール）を削除する．

例. requests モジュールをインストールする作業

```
pip install requests  Enter    ←インストールの開始
Collecting requests
  Downloading requests-2.18.2-py2.py3-none-any.whl (88kB)
    100% |********************************| 92kB 631kB/s
Collecting idna<2.6,>=2.5 (from requests)
  Using cached idna-2.5-py2.py3-none-any.whl
Collecting certifi>=2017.4.17 (from requests)
  Downloading certifi-2017.7.27.1-py2.py3-none-any.whl (349kB)
    100% |********************************| 358kB 1.8MB/s
Collecting chardet<3.1.0,>=3.0.2 (from requests)
  Using cached chardet-3.0.4-py2.py3-none-any.whl
Collecting urllib3<1.23,>=1.21.1 (from requests)
  Using cached urllib3-1.22-py2.py3-none-any.whl
Installing collected packages:  idna, certifi, chardet, urllib3, requests
Successfully installed certifi-2017.7.27.1 chardet-3.0.4 idna-2.5 requests-2.18.2 urllib3-1.22
```

[48] Apple 社の OS X，macOS では pip3 コマンドで PIP を起動する．

PIP でインストールできるモジュールは wheel という形式で配布されるパッケージであり，表 18 の処理はインターネットに公開されている wheel パッケージの情報に基いて実行される．従って PIP はインターネットに接続された計算機環境で使用することが前提となっているが，wheel 形式パッケージのファイルをダウンロードして，オフライン環境で PIP を実行してそれをインストールすることも可能である．

wheel 形式パッケージはファイル名として whl という拡張子を持ち，それをインストールするには

`pip install` **wheel パッケージ名.whl**

というコマンド操作を行う．

例. ダウンロードした SciPy パッケージをオフラインでインストールする作業

```
D:￥work> pip install scipy-0.19.1-cp36-cp36m-win_amd64.whl   Enter      ←インストール開始
Processing d:￥work￥scipy-0.19.1-cp36-cp36m-win_amd64.whl
                    ⋮
                （途中省略）
                    ⋮
Installing collected packages:  scipy
Successfully installed scipy-0.19.1
```

これは SciPy パッケージのファイル scipy-0.19.1-cp36-cp36m-win_amd64.whl をインストールした例である．

B　Kivy に関する情報

■ **英語サイト：**　https://kivy.org/

B.1　Kivy のインストール作業の例

上記サイトで Kivy のインストール方法が公開されているので，それを元にしてインストールする作業を次に例示する．

【Windows の場合】

1．モジュール管理ツール群の更新

```
pip install -U pip wheel setuptools
```
Enter

この処理は，Kivy のインストール時に限らず，適宜実行するべきものである．

2．Kivy に必要なモジュール群のインストール

```
pip install docutils pygments pypiwin32 kivy.deps.sdl2 kivy.deps.glew
```
Enter

```
pip install kivy.deps.gstreamer kivy.deps.angle
```
Enter

この処理は，Kivy のインストールに先立って行う．

3．Kivy のインストール

```
pip install kivy
```
Enter

これで Kivy が利用できる．

【Mac の場合】

1．Kivy 導入のために必要なソフトウェアをインストールする

Mac のためのパッケージ管理ツール brew [49] を使用して必要なソフトウェアをインストールする．

具体的にはターミナル上で下記のようなコマンドを発行する．

```
brew install pkg-config sdl2 sdl2_image sdl2_ttf sdl2_mixer gstreamer
```
Enter

2．Cython のインストール

```
pip3 install -U Cython
```
Enter

3．Kivy のインストール

```
pip3 install kivy
```
Enter

これで Kivy が利用できる．

B.2　Kivy 利用時のトラブルを回避するための情報

B.2.1　Kivy が使用する描画 API の設定

Kivy はグラフィックス描画の基礎に OpenGL [50] を用いるが，使用する計算機環境によっては，OS が提供する OpenGL の版が Kivy に適合しない場合もある．そのような場合は Kivy を使用したアプリケーションプログラムを実行する際にエラーや例外が発生する．しかし，Kivy では使用するグラフィックス用の API を環境変数の設定により選択することができるので，この件に関する問題が発生する場合は os モジュールの environ プロパティに使用する API

[49] brew は Mac 用のパッケージ管理ツールであり，これを予め Mac にインストールしておくこと．詳しくはインターネットサイト https://brew.sh/ を参照のこと．

参考）Mac 用のパッケージ管理ツールとしては brew 以外にも MacPorts も存在する．詳しくはインターネットサイト https://www.macports.org/ を参照のこと．

[50] グラフィックス描画のための，クロスプラットフォームの API．

を設定すると良い．具体的には次のような記述となる．

```
import os
os.environ['KIVY_GL_BACKEND'] = 'angle_sdl2'
```

指定できる API の例を表 19 に挙げる．

表 19: Kivy で使用できるグラフィックス API

指定する記号	解説
gl	UNIX 系 OS 用の OpenGL
glew（デフォルト）	Windows 環境で通常の場合に用いられる API
sdl2	Windows や UNIX 系 OS で gl や glew が使用できない場合にこれを用いる． kivy.deps.sdl2 をインストールしておく必要がある．
angle_sdl2	Windows 環境で Python 3.5 以上の版を使用する際に利用できる． kivy.deps.sdl2 と kivy.deps.angle be をインストールしておく必要がある．

B.3 GUI デザインツール

Kivy のための GUI アプリケーションデザインツール Kivy Designer が現在開発中であり，インターネットサイト https://kivy-designer.readthedocs.io/ から情報が入手できる．

C 各種モジュールの紹介

表 20: 各種モジュール

科学技術関連		
モジュール名	用途	配布元
matplotlib	グラフ作成／作図	http://matplotlib.org/
NumPy	数値計算（線形代数）	http://www.numpy.org/
mpmath	多倍長精度の数値計算	http://mpmath.org/
SciPy	各種工学のための数値解析	https://www.scipy.org/
SymPy	数式処理	http://www.sympy.org/
データ処理		
モジュール名	用途	配布元
StatsModels	統計処理とモデリング	http://www.statsmodels.org/
scikit-learn	モデリングと機械学習	http://scikit-learn.org/
pandas	表形式データの処理	http://pandas.pydata.org/
seaborn	統計学に適した可視化ツール	https://seaborn.pydata.org/
画像処理関連		
モジュール名	用途	配布元
Pillow	画像処理	https://python-pillow.org/
OpenCV	画像処理, カメラキャプチャ, 画像認識	http://opencv.org/
ニューラルネットワーク関連		
モジュール名	用途	配布元
Chainer	深層ニューラルネットワーク	http://chainer.org/
Caffe	深層ニューラルネットワーク	http://caffe.berkeleyvision.org/
TensorFlow	深層ニューラルネットワーク	https://www.tensorflow.org/
マルチメディア／ゲーム関連		
モジュール名	用途	配布元
pygame	マルチメディアとゲーム	http://www.pygame.org/
PySDL2	マルチメディアとゲーム	http://pysdl2.readthedocs.io/
プログラムの高速化／共有ライブラリの呼び出し		
モジュール名	用途	配布元
Cython	Python プログラムの高速実行（C 言語変換）	http://cython.org/
Numba	Python プログラムの高速実行（LLVM の応用）	https://numba.pydata.org/
ctypes	共有ライブラリの呼び出し	標準ライブラリ

索 引