データサイエンスと機械学習
理論からPythonによる実装まで

付録D

Dirk P. Kroese, Zdravko I. Botev, Thomas Taimre, Radislav Vaisman 著
金森敬文　監訳

目次

付録 D

PYTHON入門

Python はデータサイエンスおよび機械学習における多くの研究者や実務者から選ばれるプログラミング言語となった．本付録ではこの言語について簡潔に紹介する．この言語は絶えず発展し続けており毎年多くの新たなパッケージがリリースされているため，ここでは網羅的な紹介はせず，代わりに初学者がこの美しく考え抜かれた言語を学び始めるために十分な情報を提供したい．

D・1 はじめに

Python のメイン Web サイトは

https://www.python.org/

であり，ここではドキュメント，チュートリアル，初心者向けガイド，ソフトウェアの例などがみられる．重要なこととして，Python には Python 3 および Python 2 とよばれる互換性のない二つの“枝分かれ”があることに注意しよう．この言語のさらなる開発には Python 3 のみが関係しており，この付録（および本書全体）では，Python 3 のみを考慮する．Python のインストールで頻繁に使用される相互依存パッケージが多数あるため，一つのディストリビューションをインストールすると便利である．たとえば，**Anaconda** Python ディストリビューションは以下から利用可能である．

https://www.anaconda.com/

Anaconda インストーラは，最も重要なパッケージを自動的にインストールし，*Spyder* とよばれる便利な対話型開発環境 (IDE) も提供する．

> *Spyder*，*Jupyter notebook* の起動，パッケージのインストールと更新，またはコマンドラインターミナルを開くのに，**Anaconda Navigator** を使うことができる．

はじめに[1]，以下の入力ボックスにある Python のステートメントを試してみよう．IPython のコマンドプロンプトにステートメントを入力するか，または（非常に短い）

[1]必要なファイルがすべてインストールされ，*Spyder* が起動されていることを前提とする．

Python プログラムとして実行することができるが，これら二つの入力方法の出力は若干異なる．たとえば，コンソールで変数名を入力するとその内容が自動で出力されるが，Python プログラムでは **print** 関数を呼び出して明示的に実行する必要がある．*Spyder* でプログラムの複数行を選択（ハイライト）してからファンクションキー[2] F9 を押すのと，コンソールでそれらの行を 1 行ずつ実行するのは同じことである．

Python では，データは**オブジェクト** (object) またはオブジェクト間のリレーションとして表現される（§D・2 も参照）．基本的なデータ型は，数値型（整数，ブール値，浮動小数点を含む），シーケンス型（文字列，タプル，リストを含む），集合型，マッピング型（現在，辞書が唯一の組込みマッピング型である）である．

文字列とは，一重引用符または二重引用符で囲まれた文字の列のことである．**print** 関数を使用して，文字列を出力することができる．

```
print("Hello World!")
```
```
Hello World!
```

きれいに出力するために，Python の文字列は **format** 関数を使ってフォーマット指定をすることができる．ブラケット構文`{i}`は，0 を最初のインデックスとして，`i` 番目の変数を出力させる．個々の変数は，必要に応じて個別にフォーマット指定をすることができる．フォーマットの構文については，§D・9 で詳しく説明する．

§D・9

```
print("Name:{1} (height {2} m, age {0})".format(111,"Bilbo",0.84))
```
```
Name:Bilbo (height 0.84 m, age 111)
```

リストにはさまざまな種類のオブジェクトを含めることができ，次の例のように角括弧を使って作成する：

```
x = [1,'string',"another string"]  # 引用符の種類は重要でない
```
```
[1, 'string', 'another string']
```

リストの要素は，0 から始まるインデックスをもち，**ミュータブル** (mutable)，すなわち変更可能である:

```
x = [1,2]
x[0] = 2  # 最初のインデックスは 0 となることに注意
x
```
```
[2,2]
```

一方，タプル（丸括弧付き）は**イミュータブル** (immutable)，すなわち変更不可である．文字列も同様にイミュータブルである．

```
x = (1,2)
x[0] = 2
```
```
TypeError: 'tuple' object does not support item assignment
```

[2] キーボードやオペレーティングシステムによって異なることがある．

リストは**スライス** (slice) 記法 [start:end] でアクセスできる．ただし重要なこととして，end は選択されない要素のはじめのインデックスであり，また一番最初の要素のインデックスは 0 であることに注意せよ．スライス表記に慣れるため，以下の各行を実行してみよう．

```
a = [2, 3, 5, 7, 11, 13, 17, 19, 23]
a[1:4]  # インデックスが 1 から 3 までの要素
a[:4]   # インデックスが 4 未満の全要素
a[3:]   # インデックスが 3 以上の全要素
a[-2:]  # 最後から 2 つの要素
```

```
[3, 5, 7]
[2, 3, 5, 7]
[7, 11, 13, 17, 19, 23]
[19, 23]
```

演算子 (operator) とは，一つ以上のオペランドに対して動作を実行するプログラミング言語の構成要素である．Python における演算子の動作は，オペランドの型に依存する．たとえば，+，*，−，% のような演算子は，オペランドが数値型である場合には算術演算子となるが，非数値型（文字列など）のオブジェクトに対しては異なることもある．

```
'hello' + 'world'  # 文字列の連結
```

```
'helloworld'
```

```
'hello' * 2   # 文字列の繰返し
```

```
'hellohello'
```

```
[1,2] * 2      # リストの繰返し
```

```
[1, 2, 1, 2]
```

```
15 % 4   # 15/4 の剰余
```

```
3
```

Python 演算子については§D・3で詳しく説明する．いくつかの一般的な Python の演算子を表 D・1 に示す． §D・3

D・2 Python のオブジェクト

前節で述べたように，Python のデータはオブジェクトやオブジェクト間のリレーションで表現される．基本的なデータ型には，文字列や数値型（整数，論理値，浮動小数点など）があったことを思い出そう．

Python はオブジェクト指向のプログラミング言語なので，関数もオブジェクト（すべてがオブジェクト！）である．各オブジェクトは，オブジェクトごとに固有で，イミュータブルすなわち一度作成すると変更できない識別値，オブジェクトに適用でき

る演算子を決定付け，イミュータブルと考えられる型，ミュータブルまたはイミュータブルのいずれかとなる値をもっている．オブジェクトに割り当てられた固有の識別値は，id(obj) のように **id** を呼び出すことで確認することができる．

各オブジェクトは**属性** (attribute) のリストをもち，各属性は別のオブジェクトの参照である．オブジェクトに適用される関数 **dir** は属性のリストを返す．たとえば，文字列オブジェクトは多くの便利な属性をもつことがすぐにわかるであろう．関数は__call__属性をもつオブジェクトである．

クラス (§D・8 を参照) はカスタム型のオブジェクトを作成するテンプレートと考えることができる．

```
s = "hello"
d = dir(s)
print(d,flush=True)  # リストを "flush" 形式で表示する
```

```
['__add__', '__class__', '__contains__', '__delattr__', '__dir__',
 ... (many left out) ... 'replace', 'rfind',
  'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split',
  'splitlines', 'startswith', 'strip', 'swapcase', 'title',
  'translate', 'upper', 'zfill']
```

オブジェクト obj の任意の属性 attr には，obj.attr という**ドット表記** (dot notation) を通じてアクセスできる．オブジェクトの詳細情報を得るには，**help** 関数を使用する．

```
s = "hello"
help(s.replace)
```

```
replace(...) method of builtins.str instance
    S.replace(old, new[, count]) -> str

    Return a copy of S with all occurrences of substring
    old replaced by new.  If the optional argument count is
    given, only the first count occurrences are replaced.
```

これは，**replace** という属性が実際には関数であることを示している．関数である属性は**メソッド** (method) とよばれる．**replace** メソッドを使うと，特定の文字を置換して古い文字列から新しい文字列を作ることができる．

```
s = 'hello'
s1 = s.replace('e','a')
print(s1)
```

```
hallo
```

多くの Python エディタでは，objectname.<TAB>のように TAB キーを押すと，エディタのオートコンプリート機能により，可能な属性の一覧が表示される．

D・3　型と演算子

各オブジェクトには**型** (type) がある．Python の基本的なデータ型は，`str`（文字列），`int`（整数），`float`（浮動小数点数）の三つである．関数 `type` は，オブジェクトの型を返す．

```
t1 = type([1,2,3])
t2 = type((1,2,3))
t3 = type({1,2,3})
print(t1,t2,t3)
```

```
<class 'list'> <class 'tuple'> <class 'set'>
```

代入演算子 (assignment operator) は，たとえば `x = 12` のように，オブジェクトを変数に代入する．**式** (expression) とは，値，演算子，変数の組合せであり，新たな値や変数を生成する．

> 変数名は大文字と小文字を区別し，アルファベット，数字，アンダースコアのみ使うことができる．また，最初の文字はアルファベットかアンダースコアでなければならない．`True` や `False` などの予約語も，大文字と小文字を区別することに注意せよ．

Python は動的型付け言語であり，プログラム実行中のある時点での変数の型は，直近に代入されたオブジェクトによって決定される．つまり，(C や Java のように）最初から変数の型を明示的に宣言する必要はないが，その代わりに，変数の型は現在代入されているオブジェクトによって決まる．

重要なのは，Python の変数はオブジェクトに対する**参照** (reference) であると理解することである．このことを，靴箱のラベルに例えて考えてみよう．ラベルは単純なものであっても，靴箱の**中身**（変数が参照するオブジェクト）は任意に複雑なものとなる．ある靴箱の中身を別の靴箱に移す代わりに，単にラベルを移す方がはるかに簡単である．

```
x = [1,2]
y = x   # y は x と同じオブジェクトを参照する
print(id(x) == id(y))  # オブジェクトの識別値が同じであることを確認する
y[0] = 100  # y が参照するリストの内容を変更する
print(x)
```

```
True
[100,2]
```

```
x = [1,2]
y = x   # y は x と同じオブジェクトを参照する
y = [100,2]  # ここで y は別のオブジェクトを参照するようになる
print(id(x) == id(y))
print(x)
```

```
False
[1,2]
```

表 D・1 は，数値および論理変数に対する Python 演算子の一覧である．

表 D・1　一般的な算術演算子（左）と論理演算子（右）

+	加算	~	ビット否定 NOT
-	減算	&	ビット論理積 AND
*	乗算	^	ビット排他的論理和 XOR
**	べき乗	|	ビット論理和 OR
/	除算	==	等号
//	切り捨て除算	!=	等号否定
%	剰余		

算術演算子のいくつかは，`x += 1` が `x = x + 1` を意味するように，代入演算子と組合わせることができる．また，+や*などの演算子は，他のデータ型に対しても定義することができ，異なる意味をもつようになる．これを演算子の**オーバーロード** (overloading) といい，先ほどのリストの繰返しのための`<List> * <Integer>`の使用がその一例である．

D・4　関数とメソッド

関数を使うと，複雑なプログラムを単純なパーツへと分割するのがより簡単になる．**関数** (function) を作るには，次の構文を使用する．

```
def <function name>(<parameter_list>):
    <statements>
```

関数は，オブジェクトへの参照である入力変数のリストを受取る．関数の内部では，オブジェクトを変更するいくつかのステートメントが実行されるが，参照自体が変更されることはない．さらに，関数は，出力としてオブジェクトを返すことができる（あるいは，出力を返すように明示的に指示されていない場合は，None を返す）．靴箱の例えを再び考えると，関数の入力変数は靴箱のラベルであり，その参照するオブジェクトは靴箱の中身である．次のプログラムでは，Python における変数とオブジェクトの巧妙な関係を紹介する．

関数内のステートメントはインデントされていなければならないことに注意せよ．これは，Python が関数の開始と終了を定義する方法となっている．

```
x = [1,2,3]

def change_list(y):
    y.append(100) # y で参照されるリストに要素を追加する
    y[0]=0        # 同リストの最初の要素を変更する
    y = [2,3,4]   # ここでローカルの y は別のリストを参照するようになる
                  # y が最初に参照したリストは変更されない
    return sum(y)
```

```
print(change_list(x))
print(x)
```

```
9
[0, 2, 3, 100]
```

関数内で定義された変数は，その関数内でのみ認識される**ローカルスコープ** (local scope) をもつ．これにより，同じ変数名を異なる関数内で使用しても衝突が発生しない．関数内で変数が使用されると，Python はまずその変数がローカルスコープをもつかどうかをチェックする．もしそうでなければ（その変数が関数内で定義されていなければ），Python はその変数を関数の外（グローバルスコープ）から探し出す．次のプログラムは，いくつかの重要な点を説明している．

```
from numpy import array, square, sqrt

x = array([1.2,2.3,4.5])

def stat(x):
    n = len(x)          # x の長さ
    meanx = sum(x)/n
    stdx = sqrt(sum(square(x - meanx))/n)
    return [meanx,stdx]

print(stat(x))
```

```
[2.6666666666666665, 1.3719410418171119]
```

1. `sqrt` のような基本的な数学関数は，標準的な Python インタープリタには備わっておらず，インポートする必要がある．これについては，後ろの§D・5 で詳述する．

2. すでに述べたように，インデントは非常に重要であり，関数の始まりと終わりを示すものである．

3. 行末にセミコロン[3]は必要ないが，関数定義の最初の行（ここでは 5 行目）はコロン (:) で終わる必要がある．

4. リストは配列（数値ベクトル）ではないので，リストに対してベクトル演算を行うことはできない．しかし，`numpy` モジュールは，効率的なベクトル/行列演算を念頭に置いて設計されている．3 行目のコードでは，`x` をベクトル (ndarray) オブジェクトとして定義している．このような配列には，`square`，`sum`，`sqrt` などの関数が適用される．ここでは，Python の組込み関数である `len` と `sum` を使用している．`numpy` については§D・10 で詳述する．

5. 11 行目の `print(stat(x))` の代わりに `stat(x)` を指定してプログラムを実行しても，コンソールには何も出力されない．

[3]セミコロンは，複数のコマンドを 1 行にまとめるのに使用できる．

組込み関数の全リストを表示するには，dir(__builtin__)（二重のアンダースコアを使用）と入力する．

D・5 モジュール

Python **モジュール** (module) は，コードを管理可能なパーツへと整理するのに便利なプログラミング構造である．名前 module_name をもつ各モジュールには，関数，クラス，変数などの定義や，実行可能なステートメントを含む Python ファイル module_name.py が関連付けられる．モジュールは，次の構文を使って他のプログラムにインポートすることができる：import <module_name> as <alias_name>．ここで，<alias_name>はモジュールの省略名である．

モジュールが，他の Python ファイルにインポートされると，モジュール名は**名前空間** (namespace) として扱われ，モジュール内の各オブジェクトに固有の名前をもたせる．たとえば mod1 と mod2 という異なるモジュールは，異なる sum 関数をもつことができるが，mod1.sum や mod2.sum のように，関数名の前にモジュール名をドット表記で付けることにより，それらを区別することができる．たとえば，次のコードでは，numpy モジュールの sqrt 関数を使用している．

```
import numpy as np
np.sqrt(2)
```

```
1.4142135623730951
```

Python **パッケージ** (package) は，単に Python モジュールのディレクトリである．つまり，開くための情報を追加したモジュールの集まりである（そのうちのいくつかは__path__属性をもつものから見つけられる）．Python の組込みモジュールは__builtins__とよばれる．非常に多くの有用な Python モジュールの中から，表 D・2 にいくつかを示す．

表 D・2 いくつかの便利な Python モジュール/パッケージ

datetime	日付や時刻を操作するモジュール
matplotlib	MATLAB™ タイプの作図パッケージ
numpy	乱数生成や線形代数ツールを含む，科学計算のための基本パッケージ．ユビキタスな ndarray クラスを定義している
os	Python によるオペレーティングシステムへのインターフェース
pandas	データ解析のための基本モジュール．強力な DataFrame クラスを定義している
pytorch	GPU での計算をサポートする機械学習ライブラリ
scipy	数学，科学，工学のためのエコシステムであり，積分，微分方程式の解法，最適化などの数値計算のための多数のツールを備えている
requests	HTTP リクエストを実行し，Web との連携を行うライブラリ
seaborn	統計データの可視化のためのパッケージ
sklearn	簡単に使える機械学習ライブラリ
statsmodels	統計モデルの解析のためのパッケージ

numpy パッケージには，**random**，**linalg**，**fft** などのさまざまなサブパッケージが含まれており，詳細は§D・10 で述べる．

> *Spyder* を使用する際，任意のオブジェクトの前で Ctrl+I を押すと，そのオブジェクトのヘルプファイルが別ウィンドウで表示される．

すでにみたように，次の構文を使って特定の関数だけをモジュールからインポートすることもできる：from <module_name> import <fnc1, fnc2, ...>.

```
from numpy import sqrt, cos
sqrt(2)
cos(1)
```

```
1.4142135623730951
0.54030230586813965
```

これで，モジュール名（のエイリアス）による関数の面倒な接頭辞を避けられる．しかし，大規模なプログラムでは，使用している関数がどのモジュールに属しているかを正確かつ明確に把握できるよう接頭辞/エイリアス名の構造を常に使用する方がよい．

D・6 制御文

Python のフロー制御は，多くのプログラミング言語と同様に，while や for のループと条件文がある．if-then-else フロー制御の構文は以下の通りである．

```
if <condition1>:
   <statements>
elif <condition2>:
   <statements>
else:
   <statements>
```

ここで，<condition1>と<condition2>は，True または False のいずれかをとる論理的条件である．論理的条件には，比較演算子（==, >, <=, !=など）が含まれることがよくある．上の例では，elif の部分が一つあり，“else if” の条件文を書くことができる．一般に，elif の部分は複数あってもよいし，省略することもできる．また，else の部分も省略できる．コロンはインデントと同様に必須となる．

while ループと for ループの構文は以下の通りである．

```
while <condition>:
   <statements>

for <variable> in <collection>:
   <statements>
```

上の例では，<collection> はイテラブルなオブジェクトである（§D・7 を参照）．for ループと while ループのさらなる制御として，現在のループを終了させるために break ステートメントを使ったり，現在の反復中の残りのステートメントを放棄してループの次の反復に続くために continue ステートメントを使うことができる．ここで，一例を示す．

```
import numpy as np
ans = 'y'
while ans != 'n':
  outcome = np.random.randint(1,6+1)
  if outcome == 6:
      print("Hooray a 6!")
      break
  else:
      print("Bad luck, a", outcome)
  ans = input("Again? (y/n) ")
```

D・7　反復処理

for ループで使われるような一連のオブジェクトへの反復処理は，一般的な操作である．反復処理の仕組みをよりよく理解するために，次のコードを考える．

```
s = "Hello"
for c in s:
    print(c,'*', end=' ')
```

```
H * e * l * l * o *
```

文字列は，反復処理が可能な Python オブジェクトの一例である．文字列オブジェクトのメソッドの一つに__iter__がある．このようなメソッドをもつオブジェクトは**イテラブル** (iterable) なオブジェクトとよばれる．このメソッドを呼び出すと**イテレータ** (iterator) という，反復されるシーケンスの次の要素を返すオブジェクトが生成される．これは，__next__というメソッドを通じて行われる．

```
s = "Hello"
t = s.__iter__()     # t はイテレータである．iter(s) としても同じ
print(t.__next__() ) # next(t) としても同じ
print(t.__next__() )
print(t.__next__() )
```

```
H
e
l
```

組込み関数 **next** と **iter** は，対応するオブジェクトの二重アンダースコア付き関数を呼び出すだけである．for ループを実行する際，反復処理を行うシーケンス/コレクションはイテラブルでなければならない．for ループの実行中，イテレータが作られ，次の要素がなくなるまで **next** 関数が実行される．イテレータはイテラブルであるため，for ループでも使用することができる．リスト，タプル，文字列は，いわゆ

る**シーケンスオブジェクト** (sequence object) であり，イテラブルで，要素はインデックスによって反復される．

Python で最も一般的なイテレータは **range** イテレータで，インデックスの範囲に渡って反復させることができる．`range` はリストではなく range オブジェクトを返すことに注意せよ．

```
for i in range(4,20):
     print(i, end=' ')
print(range(4,20))
```

```
4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
range(4,20)
```

> Python のスライス演算子 $[i:j]$ と同様に，イテレータ range(i, j) は i から j までのインデックス j を**含まない**範囲となる．

他にも二つ一般のイテラブルなオブジェクトとして，集合と辞書がある．Python の**集合** (set) とは，数学と同様に，一意なオブジェクトの順序付けられていない集まりである．タプルの丸括弧 ( ) やリストの角括弧 [ ] とは異なり，集合は波括弧 { } で定義される．リストとは異なり，集合には重複する要素がない．Python では，和集合 `A | B` や積集合 `A & B` など，通常の集合演算子の多くが実装されている．

```
A = {3, 2, 2, 4}
B = {4, 3, 1}
C = A & B
for i in A:
      print(i)
print(C)
```

```
2
3
4
{3, 4}
```

リストを構築する便利な方法に，次の形の式による**リスト内包表記** (list comprehension) がある．

```
<expression> for <element> in <list> if <condition>
```

集合についても同様の構築が成立する．この方法で，リストと集合は，数学と非常に似た構文を使って定義することができる．たとえば，集合 $A := \{3,2,4,2\} = \{2,3,4\}$（要素の順序や重複は無視される）と $B := \{x^2 : x \in A\}$ の数学的な定義を，以下の Python コードと比較してみよう．

```
setA = {3, 2, 4, 2}
setB = {x**2 for x in setA}
print(setB)
listA = [3, 2, 4, 2]
listB = [x**2 for x in listA]
print(listB)
```

```
{16, 9, 4}
[9, 4, 16, 4]
```

辞書 (dictionary) は集合と似たデータ構造で，波括弧で囲まれた一つ以上のキーと値のペア key:value からなる．キーは同じ型であることが多いが，必ずしもそうである必要はなく，値も同様である．ここでは,『ロード・オブ・ザ・リング』の登場人物の年齢を辞書に格納する簡単な例を示す.

```
DICT = {'Gimly': 140, 'Frodo':51, 'Aragorn': 88}
for key in DICT:
     print(key, DICT[key])
```

```
Gimly 140
Frodo 51
Aragorn 88
```

D・8　クラス

Python ではオブジェクトが基本的に重要であることを思い出そう．実際，データ型や関数はすべてオブジェクトである．**クラス** (class) はオブジェクトの型であり，クラス定義を書くことは，オブジェクトの新たな型のテンプレートを作ることと考えることができる．それぞれのクラスは，多くの組込みメソッドを含む多くの属性をもっている．クラスを作成するための基本的な構文は次の通りである：

```
class <class_name>:
   def __init__(self):
       <statements>
   <statements>
```

おもな組込みメソッドは，クラスオブジェクトの**インスタンス** (instance) を生成する__init__である．たとえば，str はクラスオブジェクト（文字列クラス）だが，s = str('Hello')，あるいは単に s = 'Hello' とすると，str クラスのインスタンス s が生成される．インスタンス属性は初期化時に作成され，その値はインスタンスごとに異なりうる．これに対して，クラス属性の値は，すべてのインスタンスで同じである．初期化メソッドの変数 self は，現在作成中のインスタンスを参照する．以下に簡単な例をあげて，属性への代入のしかたを説明する.

```
class shire_person:
   def __init__(self,name): # 初期化メソッド
       self.name = name     # インスタンス属性
       self.age = 0             # インスタンス属性
   address = 'The Shire'        # クラス属性

print(dir(shire_person)[1:5],'...',dir(shire_person)[-2:])
                               # クラス属性のリスト

p1 = shire_person('Sam')   # インスタンスを作成
```

```
p2 = shire_person('Frodo') # 別のインスタンスを作成
print(p1.__dict__)   # インスタンス属性のリスト

p2.race = 'Hobbit'   # インスタンス p2 に別の属性を追加
p2.age = 33          # インスタンス属性を変更
print(p2.__dict__)

print(getattr(p1,'address'))   # p1 のクラス属性の内容
```

```
['__delattr__', '__dict__', '__dir__', '__doc__'] ...
['__weakref__', 'address']
{'name': 'Sam', 'age': 0}
{'name': 'Frodo', 'age': 33, 'race': 'Hobbit'}
The Shire
```

クラスオブジェクトのすべての属性を__init__メソッドで作成してもよいが，上の例でみたように，属性はクラス定義の外であっても，どこでも作成したり代入することができる．より一般的には，属性は__dict__をもつどのオブジェクトにも追加できる．

「空」のクラスは，以下のように作成できる．

```
class <class_name>:
   pass
```

Python のクラスは，次の構文での**継承** (inheritance) により，親クラスから派生させることができる．

```
class <class_name>(<parent_class_name>):
   <statements>
```

派生クラスは,（初期化時点で）親クラスの属性をすべて継承する．

たとえば，以下の shire_person クラスは，親クラスである person から name, age，address の属性を継承している．これは **super** 関数を使って行われるが，ここでは親クラスの person を明示的に指名することなしに参照するように使われている．shire_person 型の新規のオブジェクトを作成する際には，親クラスの__init__メソッドが呼び出され，追加のインスタンス属性 Shire_address が作成されている．Shire_address は shire_person インスタンス固有の属性であることを **dir** 関数により確認している．

```
class person:
   def __init__(self,name):
       self.name = name
       self.age = 0
       self.address= ' '

class shire_person(person):
   def __init__(self,name):
       super().__init__(name)
       self.Shire_address = 'Bag End'
```

```
p1 = shire_person("Frodo")
p2 = person("Gandalf")
print(dir(p1)[:1],dir(p1)[-3:] )
print(dir(p2)[:1],dir(p2)[-3:] )
```

```
['Shire_address'] ['address', 'age', 'name']
['__class__'] ['address', 'age', 'name']
```

D・9 ファイル

ファイルに書き込んだり，ファイルから読み込んだりするには，まずファイルを開く必要がある．Python の **open** 関数は，イテラブルなファイルオブジェクトを生成するので，for や while ループの中で逐次処理することができる．以下に簡単な例を示す．

```
fout = open('output.txt','w')
for i in range(0,41):
    if i%10 == 0:
        fout.write('{:3d}\n'.format(i))
fout.close()
```

open の第 1 引数はファイル名である．第 2 引数は，ファイルを読み込み ('r')，書き込み ('w')，追記 ('a') などから，何のために開くかを指定する．help(open) を参照せよ．ファイルはデフォルトではテキストモードで書き込まれるが，バイナリモードで書き込むことも可能である．上のプログラムでは，0, 10, . . . , 40 という文字列を含む 5 行からなるファイル output.txt を作成している．なお，もしも上記コードの 4 行目を fout.write(i) と書いた場合，変数 i は文字列ではなく整数であるため，エラーメッセージが表示される．string.format() という式は，Python で出力文字列のフォーマットを指定する方法であることを思い出そう．

フォーマット構文{:3d}は，出力が 10 進数の数字からなり，3 文字という特定の幅に制限されることを示している．導入部で述べたように，ブラケット構文{i}は 0 を最初のインデックスとして，i 番目の変数を出力させる．出力のフォーマットはさらに{i:format}で指定され，format は通常[4]次の形式となる．

[width][.precision][type]
この仕様では：

- width は，出力の最小幅を指定する．
- precision は，f 型の浮動小数点数の場合は小数点以下の表示桁数を，g 型の浮動小数点数の場合は小数点の前後の表示桁数を指定する．
- type は，出力の型を指定する．最も一般的な型には，文字列の s，整数の d，2 進数の b，固定小数点表記での浮動小数点数 (float) の f，一般表記での浮動小数点数の g，科学的表記での浮動小数点数の e がある．

[4] フォーマットのオプションはさらに増やすことも可能である．

数値に対するフォーマットの動作を以下に示す．

```
'{:5d}'.format(123)
'{:.4e}'.format(1234567890)
'{:.2f}'.format(1234567890)
'{:.2f}'.format(2.718281828)
'{:.3f}'.format(2.718281828)
'{:.3g}'.format(2.718281828)
'{:.3e}'.format(2.718281828)
'{0:3.3f}; {2:.4e};'.format(123.456789,  0.00123456789)
```

```
'  123'
'1.2346e+09'
'1234567890.00'
'2.72'
'2.718'
'2.72'
'2.718e+00'
'123.457; 1.2346e-03;'
```

以下のコードは，テキストファイル output.txt を一行ずつ読み込んで，その出力を画面に表示するものである．改行文字を削除するために，文字列に対する **strip** メソッドを使用した．これは，文字列の先頭と末尾からあらゆる空白を削除するものである．

```
fin = open('output.txt','r')
for line in fin:
    line = line.strip()    # 改行文字を除去する
    print(line)
fin.close()
```

```
0
10
20
30
40
```

ファイルの入出力を行う際には，必ずファイルを閉じることが重要である．プログラムエラーによりプログラムが予期せず終了した場合などに，ファイルが開いたままになっていると，システムに大きな問題をひき起こす可能性がある．このような理由から，**コンテキスト管理** (context management) によってファイルを開くことを勧める．構文は以下の通りである．

```
with open('output.txt', 'w') as f:
      f.write('Hi there!')
```

コンテキスト管理は，プログラムが途中で終了した場合でも，ファイルが正しく閉じられるようにする．例として次のプログラムでは，本書のウェブページ ataleof2cities.txt からダウンロード可能なチャールズ・ディケンズの『二都物語』に登場する最も頻度の高い単語を出力する．

次のプログラムでは，ataleof2cities.txt というファイルを現在の作業ディレクトリに置く必要があることに注意せよ．現在の作業ディレクトリは，import os に続いて cwd = os.getcwd() を実行することで確認できる．

```
numline = 0
DICT = {}
with open('ataleof2cities.txt', encoding="utf8") as fin:
    for line in fin:
        words = line.split()
        for w in words:
           if w not in DICT:
               DICT[w] = 1
           else:
               DICT[w] +=1
        numline += 1

sd = sorted(DICT,key=DICT.get,reverse=True) #辞書をソートする

print("Number of unique words: {}\n".format(len(DICT)))
print("Ten most frequent words:\n")
print("{:8} {}".format("word", "count"))
print(15*'-')
for i in range(0,10):
    print("{:8} {}".format(sd[i], DICT[sd[i]]))
```

```
Number of unique words: 19091

Ten most frequent words:

word     count
---------------
the      7348
and      4679
of       3949
to       3387
a        2768
in       2390
his      1911
was      1672
that     1650
I        1444
```

D・10 NumPy

NumPy パッケージ（モジュール名 **numpy**）は，Python による科学計算のためのビルディングブロックを提供する．このパッケージには，**sin**，**cos**，**tan** などの標準的な数学関数のほか，乱数生成，線形代数，統計計算のための効率的な関数も含まれている．

```
import numpy as np   #パッケージのインポート
x = np.cos(1)
data = [1,2,3,4,5]
y = np.mean(data)
z = np.std(data)
print('cos(1) = {0:1.8f}  mean = {1}  std = {2}'.format(x,y,z))
```

```
cos(1) = 0.54030231  mean = 3.0  std = 1.4142135623730951
```

D・10・1　配列の作成と成型

numpy の基本的なデータ型は ndarray である．このデータ型は，LAPACK や BLAS などの高度に最適化された数値計算ライブラリによる高速な行列演算を可能する．これは，（ネストされた）リストとは対照的である．そのため，大量の定量的なデータを扱う際には，**numpy** がしばしば不可欠となる．

ndarray オブジェクトは，さまざまな方法で作成することができる．次のコードでは，2×3×2 の零の配列を作成している．三つの次元をもつ行列，あるいは二つのスタックされた 3×2 の行列と考えよう．

```
A = np.zeros([2,3,2])  # 2 × 3 × 2 の零配列
print(A)
print(A.shape)   # 行数と列数
print(type(A))   # A は ndarray 型
```

```
[[[ 0.  0.]
  [ 0.  0.]
  [ 0.  0.]]

 [[ 0.  0.]
  [ 0.  0.]
  [ 0.  0.]]]
(2, 3, 2)
<class 'numpy.ndarray'>
```

ここではおもに 2 次元配列を扱う．つまり，通常の行列を表す ndarrays である．また，**range** メソッドやリストを使って，**array** メソッドで ndarrays を作ることもできる．**arange** は **numpy** 版の **range** で，ndarrays オブジェクトを返すという違いがあることに注意せよ．

```
a = np.array(range(4))    # np.arange(4) と同等
b = np.array([0,1,2,3])
C = np.array([[1,2,3],[3,2,1]])
print(a, '\n', b,'\n' , C)
```

```
[0 1 2 3]
[0 1 2 3]
 [[1 2 3]
 [3 2 1]]
```

ndarray の次元は，タプルを返す **shape** メソッドで取得できる．配列は **reshape** メソッドで再成型できるが，これは現在の ndarray オブジェクトを変更するものではない．変更を永続的にするには，新しいインスタンスを作成する必要がある．

```
a = np.array(range(9)) # a は形状(9,)をもつ ndarray
print(a.shape)
A = a.reshape(3,3)   # A は形状(3,3)をもつ ndarray
print(a)
print(A)
```

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

reshape に渡す形状の次元の一つに，−1 を指定することができる．このとき，その次元は他の次元から推測されることになる．

ndarray の'T' 属性は，その転置を与える．形状が $(n,)$ の「ベクトル」の転置は，同じベクトルとなることに注意せよ．列ベクトルと行ベクトルを区別するには，ベクトルをそれぞれ $n \times 1$ と $1 \times n$ の配列へと再成型する．

```
a = np.arange(3)    #形状(3,)をもつ 1 次元配列（ベクトル）
print(a)
print(a.shape)
b = a.reshape(-1,1) #形状(3,1)をもつ配列（行列）
print(b)
print(b.T)
A = np.arange(9).reshape(3,3)
print(A.T)
```

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

配列を結合する便利な方法として，**hstack** と **vstack** があり，それぞれ配列を水平方向と垂直方向に結合する．

```
A = np.ones((3,3))
B = np.zeros((3,2))
C = np.hstack((A,B))
print(C)
```

```
[[ 1.  1.  1.  0.  0.]
 [ 1.  1.  1.  0.  0.]
 [ 1.  1.  1.  0.  0.]]
```

D・10・2　スライシング

配列は，Python のリストと同様にスライスできる．配列が複数の次元をもつ場合，次元ごとにスライスを指定する必要がある．Python のインデックスは'0' から始まり，'len(obj)-1' で終わることを思い出そう．以下のプログラムでは，さまざまなスライス操作を示している．

```
A = np.array(range(9)).reshape(3,3)
print(A)
print(A[0])     # 1 行目
print(A[:,1])   # 2 列目
print(A[0,1])   # 1 行目 2 列目の要素
print(A[0:1,1:2])   # A[0,1] = 1 を含む(1,1)ndarray
print(A[1:,-1]) # 2,3 行目の最後の列の要素
```

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

ndarrays はミュータブルなオブジェクトなので，新規のオブジェクトを作成しなくても要素を直接変更できることに注意せよ．

```
A[1:,1] = [0,0] # 上の行列 A の二つの要素を変更
print(A)
```

```
[[0, 1, 2]
 [3, 0, 5]
 [6, 0, 8]]
```

D・10・3　配列の演算

基本的な数学の演算子や関数は，ndarray オブジェクトの**要素ごと**に作用する．

```
x = np.array([[2,4],[6,8]])
y = np.array([[1,1],[2,2]])
print(x+y)
```

```
[[ 3,  5]
 [ 8, 10]]
```

```
print(np.divide(x,y))   # x/y としても同じ
```

```
[[ 2.  4.]
 [ 3.  4.]]
```

```
print(np.sqrt(x))
```

```
[[1.41421356  2.        ]
 [2.44948974  2.82842712]]
```

行列積やベクトルの内積を計算するために，ndarray インスタンスのメソッドあるいは np のメソッドとして，**numpy** の **dot** 関数を使用することができる．

```
print(np.dot(x,y))
```

```
[[10, 10]
 [22, 22]]
```

```
print(x.dot(x))    # np.dot(x,x) としても同じ
```

```
[[28, 40]
 [60, 88]]
```

Python のバージョン 3.5 以降,（np.matmul メソッドを実装した）**@演算子** (@ operator) を使用して二つの ndarray の積をとることができる．行列に対して，これは **dot** メソッドを使うのと同様である．高次元配列に対しては二つのメソッドは異なる動作をする．

```
print(x @ y)
```

```
[[10 10]
 [22 22]]
```

NumPy では，異なる形状（次元）の配列に対する算術演算が可能である．具体的に，二つの配列がそれぞれ次元 $(m_1, m_2, \ldots, m_p)$ と $(n_1, n_2, \ldots, n_p)$ をもつとする．すべての $i = 1, \ldots, p$ について次の条件が成り立つ場合に，それらの配列または形状は**そろっている** (aligned) という．

- $m_i = n_i$，または
- $\min\{m_i, n_i\} = 1$，または
- m_i と n_i の一方あるいは両方が欠落している．

たとえば，形状 $(1,2,3)$ と $(4,2,1)$ はそろっており，$(2,,)$ と $(1,2,3)$ もそろっている．しかしながら，$(2,2,2)$ と $(1,2,3)$ はそろっていない．NumPy は，小さい方の次元の配列要素を大きい方の次元に合わせて「複製」する．この処理は**ブロードキャスト** (broadcasting) とよばれ，実際にはコピーを作成せずに実行されるため，メモリを効率的に使用することができる．以下にいくつかの例を示す．

```
import numpy as np
A= np.arange(4).reshape(2,2) # (2,2)配列

x1 = np.array([40,500])         # (2,)配列
x2 = x1.reshape(2,1)          # (2,1)配列
```

```
print(A + x1) # 形状(2,2)と(2,)
print(A * x2) # 形状(2,2)と(2,1)
```

```
[[ 40 501]
 [ 42 503]]
[[   0   40]
 [1000 1500]]
```

上記では，x1 は行単位，x2 は列単位で複製されていることに注意せよ．ブロードキャストは，下図のように，行列どうしの@演算子にも適用される．以下では，行列 b が第 3 次元に沿って複製され，結果として次の二つの行列積が行われている．

$$\begin{bmatrix}0 & 1\\ 2 & 3\end{bmatrix}\begin{bmatrix}0 & 1\\ 2 & 3\end{bmatrix} \quad \text{と} \quad \begin{bmatrix}4 & 5\\ 6 & 7\end{bmatrix}\begin{bmatrix}0 & 1\\ 2 & 3\end{bmatrix}$$

```
B = np.arange(8).reshape(2,2,2)
b = np.arange(4).reshape(2,2)
print(B@b)
```

```
[[[ 2  3]
  [ 6 11]]

 [[10 19]
  [14 27]]]
```

sum，mean，std などの関数は，ndarray インスタンスのメソッドとして実行することもできる．引数 axis を渡して，どの次元に沿って関数を適用するかを指定できる．デフォルトでは axis=None となる．

```
a = np.array(range(4)).reshape(2,2)
print(a.sum(axis=0)) #行に沿って和をとることで，列和を得ている
```

```
[2, 4]
```

D・10・4　乱数

numpy のサブモジュールの一つに random があり，乱数生成のための多くの関数を備えている．

```
import numpy as np
np.random.seed(123)  # 乱数生成のシードを設定
x = np.random.random()       # (0,1) 上の一様乱数
y = np.random.randint(5,9)   # 5,...,8 の離散一様乱数
z = np.random.randn(4)       # 4 つの標準正規乱数の配列
print(x,y,'\n',z)
```

```
0.6964691855978616 7
 [ 1.77399501 -0.66475792 -0.07351368  1.81403277]
```

numpy での乱数生成の詳細については以下を参照．

https://docs.scipy.org/doc/numpy/reference/random/index.html

D・11　Matplotlib

2 次元および 3 次元プロットのための主要な Python グラフィックライブラリは **matplotlib** であり，そのサブパッケージである **pyplot** には Python のプロットを MATLAB と同様のプロットを Python でつくるための関数が集められている．

D・11・1　基本的なプロット作成

以下のコードは，プロットを作成するためのさまざまな選択肢を示している．線やマーカーのスタイルと色，ラベルのフォントサイズを変更することができる．図 D・1 にその結果を示す．

sqrtplot.py

```
import matplotlib.pyplot as plt
import numpy as np
x = np.arange(0, 10, 0.1)
u = np.arange(0,10)
y = np.sqrt(x)
v = u/3
plt.figure(figsize = [4,2])  # プロットのサイズ（インチ）
plt.plot(x,y, 'g--')         # 緑破線をプロット
plt.plot(u,v,'r.')           # 赤点をプロット
plt.xlabel('x')
plt.ylabel('y')
plt.tight_layout()
plt.savefig('sqrtplot.pdf',format='pdf')  # pdf として保存
plt.show()                   # ここで両方のプロットが描かれる
```

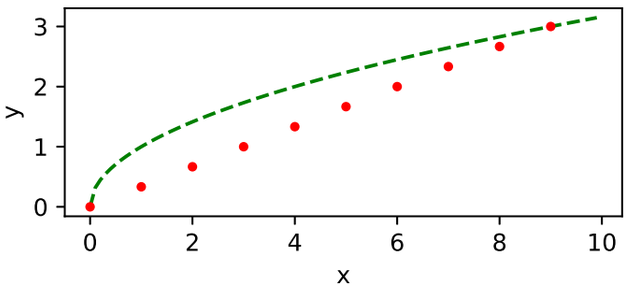

図 D・1　pyplot を用いた簡単なプロット

ライブラリ **matplotlib** では，サブプロットの作成も可能である．図 D・2 の散布図とヒストグラムは，以下のコードで作成されている．ヒストグラムを作成する際には，グラフのレイアウトに影響するいくつかのオプション引数がある．階級数はパラメータ bins で決定される（デフォルトは 10）．散布図も，点の色を決める文字列 c や，点の透明度に影響を与える alpha など，多くのパラメータを受取る．

histscat.py

```
import matplotlib.pyplot as plt
import numpy as np
x = np.random.randn(1000)
u = np.random.randn(100)
v = np.random.randn(100)
plt.subplot(121)              # 一つ目のサブプロット
plt.hist(x,bins=25, facecolor='b')
plt.xlabel('X Variable')
plt.ylabel('Counts')
plt.subplot(122)              # 二つ目のサブプロット
plt.scatter(u,v,c='b', alpha=0.5)
plt.show()
```

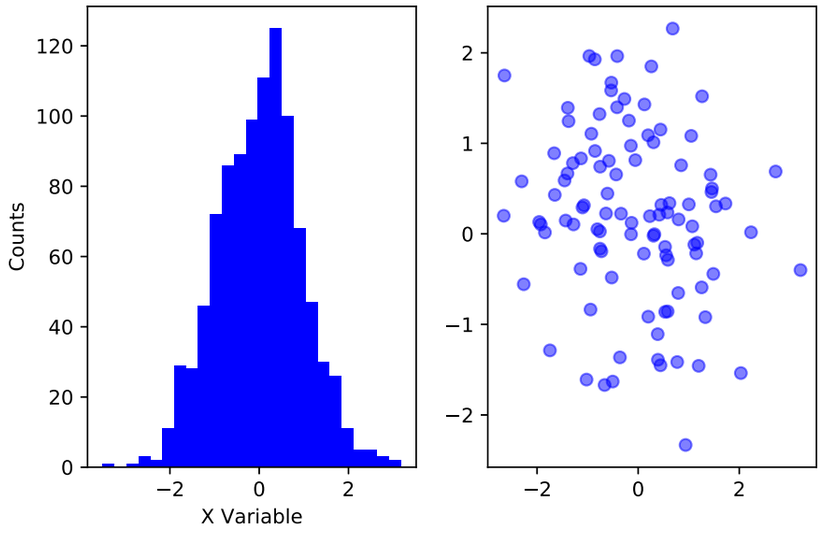

図 D・2　ヒストグラムと散布図

下に示すように 3 次元プロットを作成することもできる.

surf3dscat.py

```
import matplotlib.pyplot as plt
import numpy as np
from mpl_toolkits.mplot3d import Axes3D

def npdf(x,y):
    return np.exp(-0.5*(pow(x,2)+pow(y,2)))/np.sqrt(2*np.pi)

x, y  = np.random.randn(100), np.random.randn(100)
z = npdf(x,y)

xgrid, ygrid = np.linspace(-3,3,100), np.linspace(-3,3,100)

Xarray, Yarray = np.meshgrid(xgrid,ygrid)
```

```
Zarray = npdf(Xarray,Yarray)

fig = plt.figure(figsize=plt.figaspect(0.4))
ax1 = fig.add_subplot(121, projection='3d')
ax1.scatter(x,y,z, c='g')
ax1.set_xlabel('$x$')
ax1.set_ylabel('$y$')
ax1.set_zlabel('$f(x,y)$')

ax2 = fig.add_subplot(122, projection='3d')
ax2.plot_surface(Xarray,Yarray,Zarray,cmap='viridis',
                                      edgecolor='none')
ax2.set_xlabel('$x$')
ax2.set_ylabel('$y$')
ax2.set_zlabel('$f(x,y)$')

plt.show()
```

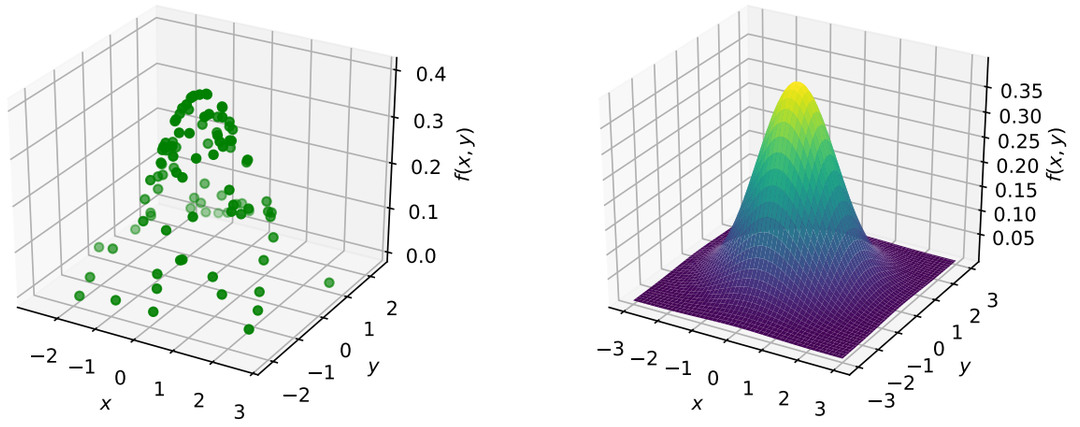

図 D・3　3 次元散布図と表面プロット

D・12　Pandas

Python の Pandas パッケージ（モジュール名 **pandas**）は，基本となる DataFrame クラスをはじめ，データ分析のためのさまざまなツールやデータ構造を提供する．

本節のコードでは，**pandas** が以下によりインポートされていることを仮定している．
`import pandas as pd.`

D・12・1　シリーズとデータフレーム

pandas の二つの主要なデータ構造は Series と DataFrame である．Series オブジェクトは，辞書と 1 次元の ndarray を組合わせたものと考えられる．Series オブジェクトを作成する構文は以下の通りである．

```
series = pd.Series(<data>, index=['index'])
```

ここで，<data>は 1 次元の ndarray，リスト，辞書などの 1 次元のデータ構造であり，index は<data>と同じ長さの名前のリストである．<data>が辞書の場合，index はその辞書のキーから作成される．<data>が ndarray で，index が省略された場合，デフォルトのインデックス [0, ..., len(data)-1] となる．

```
DICT = {'one':1, 'two':2, 'three':3, 'four':4}
print(pd.Series(DICT))
```

```
one      1
two      2
three    3
four     4
dtype: int64
```

```
years =  ['2000','2001','2002']
cost = [2.34, 2.89, 3.01]
print(pd.Series(cost,index = years, name = 'MySeries')) #名付ける
```

```
2000    2.34
2001    2.89
2002    3.01
Name: MySeries, dtype: float64
```

pandas で最もよく使われるデータ構造は，2 次元の DataFrame である．これは，pandas によるスプレッドシートの実装と考えることもできるし，辞書として各「キー」が列名に対応し，辞書の「値」がその列のデータになったと考えることもできる．DataFrame を作成するには，pandas の DataFrame メソッドを使用する．このメソッドには，data，index（行ラベル），columns（列ラベル）の三つの主要な引数がある．

```
DataFrame(<data>, index=['<row_name>'], columns=['<column_name>'])
```

インデックスが指定されていない場合，デフォルトのインデックスは [0, ..., len(data)-1] となる．データは§1・1 のように CSV ファイルや Excel ファイルから直接読み込むこともできる．データフレームを作成するのに（以下のように）辞書を使った場合，辞書のキーが列名として使われる．

§1・1

```
DICT = {'numbers':[1,2,3,4], 'squared':[1,4,9,16] }
df = pd.DataFrame(DICT, index = list('abcd'))
print(df)
```

```
   numbers  squared
a        1        1
b        2        4
c        3        9
d        4       16
```

D・12・2　データフレームの操作

DataFrame や Series オブジェクトにエンコードされたデータを抽出，変更，結合する必要が出てくることは多い．列の取得，設定，削除は，辞書と同様の方法で行うことができる．以下のコードでは，さまざまな操作を例示している．

```
ages = [6,3,5,6,5,8,0,3]
d={'Gender':['M', 'F']*4, 'Age': ages}
df1 = pd.DataFrame(d)
df1.at[0,'Age']= 60  # 要素を変更
df1.at[1,'Gender'] = 'Female' # 別の要素を変更
df2 = df1.drop('Age',1)  # 列の削除
df3 = df2.copy();    # df2 のコピーを作成
df3['Age'] = ages   # 元の列を追加
dfcomb = pd.concat([df1,df2,df3],axis=1)  # df1,df2,df3 の三つを結合
print(dfcomb)
```

```
   Gender  Age  Gender  Gender  Age
0       M   60       M       M    6
1  Female    3  Female  Female    3
2       M    5       M       M    5
3       F    6       F       F    6
4       M    5       M       M    5
5       F    8       F       F    8
6       M    0       M       M    0
7       F    3       F       F    3
```

上の DataFrame オブジェクトには二つの Age 列があることに注意せよ．式 dfcomb['Age'] は，これら両方の列をもつ DataFrame を返す．

表 D・3　データ操作に便利な pandas のメソッド

agg	一つ以上の関数を使用してデータを集計する
apply	関数を列または行に適用する
astype	変数のデータ型を変更する
concat	データオブジェクトを連結する
replace	値を検索して置換する
read_csv	CSV ファイルを DataFrame に読み込む
sort_values	行または列に沿って値をソートする
stack	DataFrame をスタックする
to_excel	DataFrame を Excel ファイルに書き出す

Python は異なる型のオブジェクトを異なる方法で扱うことがあるため，データの要約や可視化の作業に取り掛かる前に，変数のデータ型を正しく指定することが重要である．DataFrame オブジェクトの要素の一般的なデータ型は float，category，datetime，bool，int である．汎用のオブジェクト型は object である．

```
d={'Gender':['M', 'F', 'F']*4, 'Age': [6,3,5,6,5,8,0,3,6,6,7,7]}
df=pd.DataFrame(d)
print(df.dtypes)
df['Gender'] = df['Gender'].astype('category')  #型を変更
```

```
print(df.dtypes)
```

```
Gender    object
Age        int64
dtype: object
Gender    category
Age          int64
dtype: object
```

D・12・3　情報の抽出

DataFrame オブジェクトからの統計情報の抽出は，**pandas** の豊富なメソッド（関数）を利用することで容易になる．表 D・4 は，データを調べるメソッドの一部を示している．それらの実践的な使い方については，1 章を参照せよ．以下のコードは便利なメソッドのいくつかの例を示している．**apply** メソッドは，DataFrame の列や行に一般の関数を適用することができる．これらの操作では，データは変更されない．**loc** メソッドは，データフレームの要素（または範囲）にアクセスするためのもので，リストや配列のスライス操作に似た動作をするが，以下のコードに示すように *"stop"* **の値[5]が範囲に含まれる点が異なる**．

§1・1

```
import numpy as np
import pandas as pd
ages = [6,3,5,6,5,8,0,3]
np.random.seed(123)
df = pd.DataFrame(np.random.randn(3,4), index = list('abc'),
                  columns = list('ABCD'))
print(df)
df1 = df.loc["b":"c","B":"C"]       # 部分的なデータフレームを作成
print(df1)
meanA = df['A'].mean()            # 'A' 列の平均
print('mean of column A = {}'.format(meanA))
expA = df['A'].apply(np.exp)  # 'A' 列の各要素の指数関数
print(expA)
```

```
          A         B         C         D
a -1.085631  0.997345  0.282978 -1.506295
b -0.578600  1.651437 -2.426679 -0.428913
c  1.265936 -0.866740 -0.678886 -0.094709
          B         C
b  1.651437 -2.426679
c -0.866740 -0.678886
mean of column A = -0.13276486552118785
a    0.337689
b    0.560683
c    3.546412
Name: A, dtype: float64
```

DataFrame オブジェクトの **groupby** メソッドは，データを巧みに操作して要約および表示するのに便利である．一つ以上の指定された列に沿ってデータをグループ

[5]訳注：loc における範囲指定を start:stop としたときの stop の値を指すものと推測される

表 D・4　データを調べるのに便利な pandas のメソッド

columns	列名
count	欠損でないセル数をカウントする
crosstab	二つ以上のカテゴリをクロス集計する
describe	統計情報を要約する
dtypes	各列のデータ型
head	DataFrame の先頭行を表示する
groupby	データを列でグループ化する
info	DataFrame に関する情報を表示する
loc	グループや行，列にアクセスする
mean	列/行の平均
plot	列のプロット
std	列/行の標準偏差
sum	列/行の合計を返す
tail	DataFrame の末尾行を表示する
value_counts	空でない異なる値ごとのカウント
var	分散

化し，グループ化されたデータに count や mean などのメソッドを適用できるようにする．

```
df = pd.DataFrame({'W':['a','a','b','a','a','b'],
          'X':np.random.rand(6),
          'Y':['c','d','d','d','c','c'], 'Z':np.random.rand(6)})
print(df)
```

```
   W         X  Y         Z
0  a  0.993329  c  0.641084
1  a  0.925746  d  0.428412
2  b  0.266772  d  0.460665
3  a  0.201974  d  0.261879
4  a  0.529505  c  0.503112
5  b  0.006231  c  0.849683
```

```
print(df.groupby('W').mean())
```

```
          X         Z
W
a  0.662639  0.458622
b  0.136502  0.655174
```

```
print(df.groupby(['W', 'Y']).mean())
```

```
            X         Z
W Y
a c  0.761417  0.572098
  d  0.563860  0.345145
b c  0.006231  0.849683
  d  0.266772  0.460665
```

複数の関数を一度に計算するには，**agg** メソッドが利用できる．これは，関数のリスト，辞書，あるいは文字列を引数にとることができる．

```
print(df.groupby('W').agg([sum,np.mean]))
```

```
          X                   Z
        sum      mean       sum      mean
W
a   2.650555  0.662639  1.834487  0.458622
b   0.273003  0.136502  1.310348  0.655174
```

D・12・4　プロット

DataFrame の **plot** メソッドは，Matplotlib を使った DataFrame のプロットを作成する．さまざまな種類のプロットに kind = 'str' の構文でアクセスすることができ，str は line（デフォルト），bar，hist，box，kde といくつかの中から一つをとる．**matplotlib** を直接使用することで，フォントの変更などのきめ細かいコントロールができる．以下のコードは，図 D・4 の折れ線グラフと箱ひげ図を生成する．

```
import numpy as np
import pandas as pd
import matplotlib
df = pd.DataFrame({'normal':np.random.randn(100),
          'Uniform':np.random.uniform(0,1,100)})
font = {'family' : 'serif', 'size'   : 14} #フォントの設定
matplotlib.rc('font', **font)   # フォントの変更
df.plot()   # 折れ線グラフ（デフォルト）
df.plot(kind = 'box')  # 箱ひげ図
matplotlib.pyplot.show()  #プロットのレンダリング
```

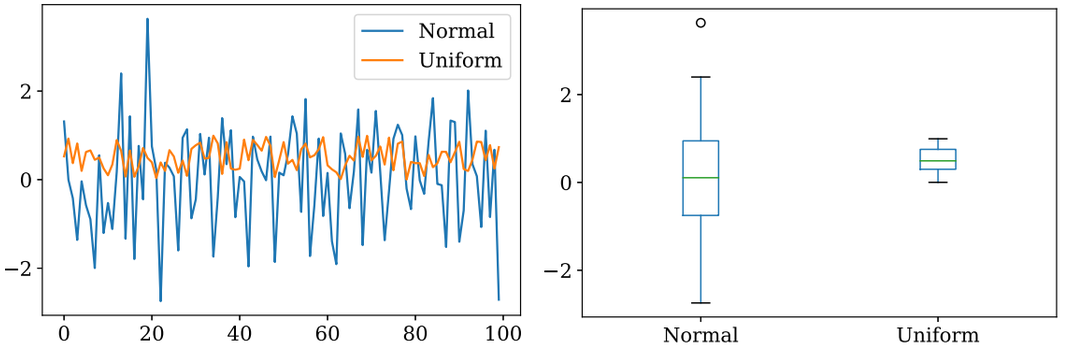

図 D・4　DataFrame の plot メソッドを使った折れ線グラフと箱ひげ図

D・13　Scikit-learn

Scikit-learn は，Python 用の機械学習およびデータサイエンスのオープンソースライブラリである．このライブラリには本書の各章に関連するさまざまなアルゴリズムが含まれており，そのシンプルさと幅広さから広く利用されている．モジュール名

は **sklearn** である．以下では，**sklearn** によるデータのモデリングについて簡単に紹介する．完全なドキュメントは以下で参照できる．

https://scikit-learn.org/

D・13・1　データの分割

モデルをテストするためにデータをランダムに分割することは，**sklearn** の関数 **train_test_split** で簡単に実現できる．たとえば，学習データが説明変数の行列 **X** と応答変数のベクトル **y** で記述されているとする．このとき次のコードは，テストセットが全体の半分となるよう，データセットをトレーニングセットとテストセットに分割している．

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                    test_size = 0.5)
```

例として，以下のコードでは，人工データセットを生成し，それを同じ大きさのトレーニングセットとテストセットに分割している．

syndat.py

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split

np.random.seed(1234)

X=np.pi*(2*np.random.random(size=(400,2))-1)
y=(np.cos(X[:,0])*np.sin(X[:,1])>=0)

X_train , X_test , y_train , y_test = train_test_split(X, y,
    test_size=0.5)


fig = plt.figure()
ax = fig.add_subplot(111)
ax.scatter(X_train[y_train==0,0],X_train[y_train==0,1], c='g',
           marker='o',alpha=0.5)
ax.scatter(X_train[y_train==1,0],X_train[y_train==1,1], c='b',
           marker='o',alpha=0.5)
ax.scatter(X_test[y_test==0,0],X_test[y_test==0,1], c='g',
           marker='s',alpha=0.5)
ax.scatter(X_test[y_test==1,0],X_test[y_test==1,1], c='b',
           marker='s',alpha=0.5)

plt.savefig('sklearntraintest.pdf',format='pdf')
plt.show()
```

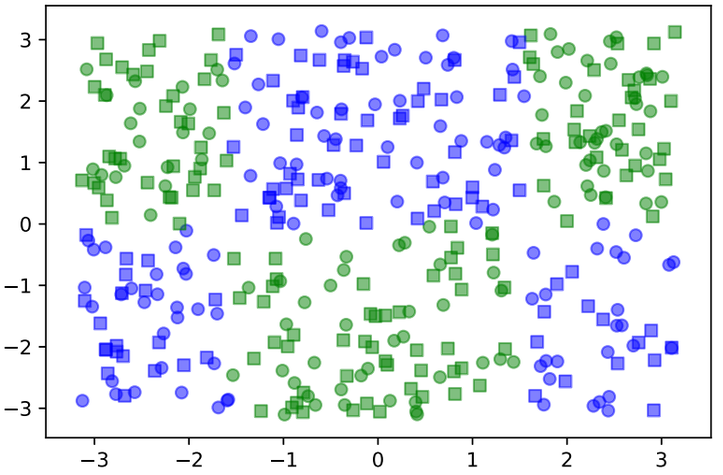

図 D・5　2 クラス分類のトレーニングセット (丸) とテストセット (四角) の例．説明変数は (x, y) 座標，クラスは 0(緑) または 1(青) となっている．

D・13・2　標準化

場合によっては，データを標準化する必要があるかもしれない．これは **sklearn** では **MinMaxScaler** や **StandardScaler** などのスケーリングメソッドで行うことができる．スケーリングは，勾配ベースの推定器の収束を改善したり，スケールが大きく異なるデータを可視化する際に有用である．次の例は，**X** を（たとえば，**numpy** 配列として格納されている）説明変数のデータとして，各値が 0 と 1 の間にあるように正規化している．

```
from sklearn import preprocessing
min_max_scaler = preprocessing.MinMaxScaler(feature_range=(0, 1))
x_scaled = min_max_scaler.fit_transform(X)
# 上と同等のコードを下記に示す：
x_scaled = (X - X.min(axis=0)) / (X.max(axis=0) - X.min(axis=0))
```

D・13・3　フィッティングと予測

データが分割され，必要に応じて標準化されたら，データを分類モデルや回帰モデルなどの統計モデルに適合させることができる．たとえば，上のデータにひき続き，以下のようにモデルをデータに適合させ，テストセットの応答変数を予測する．

```
from sklearn.someSubpackage import someClassifier
clf = someClassifier()        # 適切な分類器を選択
clf.fit(X_train, y_train)     # データにフィッティング
y_prediction = clf.predict(X_test)   # 予測
```

ロジスティック回帰，ナイーブベイズ法，線形・2 次判別分析，K-最近傍法，サ

§7・8 ポートベクトルマシンなどの具体的な分類器については，§7・8 で紹介している

D・13・4 モデルのテスト

モデルが予測を行ったら，関連する評価指標を使用してその有効性をテストすることができる．たとえば分類の場合，テストデータの混同行列を作成することができる．以下のコードは，サポートベクトルマシン分類器を用いて，図 D・5 のデータに対してこれを行っている．

```
from sklearn import svm
clf = svm.SVC(kernel = 'rbf')
clf.fit(X_train , y_train)
y_prediction = clf.predict(X_test)

from sklearn.metrics import confusion_matrix
print(confusion_matrix(y_test , y_prediction))
```

```
[[102  12]
 [  1  85]]
```

D・14 システムコール，URL アクセス，高速化

（Windows，MacOS，Linux のいずれかの）オペレーティングシステムのコマンド（ディレクトリの作成，ファイルのコピーや削除，システムシェルからのプログラムの実行など）は，**os** パッケージを使うことで Python の中で実行することができる．また，ファイルやウェブページを URL から直接ダウンロードすることができる **requests** も便利なパッケージである．次の Python スクリプトでは，この二つを使用している．また，Python での例外処理の簡単な例も示している．

misc.py

```
import os
import requests
for c in "123456":
  try:                          # 存在しない場合には
     os.mkdir("MyDir"+ c)       # ディレクトリを作成する
  except:                       # そうでない場合は
      pass                      # 何もしない

uname = "https://github.com/DSML-book/Programs/tree/master/
   Appendices/Python Primer/"
fname = "ataleof2cities.txt"
r = requests.get(uname + fname)
print(r.text)
open('MyDir1/ato2c.txt', 'wb').write(r.content) #ファイル書き込み
                                  # ここではバイナリモードにすることが重要
```

numba パッケージは，スマートなコンパイルによって計算を大幅に高速化することができる．まず，以下のコードを実行してみよう．

jitex.py

```
import timeit
import numpy as np
from numba import jit
n = 10**8

#@jit
def myfun(s,n):
   for i in range(1,n):
      s = s+ 1/i
   return s

start = timeit.time.clock()
print("Euler's constant is approximately {:9.8f}".format(
                                  myfun(0,n) - np.log(n)))
end = timeit.time.clock()
print("elapsed time: {:3.2f} seconds".format(end-start))
```

```
Euler's constant is approximately 0.57721566
elapsed time: 5.72 seconds
```

ここで,「ジャストインタイム」コンパイラを有効にするために，上のコードの @ 文字の前にある # 文字を削除する．これで 15 倍のスピードアップになる．

```
Euler's constant is approximately 0.57721566
elapsed time: 0.39 seconds
```

さらに知るには

Python を学ぶためには，文献 [82] と文献 [110] をお勧めする．しかし，Python は常に進化しているので，最新の参考文献はインターネットから入手できるだろう．