2 Python 入門

本章では，デザイン科学に関わるさまざまな問題を，フリーかつオープンソースのプログラミング言語である Python を用いて取り扱う。ここでは，Python のインストール方法とプログラミングの基本について概説する。

2.1 Python の概要と基本操作

2.1.1 Python の概要

Python はスクリプト言語に分類され，プログラムを機械語に変換するコンパイル操作なしに手軽に実行できる。また，ライセンスフリーであり，さまざまなライブラリが提供されている。このような理由から，本書では Python を使用して，デザイン科学で必要とされるさまざまなプログラミング技術を学ぶ。

Python には Python 2 の系列と Python 3 の系列があり，後者が前者の上位互換というわけではない。Python 2 の方が動作の安定したツールが多く，2017 年 6 月現在においても多くのユーザーが存在するが，Python 2 の開発はすでに終了しており，言語や標準ライブラリの新しい完全な機能のリリースは行われないこととなっている。将来的には Python 3 に移行する予定であるので，本書では Python 3 を用いることとする。

2.1.2 Python 本体と各種ライブラリのインストール

関数として利用する部品をモジュール，モジュールを複数集めてまとめたものをパッケージと呼ぶ。また，モジュールやパッケージは総称してライブラリと呼ばれる。ライブラリには，あらかじめ Python に組み込まれている標準ライブラリと，別途インストールが必要なライブラリ（サードパーティライブラリ）がある。本書では，NumPy，SciPy などのサードパーティライブラリを用いる。それらを個別にインストールするのは面倒なので，科学技術計算に必要な多くの Python モジュールを含んだ **Anaconda** を利用することを推奨する。

Anaconda とは，Python 本体に加え，科学技術，数学，エンジニアリング，データ分析など，よく利用される Python モジュール（2017 年 6 月時点で 800 以上）を一括でインストール可能にした総合パッケージであり，つぎのサイトからダウンロードできる。

https://www.continuum.io/downloads

ページの中ほどに，それぞれの OS に対応したインストーラーが用意されているので，Python 3.X（2017 年 6 月現在では 3.6）と書いてあるインストーラーをダウンロードする。ダウンロードが終了したら，ダブルクリックでインストールする。インストール場所を特に指定する必要がなければ，画面の案内に従ってつぎに進めばよい。

本書で利用するサードパーティライブラリの一覧を**表 2.1** に示す。

表 2.1 本書で利用するサードパーティライブラリの一覧

ライブラリ名	インストール方法		
	Anaconda	pip	その他
NumPy	○		
SciPy	○		
NetworkX	○		
matplotlib	○		
Numba	○		
SimPy		○	
Graphillion		○	
aima-python			○

Anaconda に組み込まれていないライブラリの多くは，**pip** を用いてインストールすることができる。pip とは，Python のパッケージ管理システムであり，すでに Anaconda に組み込まれている。pip を利用することで，インターネット上からさまざまなライブラリを追加でインストールしたり，すでにインストール済みのライブラリをバージョンアップすることができる。

Windows の場合は，pip はコマンドプロンプト上で利用する。コマンドプロンプトは，「スタートメニュー＞すべてのプログラム＞アクセサリ＞コマンドプロンプト」と進んで起動する。

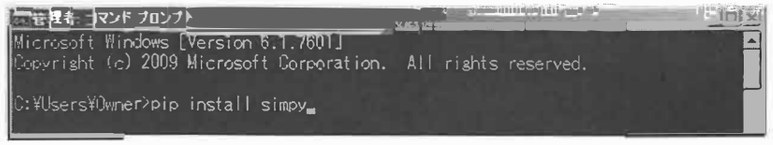
```
pip install ライブラリ名
```

のように打ち込んで Enter キーを押せば，ライブラリのインストールが完了する。例えば，SimPy をインストールする場合はつぎのようにする。

コマンドプロンプトを立ち上げて

```
pip install simpy
```

と入力する。

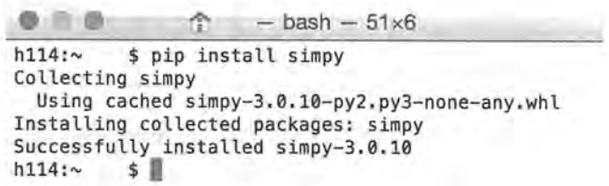


インターネットに接続した状態で，Enter を押す。つぎのように表示されれば，インストールは完了である。

Mac の場合は，ターミナル上で利用する。ターミナルは，「アプリケーション＞ユーティリティ＞ターミナル」に格納されている。Windows の場合と同様に

pip install ライブラリ名

のように打ち込んで Enter を押せば，ライブラリのインストールが完了する。SimPy をインストールする場合はつぎのようになる。



なお，Windows の場合，Graphillion を pip を用いてインストールするためには環境変数の設定が必要となる。また，aima-python についてはインターネットから直接ライブラリをダウンロードする必要がある。詳細は，それぞれ 4.2.3 項と 6.2.7 項で説明する。

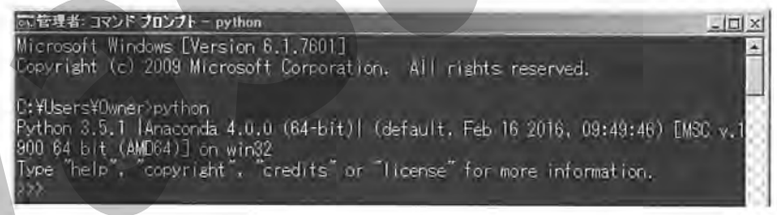
2.1.3　基本的な操作と演算

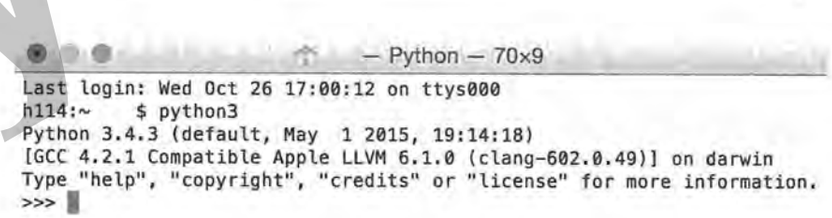
本項での基本的な作業は，以下のいずれかの方法で実行できる。

対話モードを用いた方法　Python 3.6 の Python（command line）を実行すると，対話モードが起動する。対話モードとは，対話をしながら Python を操作できる機能のことであり，タイピングをして命令の結果を即時確認しながらプログラミングができる。

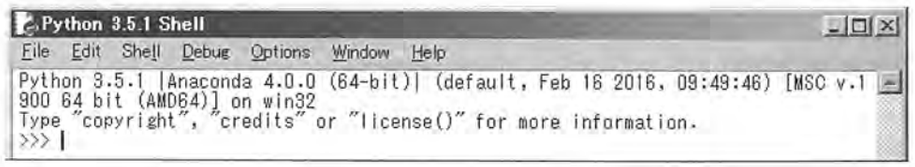
***.py ファイルを用いた方法**　対話モードを用いた場合は，通常の操作では入力したコードを保存することができない。入力履歴を保存できるモジュールは存在するが，タイピングミスも含めて保存されてしまうため，繰り返し利用するコードや長いコードは，テキストエディタでファイルに記述して保存するのがよい。ファイルの拡張子を「.py」とすると，Python のスクリプトファイルとして保存できる。Anaconda をインストールした初期の環境では，「.py」の拡張子が Python の実行ファイルに関連付けられているので，このスクリプトファイルをダブルクリックすることで実行できる。このほかにも，Windows 付属のコマンドプロンプトから実行する方法や，Spider，IDLE，SciTE といったエディタ上で実行するという方法もある（*.py の*は任意の文字列の代わりという意味）。

まず，対話モードを用いて，基本的な操作と演算を概説する。対話モードを起動するには，インストールした Anaconda3 フォルダ以下にある python.exe をダブルクリックするか，Windows の場合はコマンドプロンプト（Mac の場合はターミナル）から python と打ち込んで Enter を押せばよい。対話モードが起動されると，つぎのようにプロンプト「>>>」が表示され，入力待ち状態になる。





あるいは，付属のエディタである IDLE を用いれば，より効率的なスクリプティングが可能である。IDLE を用いた対話モードの起動は，インストールした Anaconda3 フォルダの Scripts の中にある idle.exe をダブルクリックするか，Window の場合はコマンドプロンプトから idle と打ち込んで Enter を押せばよい（Mac は環境によってはデフォルトで入っている idle が起動することもあることに注意）。



以下のように，「10+2*5」と入力して Enter キーを押すと計算結果を返す。

```
>>> 10 + 2 * 5
20
```

算術演算子は以下のとおりである。べき乗は「^」ではないことに注意する。

加算	減算	除算	乗算	べき乗	整数の剰余	整数の切り捨て除算
+	−	/	*	**	%	//

つぎのように，変数を用いて計算し，print 文で変数の値を表示できる。

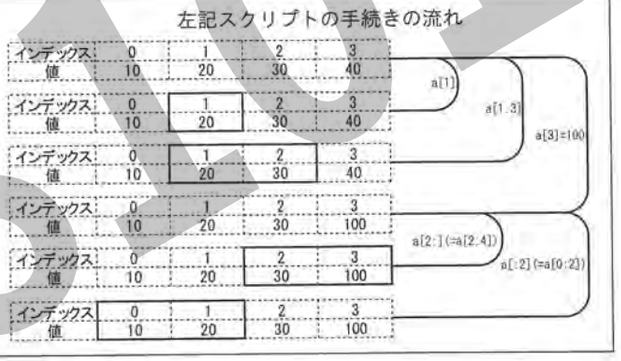
```
>>> a = 100
>>> b = 250
>>> a + b
350
>>> c = b - a
>>> print(a, b, c)
100 250 150
```

変数名には，アルファベット，数字とアンダースコア「_」を使うことができるが，数字で始めることはできない。

インデックスで要素を指定できる変数を，シーケンス型の変数といい，**リスト**や**タプル**などがある。リストは［　］で定義する。また，リストのインデックス（要素の番号）は0から始まることに注意する。要素の範囲は［n:m］のようにコロンで指定する。このような操作をスライスという。ただし，範囲をnからmと指定すると，n+1番目の要素からm番目の要素を指定したことになる。n，mを省略した場合，それぞれ最初あるいは最後のインデックスを指定したことになる。［10，20，30，40］に対してのリスト操作の例を以下に示す。

```
>>> a = [10, 20, 30, 40]
>>> print(a[1])
20
>>> print(a[1:3])
[20, 30]
>>> a[3] = 100
>>> print(a)
[10, 20, 30, 100]
>>> print(a[2:])
[30, 100]
>>> print(a[:2])
[10, 20]
```

左記スクリプトの手続きの流れ



リストの操作のために，つぎのようなさまざまな関数が定義されている。


```
>>> a = [100, 10, 50, 60, 40]
>>> a.append(20)
>>> a
[100, 10, 50, 60, 40, 20]
>>> del(a[1])
>>> a
[100, 50, 60, 40, 20]
>>> max(a)
100
>>> a.sort()
>>> a
[20, 40, 50, 60, 100]
```

a.append(20) のように，リストの class（クラス）（2.4節参照）に定義されてる関数と，del(a[1]) のように，リストを引数（ひきすう）とする関数の2種類あることに注意する。また，リストaをbにコピーする際には，b=a[:] とする必要がある。単にb=aとすると，コンピュータの記憶領域内でのaの位置とbの位置を同一にしたことになり，いずれかの要素の値を変更すると，他方の要素の値も変更されてしまうためである。

一方，タプルは（　）で定義する。基本的にはリストと同じであるが，リストのように要素の値を上書き変更することができず，append や sort などの配列操作も行えないため，変更を望まない配列に対して用いられる。

Python の便利な特徴の1つとして，文字列を数値と同じように使用できることがあげられる。文字列はシングルクオーテーション「'」もしくはダブルクオーテーション「"」で定義する。以降本書では，文字列の記述はシングルクオーテーションで統一する。

```
>>> s1, s2 ='Design1', 'Design2'
>>> print(s1, s2)
Design1 Design2
```

文字列の足し算や掛け算も可能である。

```
>>> s1, s2 = 'Design1', 'Design2'
>>> s1 + s2
'Design1Design2'
>>> s1 * 3
'Design1Design1Design1'
```

2.1.4　条件分岐と繰り返し処理

条件によって処理を変更したいときには **if** 文を用いる。if 文はコロン「:」で区切り，改行の後にインデントを用いて if ブロックをまとめる。インデントにはタブやスペースを用いる。ブロックの文を記述した後，何も入力しないで改行すると，それまでの処理が実行

される。

```
>>> a = 1000
>>> if a > 100:
...     print('Big number')
... else:
...     print('Small number')
...
Big number
```

比較演算子は，**表 2.2** のように定義されている。

表 2.2 比 較 演 算 子

比較演算子	説　明
X==Y	X と Y が等しい場合に True
X!=Y，X<>Y	X と Y が等しくない場合に True
X>Y，X<Y	X と Y の大きさを比較
X>=Y，X<=Y	X と Y が等しい場合を含み X と Y の大きさを比較

また，同じ処理を決められた回数実行するループを組みたいときは，**for** 文を用いる。if 文と同様に，繰り返し実行したい行をインデントして，ブロックとして記述する。例えば，数字 10，20，30 を表示したい場合はつぎのようになる。

```
>>> for i in range(3):
...     print((i + 1) * 10)
...
10
20
30
```

ここで，i は繰り返しを制御する変数であり，range(n) は 0 から n−1 までの整数のシーケンスを返す関数である。range(a,n,b) とすることで，a から n−b までの整数のシーケンスを b の刻みで返すこともできる。例えば，数字 30，20，10 を表示したい場合は以下のようになる。

```
>>> for i in range(30, 0, -10):
...     print(i)
...
30
20
10
```

同様の操作は，**while** 文を用いても可能である。


```
>>> i = 30
>>> while i > 0:
...     print(i)
...     i = i - 10
30
20
10
```

while 文では，while 以下の条件を満たしている間だけ，インデントされた部分が繰り返し実行される。

なお，IDLE を用いた場合は，if，for 文以下は自動でインデントされるが，Windows のコマンドプロンプト上で直接 Python を実行している場合には，手動でインデントする必要があるので注意する。

2.1.5 関 数 の 定 義

自分で関数を定義したい場合には，**def** 文を用いる。if 文と同様に，関数として定義したい行をインデントして，ブロックとして記述する。前述の for 文を func1 という名前の関数として定義するとつぎのようになる。

```
>>> def func1():
...     for i in range(30, 0, -10):
...         print(i)
...
>>> func1()
30
20
10
```

上記の例では，引数と戻り値をいずれも指定していないが，一般的には，

　def 関数名 (引数)

の形で記述し，戻り値を返す場合は，**return** 文で終える。例えば，a,b,c を引数に取り，それぞれの値をリストとして返す関数は，つぎのように記述できる。

```
>>> def func2(a, b, c):
...     answer = []
...     for i in range(a, b, c):
...         answer.append(i)
...     return answer
...
>>> a = func2(30, 0, -10)
>>> print(a)
[30, 20, 10]
```

2.1.6 スクリプトファイルの作成とファイルの入出力

これまでは，コマンドプロンプトに直接命令を入力することによって，プログラムを実行した。しかし，プログラムが長くなると，コマンドとして入力するのは不便である。また，何度も使用するプログラムや関数は，保存できるのが望ましい。

このような目的のために，プログラムをテキストファイルとして保存したスクリプトファイルを用いることができる。頻繁に実行するプログラムを関数として記述したスクリプトを，Python ではモジュールといい，拡張子「.py」をもつスクリプトファイルとして保存する。

また，2.2 節で紹介するようなさまざまな関数を用いて処理を行うとき，データをファイルに保存できれば便利である。

つぎのようなデータ data1.dat を変換するプログラムを作成してみる。

data1.dat

```
first,1,10.00
square,2,3.162
cube,3,2.154
```

open 文と for 文を用いて data1.dat の各値を読み込み，その 2 乗を計算し，data2.dat に値を書き込むプログラム 2.1 を，テキストファイルとして作成し，convert.py という名前で保存する。open 文は，

```
open('ファイル名','入出力モード')
```

の順で記述する。

入出力モードは，ファイルを読み込む場合は 'r'，書き込む場合は 'w' とする。r は read，w は write を意味する。また，# は説明のためのコメントであり，プログラムとは無関係である。日本語のコメントを含むプログラムを実行する場合，エディタによっては Python 側で文字コードが認識できずにエラーを生じる場合がある。そのため，1 行目に

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

を追加してプログラムの文字コードを明示している。後の例では，簡略化のためこの行は省略する。

プログラム 2.1 convert.py

```
f1 = open('data1.dat', 'r')     # data1.dat を読込みモードで開く
f1_lines = f1.readlines()       # ファイルを 1 行ずつ全てを str タイプで読み込み
f1.close()  # data1.dat を閉じる
print('f1:', f1_lines)
f2_lines = []                   # データを格納するリスト
for f1_line in f1_lines:
    x1, y1, z1 = f1_line.split(',')    # カンマを区切りとしてリストに分割
    x2 = x1 + ' root'    # x1 に文字列を追加
    y2 = '%s^%s' % (float(z1), int(y1))  # 文字列の置換
    z2 = str(float(z1) ** int(y1))       # z1 を浮動点少数にして y1 乗
    f2_line = ';'.join([x2, y2, z2]) + '\n'  # リストをセミコロン区切り結合して改行文字を足す
    f2_lines.append(f2_line)
print('f2:', f2_lines)
f2 = open('data2.dat', 'w')     # data2.dat を書込みモードで開く
f2.writelines(f2_lines)         # 2 乗値と空白，改行コードを書き込み
f2.close()                      # data2.dat を閉じる
```


convert.py を実行すると，data1.dat の内容が実数に変換され，以下のような data2.dat が作成される。

data2.dat

```
1 10000.0
2 40000.0
3 160000.0
```

2.2 ライブラリの利用

2.2.1 標準ライブラリの利用

Python に組み込まれている標準ライブラリを利用することで，数値計算やデータ変換，ファイル操作などを効率よく行うことができる。例えば，三角関数や対数のような数学演算を行うための関数は，**math** という名前の標準ライブラリに定義されている。ライブラリは，**import** 文の後にライブラリの名前を記述して利用する。math ライブラリを用いた計算の例をつぎに示す。

```
>>> import math
>>> p = math.pi
>>> print(p)
3.141592653589793
>>> math.cos(p/4.0)
0.7071067811865476
```

math.pi は，math に定義されている定数であり，math.cos() は余弦関数である。このように，「ライブラリ名.関数名」の形で関数を使用できる。ライブラリの名前が長いときには，つぎのように短縮名を用いることもできる。

```
>>> import math as m
>>> m.cos(0)
1.0
```

さらに，from 文を使ってライブラリを読み込み，ライブラリ名を省略して関数を呼び出すことができる。ライブラリに定義されているすべての関数や定数といったオブジェクトをインポートする場合は，つぎのように「*」を用いる。

```
>>> from math import *
>>> cos(0)
1.0
```

しかし，「*」を用いている場合は，他にインポートしているライブラリとの関数名の競合について，注意する必要がある。math ライブラリで利用可能な関数とオブジェクトの例を**表 2.3** に示す。

表 2.3　math ライブラリの関数とオブジェクトの例

関　数	説　明
sin(x)，cos(x)，tan(x)	三角関数
log(x [,y])	自然対数（y を指定すると，y を底とした対数）
degrees(x)	ラジアンから度数に変換
radians(x)	度数からラジアンに変換
pi	数学定数　π
e	自然対数の底　e

また，本書で示す応用例では，乱数や確率変数を使用することがある。その際には，**random** ライブラリを用いる。例えば，0 ～ 1 の一様乱数は，つぎのように生成できる。

```
>>> import random
>>> random.random()
0.3368945444551623
>>> random.random()
0.2918789131039383
```

同じ乱数列を再現できるようにしたい場合は，random.seed(i) で乱数発生のための初期パラメータを設定する。一様分布以外にもさまざまな分布を生成可能であり，例えば平均 x，分散 y の正規分布は，random.normalvariate(x,y) で得られる。

また，ファイルの入出力を簡便に行うライブラリに csv がある。一般に，数値データはエクセルなどの表計算ソフトで管理されることが多い。csv ファイル（カンマ区切りのファイル）は，エクセルで入出力や編集が可能なファイル形式の 1 つである。data1.dat と同じデータを有する csv ファイル

data1.csv

```
1,100
2,200
3,400
```


に対して，プログラム 2.1 と同様の操作を行うプログラムは，ライブラリ **csv** を用いてつぎのように簡潔に書ける。なお，csv の仕様上，2.1.6 項と同様の方法では出力の際に空行が挿入されてしまう。それを防ぐためには，open 文に newline='' の追記が必要となることに注意する。

プログラム 2.2 csvtest.py

```
import csv  # csv ファイルのインポート。このファイル自体を csv.py にすると競合するので注意。
f1 = open('data1.csv', 'r')  # data1.csv を入力のために open
f2 = open('data2.csv', 'w', newline='')  # data2.csv を出力のために open
reader = csv.reader(f1)  # data1.csv を csv 形式で認識
writer = csv.writer(f2)  # data2.csv を csv 形式で認識
for row in reader:  # csv ファイルの内容を 1 行ずつリストとして読み込む
    data1, data2 = float(row[0])**2, float(row[1])**2  # 実数に変換し 2 乗する
    writer.writerow([data1, data2])  # 1 行ずつ書き込む
```

csv ライブラリを使うことで，プログラム 2.1 のように数値を文字列に変換しなくても値をファイルに書き込むことができる。csvtest.py を実行すると，つぎの csv ファイルが作成される。

data2.csv

```
1.0,10000.0
4.0,40000.0
9.0,160000.0
```

また，単純な平面図形であれば，標準ライブラリである **turtle** ライブラリを利用することで，描画することができる。まず，turtle ライブラリをインポートし，線分を引いてみる。つぎのコマンドを実行すると，ウインドウが現れ，5 の線幅，青色で**図 2.1** のように右向きに矢印の付いた線が引かれる。

```
>>> from turtle import *
>>> pensize(5)
>>> pencolor('blue')
>>> forward(400)
```

forward(x) は，x の距離だけ前進するという関数である。pensize，pencolor は線幅，線色の指定であり，省略すると，線幅 1 の黒色となる。三角形はつぎのコマンドによって**図 2.2** のように描画できる。

```
>>> from turtle import *
>>> penup()
>>> setpos(-200, -100)
```

```
>>> pendown()
>>> for i in range(3):
...     forward(400)
...     left(120)
...
```

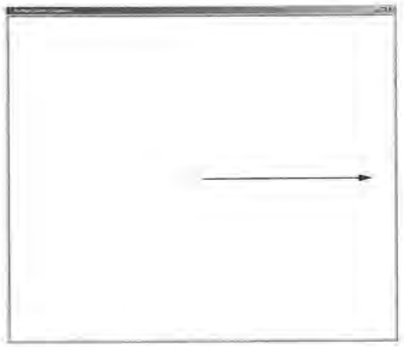


図 2.1　turtle による矢印の描画

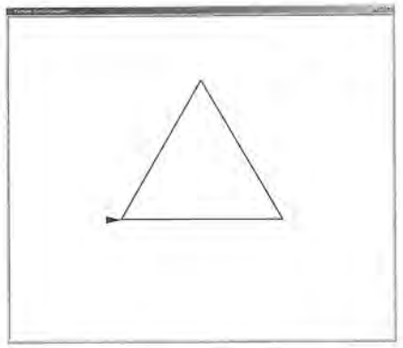


図 2.2　turtle による三角形の描画

単純に三角形を描くだけであれば，penup( )～pendown( ) は不要である。ここでは，画面の中ほどに三角形を描画するために，描画のスタート地点を（−200, −100）の位置に調整している。left(x) 関数は，進む方向を引数の角度（単位は度〔°〕）だけ反時計回りに回転する関数である。turtle は，いわゆる turtle graphics を実現するためのライブラリであり，矢印の位置に，矢印の方向を向いた亀がいて，それが動いた軌跡として線が引かれると考えればよい。left(x) は，亀の向きを変える関数である。turtle ライブラリの関数やオブジェクトの例を**表 2.4** に示す。

表 2.4　turtle ライブラリの関数とオブジェクトの例

関　数	説　明
right(x)	x〔°〕時計回りに回転する
setpos(x, y)	座標 (x, y) まで移動する
reset()	キャンバスをリセットし，初期状態に戻す。
penup()	ペン先をキャンバスから離す
pendown()	ペン先をキャンバスに置く
pencolor('blue')	線の色を変更する
pensize(3)	線の太さを変更する
circle(x [,y])	半径 x で円を描く。x の値が正のときは反時計回り，負のときは時計回り。y を指定したときは，y〔°〕度まで円弧を描く。

つぎのコマンドを実行すると，**図 2.3** のように星が描画される。

```
>>> from turtle import *
>>> penup()
>>> setpos(-300, -100)
```



```
>>> pendown()
>>> for i in range(5):
...     forward(600)
...     left(144)
...
```

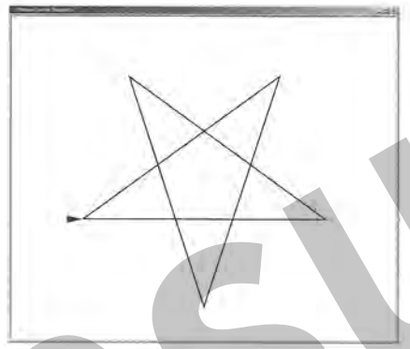


図 2.3　turtle による星の描画

2.2.2　matplotlib を用いたグラフの作成

matplotlib というライブラリを用いることで，簡単にさまざまなグラフを作成することができる。つぎのように散布図を描いてみる。

```
>>> import matplotlib.pyplot as plt
>>> plt.plot(0.3, 5.01, color='blue', marker='o')
>>> plt.plot(1.02, 6.35, color='red', marker='D')
>>> plt.plot(2.5, 7.4, color='yellow', marker='v')
>>> plt.plot(3.2, 8.3, color='gray', marker='h')
>>> plt.plot(3.97, 8.66, color='black', marker='1')
>>> plt.plot(5.4, 10.2, color='crimson', marker='o', markersize=20)
>>> plt.show()
```

1 行目の import 文によって，matplotlib の pyplot ライブラリを plt という名前でインポートする。2 行目から 7 行目では散布図のデータをメモリの中に作成する。最後に plt.show( ) で蓄積されたデータを**図 2.4** に描画する。

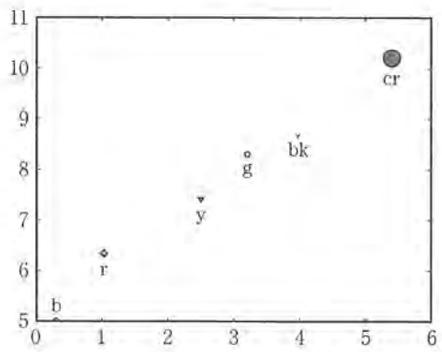


図 2.4　matplotlib を用いた散布図の描画

plot(x,y,color='c',marker='m',markersize='s') の x, y は座標である。color は色の指定であり，**図 2.5** に示す色が指定できる。また，color='#eeefff' などのように，色コードで指定してもよい。

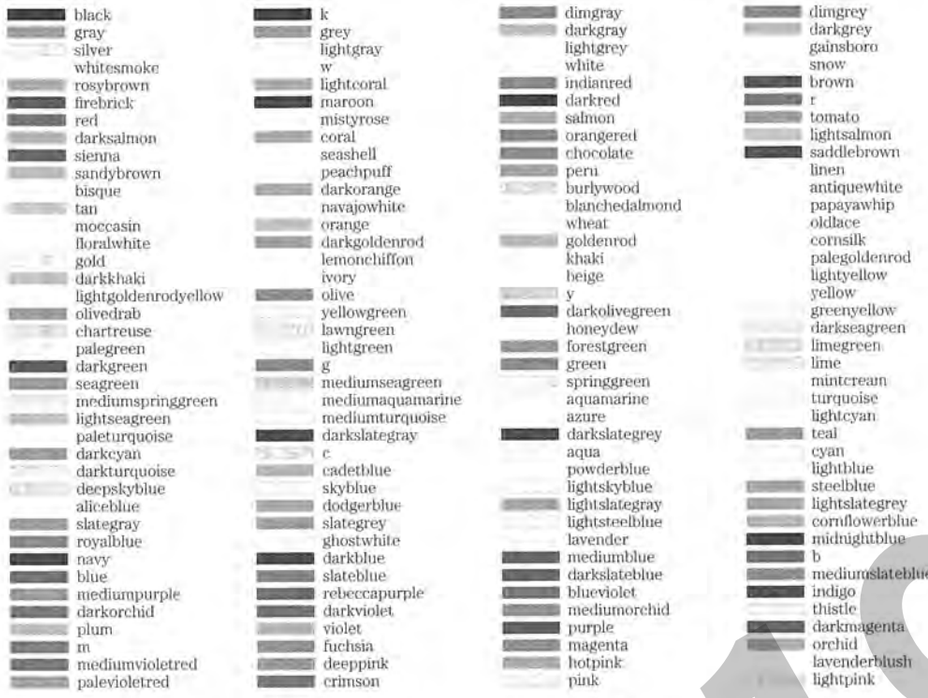


図 2.5　matplotlib の color の一覧

なお，よく使われる 'blue','green','red','cyan','magenta','yellow','black','white' の 8 つの色については，それぞれ 'b', 'g', 'r', 'c', 'm', 'y', 'k', 'w' の略記を用いることができる。

marker はマーカーの指定であり，**表 2.5** に示すマーカーが指定できる。

なお，略記を用いることのできる 8 つの色を用いる場合については，plot(x,y,'bo') のように簡潔な形で色とマーカーを指定することもできる。これは，plot(x,y,color='b',marker='o') あるいは plot(x,y,color='blue',marker='o') と同じである。オプションで，markersize='s' の形でマーカーの大きさを数値で指定することもできる。そのほかにも，matplotlib にはグラフの体裁を整える豊富なオプションが用意されているので，詳細は公式 HP（http://matplotlib.org/examples/color/named_colors.html（2017））を参照されたい。

表 2.5　matplotlib の marker の一覧

marker の表記	marker の表示	marker の表記	marker の表示
'.'	point	's'	square
','	pixel	'p'	pentagon
'o'	circle	'*'	star
'v'	triangle_down	'h'	hexagon1
'^'	triangle_up	'H'	hexagon2
'<'	triangle_left	'+'	plus
'>'	triangle_right	'x'	x
'1'	tri_down	'D'	diamond
'2'	tri_up	'd'	thin_diamond
'3'	tri_left	'|'	vline
'4'	tri_right	'_'	hline
'8'	octagon	' '	nothing

2.2.3　NumPy / SciPy を用いた数値解析

NumPy は数値解析の基本的なパッケージであり，行列演算，線形代数などの基本的なライブラリを提供している。例えば，2 元の線形連立 1 次方程式

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}=\begin{pmatrix} 5 \\ 11 \end{pmatrix} \tag{2.1}$$

を解くプログラムはつぎのとおりである。

プログラム 2.3 solve.py

```
import numpy as np
a = np.array([[1, 2], [3, 4]])  # 行列の定義
b = np.array([5, 11])  # ベクトルの定義
x = np.linalg.solve(a, b)  # 連立 1 次方程式の解
print(x)
```

実行結果は

```
[1. 2.]
```

となる。

np.array は，NumPy で定義された配列であり，行列は 2 次元配列で定義される。linalg は線形代数のライブラリであり，solve は，連立 1 次方程式を解く関数である。逆行列はつぎのように計算できる。

プログラム 2.4 inverse.py

```
import numpy as np
a = np.array([[1, 2], [3, 4]])  # 行列の定義
c = np.linalg.inv(a)  # 逆行列の計算
print(c)
```

実行結果はつぎのとおりである。

```
[[ 2. 1. ]
 [ 1.5 0.5]]
```

より高度な科学技術計算を行いたい場合には NumPy では不十分であり，**SciPy** を用いるのが効果的である。SciPy は，応用数学，科学，工学のための高水準の科学技術計算パッケージである。

例えば，次式の積分計算により円周率を求める場合を考える。

$$\pi = \int_0^1 \frac{4}{1+x^2}\,dx \tag{2.2}$$

SciPy によるプログラムはつぎのようになる。

プログラム 2.5 pifunc.py

```
import numpy as np
from scipy import integrate as itgr  # scipy内の integrate 関数をインポート


def pi(x):  # 被積分関数の定義
    return 4.0 / (1.0 + x**2)
answer = itgr.quad(pi, 0, 1)  # (被積分関数，積分区間下，積分区間上)
print(answer)
```

実行すると，(解，想定される数値計算上の誤差) がタプルの形でつぎのように得られる。

```
(3.1415926535897936, 3.4878684980086326e-14)
```

ここで，e-14 は 10^{-14} を意味する。解だけを取り出したい場合は，answer[0] とすればよい。なお，関数が複雑でない場合，上記のプログラムは，被積分関数の引数を

　　lambda 変数:関数

の形で記述することで，つぎの pilambda.py のように関数定義を直接組み入れることもできる。

プログラム 2.6 pilambda.py

```
import numpy as np
from scipy import integrate as itgr


pi = lambda x: 4.0 / (1.0 + x**2)
answer = itgr.quad(pi, 0, 1)
print(answer)
```

SciPy には，上記の関数以外にも，微分，固有値解析，疎行列の高速演算，非線形方程式の求解，最適化などさまざまな数値計算ライブラリがある。

2.2.4　ユーザーライブラリの作成

ライブラリは自分で作成することもできる。例として，csv ファイルからデータを読み込んで出力する関数と，与えられたデータを散布図にプロットする関数をそれぞれ作成し，ライブラリ module1.py として保存してみる。

プログラム 2.7 module1.py

```
def read_data(file_name):
    import csv
    reader = csv.reader(open(file_name, 'r'))
    X, Y = [], []
    for row in reader:
        X.append(row[0]), Y.append(row[1])
    return X, Y  # データを返す


def draw_graph(X, Y, xmin, xmax, ymin, ymax, Lc, Ls, Lw, title, xlabel, ylabel):
    import matplotlib.pyplot as plt
    plt.xlim(xmin, xmax)  # X の範囲の指定
    plt.ylim(ymin, ymax)  # Y の範囲の指定
    plt.title(title)  # グラフタイトル
    plt.xlabel(xlabel)  # X 軸タイトル
    plt.ylabel(ylabel)  # Y 軸タイトル
    plt.plot(X, Y, color=Lc, linestyle=Ls, linewidth=Lw)  # グラフをメモリ上に
作成
    plt.show()  # グラフの描画
```

read_data はファイル名を受け取って読み込んだデータを返す関数，draw_graph は X，Y 座標のリストと描画範囲，線の色，スタイル，幅，グラフのタイトル，X，Y 軸のラベル名を受け取って線グラフを描く関数である。module1.py と同じフォルダ内に以下のような data3.csv というファイルを作成して保存する。

data3.csv

```
0.3,5.01
1.02,6.35
2.5,7.4
3.2,7.3
3.97,7.66
5.4,4.2
6.7,2.1
7.1,1.2
8.4,9.2
9.6,5.4
```

このデータを，つぎのプログラム plot.py を実行して描画する。

プログラム 2.8 plot.py

```
import module1  # 作成したモジュール module1.py の読み込み

file_name = 'data3.csv'  # ファイル名
```

```
x, y = module1.read_data(file_name)　　# module1 内の関数 read_data によりデータを取得
xmin, xmax, ymin, ymax = 0, 10, 0, 10　# 描画範囲
lc, ls, lw = 'black', '-', 2.0　# 描画オプション
title, xl, yl = 'LineGraph', 'X-Axis', 'Y-Axis'　# タイトル，ラベル
module1.draw_graph(x, y, xmin, xmax, ymin, ymax, lc,
                   ls, lw, title, xl, yl)　# module1 内の
# 関数 draw_graph によりグラフ描画
```

実行すると**図 2.6** のようなグラフが描画される。

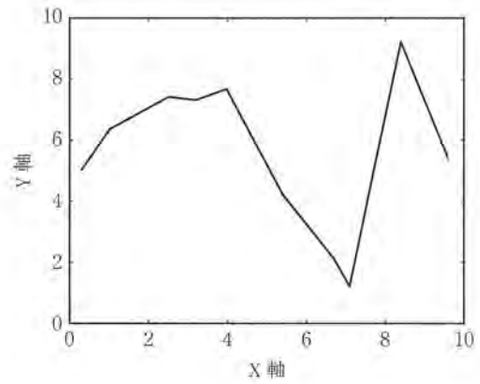


図 2.6　plot.py の実行結果

2.3　再帰プログラミング

ある関数から自分自身を呼び出すことを，**再帰呼出し**という。また，再帰呼出しを用いたプログラムを作成することを，**再帰プログラミング**という。再帰呼出しを用いると，複雑な操作を簡略なプログラムで実行できる場合がある。

文字列を再帰的に書き換えて成長させるプログラム grow.py を以下に示す。

プログラム 2.9 grow.py

```
def grow(s, r):  # 文字列 s と繰り返し回数 r を入力
    n = len(s)  # 入力した文字列の長さ
    ss = ''  # 出力する文字列を初期化
    for i in range(n):
        if s[i] == 'f':  # 'f' を 'fg' に書き換え
            ss = ss + 'fg'
        else:
            if s[i] == 'g':  # 'g' を 'gh' に書き換え
                ss = ss + 'gh'
            else:
                ss = ss + 'h'  # その他の文字（h）のときそのまま
    print(ss)
    r -= 1  # 残りの繰り返し回数を 1 減らす
    if r > 0:  # 繰り返し回数が 0 でないとき，自分自身を呼び出す
        grow(ss, r)
    return ss

grow('fgh', 2)  # grow を 2 回実行
```


このプログラムの出力結果は fgghghhh である。

つぎに，単純なルールを再帰的に適用することによって図形を描いてみる。プログラムファイル rose.py 内に，四角形が入れ子になった図形が描く関数 rose_window_recursion を記述し，保存する。

プログラム 2.10 rose.py

```
# -*- coding: utf-8 -*-
from turtle import *  # 描画環境 turtle をインポート


# rose_window_recursion(四角形の 4 頂点，内分比，繰り返し回数)
def rose_window_recursion(points, ratio, depth):
    rectangle(points)
    new_points = deviding_points(points, ratio)
    if depth == 0:
        up()
        setpos(-200, -200)
    else:
        rose_window_recursion(new_points, ratio, depth - 1)


def deviding(p0, p1, r):
    return p0 * (1 - r) + p1 * r


#---------- 以下は補助的な関数 ----------
# rectangle(四角形の 4 頂点)
def rectangle(points):
    [[x0, y0], [x1, y1], [x2, y2], [x3, y3]] = points
    up()
    setpos(x0, y0)
    down()
    setpos(x1, y1)
    setpos(x2, y2)
    setpos(x3, y3)
    setpos(x0, y0)


# 2 点の内分点を求める．
# deviding_point(点 A，点 B，内分比)
def deviding_point(p0, p1, ratio):
    [x0, y0] = p0
    [x1, y1] = p1
xr = deviding(x0, x1, ratio)
    yr = deviding(y0, y1, ratio)
    return [xr, yr]
```

```
# 四角形の各辺の内分点を求める.
# deviding_points(四角形の4頂点, 内分比)
def deviding_points(points, ratio):
    [p0, p1, p2, p3] = points
    pr0 = deviding_point(p0, p1, ratio)
    pr1 = deviding_point(p1, p2, ratio)
    pr2 = deviding_point(p2, p3, ratio)
    pr3 = deviding_point(p3, p0, ratio)
    return [pr0, pr1, pr2, pr3]
```

rose.py をライブラリとして呼び出すつぎのファイル plotrose_1.py を実行することで，**図 2.7** のような図形が描画できる。

プログラム 2.11 plotrose_1.py

```
from turtle import *  # 描画環境 turtle をインポート
from rose import *  # plot1.py と同一フォルダにある rose.py をインポート
hideturtle()
rose_window_recursion(
    [[-100, -100], [100, -100], [100, 100], [-100, 100]], 0.1, 40)
done() # turtle の終了処理
```

関数のパラメータ（四角形の各頂点，内部の四角形の頂点位置を指定する変数，再帰回数）をつぎの plotrose_2.py のように変更することで，plotrose_1.py とは異なった**図 2.8** が描画できる。

プログラム 2.12 plotrose_2.py

```
from turtle import *  # 描画環境 turtle をインポート
from rose import *  # plot1.py と同一フォルダにある rose.py をインポート
hideturtle()
rose_window_recursion(
    [[-100, -100], [100, -100], [100, 100], [-100, 100]], 0.25, 10)
done()  # turtle の終了処理
```

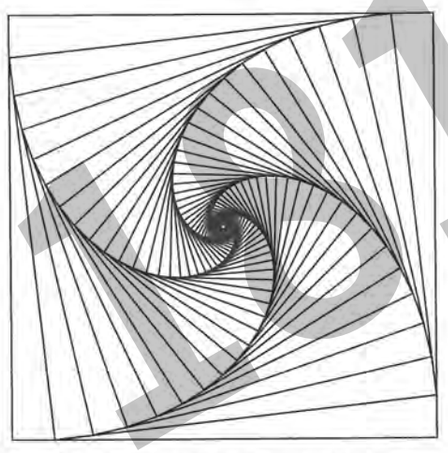


図 2.7　plotrose_1.py の実行結果

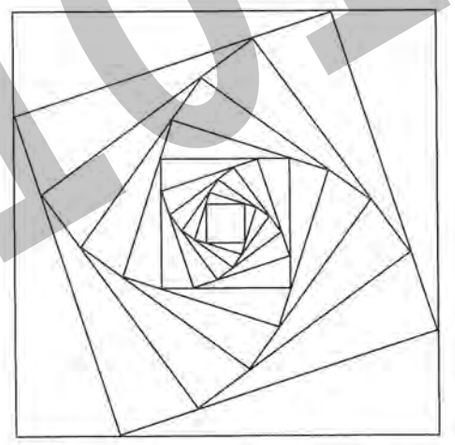


図 2.8　plotrose_2.py の実行結果

2.4 クラス

Python はオブジェクト指向言語に分類され，**クラス**（Class）によってさまざまな**オブジェクト**を定義できる。2.5 節以降でクラスの枠組みを使ったデータ構造を利用するので，本節でその基本的な内容について概説する。

クラスはオブジェクトの共通の変数（データ属性）や関数（メソッド）を定義した枠組みである。例えば，つぎのように長方形を表すクラス Rectangle を作成し，rect.py というファイルで保存する。

プログラム 2.13 rect.py

```
class Rectangle:

    def __init__(self, dx, dy):  # 初期化関数
        self.dx = dx
        self.dy = dy

    def cal_area(self):  # 面積を計算する関数
        self.area = self.dx * self.dy
        return self.area
```

すべてのクラスには，初期化する（オブジェクトを生成する）ときに実行される関数 __init__(self) が定義されなければならない。ここで，self はオブジェクト自身を意味する。cal_area(self) は，長方形の面積を計算する関数である。

つぎのようなプログラムを作成して，rect1 というオブジェクトを生成する。

プログラム 2.14 object.py

```
from rect import *  # rect.py の内容をインポートする
rect1 = Rectangle(200, 100)  # Rectangle クラスのオブジェクト rect1 を生成
print(' Side lengths of rect1: ', rect1.dx, rect1.dy)
aa = rect1.cal_area()  # rect1 の面積の計算
print(' Area of rect1: ', aa)
```

実行結果はつぎのとおりである。

```
Side lengths of rect1: 200 100
Area of rect1: 20000
```

つぎに，長方形の一種である正方形のクラスを作ってみる。その際，長方形で定義された関数と変数を用いることにする。1 つのクラスの定義を継承するようなクラスを**サブクラス**という。クラス Square を，Rectangle のサブクラスとして以下のように定義する。

プログラム 2.15 square.py

```
from rect import * # rect.pyの内容をインポートする


class Square(Rectangle):

    def __init__(self, dx):
        self.dx = dx
        self.dy = self.dx # 2辺の長さを等しくする
```

Square のインスタンス sq1 を定義し，1辺の長さを 100 とするつぎの2行を square.py に追記する。

```
sq1= Square(100)
print('Area of sq1: ', sq1.cal_area())
```

実行結果は

```
Area of sq1: 10000
```

となる。

リストのコピーと同様に，クラスのオブジェクトをコピーする際には注意が必要である。通常の変数のような代入操作では，記憶領域内での位置がコピーされたことになるため，標準モジュール copy を用いて，b=copy.copy(a) あるいは b=copy.deepcopy(a) のような手続きでコピーする。

2.5　CAD・CG ソフトウェアとの連係

2.5.1　Python で絵を描く方法

建築のデザイン・コンピューティングを学ぼうとする多くの読者は，早く「絵」をプログラミングを用いて描いてみたいと考えるだろう。では，2次元または3次元の図形を描画するにはどうすればよいだろうか。1つの方法は，Python 付属の turtle ライブラリや，グラフ描画によく用いられる matplotlib ライブラリを import して利用することである。もう1つの方法は，Python での操作に対応している **CAD**（computer aided design）ソフトウェアや **CG**（computer graphics）ソフトウェアを利用することである。建築設計でもよく使用されるモデリングソフト Rhinoceros とそのプラグインの Grasshopper や，本節で解説する Blender は，Python を用いて操作することができる。

本節では 3DCG アプリケーションである Blender を採用する。採用する理由としては，無料であること，Windows や Mac，Linux といった OS を選ばずに使用できること，オープンソースながらも商用ソフトウェアに劣らない機能が利用できることがあげられる。Blender を使用した経験がない，もしくはこのような 3DCG ソフトウェアを触るのがはじめてという読者もいるだろうが，ヘルプやインターネット上のチュートリアルなどを検索して基本操作を確認しながら進めていってほしい。

2.5.2　Blender のインストールと Python スクリプトの実行

（1）**インストール**　まず，Blender をインストールしよう。公式ウェブサイト（https://www.blender.org/download/）にアクセスし，OS に合せて適切なものをインストールする。インストールがすんだら Blender を起動してみよう。基本的なユーザーインターフェースの名称や操作については，公式マニュアル（Help → Manual から開く）を参照してほしい。

（2）**スクリプティングスクリーン**　上部にある Info Editor から Scripting Screen を図 2.9 のように選択することで，デフォルトの Editor 構成からスクリプティング用の Editor 構成に切り替わる。

図 2.9　Blender での Scripting Screen の選択

Scripting Screen では，中央左側の Text Editor で Python スクリプトが実行でき，下側の Python Console では対話型で実行できる。

（3）**Python Console による操作**　つぎのように，Python Console を使ってみよう。

```
>>> import bpy
>>> list(bpy.data.objects)
[bpy.data.objects['Camera'], bpy.data.objects['Cube'], bpy.data.objects['Lamp']]

>>> bpy.ops.mesh.primitive_cube_add(radius=2, location=(5,0,0))
{'FINISHED'}

>>> list(bpy.data.objects)
[bpy.data.objects['Camera'], bpy.data.objects['Cube'], bpy.data.objects
```