Another Book for Python

2021 年 4 月 3 日

S.Matoike

目次

第 1 章

はじめに

記号の読み方

記号	読み	記号	読み
～	ティルダ	{	中括弧　開く
＝	イコール	}	中括弧　閉じる
@	アット・マーク	[	大括弧　開く
#	ハッシュ（シャープ or 井桁）	]	大括弧　閉じる
$	ドル・マーク	|	バーティカル・バー（縦線）
%	パーセント	!	エクスクラメーション・マーク（感嘆符）
^	キャレット	:	コロン
&	アンパサンド	;	セミ・コロン
＊	アスタリスク	”	ダブル・プライム（ダブル・クォーテーション・マーク）
(	小括弧　開く	’	プライム（クォーテーション・マーク or アポストロフィ）
)	小括弧　閉じる	＜	（A は B）より小さい
_	アンダー・スコア（下線）	＞	（A は B）より大きい
＋	プラス	.	ピリオド（小数点）
－	ハイフン（マイナス）	,	カンマ（コンマ）
\ (¥)	バック・スラッシュ（円マーク）	/	スラッシュ（フォワード・スラッシュ or スラント）
?	クェスチョン・マーク	‘	dumb quotes「まぬけな引用符」か？（ほとんど使わない）

文字コードの歴史的な事情（ASCII コードを元に JIS X 0201 が策定された際に、バック・スラッシュ（＼）の文字コードの所に円マーク（¥）が割り当てられた事）により、Windows ではアプリケーションプログラム毎に、その作成者の都合でどちらか一方が表示されるという、ややこしい事になっています。

Windows 以外の OS(Mac や Linux などの Unix 系 OS) では、どちらの文字もキーボードから入力で

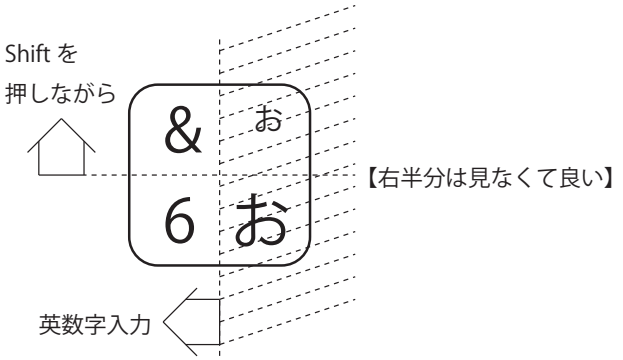
きるので混乱はありません。プログラムの編集などではむしろ、バック・スラッシュを表示するのがいいかもしれません。（グローバルに活躍できる技術者を目指すならば、、、）

最近では、UTF-8 などの Unicode と呼ばれる文字コードが主流となり、それが標準となってきていることから、Windows でも区別して扱えるように対応されつつあります。

日本語キーボードは不要

キーボードから直接日本語を入力することはありません
日本語はアルファベットを使ってローマ字で入力し、それを漢字に変換して入力します
キーボードに書かれている日本語の表記は、プログラミング学習では不要です。
キートップの右半分は無視しましょう。

また、半角のアルファベットの文字は、そのままキーを押すと小文字が入力され、
Shift キーを押しながらキーを押すと大文字を入力できます。



漢字の全角文字と半角の英数字を、きちんと区別して入力できるようになりましょう。
Python では特に、半角の空白文字による段付け（インデント）が、文法上重要な役目を担っています。
C 言語や Java では、インデントが不自然でもそのプログラムは動作しましたが、
Python では、インデントがいい加減だと意味不明としてエラーになります。
半角の空白文字が必要なところを全角の空白文字で埋めてしまうと、見た目では判別しづらいのですがプログラムではエラーになってしまいます。

記号	読み	記号	読み
~		{	
=		}	
@		[	
#	ハッシュ（シャープ or 井桁）	]	
$		|	
%		!	
^		:	
&		;	
*		”	
(		’	
)		<	（A は B）より小さい
_		>	（A は B）より大きい
+		.	
-		,	
\ (¥)		/	
?		‘	dumb quotes「まぬけな引用符」か？（ほとんど使わない）

第 2 章

変数

2.1　変数

プログラミングにおける変数とは「プログラムで使うデータを一時的に格納する入れ物」のようなもの
変数には、数値の他、文字列やリスト（値の並び）など、様々なデータを入れられます
変数にデータを格納することを「代入する」といいます
変数に値を代入するには「変数＝値」の様に記述し、「＝」を「代入演算子」といいます
この時「＝の右側（右辺）にある値を、＝の左側（左辺）にある変数に代入する」という意味です
「右辺と左辺が等しい」という意味ではありません
Python では変数の型宣言は基本的に必要ないので、変数を使いたいときに値を代入するだけです
C 言語や Java では変数の型を指定して宣言する必要がありました（静的型付け言語といいます）

整数型変数

変数の命名規則（この規則は守らないとエラーになる）

- 変数名には半角文字を使う
- 変数名は数字から始まらない
- アルファベットの大文字、小文字、及び数字とアンダー・スコア（_）から構成する
- アンダー・スコア（_）から始まる変数名には、Python の文法上の意味がある
- Python の文法上使われる予約語は変数名に使えない

変数、定数などの命名に関する暗黙の規則（この規則は紳士協定です）

- どのようなデータが格納されているのか、想像しやすい名前を、英単語を基本として命名する
- 変数名では、英単語の単語と単語の間をアンダー・スコア（_）で接続する（snake case）
- 定数名には、アルファベットの小文字を含まない名前を使用する
- クラス名には、単語と単語の間はつめて、単語の先頭文字を大文字にする（camel case）

＃の右側には、コメントを記述できます
＃によるコメントは独立した行に書くのではなく、Python の文の右側に書くようにします（これは、Python のプログラム記述上の紳士協定ですが、本稿では解説する都合に合わせて、独立した行にコメントを書くことが多いので注意してください）

ソースコード 2.1　変数と定数

```
1  # 変数の名前にはアルファベットの小文字を使う
2  # 変数price に、値 100 を代入
3  price = 100
4  # 変数price の内容を印刷
5  print( price )
6
7  # 定数の名前にはアルファベットの大文字を使う
8  # 定数TAX_RATE を定義(税率 8%のつもり)
9  TAX_RATE = 1.08
10 # 税込み価格を印刷
11 xprice = price * TAX_RATE
12 print( xprice )
13
14 # 複合代入演算子:代入＝と同時に行う処理を＝の前に書く
15 # インクリメント(1を足すこと)を複合代入演算子で書いてみる
16 price += 1
17 print( price )
18 # デクリメント(1を減じること)を複合代入演算子で書いてみる
19 price -= 1
20 print( price )
21
22 # 四則演算など
23 a = 10
24 b = 3
25
26 print( '\na=', a )
27 print( 'b=', b )
28 print( '\na+b=', a + b )
29 c = a
30 c += b
31 print( 'c=', c )
32 print( '\na-b=', a - b )
33 c = a
34 c -= b
35 print( 'c=', c )
36 print( '\na*b=', a * b )
37 c = a
38 c *= b
39 print( 'c=', c )
40 print( '\na/b=', a / b )
41 c = a
42 c /= b
43 print( 'c=', c )
44 print( '\na//b=', a // b )
45 c = a
46 c //= b
```

```
47 print( 'c=', c )
48 print( '\na%b=', a % b )
49 c = a
50 c %= b
51 print( 'c=', c )
52 print( '\na**b=', a ** b )
53 c = a
54 c **= b
55 print( 'c=', c )
```

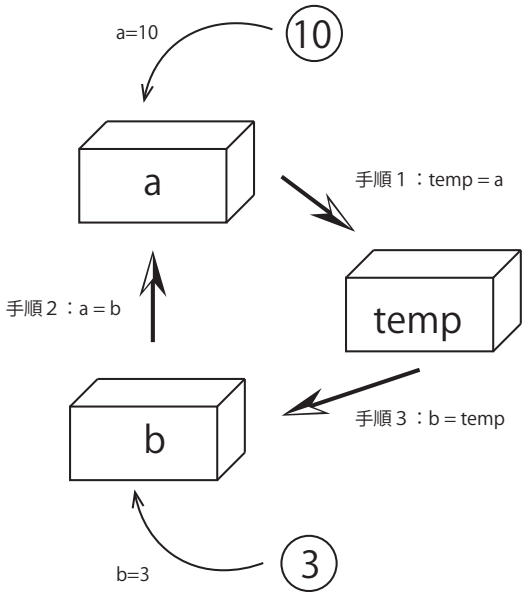
$a+b$ や $a-b$ など、2 つの項 a と b の間の演算を行っているので、2 項演算子といいます

C 言語や Java には、$a//b$ や $a**b$ のような演算子はないので注意しましょう

複合代入演算子（＋＝や－＝）は、代入演算子（＝）の操作と共に＋やーなどの処理も同時に行う演算子です。インクリメントの演算 $a+=1$ は、 $a=a+1$ と同じ意味です。

ソースコード 2.2　代入操作

```
1  # 代入操作について理解を深めましょう
2  a = 10
3  b = 3
4  # a と b を入れ替える前の状態
5  print( '\na=', a )
6  print( 'b=', b )
7
8  # 入れ替えの操作は次の 3ステップ
9  temp = a
10 a = b
11 b = temp
12
13 # 入れ替えた後のa と b の状態
14 print( '\na=', a )
15 print( 'b=', b )
16
17 # Python らしい方法(
       a と b の入れ替え)
18 a, b = b, a
19 print( '\na=', a )
20 print( 'b=', b )
21
22 # 一気に代入する
23 c = 0
24 a = b = c
25 print( '\na=', a )
26 print( 'b=', b )
```



2 つの変数の中身を入れ替える処理はしばしば必要になります

3 ステップで行う変数の内容の入れ替え操作についてよく考えて、代入操作について理解して下さい。

なお、C 言語や Java では、$a, b = b, a$ のように、代入文を 1 行で書く方法はありません（これは Python 独自です）

文字列型変数

ソースコード 2.3　文字列の変数

```
1  # 文字列はクォーテーション・マークで挟む
2  msg = 'Hello'
3  print( msg )
4
5  # 文字列は、プラスの演算子によって結合できる
6  msg2 = msg + '␣' + 'World!'
7  print( msg2 )
8
9  # 文字列は、ダブル・クォーテーション・マークで挟んでも良い
10 msg3 = "Hello␣" + 'everyone.'
11 print( msg3 )
12
13 # 文字列の長さは、len()関数で得られる
14 print( '文字列の長さは、', len( msg ) )
15
16 # エスケープシーケンス
17 print('abcd\tefgh\tijkl\tmnop\nqrst\tuvwx\tyz\n')
18 print('ABCDEFGHIJKLM\nNOPQRSTUVWXYZ\n')
```

文字列はシングル・クォートで挟みます（あるいはダブル・クォートで挟んでもよい）

C 言語や Java における文字列は、ダブル・クォートで挟みました

msg という名前の変数に、”Hello”という文字列をその値として格納しています

print 関数を使って、変数 msg に格納されている値を表示しています

プラス「＋」の演算子によって、文字列を結合することができます

文字列の長さは、len(文字列) 関数によって得ることができます

あたかも 2 文字からなる文字列のように書かれている’\t’ は、タブ文字と呼ばれる特殊な 1 文字を表しています

バック・スラッシュの後に 1 文字を続けて書いて、特殊な機能を持つ 1 文字を表現する方法を「エスケープ・シーケンス」といいます

タブ文字（’\$t$’）以外に、改行文字（’\$n$’）もよく使われます（C 言語の printf 文などで改行文字は多用されています）

整数、実数と文字列の変換

数値同士の演算は普通に行えますが、文字列は足したり引いたりと言うわけにはいきません

また、「＋」演算子によって文字列同士の連結ができますが、数値を文字列の様に繋ぐことはできません

そこで、以下のようにして変数の型を変換する必要がしばしば生じます

str(整数型変数)　によって、整数を文字列に変換できます

str(実数型変数)　によって、実数（浮動小数点数）を文字列に変換できます
int(文字列型変数)　によって、整数を表す文字列を整数に変換できます
float(文字列型変数)　によって、実数を表す文字列を実数（浮動小数点数）に変換できます

ソースコード 2.4　キャストによる型の変換

```
1  # str( 整数型変数 ) によって「整数→文字列」の変換ができる
2  price = 100
3  msg = 'price␣is␣' + str( price ) + '␣yen.'
4  print( msg )
5  # msg = 'price is ' + price + 'yen.' はエラーになる
6
7  # int( 数値の文字列 ) によって「文字列→整数」の変換ができる
8  nebiki = '10'
9  urine = price - int( nebiki )
10 print( urine )
11 # urine = price - nebiki はエラーになる
12
13 # float( 数値の文字列 )によって、「文字列→実数」の変換ができる
14 pi = float( '3.14159265' )
15 r = 10.0
16 area = pi*r**2
17 # str( 実数型変数 )によって、「実数→文字列」の変換ができる
18 print( '円の面積は、'+ str(area) )
19 print( '円周の長さは、', 2.0*pi*r )
```

その他の変数の型

変数の型を、type 関数を使って確認できます

ソースコード 2.5　変数の型

```
1  # i は、整数(Integer)型変数
2  i = 10
3  print( type( i ), '\t', i )
4
5  # a は、浮動小数点(float)型変数
6  a = 3.14159265
7  print( type( a ), '\t', a )
8
9  # x も、浮動小数点型変数
10 x = 271828e-5
11 print( type( x ), '\t', x )
12
13 # s は、文字列(str)型変数
14 s = 'I␣am␣a␣student.'
15 print( type( s ), '\t', s )
16
17 # b は、論理(bool)型変数
```

```
18 b = 10<30
19 print( type( b ), '\t', b )
20
21 # c は、複素数(complex)型変数
22 # j は、虚数単位
23 c = 2 + 1j
24 print( type( c ), '\t', c )
```

2.2　標準入出力

コンピュータには、色々な方法でデータを取り込む（データを入力する）ことができます
例えば、デジカメで撮った画像を取り込んだり、スキャナを使って文書をコンピュータに入力したり
コンピュータへデータを入力する最も手軽な手段は、キーボードから文字を入力することです
キーボードからの入力のことを、「標準入力」という言い方をします（Python では input 関数）

一方コンピュータは、色々な装置にデータを取り出す（データを出力する）ことができます
例えば、プリンタへ印刷するとか、プロッタに作図するとか、ディスクや DVD に保存するとか
コンピュータからデータを出力する最も身近な装置は、画面（CRT など）です
画面への出力のことを、「標準出力」といいます（Python では print 関数）

2.2.1　標準入力

標準入力 (input() 関数) では、キーボードから Enter キーが押されるまでコンピュータは待っています
Enter キーを押すまでに叩いた一連のキーを、文字列としてコンピュータに取り込むことになります

```
1  # 標準出力へprint()関数を使って、文字列を表示する(出力)
2  print('メッセージを入力して下さい')
3  # 標準入力からinput()関数を使って、文字列を受け取る(入力)
4  str = input()
5  # 入力した文字列を、確認のため画面に印刷(出力)してみる
6  print('あなたは、「', str, '」を入力しました')
7  # 画面にメッセージを出力して、入力操作を促すことはよくあることです
8  # input()関数の括弧の中に、メッセージを書いて上の操作を短く記述できます
9  str = input('メッセージを入力して下さい')
10 print('あなたは、「', str, '」を入力しました')
11 # 入力されたのは「文字列」であることを確認しておきましょう
12 # 数字を入力したとしても、それは「数字」=「数値を表す文字列」です
```

2.2.2　標準出力

標準出力の装置である画面に情報を表示する（出力する）とき、体裁を整えたいことがあります

タブ文字、改行文字を出力文字列へ挿入

縦に揃えるための最も手軽な方法は、タブ文字（$\backslash t$）や改行文字（$\backslash n$）を出力することです

行末の文字の指定

print() 関数は、特に指定しない場合、最後に改行文字（$\backslash n$）が出力されます

最後に出力される文字を、他の文字を指定して、改行文字から変更することができます

```
print('print 文の終わりには、\\n␣が自動的に付加されています')
print('終わりの␣\\n␣を,例えば␣===␣に変更してみると、', end="␣===␣")
print('改行␣\\n␣ではなく、␣===␣が出力されています')
print('区切り文字には', '半角スペース 1個', 'が自動的に', '付加されます')
print('区切り文字の', '半角スペース 1個', 'をコロン', 'に変更すると', sep="␣:␣")
```

format による書式指定

書式の指定は、{ インデックス番号：書式指定 } という書き方をします

インデックス番号は、最も左の 0 から順にふられます。自明の場合は省略できます

```
line = '{0}さんの身長は{1}cm、体重は{2}kg です。'.format('太郎', 175, 68.3)
print(line) # 太郎さんの身長は 175cm、体重は 68.3kg です。

line = "{0}さんの身長は{1}cm。{0}さんの体重は{2}kg。".format("太郎", 175, 68.3)
print(line) # 太郎さんの身長は 175cm。太郎さんの体重は 68.3kg。

line = '{}さんの身長は{}cm、体重は{}kg です。'.format("太郎", 175, 68.3)
print(line) # 太郎さんの身長は 175cm、体重は 68.3kg です。

line = "{0}さんの身長は{1:.1f}cm、体重は{2:.1f}kg です。".format('太郎', 175, 68.3)
print(line) # 太郎さんの身長は 175.0cm、体重は 68.3kg です。

line = '{:<20,.3f}'.format(12345.67)
print(line) # '12,345.670 '
```

記号	型
s	文字列 (デフォルト, 通常は省略)
d	整数（10 進数）
b	整数（2 進数）
x 又は X	整数（16 進数）
e 又は E	実数（指数表記）
f 又は F	実数（小数点表記）

第 3 章

制御構文

段付け（インデント）は、Python に特徴的で重要な記述上の規則です
本テキストにおいては、インデントを行う際は「半角の空白文字を 4 個」と決めておきます
全角スペースを段付けに使うことはできません
インデントの際、タブ（'\t'）と空白文字を混在させてはいけません。タブは使わないことにしましょう

3.1　条件分岐

表 3.1　if/elif/else 文

if/elif/else 文

```
if 条件：
ΔΔΔΔ 処理
ΔΔΔΔ 処理
elif 条件：
ΔΔΔΔ 処理
ΔΔΔΔ 処理
else：
ΔΔΔΔ 処理
ΔΔΔΔ 処理
```

if/elif/else 文 の例

```
if n%2==0：
ΔΔΔΔ print('n は偶数です')
ΔΔΔΔ print('% は余りを求める演算子です')
elif 99<n：
ΔΔΔΔ print('n は奇数です')
ΔΔΔΔ print('n は 99 より大きい数です')
else：
ΔΔΔΔ print('n は奇数です')
ΔΔΔΔ print('n は 99 以下の数です')
```

Δ によって半角スペースによるインデントを明示しています。
条件の記述の終わりにはコロン（：）を置きます。
コロンの次の行は、必ずインデントが行われなければなりません。「条件」が成り立った（True になった）場合に実行する処理の範囲（コードブロック）は、インデントされていなければならないのです。
「if」の行の先頭の文字「i」と、「elif」の行の先頭の文字「e」、そして「else」の行の先頭の文字「e」の位置は、縦に揃っていなければなりません。また、インデントされた各処理の先頭の文字（例えば「print(...」の「p」）の位置も、縦に並んでいることを確認しましょう。
C 言語や Java でもインデントは行われますが、これはプログラムを読みやすくするためのインデント

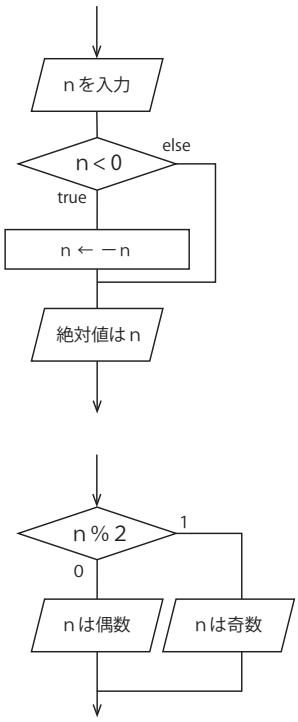
でした。条件が成り立った時に処理される範囲、コードブロックは、中括弧で挟むことで示せたからです。従って、インデントをいい加減に書いても実行結果に影響しませんでした。

しかし Python におけるインデントは文法上の要求であり、見やすさのためだけに行うものではありません。Python では、条件が成立した時に実行される処理の範囲を明示する方法が、インデントしかないのです。インデントが間違っていると、エラーになったり、間違った動作をしたりすることになります。

なお、elif 節や else 節は、常に両方全てを同時に備えている必要はありません。必要になった場合に、elif 節や else 節を追記して if 文の全体を構成することになります。

ソースコード 3.1　if 文

```
1  # 絶対値を求めます
2  s = input('整数を入力：')
3  # 入力された文字列 s を整数 n に変換します
4  n = int( s )
5
6  # 正の数或いは 0だったら､何もしません（そのままpass）
7  # 負の数だったら､－１をかけて正の数にします
8  if n<0:
9      n = -n
10 # 求めた絶対値を出力します
11 print('絶対値は', n)
12
13 # その値は偶数か奇数かを判定します
14 if n%2==0:
15     print( str(n) + 'は偶数です')
16 else:
17     print( str(n) + 'は奇数です')
```



input(' 表示するメッセージ') 関数を使って、標準出力 (画面) にメッセージを表示させ、かつ標準入力（キーボード）から入力された値を、「文字列で」受け取ります。

受け取った文字列のままでは、かけ算をしたり割り算の余りを求めたりするような計算はできません。演算を行うために、入力された文字列を int() によって整数に変換しています。

表 3.2　比較演算子

演算子	意味	使用例（True になる場合）
==	左辺と右辺が等しいなら True	10==10
!=	左辺と右辺が等しくないなら True	32!=10
>	左辺が右辺より大きいなら True	64>32
>=	左辺が右辺より大きいか又は等しいなら True	30>=20
<	左辺が右辺より小さいなら True	20<30
<=	左辺が右辺より小さいか又は等しいなら True	20<=20

>= という演算子はありますが、=>という書き方はないので注意しましょう。

<= という演算子はありますが、=<という書き方はないので注意しましょう。

3.1.1　elif

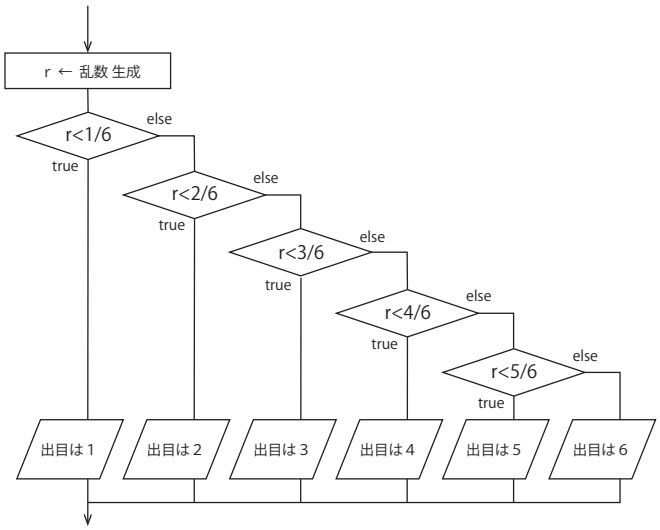
if と else だけを使って記述すると、if 文の入れ子（ネスト）が深くなってしまいます。

ソースコード 3.2　if-else

```
# 乱数を生成するにはrandom というライブラリを import する必要があります
import random
# import した random の中にある、random()という名前の関数を呼び出します
r = random.random()
# r には、「0 以上 1 未満の実数」が代入されます
print('乱数が出した値は、',r,'でした')

'''
 このr の値を、サイコロを振ったときの出目に対応づけます
 0以上 1/6未満だったら、1の目が出たことにします
 1/6以上で 2/6未満だったら、2の目が出たことにします
 2/6以上で 3/6未満だったら、3の目が出たことにします
 3/6以上で 4/6未満だったら、4の目が出たことにします
 4/6以上で 5/6未満だったら、5の目が出たことにします
 5/6以上で 6/6=1未満だったら、6の目が出たことにします
'''

# if と else だけでサイコロを記述しようとすると
fstr = '出目は{:2d}␣({:.3f}␣<=␣{:.3f}␣<␣{:.3f})'
if r<1/6:
    print(fstr.format(1, 0, r, 1/6))
else:
    if r<2/6:
        print(fstr.format(2, 1/6, r, 2/6))
    else:
        if r<3/6:
            print(fstr.format(3, 2/6, r, 3/6))
        else:
            if r<4/6:
                print(fstr.format(4, 3/6, r, 4/6))
            else:
                if r<5/6:
                    print(fstr.format(5, 4/6, r, 5/6))
                else:
                    print(fstr.format(6, 5/6, r, 6/6))
```



クォーテーション・マーク 3 個を連続させると、コメントの記述開始という意味になります。コメントの記述終了も、クォーテーション・マークを 3 個を連続させて指示します。# は 1 行ごとのコメントのために使い、クォーテーション・マーク 3 個を連続させるのは、複数行のコメントの場合に使います。

ここでは、乱数の発生させ方についても学習しましょう。random() という関数は「0 以上 1 未満の実数」による一様乱数を生成します。一様乱数とは、「いかさまサイコロではないよ」という意味です。乱数を使った模擬実験は、モンテカルロ・シミュレーションと呼ばれます。

elif を使うことによって、見やすいプログラムにすることができます。

ソースコード 3.3　elif

```
1  # 乱数を生成するにはrandom というライブラリを import する必要があります
2  import random
3  # import した random の中にある、random()という名前の関数を呼び出します
4  r = random.random()
5  # r には、「0 以上 1 未満の実数」が代入されます
6
7  # elif を使って記述すると次の通り
8  if r<1/6:
9      print('1の目が出た')
10 elif 1/6<=r and r<2/6: # Python 独自の条件の書き方に注目
11     print('2の目が出た')
12 elif 2/6<=r and r<3/6:
13     print('3の目が出た')
14 elif 3/6<=r<4/6:
15     print('4の目が出た')
16 elif 4/6<=r<5/6:
17     print('5の目が出た')
18 else:
19     print('6の目が出た')
20
21 # もちろん、次の様に書いても同じ結果になりますね
22 if r<1/6:
23     print('r<1/6なので、出目は 1')
24 elif r<2/6:
```

```
25      print('r<2/6なので、出目は 2')
26  elif r<3/6:
27      print('r<3/6なので、出目は 3')
28  elif r<4/6:
29      print('r<4/6なので、出目は 4')
30  elif r<5/6:
31      print('r<5/6なので、出目は 5')
32  else:
33      print('r<1なので、出目は 6')
```

Python では、「3/6<= r <4/6」 という条件の書き方が許されています。日本語の通常の言い方をすると「r は 3/6 と 4/6 の間の数である（また、3/6 と等しいこともあり得る)」という意味ですね。この書き方は自然な書き方の様に感じるかもしれませんが、この様な書き方ができるのは Python だけです。

例えばこれと同じ条件を C 言語や Java で記述しようとすると、「(3/6 <= r) && (r <4/6)」 という形で書かなければなりません。つまり、「r は 3/6 以上であり、かつ 4/6 未満の数である」という書き方です。(C 言語や Java で&&は、and「かつ」という意味でした)

Python における条件の書き方は簡易で分かりやすいのですが、Python 以外のプログラム言語、C 言語や Java ではこの様な条件の記述はできない事を覚えておきましょう

なお、C 言語や Java には、switch-case 文がありますが、Python にはありません。if-elif-else を使えば、switch-case 文と同様の機能を実現できます。

また、elif というキーワードは、C 言語や Java では、else if と書かれる部分に対応しています。

random のいろいろ

ソースコード 3.4　random

```
1  import random
2
3  # 実は、こうするとサイコロは簡単に実現できる
4  # 1 以上 6 以下
5  n = random.randint(1, 6)
6  print('出目は、', n )
7
8  # これでもいい
9  # 0 以上 1 未満
10 r = random.random()
11 n = int( r*6 + 1 )
12 print('出目は、', n )
13
14 # 或いは
15 # 0 以上 6 未満
16 r = random.uniform(0, 6)
17 n = int(r + 1)
18 print('出目は、', n )
19
20 # これの方が簡単
21 # 1 以上 7 未満
22 r = random.uniform(1, 7)
```

```
23 n = int( r )
24 print('出目は、', n )
```

3.2　繰り返し

3.2.1　for 文による繰り返し

```
# Q1 文字列　'繰り返し'　を 5回印刷するには
print('繰り返し')
print('繰り返し')
print('繰り返し')
print('繰り返し')
print('繰り返し')
print('繰り返し終わり')

# A1 次のように for 文 と range()関数を使って
for n in range(5):
    print('繰り返し')
print('繰り返し終わり')
```

range() 関数の引数で、反復回数を指定していることを確認しましょう。

```
# Q2 文字列　'繰り返し' と '反復' の２行を 5回印刷するには
print('繰り返し')
print('反復')
print('繰り返し')
print('反復')
print('繰り返し')
print('反復')
print('繰り返し')
print('反復')
print('繰り返し')
print('反復')
print('繰り返し終わり')

# A2 次のように for 文 と range()関数を使って
for n in range(5):
    print('繰り返し')
    print('反復')
print('繰り返し終わり')
```

for 文の右端のコロン（：）の次の行から、反復する処理の範囲をインデントしています。

```
# Q3 文字列　'繰り返し'　を 5回印刷するとともに､繰り返し回数も一緒に印刷するなら
print('0␣繰り返し')
print('1␣繰り返し')
print('2␣繰り返し')
print('3␣繰り返し')
print('4␣繰り返し')
print('繰り返し終わり')
```

```
 8
 9 # A3 次のように for 文 と range()関数を使って
10 for n in range(5):
11     print(n, '繰り返し')
12 print('繰り返し終わり')
```

for n ... において、関数 range(5) によって、n には 0 から　1,2,3,4 までの 5 個が、順に取り出されているのが分かります

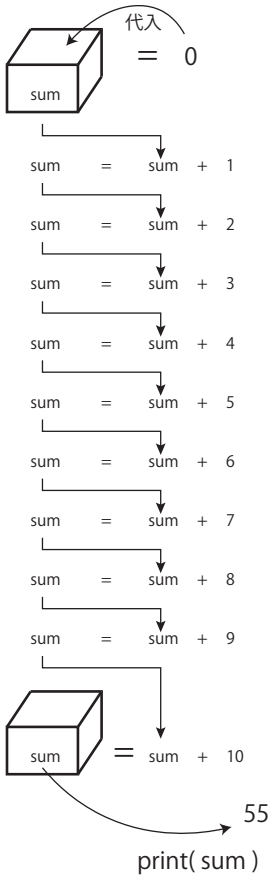
合計の計算

次の計算をする処理をプログラムしてみましょう。$n = 10$ の時、この計算結果は 55 になります。

$$sum = \sum_{i=1}^{n} i = 1 + 2 + \cdots + (n-1) + n$$

ソースコード 3.5　sum

```
 1 # 一歩一歩計算していくと
 2
 3 sum = 0
 4
 5 sum = sum + 1
 6
 7 sum = sum + 2
 8
 9 sum = sum + 3
10
11 sum = sum + 4
12
13 sum = sum + 5
14
15 sum = sum + 6
16
17 sum = sum + 7
18
19 sum = sum + 8
20
21 sum = sum + 9
22
23 sum = sum + 10
24
25 print( 'sum=', sum )
```



for 文による繰り返し処理です。for 文の右端にはコロン（：）を置き、その次の行からインデントしたコードブロックを構成しなければなりません。

range() 関数の使い方について、「要注意！」の部分をもう一度しっかり確認しましょう。

ソースコード 3.6　合計

```
# これを繰り返し処理しようとして次のように書いてみる
sum = 0
for i in range(10):
    print( 'i=', i )
    sum = sum + i
print( 'sum=', sum )

# range( 10 ) は 10を含まない　要注意！
# 11までとしてみる。11は含まないけど、10まで加算できる
sum = 0
for i in range(11):
    sum = sum + i
print( sum )

# 0を足す操作は不要なので 1から加算を始める
sum = 0
for i in range(1, 11):
    sum = sum + i
print( sum )

# 増分を 1ずつ増やす指定（特に指定しなくても、1ずつの増分になっているけど）
sum = 0
for i in range(1, 11, 1):
    sum = sum + i
print( sum )

# 増分を 2ずつ、とばして加算すると「奇数の和」になる
sum = 0
for i in range(1, 11, 2):
    print( 'i=', i )
    sum = sum + i
print( 'sum=', sum )

#「偶数の和」を求めるにはどうしたらいいかな？
```

【演習問題】次の計算を、for 文と range() 関数で記述しなさい

［1］$sum1 = 0 + (-1) + (-2) + \cdots + (-9) + (-10)$

［2］$sum2 = -10 + (-9) + \cdots + (-2) + (-1) + 0$

［3］$sum3 = -10 + (-8) + (-6) + (-4) + (-2) + 0$

［4］$factorial = 8 \times 7 \times \cdots \times 2 \times 1$

階乗の計算

合計 sum と同様な繰り返し処理で、階乗を計算するプログラムを書いてみます。

$$fact = n! = n * (n-1) * (n-2) * \cdots * 3 * 2 * 1$$

ソースコード 3.7　階乗

```
# 5の階乗を計算してみましょう
# 5! = 5 * 4 * 3 * 2 * 1
n = 5
fact = 1
fact = fact * 5
fact = fact * 4
fact = fact * 3
fact = fact * 2
fact = fact * 1
print( 'fact=', fact )

# これを繰り返し処理で記述してみる
# 最後の 6は含まれないので、5までになるはず、、、
fact = 1
for i in range(6):
    fact = fact * i
print( 'fact=', fact )

# おっと！最初にi=0をかけてしまうので、ダメですね
# 1から開始する事を指定しなければなりません
fact = 1
for i in range(1, 6, 1):
    fact = fact * i
print( 'fact=', fact )

# これで一応出来ましたが、
# n=5から 1まで、逆に 1ずつ減らしていくことを考えます
# 1 をかけるのは無駄な処理ですから、2までででいいと思って
n = 5
fact = 1
for i in range(n, 2, -1):
    fact = fact * i
print( 'fact=', fact )

# としてみたら、終わりの 2は含まれていませんでした　要注意！！
# 結局、次の形で 2までのかけ算にすることができました
n = 5
fact = 1
for i in range(n, 1, -1):
    print( 'i=', i )
    fact = fact * i
print( 'fact=', fact )
```

3.2.2　while 文による繰り返し

while に続けて指定している条件が成り立っている間、インデントされたブロックが繰り返されます

フィボナッチ数列

最初の数は 0、次の数は 1 と決めておいて、3 番目から後に続く数はすべて、直前の 2 つの数を加えた値になっているような数列を、フィボナッチ数列といいます

$$
\begin{aligned}
&F_0 = 0 \\
&F_1 = 1 \\
&F_k = F_{k-1} + F_{k-2} \qquad (where\ k \geq 2)
\end{aligned}
$$

ソースコード 3.8　while

```
1  # while を使って 1 から 10 までの合計 sum を求めます
2
3  n = 10
4  sum = 0
5  while n>0:
6      sum = sum + n
7      n = n - 1
8
9  print( 'sum=', sum )
10
11 # フィボナッチ数列
12
13 a, b = 0, 1
14 print( a, end=",␣" )
15 while b < 10:
16     a, b = b, a + b
17     print( a, end=",␣" )
18
19 print( b )
```

break と continue

while True: によって、無限ループする処理をインデントしたブロックに作ることがあります

無限ループは、どこかでその繰り返しを終える条件を書く必要があります

ソースコード 3.9　break と continue

```
1  # break と continue の使われ方を確認しましょう
2
3  n = -10
4  sum = 0
5  while True:
6      print( '途中経過␣n=', n, '␣sum=', sum )
7      n = n + 1
8      if n>10:
9          break
10     if n<=0:
11         continue
12     sum = sum + n
13
14 print( 'sum=', sum )
```

1 から 10 までの整数の足し算をするという処理なので、n が 10 を超えたら繰り返しは終えて（break して）いいのです。

また、n が 1 より小さい値の時は sum の計算をする必要がありませんから、次の値の繰り返しにすぐに入ります。

break に遭遇すると、（その時の一番内側の）繰り返しを打ち切ります。

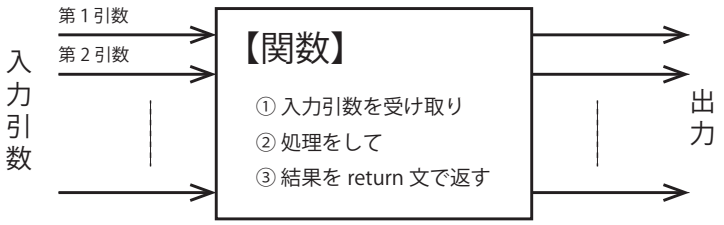
continue に当たると、その回だけは continue 文以下は実行されずに、その次のループ処理に入ります。

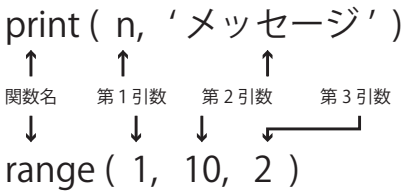
第 4 章

関数

4.1　関数、引数、戻り値

プログラミングでは、
①入力があって、②その入力を元に何らかの処理をして、③その処理の結果を返す出力がある
という、3 つを備えているものを関数と言っています





これまで、print() 関数や range() 関数を使ってきました
print() 関数や range() 関数の、括弧内に指定している情報がそれぞれの関数に渡され、それが関数では入力引数として受け取られて、それぞれの関数での処理が行われます

戻り値のある関数

ソースコード 4.1　関数定義

```
1 a = input('整数を入力してください')
2 # a は文字列
3
4 b = int( a )
5 # b は数値に変換された値
6
7 # 絶対値を求める関数を定義
8 def my_abs( x ):
```

```
    if x < 0:
        x = -x
    return x

c = my_abs( b )
d = my_abs( b ) + my_abs( c )

print( 'b=', b )
print( 'c=', c )
print( 'd=', d )
```

関数定義の引数に書いている変数 x は、関数が入力引数として受け取る「仮の変数」として使います
この「仮の変数」を使って、関数内での処理を記述していきます

戻り値のない関数

```
# 絶対値を求める処理の関数定義です
def my_abs( x ):
    if x < 0:
        return -x
    return x

# 先ほどと同じ結果になります
a = my_abs( 5 )
b = my_abs( -10 )
print( 'a=', a )
print( 'b=', b )
c = a + my_abs( -10 ) + my_abs( 8 )
print( 'c=', c )

# 次の関数定義では、どうなるでしょう
def my_abs2( x ):
    if x < 0:
        return -x

# 引数x が負の数の場合に、-x を返すのは書かれていますから
d = my_abs2( -5 )
print( 'd=', d )

# 一方、引数x が正の数あるいはゼロの場合に、
# 関数の戻り値について書かれていません
e = my_abs2( 5 )
print( 'e=', e )

# None（ナン）という結果が返されています
# 次の様に、return 文で何も指定しなかった場合も None が返されます
def my_abs3( x ):
```

```
32      if x < 0:
33          return -x
34      return
35  #
36  print( my_abs3( -5 ) )
37  print( my_abs3( 5 ) )
38
39  # ちなみに、print()関数の戻り値もNone になります
40  f = print( '戻り値を確認しよう' )
41  print( 'f=', f )
```

4.1.1　再帰呼び出し

フィボナッチ数列や階乗を求める関数は、再帰呼び出しを使って関数定義することができます。

```
1  def fib(n):
2      '''
3      n > 0 の整数を前提に、
4      n 番目のフィボナッチ数を返す
5      '''
6      if n < 2:
7          return n
8      else:
9          return fib(n-1) + fib(n-2)
10
11 n = 5
12 for i in range(n+1):
13     print('fib␣of', i, '=', fib(i
           ))
```

```
1  def factorial(n):
2      '''
3      n > 0 の整数を前提に
4      n! を返す
5      '''
6      if n <= 1:
7          return 1
8      else:
9          return n * factorial(n
               -1)
10
11 n = 5
12 for i in range(n+1):
13     print(i, '!␣=␣', factorial
           (i))
```

4.1.2　lambda 関数

匿名の（関数名を付けない）関数を定義することができます

```
1  c = (lambda a, b: 2*a + 3*b)(1.0, 4.0)
2  print(c)
3  data1 = [1, 2, 3, 4, 5]
4  data2 = [10, 9, 8, 7, 6]
5  result = list( map( lambda a, b: 2*a + 3*b, data1, data2 ) )
6  print(result)
```

4.2　順序引数、キーワード引数とデフォルト引数

関数呼び出し時に「引数を渡した位置 (順番) によって、どのパラメーターがその値を受け取るかが決定される」ような引数の渡し方を「位置引数」(positional argument) と呼ぶ。

関数呼び出し時に「実引数をどのパラメーターに渡すべきか」を「パラメーター名=実引数の値」のようにして指定する引数の渡し方を「キーワード引数」(keyword argument) と呼ぶ。

キーワード引数では、引数で受け取る値のデフォルト値を予め指定しておくことができる。その引数が指定されなかった場合に使われる。

位置引数とキーワード引数が混在する場合は、位置引数が先に書かれなければならない。

```
1 def funcArgs(arg1, arg2, arg3='three'):
2     return f'arg1={arg1},arg2={arg2},arg3={arg3}'
3
4 print( funcArgs(1, 2, 3) ) # 位置引数
5 print( funcArgs( 'one', arg2='two') ) # 引数タイプの混在とデフォルト引数
6 print( funcArgs( 'one', arg3='four', arg2='two') ) # 混在では、位置引数が先
```

4.3　変数のスコープ

```
1 a = 10
2 def func1():
3     a = 100
4     print( 'a␣は␣local：', a )
5 def func2():
6     global a
7     a = 100
8     print( 'a␣は␣global：', a )
9
10 func1()
11 print( a )
12 func2()
13 print( a )
```

```
1 a = 10
2 print('a␣は␣global')
3 for i in range(10):
4     print( a*i, end='␣' )
5
6 print('a␣は␣local')
7 for i in range(10):
8     a = 2
9     print( a*i, end="␣" )
```

ソースコード 4.2　西暦と和暦を変換する関数を作ってみましょう

```
1 def wareki( seireki ):
2     if seireki<1989: # 大正の終わりの年は 1926年
3         return '昭和' + str( seireki-1926+1 ) + '年'
4     elif year<2019: # 昭和の終わりの年は 1989年
5         return '平成' + str( seireki-1989+1 ) + '年'
6     else: # 平成の終わりの年は 2019年
```

```
7         return '令和' + str( seireki-2019+1 ) + '年'
8
9  for year in range(1970, 2021):
10     print('西暦', year, 'は、', wareki(year) )
```

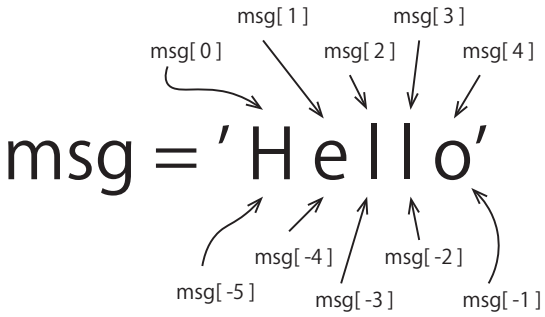
第 5 章

データ構造

5.1　文字列データの操作

ソースコード 5.1　文字列

```
1  # 文字列はクォーテーション･マークで挟む
2  msg = 'Hello'
3  print( msg )
4
5  # 文字列は配列に入っている
6  # インデックスは０から始まる
7  print( msg[0] )
8
9  # インデックスに１を指定すると、
10 # ２文字目が取り出される
11 print( msg[1] )
12 print( msg[2] )
13 print( msg[3] )
14 print( msg[4] )
15 print()
16
17 # 負のインデックスは､後ろから数える
18 print( msg[-1] )
19 print( msg[-2] )
20 print( msg[-3] )
21 print( msg[-4] )
22 print( msg[-5] )
23
24 # ６番目の文字は無いのでエラーになる
25 # IndexError: string index out of
       range
26 print( msg[5] )
```



文字列に対する色々な操作を示します

大文字、小文字の変換、及び文字列の置換は、何れも元の文字列に変更を加えているのではなく、変換した後の新しい文字列オブジェクトを生成して、それを返してくれる関数になっています

ソースコード 5.2　文字列の操作いろいろ

```
# 大文字小文字の変換
msg = 'HELLO python'
print( msg )
str0 = msg.upper()
print( str0 )
str0 = msg.lower()
print( str0 )
str0 = msg.title()
print( str0 )

# 文字列の置換
str0 = msg.replace( 'HELLO', 'Hi')
print( msg, str0, sep=' --> ' )

# 文字列を空白文字で分割
str0 = '   spam  ham    eggs  '
lst = str0.split()
print( lst )

# 左右の空白文字を削除
str1 = str0.strip()
print( "'" + str1 + "'" )

# 文字列の末尾をチェック
str0 = 'sample.jpg'
bool_result = str0.endswith( ('jpg', 'gif', 'png') )
print( bool_result )

# 文字列が数値の文字列かどうかチェック
str0 = '1234567890'
str1 = 'A234B789C'
bool_result0 = str0.isdigit()
bool_result1 = str1.isdigit()
print( bool_result0 )
print( bool_result1 )

# 文字列の長さを取得
print( len(str0), len(str1) )

# 文字列の中に任意の文字列が存在するかどうかチェック
str0 = 'HELLO python'
bool_result = 'py' in str0
print( bool_result )
```

```
bool_result = 'java' in str0
print( bool_result )

# 複数の文字列の連結
str0 = '-'.join( ['spam', 'ham', 'eggs'] )
print( str0 )

# format メソッド
lang, num, name = 'python', 10, 'taro'
str0 = '{}は{}が好きです'.format(name, lang)
print( str0 )

# 引数の順番で指定
str0 = '{1}は{0}が好きです'.format(lang, name)

# キーワード引数で指定
str0 = '{n}はC 言語よりも{num}倍{l}が好きです'.format(l=lang, n=name, num=num)
print( str0 )
```

5.2　リスト、タプル、辞書、集合（セット）

5.2.1　リスト

リストとよばれるデータ構造に対する操作は、Python において多用されます

リストは、複数の要素をカンマで区切って記述します。配列ともよばれます

例えば list という名前の変数を、[5, 4, 3, 2] の様に、4 つの要素がこの順番に入っているリスト（配列）とします

list の「0 番目」の要素は、5 です

list の「1 番目」の要素は、4 です

list の「2 番目」の要素は、3 です

list の「3 番目」の要素は、2 です

「0 番目」とか「1 番目」と言うときの数値のことを、リスト（配列）のインデックスと呼んでいます

リストの長さは、len() 関数によって得られます（文字列の長さも len() 関数で得られましたね）

入れ子にすると、2 次元配列を表せます

append や insert、pop などのメソッドを使うことによって、リストの内容を変更できます（このことをミュータブルという）

```
data = [3, 5, 2, 4, 3, 1]
print( data )

# リストの中で、一致する要素をカウントします
m = data.count( 3 )
print( m )

# リストの要素数(長さ)は
```

```
n = len( data )
print( n )

# index で要素を直接指定できます
print( data[0] )
print( data[n-1] )
print( data[-1] )

# index はコロン(:)で区切って、範囲を指定できます
print( data[1:3] )

# リストの最後に要素を追加
data.append( 8 )
print( data )
# リストの指定箇所に要素を追加
data.insert( 0, 8 )
print( data )
# リストの指定箇所の要素を取り出して削除
v = data.pop( 0 )
print( v )
print( data )
# del を使って、index で指定した要素を削除できます
del data[0]
print( data )
# index を指定しないと,リストそのものを削除してしまいます
del data
# print( data ) # data が削除された後では,これはError になります

# 2次元配列
data2 = [[3, 5, 2], [4, 6, 1]]
print( data2[0] )
print( data2[0][1] )
print( data2[1] )

# アスタリスクの使い方に注目！
gdata = [ [0.0]*6 ]*5
print( gdata[0] )
print( gdata[2][4] )
```

リストのコピー

浅いコピーと深いコピーの違いを意識しておく必要があります。

【リストを使う例題】

ソースコード 5.3　break と continue

```
list = [9, 7, 11, 5, 6, 3, 4, 9, 5]

```

```
x = [1, 2, 3]
# これは、「浅いコピー」
y = x
# リスト
    x の先頭番地が、y にコピーされた
#
    y も x も「同一のリスト要素」の先頭を保持

y[0] = 10
print( y )
print( x ) # x も更新されている
```

```
x = [1, 2, 3]
# これは、「深いコピー」
y = x.copy()
# リスト
    x の値要素が、全て y にコピーされた
#
    y は x とは独立した配列要素を持っている

y[0] = 15 # y 配列だけが更新された
print( y )
print( x ) # x は元のまま
```

```
# 偶数が含まれているかチェックする
for i in list:
    if i%2==0:
        print( i, '偶数ありました。終わります。\n')
        break;
    else:
        print( i, 'は奇数です')

# 偶数だけを印刷する
for i in list:
    if i%2==1:
        continue
    print( i, 'は偶数です')
```

リスト list の中の要素の合計を求めるプログラムです

for 文の中で enumerate() 関数を使うことによって、リストのインデックスと要素の両方を、それぞれ変数 index と、変数 item で受け取っています

ソースコード 5.4　while 文

```
# for の繰り返しで合計を求めます
list = [9, 7, 11, 5, 6, 3, 4, 9, 5]
sum = 0
for index, item in enumerate(list):
    print( 'index=', index, ':', item )
    sum += item
print( 'sum=', sum, '\n' )

# while の繰り返しで、同じく合計を求めます
length = len( list )
index = 0
sum = 0
while index < length:
    item = list[index]
    print( 'list[',index,']=', item )
```

```
16      sum += item
17      index += 1
18  print( 'sum=', sum )
19
20  # リストから繰り返しの要素を順に取り出します
21  fact = 1
22  for i in [5, 4, 3, 2]:
23      fact = fact * i
24  print( 'fact=', fact )
```

5.2.2　タプル

タプル (tuple) 型の変数は、リスト型と同様に複数の要素が順に並んでいる配列です

リストと違って、タプルで作成された要素は変更できません（読み出し専用の変数のことをイミュータブルという）

```
1  # 以下のt1,t2,t3 は,いずれもtuple です
2  # リストは大括弧で囲みましたが、
3  # タプルは小括弧で囲みます
4  t1 = (1, 2, 3, 4, 3, 2, 1)
5  print( t1 )
6  print( type(t1) )
7  print( 't1 の長さは、', len(t1) )
8  print( 't1 の中で３の出現回数は,', t1.count(3) )
9
10 # 大括弧でインデックスの位置の要素を選ぶのは、タプルもリストも同じ方法です
11 print( 't1[2]␣+␣t1[4]␣=␣', t1[2] + t1[4] )
12 # 小括弧ではタプルのインデックスを指定できません！
13 # print( 't1(2) + t1(4) = ', t1(2) + t1(4) ) # これは error になります
14
15 # 小括弧で囲まなくても、カンマで区切るだけでタプルを作れます
16 t2 = -1, -2, -3, -4
17 print( t2 )
18 print( type(t2) )
19 # 要素数 1個のタプルの作り方（カンマ付きに注意！）
20 t3 = (1,)
21 print( t3 )
22 print( type(t3) )
23 # カンマを書かないと、小括弧で囲んでも単なる数値と見なされてしまいます
24 t4 = (1)
25 print( t4 )
26 print( type(t4) )
```

5.2.3　内包表記

内包表記は、リストやタプルなどの順序型の変数の中の各要素に対して、簡易に演算を適用する方法のことです。例えば、変数 list を、次の様にして作ることがあります

```
list =  [ i for i in range(5, 1, -1) ]
```

変数 list は、そもそも ［i］ という形のリストなのですが、この変数 i を range(5, 1, -1) の範囲で変化させることを、for i in range(5, 1, -1) の部分で行って、［5, 4, 3, 2］の形のリストを作っているのです
リストの中で処理を書いてしまうような方法をリスト内包表記といい、Python では多用されます

ソースコード 5.5　リストによる繰り返し

```
# リスト内包表記で、予めリストを作っておいて
n = 5
list = [i for i in range(n, 1, -1)]
print( 'list は、', list )

# そのリストを使って階乗の計算をすると次の通りです
# i に list の中の要素が、順番に取り出されているのが分かります
fact = 1
for i in list:
    print( 'i=', i )
    fact = fact * i
print( 'fact=', fact )
```

内包表記に、if 文が書かれることもあります

```
# タプルを用意しておいて
t = (1, 2, 3, 4, 5)
# そのタプルのデータを元にしてリストを作っています
lst1 = [i for i in t]
print( lst1 )
# タプルt の中の要素を、for で順に取り出し、
# それが偶数の場合だけを、リストの要素にしています
lst2 = [i for i in t if i%2 == 0]
print( lst2 )
```

5.2.4　辞書型

辞書 (dict) 型の変数は、キー (key) と値 (value) がコロンで区切られたペアとして構成されます
辞書の各要素は順序づけられていないため、「何番目の要素」の様なアクセスはできません
各要素へは、キーを指定して、そのキーを持つ値にアクセスすることができます

```
# 一行が長くなるときは、バックスラッシュ(¥)を使って複数行に書くことができます
month = { '睦月': 1, '如月': 2, '弥生': 3, '卯月': 4,\
          '皐月': 5, '水無月': 6, '文月': 7, '葉月': 8,\
          '長月': 9, '神無月': 10, '霜月': 11, '師走': 12}
```

```
print( month )
dist = month['長月'] - month['卯月']
print( '長月'+'と'+'卯月'+'は、', dist, 'ヶ月はなれています\n' )

month['睦月'] = 'むつき'
month['如月'] = 'きさらぎ'
month['弥生'] = 'やよい'
month['卯月'] = 'うづき'
month['皐月'] = 'さつき'
month['水無月'] = 'みなづき'
month['文月'] = 'ふみづき'
month['葉月'] = 'はづき'
month['長月'] = 'ながつき'
month['神無月'] = 'かんなづき'
month['霜月'] = 'しもつき'
month['師走'] = 'しわす'
print( 'キーだけの一覧：', month.keys(), '\n' )
print( '値だけ　の一覧：', month.values(), '\n' )
print( '辞書　　の全体：', month, '\n' )
print( '如月␣は、「' + month['如月'] + '」と読みます')
print( '皐月␣は、「' + month['皐月'] + '」と読みます')
```

5.2.5　集合（セット）

中括弧で要素を囲むとセット型オブジェクトを生成できる

重複する値がある場合は無視されて、一意な値のみが要素として残る

int や float のように異なる型でも値が等価であれば重複していると見なされる

異なる型を要素として持つこともできる。ただし、リスト型のような更新可能なオブジェクトは登録できない。タプルは OK

set 型は順序をもたないので、生成時の順序は記憶されない

空の波括弧は辞書型 dict と見なされるため、空の set 型オブジェクト（空集合）を生成するには特別のコンストラクタ set() を使う

```
# 中括弧{}で要素を囲むとセット型オブジェクトを生成できる
s = {1, 2, 2, 3, 1, 4}
print(s)
print(type(s))

# 集合の要素数は組み込み関数len()で取得できる
print( len(s) )

# set 型は順序をもたないので、生成時の順序は記憶されない
s = {1.23, '百', (0, 1, 2), '百'}
print(s)

# int や float のように異なる型でも値が等価であれば重複していると見なされる
```

```
14 s = {100, 100.0}
15 print(s)
16
17 # リスト内包表記と同様に集合内包表記もある。
18 # リスト内包表記の角括弧 []を波括弧{}にするだけ
19 s = {i**2 for i in range(5)}
20 print(s)
21
22 # 集合に要素を追加するには、add()メソッドを使う
23 s = {0, 1, 2}
24 s.add(3)
25 print(s)
26
27 # 集合から要素を削除するには、discard(), remove(), pop(), clear()メソッドを使う
28 # discard()メソッドは引数に指定した要素を削除する。
29 # 集合に存在しない値を指定すると何もしない
30 s = {0, 1, 2}
31 s.discard(1)
32 print(s)
33
34 s = {0, 1, 2}
35 s.discard(10)
36 print(s)
37
38 # remove()メソッドも引数に指定した要素を削除するが、
39 # 集合に存在しない値を指定するとエラーKeyError になる
40 s.remove(1)
41 print(s)
42
43 s = {0, 1, 2}
44 s.remove(10)
45 # KeyError: 10
46
47 # pop()メソッドは集合から要素を削除し、その値を返す
48 # どの値を削除するかは選択できない。空集合だとエラーKeyError になる
49 s = {2, 1, 0}
50 v = s.pop()
51 print(s)
52 print(v)
53
54 s = {2, 1, 0}
55 print(s.pop())
56 print(s.pop())
57 print(s.pop())
58 print(s.pop())
59 # KeyError: 'pop from an empty set'
60
```

```
# clear()メソッドはすべての要素を削除し、空集合とする
s = {0, 1, 2}
s.clear()
print(s)
```

和集合や積集合など、集合の演算に関するメソッドが用意されています。

```
# set 型のオブジェクトはコンストラクタ set()でも生成できる
# 引数としてリストやタプルのようなイテラブルオブジェクトを指定すると、
# 重複する要素が除外されて一意な値のみが要素となるset オブジェクトが生成される
l = [1, 2, 2, 3, 1, 4]
print(l)
print(type(l))
s_l = set(l)
print(s_l)
print(type(s_l))

# イミュータブルなfrozenset 型はコンストラクタ frozenset()で生成する
fs_l = frozenset(l)

print(fs_l)
print(type(fs_l))

# 引数を省略すると、空のset 型オブジェクト（空集合）が生成される
s = set()
print(s)
print(type(s))

# set()を利用してリストやタプルから重複した要素を取り除くことができるが、
# 元のリストの順序は保持されない
# set 型をリストやタプルに変換するには list(), tuple()を使う
l = [2, 2, 3, 1, 3, 4]
l_unique = list(set(l))
print(l_unique)

# 順序を保持したまま重複した要素を削除する場合や、
# 重複した要素のみを抽出したりする場合
# 二次元配列(リストのリスト)の重複要素を処理したい場合などについては別途

# 和集合(合併、ユニオン)は|演算子またはunion()メソッドで取得できる
s2 = {1, 2, 3}
s3 = {2, 3, 4}
s_union = s1 | s2
print(s_union)
s_union = s1.union(s2)
print(s_union)

# メソッドには複数の引数を指定することができる。また集合set 型だけでなく、
```

```
# set()でset 型に変換できるリストやタプルなども引数に指定できる。
# 以降の演算子、メソッドも同様
print(s_union)
s_union = s1.union(s2, [5, 6, 5, 7, 5])
print(s_union)

# 積集合(共通部分、交差、インターセクション)は&演算子
# またはintersection()メソッドで取得できる
print(s_intersection)

s_intersection = s1.intersection(s2)
print(s_intersection)

s_intersection = s1.intersection(s2, s3)
print(s_intersection)

# 差集合は-演算子またはdifference()メソッドで取得できる
s_difference = s1 - s2
print(s_difference)

s_difference = s1.difference(s2)
print(s_difference)

s_difference = s1.difference(s2, s3)
print(s_difference)

# 対称差集合(どちらか一方にだけ含まれる要素の集合)は^演算子
# またはsymmetric_difference()で取得できる
# 論理演算における排他的論理和(XOR) に相当
s_symmetric_difference = s1 ^ s2
print(s_symmetric_difference)

s_symmetric_difference = s1.symmetric_difference(s2)
print(s_symmetric_difference)

# ある集合が別の集合の部分集合かを判定するには、<=演算子
# またはissubset()メソッドを使う
s1 = {0, 1}
s2 = {0, 1, 2, 3}

print(s1 <= s2)
print(s1.issubset(s2))

# <=演算子もissubset()メソッドも、等価な集合に対してTrue を返す
# 真部分集合であるかを判定するには、等価な集合に対してFalse を返す<演算子を使う
print(s1 <= s1)
print(s1.issubset(s1))
```

```
89 print(s1 < s1)
90
91 # ある集合が別の集合の上位集合かを判定するには、>=演算子またはissuperset()を使う
92 s1 = {0, 1}
93 s2 = {0, 1, 2, 3}
94 print(s2 >= s1)
95 print(s2.issuperset(s1))
96
97 # >=演算子もissuperset()メソッドも、等価な集合に対してTrue を返す
98 # 真上位集合であるかを判定するには、等価な集合に対してFalse を返す>演算子を使う
99 print(s1 >= s1)
100 print(s1.issuperset(s1))
101 print(s1 > s1)
102
103 # 二つの集合が互いに素かを判定するには、isdisjoint()メソッドを使う
104 s1 = {0, 1}
105 s2 = {1, 2}
106 s3 = {2, 3}
107 print(s1.isdisjoint(s2))
108 print(s1.isdisjoint(s3))
```

第 6 章

実践演習

6.1　数独

再帰呼び出しについて学習する一つの例として、数独を解くプログラムを考えます
最初に、board リストに盤面を定義し、盤面の印刷を行う関数 print_board() を定義します
まずは、print_board() をそのまま呼び出して動作を確認してみましょう

ソースコード 6.1　board

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

def print_board():
    global board
    for y in range(9):
        for x in range(9):
            print('␣',end='')
            if x in [2,5]:
                print(board[y][x], end='␣|')
            else:
                print(board[y][x], end='␣')
        if y in [2,5]:
            print('\n---------|---------|--------')
        else:
            print()

# 動作確認
print_board()
```

```
 5  3  0 | 0  7  0 | 0  0  0
 6  0  0 | 1  9  5 | 0  0  0
 0  9  8 | 0  0  0 | 0  6  0
---------|---------|--------
 8  0  0 | 0  6  0 | 0  0  3
 4  0  0 | 8  0  3 | 0  0  1
 7  0  0 | 0  2  0 | 0  0  6
---------|---------|--------
 0  6  0 | 0  0  0 | 2  8  0
 0  0  0 | 4  1  9 | 0  0  5
 0  0  0 | 0  8  0 | 0  7  9
```

次に、引数 y と x で指定されたスロットに、第三引数で受け取った n を置く事ができるか否かを判定する関数 possible() を定義します

(1) 縦一列の中に、n と同じ数字があったら置けませんので、False を返します

(2) 横一行の中に、n と同じ数字があったら置けませんので、False を返します

(3) 3×3 の枠の中に、n と同じ数字があったら置けませんので、False を返します

上記 (1)、(2)、(3) の何れでもないなら、True を返します

5 行 5 列目 (board リストの 0 行目や 0 列目から数え始めるので、引数は x=y=4) の空きスロットに値を指定して、possible() の動作を確認しましょう

ソースコード 6.2　possible

```
1  def possible(y,x,n):
2      global board
3      for i in range(0,9):
4          if board[y][i] == n:
5              return False
6      for i in range(0,9):
7          if board[i][x] == n:
8              return False
9      x0 = (x//3)*3
10     y0 = (y//3)*3
11     for i in range(0,3):
12         for j in range(0,3):
13             if board[y0+i][x0+j] == n:
14                 return False
15     return True
16
17 # 動作確認
18 print( possible(4,4,4) )
19 print( possible(4,4,5) )
```

最後に、問題を解く関数 solve() を定義します

y 行 x 列目のスロットに着目して、そこが 0 ならば空きスロットですから、そのスロットに、1 から 10 までの数字を順に指定して、possible() を呼びだしてはチェックしていきます

もし、possible() 関数が True を返したら、それは一つの候補ですので、引き続き solve() を繰り返して空きスロットを埋めていきます

solve() が return で None を返したときは、選ばれた候補は使えなかったという事ですので、盤面のスロットを空（0）に戻しています

全ての空欄が埋まったなら、print_board() を呼んで結果（答え）を印刷しています

ソースコード 6.3　solve

```
def solve():
    global board
    for y in range(9):
        for x in range(9):
            if board[y][x] == 0:
                for n in range(1,10):
                    if possible(y,x,n):
                        board[y][x] = n
                        solve()
                        board[y][x] = 0
                return
    print_board()

# 動作確認
solve()
```

solve() 関数の中から、自分自身である solve() 関数を呼び出しています。再帰呼び出しです。再帰呼び出しを使うことによって、プログラムを短く記述できています

実行結果

```
 5  3  4 | 6  7  8 | 9  1  2
 6  7  2 | 1  9  5 | 3  4  8
 1  9  8 | 3  4  2 | 5  6  7
---------|---------|--------
 8  5  9 | 7  6  1 | 4  2  3
 4  2  6 | 8  5  3 | 7  9  1
 7  1  3 | 9  2  4 | 8  5  6
---------|---------|--------
 9  6  1 | 5  3  7 | 2  8  4
 2  8  7 | 4  1  9 | 6  3  5
 3  4  5 | 2  8  6 | 1  7  9
```

プログラムで解けてしまうとなると、しかも「こんなに短いプログラムで」解いたとなると、数独そのものはつまらないものになりますね。また「1 から 9 まで総当たりで試しながら」＝「コンピュータが力ずくで」なので、芸がないという気がします

このゲームは、まずは鉛筆と消しゴムを使って、自分で解いてみることが楽しむための重要なポイントのようです

【演習】

6.2　三目並べ（Tic-Tac-Toe）

プログラムを実行すると盤面が表示され、×の石を置く場所を指定するよう促されます

既に石の置かれているスロット番号は指定できませんし、スロット番号として 0 〜 8 の数値を指定しなければなりません

キーボードから番号を入力すると、その番号のスロットに×の石が置かれた盤面が表示され、次の手番のコンピュータは、乱数で決めた場所に○を置いてきます

手番を交互に変えながらゲームは進み、縦、横、斜めの何れかに、先に一列に自分の石を並べた方が勝ちとなります

```
スタート！ [Tic Tac Toe]
/---|---|---\
| 0 | 1 | 2 |
|---|---|---|
| 3 | 4 | 5 |
|---|---|---|
| 6 | 7 | 8 |
\---|---|---/
あなたの手番です。スロット番号は？ : 4
/---|---|---\
| 0 | 1 | 2 |
|---|---|---|
| 3 | X | 5 |
|---|---|---|
| 6 | 7 | 8 |
\---|---|---/
/---|---|---\
| 0 | 1 | O |
|---|---|---|
| 3 | X | 5 |
|---|---|---|
| 6 | 7 | 8 |
\---|---|---/
あなたの手番です。スロット番号は？ :
    •
    •
    •
```

三目並べの盤面 board をリストで用意し、それを画面に表示する関数 print_board() を定義します

空きスロットは 0（スペース）、人が置いた石は 1（X）、コンピュータが置いた石は 2（O）を、board に代入することにします

ソースコード 6.4　board

```
1 board = [0,0,0,\
2          0,0,0,\
3          0,0,0]
4
5 def print_board():
6     global board
```

```
7     bd = ['', '', '',\
8           '', '', '',\
9           '', '', '']
10    for val in board:
11        if val==0:
12            bd[n] = '␣'
13        elif val==1:
14            bd[n] = 'X'
15        elif val==2:
16            bd[n] = 'O'
17        n += 1
18    print( '/---|---|---\\')
19    print( '|␣{}␣|␣{}␣|␣{}␣|'.format(bd[0], bd[1], bd[2]) )
20    print( '|---|---|---|')
21    print(f'|␣{bd[3]}␣|␣{bd[4]}␣|␣{bd[5]}␣|')
22    print( '|---|---|---|')
23    print(f'|␣{bd[6]}␣|␣{bd[7]}␣|␣{bd[8]}␣|')
24    print( '\\---|---|---/')
25
26 # 動作確認
27 print_board()
```

コンピュータは乱数でスロット番号を選び、そこが空きスロットなら 2 を board に代入します

ソースコード 6.5　machine$_put$

```
1 from random import randint
2 def machine_put():
3     global board
4     while True:
5         n = randint(0:8)
6         if board[n]==0:
7             board[n] = 2
8             break
9
10 # 動作確認
11 machine_put()
12 board_print()
```

人の手番では入力を促すメッセージを表示し、（整数の入力を期待していますが）キーボードから入力された整数が空きのスロット番号として有効なら、そこに 1 を代入します

ソースコード 6.6　human$_put$

```
1 def human_put():
2     global board
3     while True:
4         instr = input("あなたの手番です。スロット番号は？␣:␣")
5         n = int( instr )
6         if 0<=n<9 and board[n] == 0:
```

```
7             board[n] = 1
8             break
9
10 # 動作確認
11 human_put()
12 print_board()
```

縦、横、斜めに一致したなら、勝敗がついたという意味で True を返し、そうでなければ、ゲーム継続ですので False を返します

ソースコード 6.7　judge

```
1  def judge():
2      global board
3      if board[0]==board[1]==board[2] or\
4          board[3]==board[4]==board[5] or\
5          board[6]==board[7]==board[8] or\
6          board[0]==board[3]==board[6] or\
7          board[1]==board[4]==board[7] or\
8          board[2]==board[5]==board[8] or\
9          board[0]==board[4]==board[8] or\
10         board[2]==board[4]==board[6]:
11         return True
12     return False
```

ゲーム全体の流れは、start_game() 関数に定義します

ソースコード 6.8　game

```
1  def start_game():
2      print("スタート！␣[Tic␣Tac␣Toe]")
3      machine_turn = False
4      while True: # 以下をゲームオーバーまで繰り返す
5          board_print() # 盤面を表示して
6          if machine_turn: # machine の手番なら
7              machine_put()
8          else: # human の手番なら
9              human_put()
10         if judge(): # 勝敗の判定
11             break
12         machine_turn = not machine_turn # 手番を交代する
13
14 # 動作確認
15 start_game()
```

【演習】

6.3　ライフゲーム

http://vigne-cla.com/17-1/

フィールドは正方形のマス目で、各マスの状態が変化するためのルールは次の２つだけです。

現在の状態 隣接する８セルの条件 次の状態生存（１） 生存セルが０～１または４～８個 死（０）死（０） 生存セルが３個 生存（１）これに当てはまらない場合は現状維持となります。

別の表現として、生きるための条件に着目して書くとこうなります。

次のステップでの生存条件生存セル 隣接する８セルのうち生存セルが２、３個死セル 隣接する８セルのうち生存セルが３個

ソースコード 6.9　life00

```
import numpy as np
import numpy.random as nr
import matplotlib.pyplot as plt

#============================
#初期状態の設定（グライダー）
#============================

#高さ、幅
h, w = 10, 10

#任意の状態を用意
state = np.array([[0,0,1],
                  [1,0,1],
                  [0,1,1]])

#フィールドのどこに置くか（左上点を指定）
p = (0, 0)

#終了ステップ数
max_step = 35

#============================
#メイン処理
#============================

#フィールドの生成
f = np.zeros((h, w), dtype=bool)

#任意の状態を置く
f[p[0]:p[0]+len(state), p[1]:p[1]+len(state[0])] = state

#初期状態の表示
#plt.figure(figsize=(10, 10))
plt.imshow(f, cmap='inferno')
#plt.savefig('save/{}.png'.format(0), bbox_inches='tight', pad_inches=0)
plt.show(), print()

#状態の更新
```

```
for i in range(1, max_step + 1):

    #周囲の生存マス数を記録するための配列
    mask = np.zeros((h, w))

    #周囲の生存マスを足し込む
    mask[1:, :] += f[:-1, :] #上
    mask[:-1, :] += f[1:, :] #下
    mask[:, 1:] += f[:, :-1] #左
    mask[:, :-1] += f[:, 1:] #右
    mask[1:, 1:] += f[:-1, :-1] #左上
    mask[1:, :-1] += f[:-1, 1:] #右上
    mask[:-1, 1:] += f[1:, :-1] #左下
    mask[:-1, :-1] += f[1:, 1:] #右下

    #未来のフィールド（すべて死状態）
    future = np.zeros((h, w), dtype=bool)

    #生きているマスが生きる条件（＝生存）
    future[mask*f==2] = 1
    future[mask*f==3] = 1
    #死んでいるマスが生きる条件（＝誕生）
    future[mask*~f==3] = 1

    #フィールドの更新（浅いコピーに注意）
    f = future

    #表示
    #plt.figure(figsize=(10, 10))
    plt.imshow(f, cmap='inferno')
    #plt.savefig('save/{}.png'.format(i), bbox_inches='tight', pad_inches=0)
    plt.show(), print()
```

ソースコード 6.10　life01

```
#============================
#初期状態の設定（銀河）
#============================

#高さ、幅
h, w = 15, 15

#任意の状態を用意
state = np.array([[1,1,0,1,1,1,1,1,1],
                  [1,1,0,1,1,1,1,1,1],
                  [1,1,0,0,0,0,0,0,0],
                  [1,1,0,0,0,0,0,1,1],
                  [1,1,0,0,0,0,0,1,1],
```

```
14                  [1,1,0,0,0,0,0,1,1],
15                  [0,0,0,0,0,0,0,1,1],
16                  [1,1,1,1,1,1,0,1,1],
17                  [1,1,1,1,1,1,0,1,1]])
18
19 #フィールドのどこに置くか（左上点を指定）
20 p = (3, 3)
21
22 #終了ステップ数
23 max_step = 15
```

ソースコード 6.11　life02

```
1  #============================
2  #初期状態の設定（ダイハード）
3  #============================
4
5  #高さ、幅
6  h, w = 25, 25
7
8  #任意の状態を用意
9  state = np.array([[0,0,0,0,0,0,1,0],
10                  [1,1,0,0,0,0,0,0],
11                  [0,1,0,0,0,1,1,1]])
12
13 #フィールドのどこに置くか（左上点を指定）
14 p = (5, 9)
15
16 #終了ステップ数
17 max_step = 135
```

ソースコード 6.12　life03

```
1  #============================
2  #初期状態の設定（十字スタート）
3  #============================
4
5  #高さ、幅
6  h, w = 401, 401
7
8  #任意の状態を用意
9  state = np.zeros((h, w))
10 state[200, :] = 1
11 state[:, 200] = 1
12
13 #フィールドのどこに置くか（左上点を指定）
14 p = (0, 0)
15
```

```
16 #終了ステップ数
17 max_step = 135
```

第 7 章

おわりに

謝辞

参考文献

Brett Slatkin 著、黒川利明 訳（株式会社オライリー・ジャパン）「Effective Python 第 2 版ー Python プログラムを改良する 90 項目」