第P章 プログラミング言語 Python

Python は 1990 年に Guido van Rossum により発表された、マルチパラダイム（手続き型 + ＿＿＿＿＿＿＿＿ + ＿＿＿＿）・動的型付けのプログラミング言語である。ライブラリー（前もって用意されている関数など）が豊富で、Web アプリケーション のほか、＿＿＿＿＿＿ などデータサイエンス分野で広く用いられるため、 2010 年代後半から急速に人気が高まってきている。

P.1 Python の実行

対話的な処理系は ＿＿＿＿＿＿ というコマンドで起動できる。対話的な処理系では、プロンプト（通常、>>>）のあとに式を入力すれば、その値を出力する。

```
>>> 1 + 1
2
```

＿＿＿＿＿＿ または ＿＿＿＿＿＿ と入力すれば対話的な処理系を終了する。

また、Python プログラムのファイルを作成して、`python ファイル名` というコマンドで直接実行することもできる。例えば、factorial.py というファイルを次のような内容で作成すると、

ファイル名 factorial.py

```
1 def fact(n):
2     if n == 0:
3         return 1
4     else:
5         return n * fact(n - 1)
6
7 if __name__ == '__main__':
8     print(fact(20))
```

シェルから、次のように実行できる。

```
> python factorial.py
```

（「>」はシェルのプロンプト）

> `__name__` という特別な変数は、モジュール（≒ファイル）が直接実行される場合は `'__main__'` という値を持ち、 インポート（後述）された場合はモジュールの名前を持つ。 だから、`if __name__ == '__main__':` のあとに、モジュールが直接実行された場合のみに実行するコードを記述することができる。

Python を実行する環境として、IDLE（Python をインストールしていれば `idle` というコマンドで起動する）, PyCharm, Spyder などいくつかの統合開発環境 (Integrated Development Environment, IDE) や Jupyter Notebook というアプリケーションもよく使われるが、ここでは説明を割愛する。

P.2 代入と関数定義

変数と代入

変数は、次のように記号「 = 」の左辺に書き、右辺の値を代入する。（変数の宣言は特になく、初めて使う変数に代入したときに変数が用意される。このため、綴りの間違いには気を付ける必要がある。）

```
>>> x = 2
>>> x * 3
6
```

⚡ Python の変数名には英字（大文字、小文字）、アンダースコア「_」と先頭以外では数字（「0」～「9」）を用いることができる。その他にもかなりの Unicode の文字を用いることができるが、ここでは詳細は割愛する。

また Python のキーワードには以下のようなものがある。これらと全く同一の変数名を使用することはできない。

False	await	else	import	pass
None	break	except	in	raise
True	class	finally	is	return
and	continue	for	lambda	try
as	def	from	nonlocal	while
assert	del	global	not	with
async	elif	if	or	yield

関数定義

関数はキーワード により定義することができる。

```
>>> def fact(n):
...     if n == 0:
...         return 1
...     else:
...         return n * fact(n - 1)
...
>>> fact(10)
3628800
```

入力が完了していないと判断すると Python 処理系は、上のように第２プロンプト「...」を表示して入力の継続を促す。

関数定義は、キーワード `def` のあとに関数名（この例では `fact`）、丸括弧「(」～「)」の中の仮引数のコンマ区切りの並び（この例では `n` のみ）、コロン「:」で始まる。改行後に関数の本体を記述する。

関数の本体はインデント（字下げ）する。つまり `def` の位置よりも数文字（この例では４文字）分下げる。インデントしている限り関数の本体が続いていると見なされる。

このように Python はインデンテーションが文法上＿＿＿＿＿言語である。（一方、C や Java などの言語ではどのようにインデントしてもプログラムの意味は＿＿＿＿＿。）

> ⚡ 慣習として、変数名や関数名は英字はすべて小文字を使い、また複数の単語からなる場合は、アンダースコア「_」で区切る。つまり `myFavoriteWord` **ではなく** `my_favorite_word` のような変数名・関数名を使う。

`return` の意味は（C 言語などと同じで）あとに書かれた式を評価し、それを戻り値として関数呼出しから抜け出す。あとに式がなければ `None` という値が戻り値になる。

キーワード引数

Python では実引数を渡すときに、位置によって仮引数と対応させるだけではなく、仮引数の名前を指定して実引数を渡すこともできる。これをキーワード引数と言う。例えば次のように定義された関数 `foo` があるとする。

```
def foo(x, y, z):
    return 2 * x + 3 * y + 5 * z
```

この関数を呼び出すときに、`foo(1, 2, 3)` のように位置で指定することもでき、 `x` に 1, `y` に 2, `z` に 3 が代入されて、 `foo` が実行され戻り値が 23 になるが、`foo(z=4, x=5, y=6)` のように仮引数の名前を指定して呼び出すこともできる。このときは当然 `x` に 5, `y` に 6, `z` に 4 が代入されて、`myfunc` が実行され、戻り値が 48 になる。

デフォルト引数

Python は関数を定義するときに仮引数にデフォルトの値を与えることができる。これをデフォルト引数と言う。例えば次のように定義された関数 `bar` があるとする。

```
def bar(x, y=2, z=3):
    return 7 * x + 11 * y + 13 * z
```

この関数を呼び出すときに、`bar(1, 1, 1)` とすると、`x` に 1, `y` に 1, `z` に 1 が代入されて、`bar` が実行され戻り値が 31 になる。一方 `bar(3)` と呼び出すと、`x` に 1, `y` に 2, `z` に 3 が代入されて、`bar` が実行され戻り値が 68 になる。

インポート文

通常は、ファイルに関数などの定義を記述して、`import` 文で読み込む。また Python のソースファイルには ＿＿ という拡張子をつけるのが通例である。

ファイル名 factorial.py

```python
def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n - 1)
```

```
>>> from factorial import *
>>> fact(20)
2432902008176640000
```

この `from ～ import *` という形式の import 文は、～というモジュール（ファイル名から拡張子 `.py` を除いたもの）から変数や関数などの定義を読み込むことを意味する。

一方、`import ～` という形の import 文もある。これは～というモジュールを読み込むが、この形の import 文で読み込んだ関数や変数を使うときは、モジュール名+「.」を接頭辞として付ける必要がある。例えば、次のように書く。

```python
import factorial
print(factorial.fact(5))
```

このように書くと、同じ名前の関数や変数が他のモジュールにあっても区別できる。

このほかに `as` というキーワードを使って、モジュール名や関数名に別名をつけるやり方もあるが、実例が出てきたときに詳しく説明する。

P.3 数と文字列

数値リテラル

整数や浮動小数点数のリテラルは他の言語と大きな違いはない。また、接尾辞 `j` を整数や浮動小数点数リテラルにつけることで＿＿を表すことができる。

```
>>> (1 + 1j) / (1 - 2j)
(-0.2+0.6j)
```

文字列リテラル

文字列型は Python ではテキストシーケンス型と呼ばれる。文字列リテラルは、二重引用符「"」 または一重引用符「'」 で囲まれた文字列である。一方 Python に文字リテラルは存在しない、つまり一文字のみの定数を特別扱いすることはないので、一重引用符／二重引用符どちらを使っても意味は変わらない。

```
>>> "hello"
'hello'
>>> 'world'
'world'
```

また三連続の二重引用符「＿＿」または一重引用符「＿＿」で囲むと改行や一重引用符・二重引用符を含む文字列を表すことができる。

```
>>> """Mike said "Hello!"
... and Anne said "Good Bye!"
... """
'Mike said "Hello!"\nand Anne said "Good Bye!"\n'
```

引用符の前に接頭辞 `_` または `F` をつけると、波括弧「`{`」～「`}`」に囲まれた部分が式として評価され、文字列中に埋め込まれる。これをフォーマット済文字列リテラル、あるいは **f 文字列**という。

```
>>> x = 13
>>> f"The factorial of {x} is {fact(x)}."
'The factorial of 13 is 6227020800.'
```

問 P.3.1 では、f 文字列のなかに波括弧（「`{`」または「`}`」）を含めたいときはどうすれば良いか調べよ。

文字列は「`+`」演算子で連接することができ、「`*`」演算子で自身を繰り返し連接することができる。

```
>>> "hello, " + 'world'
'hello, world'
>>> "hey!" * 3
'hey!hey!hey!'
```

`str` 関数

`str` という関数を使うと他のデータ型の値を文字列に変換することができる。例えば `str(123)` は文字列 `'123'` を返す。また、 `str(3.1415)` は文字列 `'3.1415'` を返す。

`[]` による文字の取出し

文字列から角括弧 `[` ～ `]` で個々の文字を取り出すことができる。例えば、次のように書く。

```
>>> s = "Takamatsu"
>>> s[0]
'T'
>>> s[4]
'm'
```

このように、文字列の各文字は添字 (index) で指定することができる。添字は 0 から始まる位置を表す整数である。0 から始まることを強調するためにオフセット (offset) と言うこともある。また、添字は負の値を指定することもできる。負の添字は、文字列の末尾からの位置を表す。例えば、次のように書く。

```
>>> s[-1]
'u'
>>> s[-3]
't'
```

なお、Python には一文字のためのデータ型（C や Java の `char` のような型）は存在しないので、文字列の一文字も要素数が 1 の文字列型である。（文字コードではない。）

スライス

角括弧のなかにコロン「:」を使って範囲を指定すると、文字列の一部分を取り出すことができる。これを**スライス**と呼ぶ。例えば、次のように書く。

```
>>> s = "Takamatsu"
>>> s[1:4]
'aka'
>>> s[4:7]
'mat'
>>> s[:4]
'Taka'
>>> s[5:]
'atsu'
```

`1:4` のように書くと、オフセット 1 から 3 までの文字を取り出す。 つまり、左端の添字は含まれ、右端の添字は含まれない。コロンの左側、右側を省略すると、 それぞれ文字列の先頭から、末尾まで、を表す。また両側を省略して `s[:]` と書くと、元の文字列と同じ内容だが、元の文字列のコピーを返す。

さまざまな文字列の関数

Python には文字列に対する様々な関数が組み込みで用意されている。例えば次のような関数はよく使われる。

関数	説明
`len(s)`	文字列 `s` の長さ（文字数）を返す。
`s.split(sep)`	文字列 `s` を区切り文字 `sep` で分割し、リストを返す。
`s.startswith(pre)`	文字列 `s` が指定した接頭辞 *`pre`* で始まるかどうか、真偽値を返す。
`s.endswith(suf)`	文字列 `s` が指定した接尾辞 *`suf`* で終わるかどうか、真偽値を返す。
`s.find(sub)`	文字列 `s` の中に部分文字列 *`sub`* が最初に現れるオフセットを返す。 見つからない場合は -1 を返す。
`s.replace(old, new)`	文字列 `s` の中の部分文字列 *`old`* を *`new`* に置き換えた新しい文字列を返す。

この他の標準的な文字列に関する関数は、 Python 標準ライブラリーのドキュメントの組み込み型を探せば見つけることができる。

P.4 演算子と組み込み関数

算術演算子

演算子は C や Java の演算子と似ているが、「`/`」は整数同士の演算でも通常の（小数になる可能性のある）除算である。整数としての除算の商を求めるには「`//`」を用いる。余りを求める演算子は「`%`」である。

```
>>> 1 / 3
0.3333333333333333
>>> 17 // 6
```

```
2
>>> 17 % 6
5
```

「**」は　　を求める演算子である。

```
>>> 2 ** 10
1024
>>> 2 ** 0.5
1.4142135623730951
```

なお、Python にも += のような複合代入演算子はある。例えば x += 10 は、x = x + 10 と同じ意味である。ただし、++、-- などはない。

print 関数と input 関数

画面に出力するには print 関数を用いる。いくつかの引数をコンマで区切って与えるとそれらを順に出力する。

```
>>> y = 23
>>> print(x, "+", y, "=", x + y)
13 + 23 = 36
```

最後に、sep=～ とキーワード引数を指定すると、区切り文字を～に変えることができる。（区切り文字を何も指定しないと上の例のように空白文字が区切りに使われる。）

```
>>> print(x, "+", y, "=", x + y, sep=',')
13,+,23,=,36
```

一方、input 関数はキーボードから文字列を読みこむ関数である。　　関数（整数への変換）や　　関数（浮動小数点数への変換）と組み合わせることで、文字列をそれぞれ整数や浮動小数点数に変換することができる。

ファイル名 temp.py

```
1 x   = float(input('x を入力してください: '))
2 y   = float(input('y を入力してください: '))
3 z   = int(input('z を入力してください: '))
4 age = int(input('年齢を入力してください: '))
```

なお int 関数を浮動小数点数に適用すると切り捨てになる。

```
>>> int(1.5)
1
>>> int(-2.4)
-2
```

また round という関数は四捨五入に近い動作をするが、厳密に言うと、偶数丸め（銀行家の丸め、bankers' rounding）という動作になる。つまり、.5 は最も近い偶数に丸める。

```
>>> round(3.14)
3
>>> round(4.9)
```

```
5
>>> round(4.5)
4
>>> round(7.5)
8
```

（そんなことが実際あるかどうかはさておき）本当の四捨五入が必要な場合は Python の標準ライブラリーの `decimal` モジュールを使えば可能である（が、ちょっと面倒である）。

P.5 制御構造

if 文

条件分岐を表す if 文は `if` というキーワードのあと、条件式、コロン「:」で始まり、改行後に条件が成り立つときに実行する文を並べて書く。

```
1 if x < 0:
2     print('x は負の数です。')
3     print('正の数や 0 ではありません。')
```

条件式には「<」「<=」「>=」「>」「==」「!=」などの比較演算子を用いることができる。条件が成り立つときに実行する文はインデント（字下げ）する。つまり `if` の位置よりも数文字（この例では 4 文字）分開始を下げる。ここに複数の文を並べることもできる。同じ字下げ幅である限りは、条件が成り立つときには実行する。

条件が成り立たないときに実行する文は、キーワード `else`、コロン「:」のあとに改行して書く。

```
1 if y < 0:
2     print('y は負の数です。')
3     print('正の数や 0 ではありません。')
4 else:
5     print('y は正の数または 0 です。')
6     print('負の数ではありません。')
```

また、`if` と `else` の間に `elif` というキーワード、条件式、コロン「:」で始まり、改行後にインデントした文の並び、というかたちをはさむことができる。この場合、上から順に条件式を評価し、成り立つときに対応する文の並びを実行する。

```
1 if z <= 0:
2     print('z は負の数か 0 です。')
3 elif z < 10:
4     print('z は一桁の正の数です。')
5 elif z < 100:
6     print('z は二桁の正の数です。')
7 else:
8     print('z は三桁以上の正の数です。')
```

論理演算子

条件式は `and`（～かつ～）、`or`（～または～）、`not`（～でない）などの論理演算子で組み合わせることができる。

```
if 13 <= age and age <= 19:
    print('あなたはティーンエイジャーです。')
```

Python では（C や Java と異なり）比較演算子は連鎖させることができるので、この例は次のように書くこともできる。

```
# x < y < z は x < y and y < z と同じ。
# ただし、前者では y は一度しか評価されない。
if 13 <= age <= 19:
    print('あなたはティーンエイジャーです。')
```

コメント

上の例で使われているように、「`#`」から行末まではコメントである。

なお、関数定義の中で、最初に説明のために文字列リテラルを記述することがある。実行結果には影響しないのでコメントのようなものであるが、正確には関数のドキュメンテーション文字列 (docstring) と呼ばれる。

```
def fact(n):
    """n の階乗を求める関数。
    n は 0 以上である必要がある。
    """
    if n == 0:
        return 1
    else:
        return n * fact(n - 1)
```

for 文

決まった回数の繰り返しを実現するときには for 文が使われる。次のようにキーワード `for`、変数、キーワード `in`、式、コロン「`:`」という形式で使われる。このあとに改行して、繰り返し本体をインデントして書く。

ファイル名 temp2.py

```
for i in range(5):
    print('Hello')
    print('  ', i, '回目')
```

ここで `range` 関数は、`range(n)` が、0 から n - 1 までの整数の列を返すような関数である。

これを実行すると、変数 `i` に 0, 1, ..., 4 が順に代入され、次のように出力される。

```
Hello
   0 回目
Hello
   1 回目
Hello
```

```
   2 回目
Hello
   3 回目
Hello
   4 回目
```

他に range 関数は次のようなカタチでも使われる。

range(n)	$0, 1, ..., n-1$ を返す
range(m, n)	_____ を返す
range(m, n, s)	_____ を返す

ただし、k は $m + k \cdot s$ が n 未満となるような最大の k である。

Q P.5.1 次のプログラムの断片はそれぞれどう出力するか？

1.
```
for i in range(3):
    print(i, end=', ')
```
_____

2.
```
for i in range(3, 6):
    print(i, end=', ')
```
_____

3.
```
for i in range(2, 8, 3):
    print(i, end=', ')
```
_____

なお range 関数の返す値は後述のリストではないが、list 関数でリストに変換することができる。

```
>>> range(2, 5)
range(2, 5)
>>> list(range(2, 5))
[2, 3, 4]
```

while 文

繰り返しを表す while 文は while というキーワードのあと、条件式、コロン「:」で始まり、改行後に繰り返す文をインデントして並べて書く。

ファイル名 while.py

```
1 n = 1000
2 while n > 0:
3     print(n, end=', ')
4     n = n // 2    # n //= 2 と書いてもよい
5 print()          # 改行のみ出力する
```

ここで print 関数に、end=～ とキーワード引数を指定すると最後に出力する文字を変えることができる。（何も指定しないと改行文字が最後に出力され

る。）

これを実行すると、次のように出力される。

```
1000, 500, 250, 125, 62, 31, 15, 7, 3, 1,
```

なお、C や Java の do～while 文に相当する（ループ本体を条件式より先に実行する）構文は Python にはない。

continue 文と break 文

Python にも continue 文と break 文がある。意味は C や Java のそれと同じである。つまり、 while 文や for 文の中で、＿＿＿＿文を使うと、その文の後の処理を飛ばして、次の繰り返しの先頭に戻る。また、＿＿＿文を使うと、繰り返しを中断して繰り返しの外に出る。

while ～ else 文、 for ～ else 文

Python では、while 文や for 文で、繰り返しが正常に終了した場合（つまり break 文で中断しなかった場合）ループのあとにに実行される文を else 節として書くことができる。

ファイル名 forelse.py

```
for i in range(3):
    print(i, end=', ')
    if (input('コンティニュー？ (y/n) ') != 'y'):
        break
else:
    print('繰り返しが正常に終了しました。')
```

この例では、コンティニュー？の入力に対して y 以外を入力すると、繰り返しが中断され、その場合は else 節の内容である「繰り返しが正常に終了しました。」の出力はされない。

try 文

Python には例外処理を行うための try 文がある。**例外** (exception) というのは、初期化されていない変数を使った、0 で割り算した、存在しないファイルをオープンしようとした、などの、正常でない状況である。例外が発生すると何もしなければプログラムが終了してしまうが、try 文を使うことで例外が発生しても、プログラムを終了させずに処理を続けることができる。

try 文は次の例のように使われる。

ファイル名 tryexample.py

```
arr = [98, 99]
for i in range(-2, 3):
    try:
        print(1 / i, end=', ')
        print(arr[i], end=', ')
    except ZeroDivisionError:
        print('ゼロ除算エラーが発生 ', end='')
```

```
    except IndexError:
        print('インデックスエラーが発生 ', end='')
    else:
        print("... ", end='')
    finally:
        print("end")
```

一般的には try というキーワードのあとのコロン「:」で始まり、改行後に例外が発生するかもしれない文をインデントして並べて書く。そのあと except というキーワードを書き、例外の種類のあとに、コロン「:」の行のあとに改行して、例外が発生したときの処理をインデントして並べて書く。この except 節は複数個書くこともでき、その場合は、発生した例外の種類に対応する except 節が実行される。

またオプショナルの else: を使うと、例外が発生しなかった場合に実行する文をインデントして並べて書くことができる。 最後にやはりオプショナルの finally: を使うと、 例外が発生してもしなくても最後に実行する文をインデントして並べて書くことができる。

結局、上のプログラムは次のように出力する。

```
-0.5, 98, ... end
-1.0, 99, ... end
ゼロ除算エラーが発生 end
1.0, 99, ... end
0.5, インデックスエラーが発生 end
```

よく使われる例外の種類としては次のようなものがある。

例外の種類	説明
ZeroDivisionError	0 で割り算をしたとき
IndexError	リストや文字列のインデックスが範囲外の とき
FileNotFoundError	存在しないファイルをオープンしようとしたとき
ValueError	int などの関数に不正な値を渡したとき
NameError	存在しない変数を参照したとき

例外を発生させるには、raise 文を使うが、ここでは説明を割愛する。

P.6 リスト

リストは単純だが有用性の高いデータ型で、関数型言語などでも多用されるデータ型である。配列と同様に（同種の）データを順序付けて集めたものだが、要素の追加・削除が可能である。ただし、配列のように各要素に等時間で高速にアクセスすることはできない。

リストリテラル

リストを構成するためには、＿＿＿＿（[～]）で囲み、各要素をコンマ「,」で区切って並べる。例えば、 [] は空リストを表し、[2, 3, 5] は 3 つの要素からなるリストを表す。

また、リストは「+」演算子や「+=」演算子で連接することができる。

```
>>> w = [1, 3] + [0]
>>> w += [6, 2, 3]
>>> w
[1, 3, 0, 6, 2, 3]
```

「*」演算子や「*=」で、自身を繰り返し連接することもできる。

```
>>> v = [0]
>>> v * 5
[0, 0, 0, 0, 0]
>>> u = [1, 2, 3]
>>> u *= 3
>>> u
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

リストに関する関数

`list` という関数はいろいろなものをリストに変換することができる。 `range` 関数の結果をリストに変換できることは既に述べた。そのほか、文字列やタプル（後述）などもリストに変換することができる。例えば、`list('abc')` は `['a', 'b', 'c']` というリストになる。

リストに関しても文字列と同様に、さまざまな操作を行うためのメソッドが用意されている。以下はそのごく一部である。

メソッド	説明
`len(list)`	リスト `list` の要素数を返す
`list.append(`*item*`)`	リスト `list` の末尾に *item* を追加する
`list.sort()`	リスト `list` の要素を昇順にソートする
`list.pop()`	リスト `list` の末尾の要素を削除し、その値を返す

実は n 番めの要素 `list[`*n*`]` やスライス `list[2:5]` などの記法も文字列と共通である。

リスト内包表記

リスト内包表記 (list comprehension) は数学で使われる集合の表記に似た糖衣構文 (syntax sugar) である。

```
>>> [x * y for x in range(1, 5) for y in range(5, 8)]
[5, 6, 7, 10, 12, 14, 15, 18, 21, 20, 24, 28]
>>> [x * x for x in range(1,11) if x % 2 == 1]
[1, 9, 25, 49, 81]
```

リスト内包表記は、角括弧（`[`～`]`）のなかに最初に式を一つ書き、そのあとに、「`for` 変数 `in` 式」というカタチか「`if` 式」というカタチを並べたものである。（ただし並びの最初は「`for` ～」のカタチでなければいけない。）その値は「`for` 変数 `in` 式」というカタチで与えられた繰り返しの中で「`if` 式」というカタチで与えられた条件が成り立つときの、角括弧のあとの最初の式の値を順に並べたものになる。

Q P.6.1 次のリスト内包表記の値は何か?

① `[x * y for x in [1,2] for y in [3,5,7]]`

P.7 タプル

タプル（tuple, 組）は要素を「,」（コンマ）で区切って並べ、丸括弧「(」と「)」で囲んで表す。（文脈によっては丸括弧がなくても良い場合がある。）リストは通常、各要素は同種のものからなるが、タプルは要素の種類が同一である必要はない。

タプルは以下の例のように関数の戻り値に使うこともできるし、代入文の左辺に書くこともできる。

ファイル名 temp3.py

```
1  def sort2(m, n):
2      if m > n:
3          return (m, n)
4      else:
5          return (n, m)
6
7  i1 = int(input('整数 1 を入力してください: '))
8  i2 = int(input('整数 2 を入力してください: '))
9
10 (j1, j2) = sort2(i1, i2)
11 print('大きいほうは', j1, '小さいほうは', j2, 'です。')
```

特に次のように書くと左辺の 2 つの変数への代入が同時に行われるので、2 つの変数の内容を入れ替えることができる。

```
y, x = x, y
```

（この例ではタプルのまわりの丸括弧が省略されている。）

Q P.7.1 次のリスト内包表記の値は何か?

① `[(x, y) for x in [1,4,7] for y in [2,5,8] if x < y]`

問 P.7.2 3 つの実数 a, b, c を受け取り、二次方程式 $ax^2 + bx + c = 0$ の 2 つの解を組として返す関数 `quadratic` を定義せよ。

標準ライブラリー関数の `zip` は 2 つ以上のイテラブルの同じ位置の要素をタプルにしたもののイテラブルを返す関数である。イテラブルの長さが異なる場合は、短いほうにあわせる。（イテラブル (iterable) は、いくつかの要素を列挙したデータと考えておけば良い。リストもイテラブルの一種である。）

```
>>> list(zip([1,3,5,7,11], [2,4,6,10]))
[(1, 2), (3, 4), (5, 6), (7, 10)]
```

可変長引数

Python では、関数の引数を可変長にすることができる。具体的には、関数を定義するときに最後の仮引数の前にアスタリスク「*」をつけると、この仮引数は余ったすべての引数を受け取ったタプルになる。 例えば、次のように関数の定義を開始することができる。

```
def varargs_func(x, y, *args):
    ⋮
```

この関数を例えば `varargs_func(2, 3, 5, 7, 11)` のように呼び出すと、`x` の値が 2、`y` の値が 3、`args` の値が `(5, 7, 11)` となる。

また、最後の仮引数の前にダブルアスタリスク「**」をつける形もあり、これは余ったキーワード引数を、後述の辞書型としてまとめて受け取ることができる。詳しい説明は割愛する。

P.8 辞書型

Python には、キーと値のペアを保持するための組み込みのデータ型として、辞書 (dictionary) 型がある。他の言語ではハッシュマップや連想配列と呼ばれるものに相当する。配列やリストは、個々の要素をアクセスするときに整数の添字（オフセット）を使うが、辞書型はより一般的なキーを使って要素にアクセスする。キーにはいろいろな型を使うことができるが、一般的には文字列や整数が使われる。

Python の辞書型は波括弧（`{`〜`}`）のなかにキーと値のペアをカンマ区切りで並べることで表現される。キーと値のペアはコロン「:」で区切る。例えば `{'x': 1, 'y': 3.2, 'foo': -2.9}` は `'x'` という文字列のキーに対して 1 という値、`'y'` に対して 3.2 という値、`'foo'` に対して -2.9 という値を関連付けた辞書を表す。

辞書に対しては、リストや文字列と同じように 角括弧 `[` 〜 `]` を使って要素にアクセスする。また、要素を削除するには `del` 文を用いる。

ファイル名 dictexample.py

```python
colors = {
    'red':    0xff0000,
    'green':  0x00ff00,
    'blue':   0x0000ff,
    'salmon': 0xfa8072,
}

print(f'0x{colors['red']:06x}')
print(f'0x{colors['green']:06x}')
print(f'0x{colors['blue']:06x}')

colors['yellow'] = 0xffff00
print(f'0x{colors['yellow']:06x}')
del colors['salmon']
print(f'0x{colors['salmon']:06x}')
# print(f'0x{colors.get('salmon', 0):06x}')
```

この例では f 文字列のなかで `{num:06x}` という形を用いているが、この「:」以降はフォーマット指定子と言って、`:06x` は値を 16 進数で表し、ゼロ埋めして 6 桁の文字列にすることを意味する。このプログラムは `0xff0000`, `0x00ff00`, `0x0000ff`, `0xffff00` と順に出力し、15 行目で `colors['salmon']` がないので KeyError という例外が発生する。キーが存在しない場合に例外を発生させずにデフォルト値を返すには、`colors.get('salmon', 0)` のように `get` というメソッドを使うことができる。`get` の第 2 引数がデフォルト値である。上のプログラム例の 15 行目の `colors['salmon']` を `colors.get('salmon', 0)` に書き換えると、`0x000000` と出力する。

P.9 ⚡ 高階関数とラムダ式

高階関数は＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿関数である。

リストに対しては、いくつかの高階関数が標準ライブラリーに用意されている。例えば、＿＿＿＿は、リストの要素に一斉に関数を適用し、その戻り値のリストを返す関数である。（正確にはイテラブルを受け取って、イテラブルを返す関数である。一般にイテラブルを表示するためには `list` 関数でリストに変換する必要がある。）

このメソッドは次のように使用することができる。

```
>>> list(map(chr, [97, 98, 99, 100]))
['a', 'b', 'c', 'd']
>>> def twice(n):
...     return 2 * n
>>> list(map(twice, [97, 98, 99]))
[194, 196, 198]
```

ここで、`chr` は文字コードに対し対応する一文字からなる文字列を返す関数である。

ところで高階関数の引数として使う `twice` のような小さな関数にいちいち名前をつけるのは面倒なので、名前をつけずに関数を表現する記法が用意されている。これを＿＿＿＿＿＿という。（この名前は、かつて数学の一分野で、この目的のためにギリシャ文字の「λ」が使われたことに由来する。）

例えば、`lambda x: 2 * x` という式で `twice` と同等の関数を表す。キーワード `lambda` とコロン「:」の間に仮引数のコンマ区切りの並びを、コロンの右側に戻り値の式を書く。次はラムダ式の使用例である。

```
>>> list(map(lambda x: x * x, [2, 3, 5]))
[4, 9, 25]
```

また、＿＿＿＿＿＿は、リストの要素の中で、与えられた関数の値を真にする要素だけのリストを返すメソッドである。

```
>>> list(filter(lambda x: x % 2 == 0, [2, 3, 5, 8]))
[2, 8]
```

P.10 ジェネレーター

通常の関数は、リターンする（戻り値を返す）と、次に実行されるときはもう一度関数の最初から実行される。しかし、Python の**ジェネレーター関数** (generator function) は前回に値を渡した地点の**続きから**実行が再開される。

ファイル名 fib.py

```
def gfib(n):
    a = 1
    b = 1
    while a < n:
        yield a
        a, b = b, a + b
```

Python のジェネレーター関数では＿＿＿＿というキーワードを使って値を渡す。yield 文は return 文と似ていて、`yield 式` という形で用いる。

ジェネレーター関数を呼び出すと、すぐに関数内部のコードが実行されるのではなく、一旦、ジェネレーターイテレーター (generator iterator) というオブジェクトが作られて返される。このジェネレーターイテレーターを＿＿＿関数に渡すと、ジェネレーター関数内部のコードが実行され、`yield` された値を返す。さらに `next` 関数を呼び出すと yield 文の＿＿＿＿＿実行が再開され、やはり、次の yield された値を返す。

```
gen = gfib(100)
print(next(gen))          # 1 を出力する
print(next(gen))          # 1 を出力する
print(next(gen))          # 2 を出力する
print(next(gen))          # 3 を出力する
print(next(gen))          # 5 を出力する
print(next(gen))          # 8 を出力する
```

`yield 式` の代わりに、`yield from 式` というカタチを使うと、**式** に配列やリストのような イテラブルが与えられたとき、イテラブルの要素を順に `yield` することになる。 例えば、`yield from range(5)` とすると、0 から 4 までの整数を順に `yield` することになる。

ジェネレーターイテレーターは `for` 文の `in` の次の式としても使うことができる。この場合の `for` 文はジェネレーターイテレーターを受け取った `next` 関数によって返される値を順に変数に代入してループする。ジェネレーター関数の中のコードの実行が return 文によりリターンするか、関数を抜けるとループを終了する。

```
for v in gfib(20):
    print(v, end=' ')
```

この部分は「＿＿＿＿＿＿＿＿＿＿」と出力する。

ジェネレーターは必ずしも有限個の要素の生成で終わる必要はない。次の例は無限に yield する例である。

ファイル名 ifib.py

```
def ifib():
    a = 1
    b = 1
    while True:
        yield a
        a, b = b, a + b
```

次のようにすると、

```
for i, v in zip(range(15), ifib()):
    print(v, end=' ')
print()
```

この部分は「1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 」と出力する。

問 P.10.1 整数 n を引数として受け取り、最初は n を yield し、以降は、

① 直前に yield した値が偶数ならば $n/2$ を yield する、
② 直前に yield した値が奇数ならば $3n+1$ を yield する、

という処理を繰り返すジェネレーター関数 hailstone を定義せよ。

next 関数の代わりに、ジェネレーターイテレーターの send メソッドを使うと、ジェネレーター関数に値を送って、実行を再開させることができる。 send メソッドの引数は、ジェネレーター関数の中では yield 文の戻り値として受け取ることができる。

次の例ではアリコット (aliquot) 数列という数列を生成するジェネレーターを定義しているが、 send メソッドを使って、ユーザーが入力した値をジェネレーター関数に送っている。

ファイル名 aliquot.py

```
def sum_of_divisors(n):
    """n の約数の和を求める"""
    s = 0
    for i in range(1, n // 2 + 1):
        if n % i == 0:
            s += i
    return s

def aliquot_sequence(n):
    """約数の和の数列を生成するジェネレーター"""
    while True:
        received = yield n
        if received is not None: # send された場合
            n = received
        if n <= 0:
            return
        n = sum_of_divisors(n)

if __name__ == "__main__":
    seq = aliquot_sequence(12)
```

```
    print(next(seq))
    while True:
        try:
            # try 276, 552, 564, 660, 966
            n = int(input("正の整数: "))
            if n <= 0:
                break
            ret = seq.send(n) # 入力された整数を送る
        except ValueError:
            ret = next(seq)    # 入力が整数以外の場合
        print(ret)
        if ret == 0:
            break
    print("終了しました。")
```

P.11 ⚡ 例: エラトステネスの篩（ふるい）

最後に、素数列を生成するプログラムを例として挙げる。（ただし、この定義は効率面での改良の余地は大いにあると思われる。）

ファイル名 primes.py

```
def ifrom(n):
    while True:
        yield n
        n += 1

def sieve(n, xs):
    for i in xs:
        if i % n != 0:
            yield i

def primes():
    xs = ifrom(2)
    while True:
        n = next(xs)
        yield n
        xs = sieve(n, xs)
```

この `primes()` は無限に素数を生成するので、例えば `range` のように有限のものと `zip` する。

```
for i, p in zip(range(20), primes()):
    print(p, end=' ')
```

この for 文は、

```
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71
```

と 20 個の素数を出力する。