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。

ここでは C 言語と異なる部分を中心に説明する。

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` ファイル名 というかたちで直接実行することもできる。（「>」はシェルのプロンプト）

```
> python factorial.py
```

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。（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
```

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 はどのようにインデントしてもプログラムの意味は＿＿＿＿＿。）

インポート

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 factorial.py

```
1  def fact(n):
2      if n == 0:
3          return 1
4      else:
5          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。

P.3 リテラル

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。文字リテラルは存在し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 `f` 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。

```
>>> 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!'
```

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
```

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=～ と指定すると、区切り文字を～に変えることができる。このように「キーワード=式」のカタチで与える引数をキーワード引数という。（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: '))
```

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＿＿、コロン「:」のあとに改行して書く。

```
1 if y < 0:
```

```
    print('y は負の数です。')
    print('正の数や 0 ではありません。')
else:
    print('y は正の数または 0 です。')
    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。

```
if z <= 0:
    print('z は負の数か 0 です。')
elif z < 10:
    print('z は一桁の正の数です。')
elif z < 100:
    print('z は二桁の正の数です。')
else:
    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）比較演算子は連ね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。

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。

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。

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、Python に do～while 文に相当する構文はない。

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を省略できる場合がある。）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

```
def sort2(m, n):
    if m > n:
        return (m, n)
    else:
        return (n, m)

i1 = int(input('整数 1 を入力してください: '))
i2 = int(input('整数 2 を入力してください: '))

(j1, j2) = sort2(i1, i2)
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)]
```

P.8 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、`map` 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.9 ジェネレーター

Python のジェネレーター関数 (generator function) は＿＿＿＿＿＿＿(coroutine) の一種を提供する。コルーチンとは、2 つ以上のプログラムの実行単位が、＿＿＿制御を受け渡しながら実行されていく方式のことである。通常の関数（サブルーチン）はリターンする（戻り値を返す）と、次に実行するときはもう一度最初からになるが、コルーチンは次に実行するときに前回リターンした地点の続きから実行する。

ファイル名 fib.py

```
1 def gfib(n):
2     a = 1
3     b = 1
4     while a < n:
5         yield a
6         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生成する。

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) 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引数として ＿＿＿関数を呼び出すと、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
```

ジェネレーターイテレーターは for 文の in の次の式でも使うことができる。

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.9.1 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。

P.10 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。