第１章

基本的なアルゴリズム

本章では、「アルゴリズム」の定義や、各種の基礎的なアルゴリズムを学習します。

- アルゴリズムの定義
- 流れ図／フローチャート
- 決定木
- 構造化プログラミング（整構造プログラミング）
- 順次構造／選択構造／繰返し構造
- 前判定繰返しと後判定繰返し
- 繰返しの過程における条件判定
- 繰返しの中断とスキップ
- 無限ループ
- 多重ループ
- 終了条件と継続条件
- ド・モルガンの法則
- ３値の最大値
- ３値の中央値
- ２値の交換
- ２値のソート
- 約数の列挙
- 連続する整数の総和とガウスの方法

1–1 アルゴリズムとは

本節では、短く単純なプログラムを題材として、《アルゴリズム》とは何かを、その定義を含めて学習していきます。

3値の最大値

まず最初に、**“そもそもアルゴリズム**（*algorithm*）**とは何か?”**を、短く単純なプログラムを例に考えていきましょう。題材として取り上げる **List 1-1** は、**三つの値の《最大値》を求めるプログラム**です。

変数 a，b，c に代入されるのは、キーボードから読み込んだ整数値です。それら3値の最大値が、変数 *maximum* に求められて表示されます。

まずは、プログラムを実行して、動作を確認しましょう。

List 1-1 chap01/max3.py

```
# 三つの整数値を読み込んで最大値を求めて表示

print('三つの整数の最大値を求めます。')
a = int(input('整数aの値：'))
b = int(input('整数bの値：'))
c = int(input('整数cの値：'))

maximum = a                          ←1
if b > maximum: maximum = b          ←2
if c > maximum: maximum = c          ←3

print(f'最大値は{maximum}です。')
```

実行例

```
三つの整数の最大値を求めます。
整数aの値：1↵
整数bの値：3↵
整数cの値：2↵
最大値は3です。
```

変数 a，b，c の最大値を *maximum* に求めるのが、1〜3の箇所です。その手順は、次のようになっています。

1. ***maximum* に a の値を代入する。**
2. **b の値が *maximum* よりも大きければ、*maximum* に b の値を代入する。**
3. **c の値が *maximum* よりも大きければ、*maximum* に c の値を代入する。**

これら三つの文は、**順番に**実行されます。このような、一つずつ順番に処理が実行される構造は、**順次**（*concatination*）構造と呼ばれます。

＊

さて、プログラムの1は**代入文**です。Python の代入文は、**単純文**の一種です。

残る2と3は、**if 文**です。Python の `if` 文は、単純文ではなく、**複合文**の一種です。

`if` と `:` で囲まれた式（本書では**判定式**と呼びます）の評価結果に応じて、プログラム実行の流れを変更する `if` 文は、**選択**（*selection*）構造と呼ばれます。

Column 1-1　キーボードからの文字列と数値の読込み

次に示すのは、キーボードから名前を文字列として読み込んで、挨拶を表示する対話的なプログラムです（'chap01/input1.py'）。

```
# 名前を読み込んで挨拶
print('お名前は：', end='')
name = input()
print(f'こんにちは{name}さん。')
```

```
お名前は：福岡　太郎↵
こんにちは福岡　太郎さん。
```

input 関数は、キーボードから文字列を読み込んで**返却**します（**Fig.1C-1**）。読込みは改行に相当するエンターキーまでの1行分ですが、返却する文字列には、末尾の改行文字は含まれません。

実行例の場合、呼出し式 **input()** を評価すると、読み込んだ文字列型（**str** 型）の '福岡　太郎' が得られ、その文字列が変数 ***name*** に代入されます。

なお、図に示すように、**input** 関数は、実引数として文字列を与えた、**input(文字列)** の形式でも呼び出せます。この形式では、画面に「文字列」が表示され（このとき改行文字は出力されません）、それから文字列の読込みが行われます。

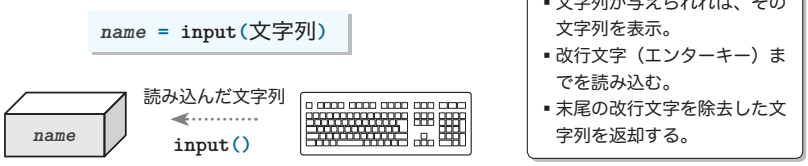


Fig.1C-1　キーボードからの読込み

そのため、網かけ部の2行は、以下の1行にまとめられます（'chap01/input2.py'）。

```
name = input('お名前は：')
```

さて、**List 1-1** で必要なのは、文字列ではなくて整数です。**input** 関数が返却する文字列を整数に変換する（たとえば、**str** 型の文字列 '3' を **int** 型の整数値 **3** に変換する）のが、引数に受け取った値を **int** 型の整数値に変換する **int 関数**です。**Fig.1C-2** に示すように、**int(文字列)** と呼び出すと、文字列を整数値に変換した値が返却されます。

なお、2進、8進、10進、16進の整数を表す文字列の整数への変換は、**int(文字列, 基数)** で行い、文字列から **float** 型の実数値への変換は、**float 関数**を呼び出す **float(文字列)** で行います。

なお、数値に変換できない文字列を与えて呼び出す式（たとえば **int('H2O')** や **float('5X.2')**）は、エラーになります。

`int('17')` ➡ 17	int(文字列)	10進整数とみなして変換
`int('0b110', 2)` ➡ 6 `int('0o75', 8)` ➡ 61 `int('13', 10)` ➡ 13 `int('0x3F', 16)` ➡ 63	int(文字列, 基数)	指定された基数の整数とみなして変換
`float('3.14')` ➡ 3.14	float(文字列)	浮動小数点数とみなして変換

Fig.1C-2　文字列から整数への変換

3値の最大値を求める手続きを図で表したのが **Fig.1-1** です。プログラムの構造や流れなどを表す図には、いろいろな種類があり、ここでは**流れ図**＝**フローチャート**（*flowchart*）と呼ばれる図を使っています。

▶ フローチャートの主要な記号は、p.12 でまとめて学習します。

```
maximum = a
if b > maximum: maximum = b
if c > maximum: maximum = c
```

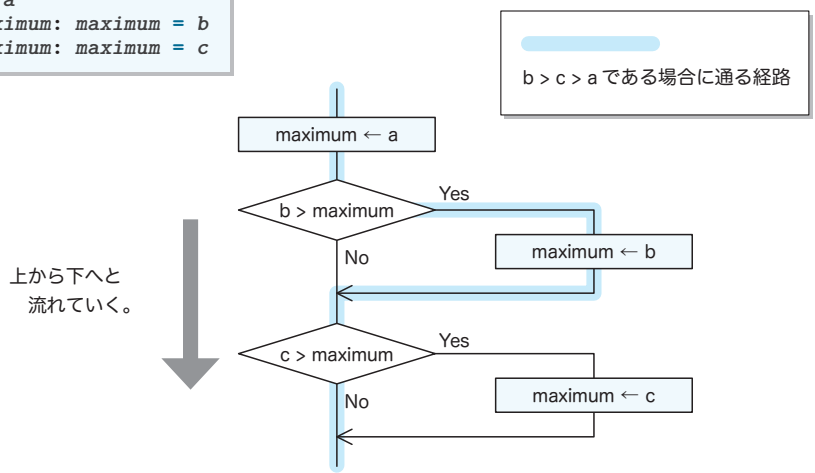


Fig.1-1　3値の最大値を求めるアルゴリズムの流れ図

プログラムの流れは、── に沿って上から下へと向かい、その過程で □ 内の処理が実行されます。

ただし、◇ を通過する際は、その中に記された《**条件**》の評価結果に応じて、Yes と No の**いずれか一方**をたどります。

そのため、*b* > *maximum* や *c* > *maximum* の判定が成立すれば（判定式 *b* > *maximum* あるいは判定式 *c* > *maximum* を評価した値が**真**であれば）、Yes と書かれた**右側**に進み、そうでなければ No と書かれた**下側**に進みます。

＊

プログラムの流れは、二つの分岐のいずれか一方を通るため、`if` 文によるプログラムの流れの分岐は、**双岐選択**と呼ばれます。

なお、□ 内の矢印記号 ← は、値の代入を表します。たとえば“*maximum* ← a”は、

> 『変数 a の値を変数 *maximum* に代入せよ。』

という指示です。

p.2 の実行例のように、変数 a，b，c に対して 1，3，2 を入力すると、プログラムの流れはフローチャート上の**青い線**の経路をたどります。

それでは、他の値を想定して、フローチャートをなぞってみましょう。

変数 a，b，c の値が、1，2，3 や 3，2，1 であっても、正しく最大値を求められます。また、三つの値が 5，5，5 とすべて等しかったり、1，3，1 と二つが等しくても、正しく最大値を求められます（**Fig.1-2**）。

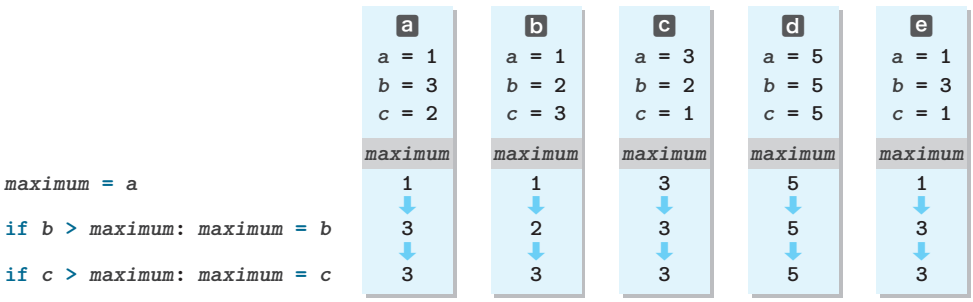


Fig.1-2　3値の最大値を求める過程における変数 maximum の値の変化

三つの変数 a，b，c の値が、6，10，7 や -10，100，10 であっても、フローチャート内の**青い線**をたどります。すなわち、b ＞ c ＞ a であれば、**必ず同じ経路をたどります**。

Column 1-2　if 文の構文

if 文や while 文などの複合文内の冒頭部は、if や while などのキーワードで始まって、コロン：で終わります。この部分は**頭部**（ヘッダ）（*header*）と呼ばれます。頭部の末尾のコロン：は、『この後にスイートが続きますよ。』という目印です。

Fig.1C-3 に示すのが、if 文の構文の概略です。

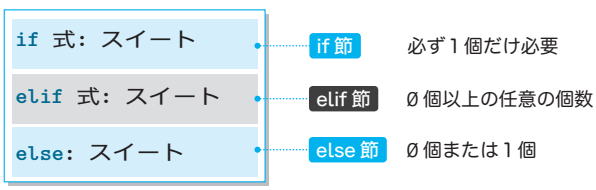


Fig.1C-3　if 文の構文の概略

3値の**具体的な値**ではなく、**すべての大小関係**に対して、最大値を正しく求められるかどうかを確認しましょう。確認を手作業で行うのは大変ですから、プログラムで行います。

List 1-2 に示すのが、そのプログラムです。

List 1-2 chap01/max3_func.py

```
# 三つの整数値の最大値を求めて表示（すべての大小関係に対して確認）

def max3(a, b, c):
    """a, b, cの最大値を求めて返却"""
    maximum = a
    if b > maximum: maximum = b
    if c > maximum: maximum = c
    return maximum

print(f'max3(3, 2, 1) = {max3(3, 2, 1)}')   # [A] a>b>c
print(f'max3(3, 2, 2) = {max3(3, 2, 2)}')   # [B] a>b=c
print(f'max3(3, 1, 2) = {max3(3, 1, 2)}')   # [C] a>c>b
print(f'max3(3, 2, 3) = {max3(3, 2, 3)}')   # [D] a=c>b
print(f'max3(2, 1, 3) = {max3(2, 1, 3)}')   # [E] c>a>b
print(f'max3(3, 3, 2) = {max3(3, 3, 2)}')   # [F] a=b>c
print(f'max3(3, 3, 3) = {max3(3, 3, 3)}')   # [G] a=b=c
print(f'max3(2, 2, 3) = {max3(2, 2, 3)}')   # [H] c>a=b
print(f'max3(2, 3, 1) = {max3(2, 3, 1)}')   # [I] b>a>c
print(f'max3(2, 3, 2) = {max3(2, 3, 2)}')   # [J] b>a=c
print(f'max3(1, 3, 2) = {max3(1, 3, 2)}')   # [K] b>c>a
print(f'max3(2, 3, 3) = {max3(2, 3, 3)}')   # [L] b=c>a
print(f'max3(1, 2, 3) = {max3(1, 2, 3)}')   # [M] c>b>a
```

求めた最大値を呼出し元に返却

実行結果

```
max3(3, 2, 1) = 3
max3(3, 2, 2) = 3
max3(3, 1, 2) = 3
max3(3, 2, 3) = 3
max3(2, 1, 3) = 3
max3(3, 3, 2) = 3
max3(3, 3, 3) = 3
max3(2, 2, 3) = 3
max3(2, 3, 1) = 3
max3(2, 3, 2) = 3
max3(1, 3, 2) = 3
max3(2, 3, 3) = 3
max3(1, 2, 3) = 3
```

▶ コメントの [A] ～ [M] は、**Fig.1C-5**（p.8）の A～M に対応しています。

最大値を求める部分は、何度も繰り返して利用されるため、独立した**関数**（*function*）として実現しています。薄い水色の部分が、受け取った三つの仮引数 `a`，`b`，`c` の最大値を求めて、それを返却する `max3` の**関数定義**（*function definition*）です。

プログラムのメイン部では、関数 `max3` に対して三つの値を実引数として与えて呼び出して、その返却値（**Column 1-3**）を表示する処理を 13 回行っています。

計算結果が正しいかどうかを確認しやすくするために、本プログラムでは、すべての呼出しにおいて、最大値が 3 となるように組み合わせた値を与えています。

Column 1-3 関数の返却値と関数呼出し式の評価

関数は、処理を行った結果の値を、`return` 文で呼出し元に返却します。関数 `max3` の場合、関数の末尾で変数 `maximum` の値を返却しています。

返却された値は、**呼出し式**の**評価**によって得られます。たとえば、`max3(3, 2, 1)` と呼び出した場合、**Fig.1C-4** に示すように、その呼出し式 `max3(3, 2, 1)` を評価した値が、`int` 型の 3 となります。

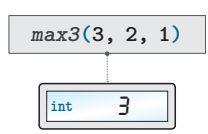


Fig.1C-4 呼出し式の評価

プログラムを実行してみましょう。13 種類すべての組合せに対して **3** と表示され、正しく最大値を求めていることが確認できます。

▶ 大小関係が全部で 13 種類であることについては、次ページの **Column 1-5** で学習します。

JIS X0001 では、《**アルゴリズム**》は次のように定義されています。

問題を解くためのものであって、明確に定義され、順序付けられた有限個の規則からなる集合。

もちろん、いくら曖昧さのないように記述されていても、変数の値によって、解けたり解けなかったりするのでは、正しいアルゴリズムとはいえません。

ここでは、3値の最大値を求めるアルゴリズムが正しいことを、論理的に確認するとともに、プログラムの実行結果からも確認しました。

▶ JIS（*Japanese Industrial Standards*）すなわち**日本工業規格**は、工業標準化法によって制定される鉱工業品に関する国の規格です。

Column 1-4 スイートの記述

複合文中の頭部の末尾に置かれたコロン：は、『この後に**スイート**（*suite*）が続きますよ。』という目印です（**Column 1-2**：p.5）。

スイートは、『一式』という意味であり、次のように記述します。

▪ 頭部の次の行に、1レベル深くインデント（字下げ）して（スペースの個数を多くして）文を置きます。スイート内の文が複数の場合は、それらの文をすべて同じレベルのインデントで置きます。

なお、置く文は、単純文でも複合文でも構いません（後者であれば、複合文の中に複合文が入る、入れ子の構造となります）。

なお、インデントのためのスペースは、最低でも1個必要です。

例
```
if a < b:
    min2 = a
    max2 = b
```

▪ スイートが単純文のみで構成される場合に限り、頭部と同じ行（すなわち：から改行までのあいだ）にスイートを置けます。

なお、単純文が2個以上であれば、各文をセミコロン；で区切ります（さらに、セミコロン；は最後の単純文の後ろにも置けるようになっています）。

例
```
if a < b: min2 = a
```

例
```
if a < b: min2 = a; max2 = b
```

例
```
if a < b: min2 = a; max2 = b;
```

なお、頭部と同じ行に置くスイートに複合文を含めることはできません。そのため、以下のようにするとエラーになります。

```
if a < b: if c < d: x = u   # エラー コロン:の後ろに複合文は置けない
```

Column 1-5　3値の大小関係と中央値

▪ 3値の大小関係の列挙

3値の大小関係の組合せ 13 種類を列挙するのが、**Fig.1C-5** です。ちなみに、ここに示している図は、木の形をしていることから**決定木**（*decision tree*）と呼ばれます。

左端の枠（a ≧ b）からスタートして右側へと進みましょう。［　　］内の条件が成立すれば**上側の線**をたどり、成立しなければ下側の線をたどっていきます。

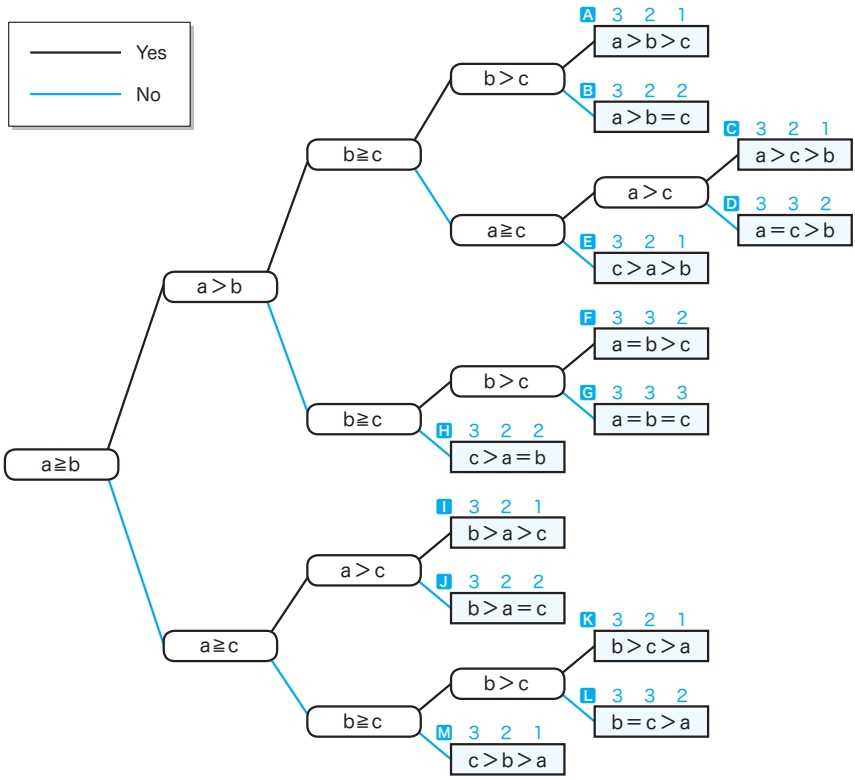


Fig.1C-5　3値の大小関係を列挙する決定木

右端の［　　］内に示しているのが、三つの変数 **a**，**b**，**c** の大小関係です。その上に示している青い数値は、**List 1-2** のプログラムで利用している、三つの値です（このプログラムでは、A〜Mの 13 種類に対して、最大値を求めています）。

▪ 3値の中央値

最大値・最小値とは異なり、**中央値**を求める手続きは、複雑です（そのため、数多くのアルゴリズムが考えられます）。

List 1C-1 に示すのが、プログラムの一例です。各 `return` 文の横のA〜Mは、**Fig.1C-5** と対応しています。

List 1C-1 chap01/median3.py

```
# 三つの整数値を読み込んで中央値を求めて表示

def med3(a, b, c):
    """a, b, cの中央値を求めて返却"""
    if a >= b:
        if b >= c:
            return b ← A B F G
        elif a <= c:
            return a ← D E H
        else:
            return c ← C
    elif a > c:
        return a ← I
    elif b > c:
        return c ← J K
    else:
        return b ← L M

print('三つの整数の中央値を求めます。')
a = int(input('整数aの値：'))
b = int(input('整数bの値：'))
c = int(input('整数cの値：'))

print(f'中央値は{med3(a, b, c)}です。')
```

実行例

```
三つの整数の中央値を求めます。
整数aの値：1↵
整数bの値：3↵
整数cの値：2↵
中央値は2です。
```

プログラムをじっくりと読んで、解読しましょう。

さて、中央値を求める関数 *med3* は、以下のようにも実現できます（'chap01/median3a.py'）。

```
def med3(a, b, c):
    """a, b, cの中央値を求めて返却（別解）"""
    if (b >= a and c <= a) or (b <= a and c >= a):
        return a
    elif (a > b and c < b) or (a < b and c > b):
        return b
    return c
```

コードは短くなるものの、効率が悪くなります。その理由を考えていきましょう。

まずは、`if` 節の判定式に着目します。

```
if (b >= a and c <= a) or (b <= a and c >= a):
```

ここで、`b >= a` および `b <= a` の判定を裏返した判定（実質的に同一の判定）が、続く `elif` 節で

```
elif (a > b and c < b) or (a < b and c > b):
```

と行われます。`if` 節の判定式が成立しなかった場合、続く `elif` 節でも（実質的に）同じ判定を行っていることが、効率の低下につながっています。

なお、3値の中央値を求める手続きは、『クイックソート』の改良アルゴリズム（第6章：p.216）などで応用されます。

条件判定と分岐

List 1-3 は、読み込んだ整数値の符号（正／負／0）を判定・表示するプログラムです。このプログラムを通じて、プログラムの流れの分岐に対する理解を深めましょう。

List 1-3 chap01/judge_sign.py

```
# 読み込んだ整数値の符号を表示

n = int(input('整数：'))

if n > 0:
    print('その値は正です。')  ← 1
elif n < 0:
    print('その値は負です。')  ← 2
else:
    print('その値は0です。')   ← 3
```

実行例

```
1 整数：17↵
  その値は正です。
2 整数：-5↵
  その値は負です。
3 整数：0↵
  その値は0です。
```

Fig.1-3 に示すのは、網かけ部のフローチャートです。変数 n の値が正であれば 1 が実行され、負であれば 2 が実行され、0 であれば 3 が実行されます。

すなわち、実行されるのは、いずれか**一つだけ**です。どれか二つが実行されたり、一つも実行されなかったり、ということはありません。プログラムの流れは**三つに分岐します**。

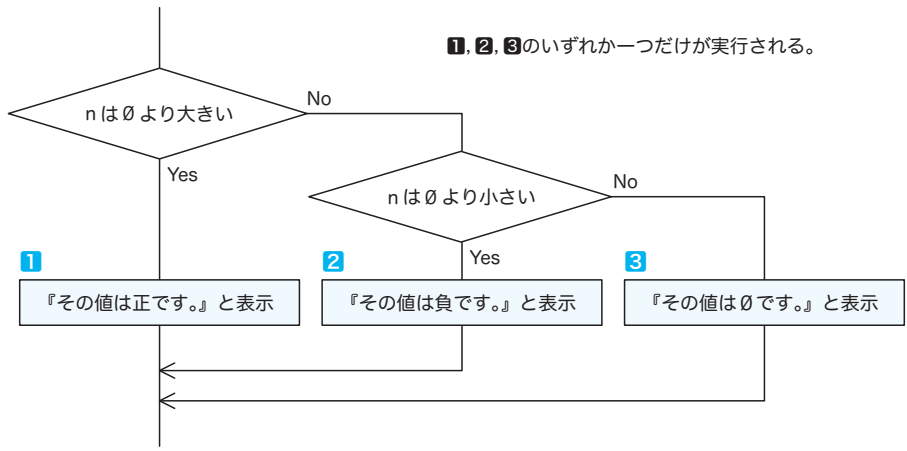


Fig.1-3　変数 n の符号の判定

引き続き、本プログラムと見た目が似ている **List 1-4** と **List 1-5** のプログラムの動作を検証しましょう。プログラムの行数が同じですから、プログラムの流れを同じように三つに分岐させるように感じられます。

いずれのプログラムも、n の値が 1 であれば『A』と表示し、2 であれば『B』と表示し、3 であれば『C』と表示します。ところが、それ以外の数値の場合の挙動は異なります。

List 1-4 chap01/branch1.py

```
# 整数値の判定（その１）

n = int(input('整数：'))

if n == 1:
    print('A')
elif n == 2:
    print('B')
else:
    print('C')
```

実行例

```
① 整数：3↵
C
② 整数：4↵
C
```

List 1-5 chap01/branch2.py

```
# 整数値の判定（その２）

n = int(input('整数：'))

if n == 1:
    print('A')
elif n == 2:
    print('B')
elif n == 3:
    print('C')
```

実行例

```
① 整数：3↵
C
② 整数：4↵
```

▪ **List 1-4** の動作（3分岐）

n が 1，2 以外の数値であれば、どんな値であっても『C』と表示します（実行例①および②）。すなわち、左ページの **List 1-3** と同様に、プログラムの流れを三つに分岐します。

▪ **List 1-5** の動作（4分岐）

n の値が 1、2、3 以外の数値であれば、何も表示しません（実行例②）。`if` 節と、続く2個の `elif` 節で構成されるため、プログラムの流れを三つに分岐しているように見えますが、そうではありません。

このプログラムの正体は **List 1-6** です。

“何も行わない” `else` 節が隠れており、プログラムの流れを四つに分岐します。

▶ `pass` 文は、何も行わない文です。

List 1-6 chap01/branch2a.py

```
# 整数値の判定（その２の正体）

n = int(input('整数：'))

if n == 1:
    print('A')
elif n == 2:
    print('B')
elif n == 3:
    print('C')
else:
    pass
```

実行例

```
① 整数：3↵
C
② 整数：4↵
```

Column 1-6 演算子とオペランド

プログラミング言語の世界では、`+` や `-` などの演算を行う記号は**演算子**（*operator*）と呼ばれ、演算の対象となる式は**オペランド**（*operand*）と呼ばれます。たとえば、大小関係を判定する式 `a > b` では、演算子は `>` であって、オペランドは `a` と `b` の二つです。

演算子は、オペランドの個数によって、以下の3種類に分類されます。

- **単項演算子**（*unary operator*） … オペランドが1個。例：`-a`
- **2項演算子**（*binary operator*） … オペランドが2個。例：`a < b`
- **3項演算子**（*ternary operator*） … オペランドが3個。例：`a if b else c`

＊

条件演算子（*conditional operator*）という名称の、`if` ～ `else` 演算子は、唯一の3項演算子です。条件式 `a if b else c` の評価結果は、式 `b` を評価した値が真であれば `a` の値、偽であれば `c` の値です。

```
1 a = x if x > y else y
2 print('cはゼロ' if c == 0 else 'cは非ゼロ')
```

1 では、*x* と *y* の大きいほうの値が `a` に代入されます。また、2 では、変数 *c* の値が `0` であれば『`c` はゼロ』と表示され、そうでなければ『`c` は非ゼロ』と表示されます。

フローチャート（流れ図）の記号

問題の定義・分析・解法の図的表現である**流れ図＝フローチャート**（*flowchart*）と、その記号は、以下の規格で定義されています。

JIS X0121『**情報処理用流れ図・プログラム網図・システム資源図記号**』

ここでは、代表的な用語と記号の概要を学習します。

プログラム流れ図（program flowchart）

プログラム流れ図には、以下に示す記号があります。

- 実際に行う演算を示す記号。
- 制御の流れを示す線記号。
- プログラム流れ図を理解し、かつ作成するのに便宜を与える特殊記号。

データ（data）

媒体を指定しないデータを表します（**Fig.1-4**）。

Fig.1-4　データ

処理（process）

任意の種類の処理機能を表します（**Fig.1-5**）。

たとえば、情報の値・形・位置を変えるように定義された演算もしくは演算群の実行、または、それに続くいくつかの流れの方向の一つを決定する演算もしくは演算群の実行を表します。

Fig.1-5　処理

定義ずみ処理（predefined process）

サブルーチンやモジュールなど、別の場所で定義された一つ以上の演算または命令群からなる処理を表します（**Fig.1-6**）。

Fig.1-6　定義ずみ処理

判断（decision）

一つの入り口といくつかの択一的な出口をもち、記号中に定義された条件の評価にしたがって、唯一の出口を選ぶ判断機能、またはスイッチ形の機能を表します（**Fig.1-7**）。

想定される評価結果は、経路を表す線の近くに書きます。

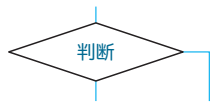


Fig.1-7　判断

ループ端（loop limit）

二つの部分から構成され、ループの始まりと終わりを表します（**Fig.1-8**）。記号の二つの部分には、**同じ名前**を与えます。

Fig.1-9 に示すように、ループの**始端記号**（前判定繰返しの場合）または**終端記号**（後判定繰返しの場合）の中に、初期値（初期化）と増分と終了値（終了条件）とを表記します。

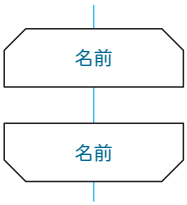


Fig.1-8　ループ端

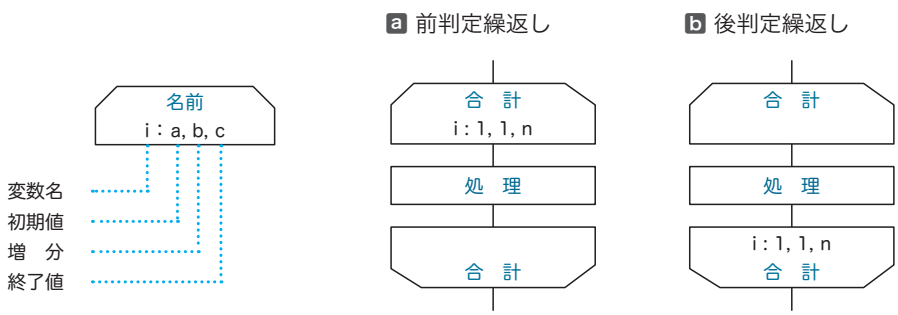


Fig.1-9　ループ端と初期値・増分・終了値

▶　図**a**と図**b**に示すのは、変数 *i* の値を 1 から *n* まで 1 ずつ増やしながら、『処理』を *n* 回繰り返すフローチャートです。なお、1, 1, *n* の代わりに、1, 2, …, *n* という表記を用いることもあります。

線（line）

制御の流れを表します（**Fig.1-10**）。

流れの向きを明示する必要があるときは、矢先を付けなければなりません。

なお、明示の必要がない場合も、見やすくするために矢先を付けても構いません。

Fig.1-10　線

端子（terminator）

外部環境への出口、または外部環境からの入り口を表します（**Fig.1-11**）。たとえば、プログラムの流れの**開始**もしくは**終了**を表します。

Fig.1-11　端子

この他に、並列処理、破線などの記号があります。

1–2 繰返し

本節では、プログラムの流れを繰り返すことによって実現される、単純なアルゴリズムを学習します。

1 から n までの整数の総和を求める

次に考えるのは、《**1 から *n* までの整数の総和を求めるアルゴリズム**》です。求める総和は、*n* が 2 であれば 1 + 2 で、*n* が 3 であれば 1 + 2 + 3 です。

プログラムを**List 1-7**に、プログラム網かけ部のフローチャートを**Fig.1-12**に示します。

List 1-7 chap01/sum1ton_while.py

```
# 1からnまでの総和を求める（while文）

print('1からnまでの総和を求めます。')
n = int(input('nの値：'))

sum = 0                          ―1
i = 1

while i <= n:    # iがn以下のあいだ繰り返す
    sum += i     # sumにiを加える   ―2
    i += 1       # iに1を加える
print(f'1から{n}までの総和は{sum}です。')
```

実行例

```
1からnまでの総和を求めます。
nの値：5
1から5までの総和は15です。
```

while 文による繰返し

ある条件が成立するあいだ、処理を繰り返し実行するのは、一般に**ループ**（*loop*）と呼ばれる**繰返し**（*repetition*）構造です。

`while` 文は、繰返しを続けるかどうかの判定を、処理実行の**前**に行うループです。このような繰返しの構造は、**前判定繰返し**と呼ばれます。

以下に示すのが、`while` 文の基本的な形式です。**判定式**の評価によって得られる値が真である限り、**スイート**が繰り返し実行されます。

```
while 判定式： スイート
```

なお、繰返しの対象となる**スイート**のことを、本書では、**ループ本体**と呼びます。

それでは、プログラムとフローチャートの 1 と 2 を理解しましょう。

1 総和を求めるための前準備です。総和を格納するための変数 `sum` の値を `0` にして、繰返しを制御するための変数 `i` の値を 1 にします。

2 変数 `i` の値が `n` 以下であるあいだ、`i` の値を一つずつ増やしていきながら、ループ本体を繰り返し実行します。繰り返すのは `n` 回です。

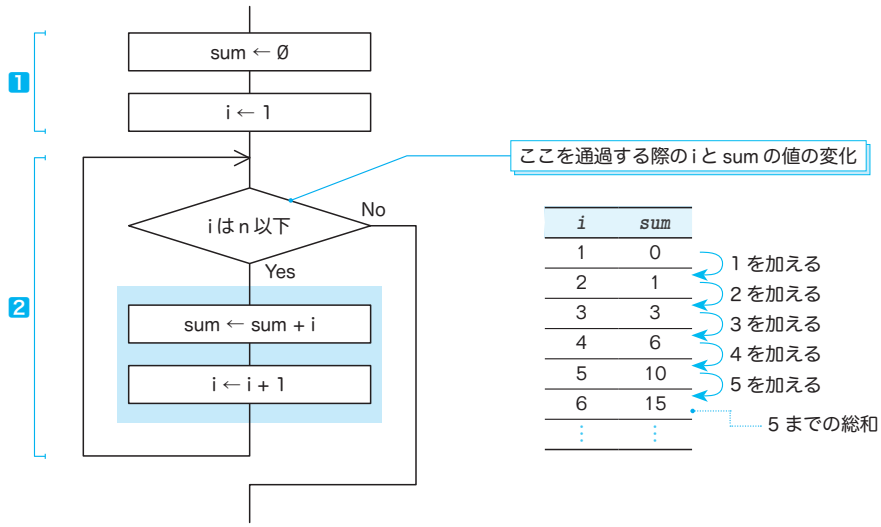


Fig.1-12　1 から n までの総和を求めるフローチャートと変数の変化

i が *n* 以下かどうかを判定する判定式 `i <= n`（フローチャートの ◇ ）を通過する際の変数 *i* と *sum* の値の変化をまとめた表と、プログラムとを見比べながら、このアルゴリズムを詳しく理解していきましょう。

判定式を初めて通過する際の変数 *i* と *sum* の値は 1 で設定した 1 と 0 です。その後、繰返しが行われるたびに変数 *i* の値はインクリメントされて一つずつ増えていきます。

変数 *sum* の値は『**それまでの総和**』であり、変数 *i* の値は『**次に加える値**』です。

たとえば、*i* が 5 のときの変数 *sum* の値は『**1 から 4 までの総和**』である 10 です（すなわち変数 *i* の値である 5 が加算される前の値です）。

*

なお、*i* の値が *n* を超えたときに `while` 文の繰返しが終了するため、最終的な *i* の値は、**n ではなく n + 1 となります**。

本プログラムの末尾に、以下の文を加えましょう（`'chap01/sum1ton_while2.py'`）。

```
print(f'iの値は{i}です。')
```

```
1からnまでの総和を求めます。
nの値：5↵
1から5までの総和は15です。
iの値は6です。
```

右のような実行結果が得られ、最終的な *i* の値が、**n ではなく n + 1 であることが確認できます**。

*

変数 *i* のように、繰返しの制御に用いられる変数は、一般に**カウンタ用変数**と呼ばれます。

for 文による繰返し

単一の変数の値でプログラムの流れを制御する繰返しは、**while** 文ではなく**for** 文を用いたほうがスマートに実現できます。

1 から *n* までの整数の総和を **for** 文で求めるように書きかえたプログラムが **List 1-8** です。

List 1-8 chap01/sum1ton_for.py

```
# 1からnまでの総和を求める（for文）

print('1からnまでの総和を求めます。')
n = int(input('nの値：'))

sum = 0
for i in range(1, n + 1):
    sum += i    # sumにiを加える

print(f'1から{n}までの総和は{sum}です。')
```

実行例

```
1からnまでの総和を求めます。
nの値：5↵
1から5までの総和は15です。
```

網かけ部のフローチャートを **Fig.1-13** に示します。六角形の**ループ端**（*loop limit*）は、繰返しの**開始点**と**終了点**を指示する記号です。同じ名前をもったループ始端とループ終端とで囲まれた部分が繰り返されます。

したがって、変数 *i* の値を 1，2，3，… と、1 から *n* までを、1 ずつ増やしながらループ本体内の代入文 *sum* **+=** *i* を実行します。

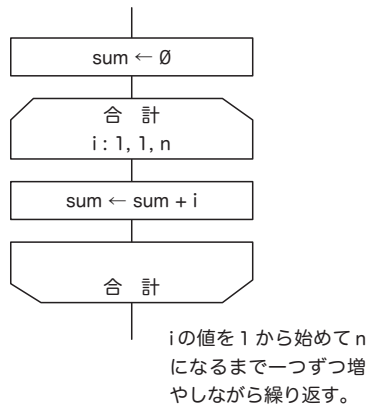


Fig.1-13 1 から n までの総和

ガウスの方法

たとえば、1 から **10** までの総和は **(1 + 10) * 5** によって求められることが知られています。

ガウスの方法と呼ばれる、この方法を用いて総和を求めるコードは次のようになり、繰返しが不要です（'chap01/sum_gauss.py'）。

```
sum = n * (n + 1) // 2
```

Column 1-7 range 関数

range 関数は、**Fig.1C-6** に示す数列（イテラブルオブジェクト）を生成します。

range(*n*)	0以上*n*未満の数値を順番に列挙した数列
range(a, *b*)	a以上*b*未満の数値を順番に列挙した数列
range(a, *b*, *step*)	a以上*b*未満の数値を*step*個おきに順番に列挙した数列

Fig.1C-6 range 関数が生成する数列

2値のソートと2値の交換

総和を求める対象の開始値を、1ではなくて任意の整数値に変更しましょう。**List 1-9** は、二つの整数 a と b を含め、そのあいだの全整数の総和を求めるプログラムです。

List 1-9 chap01/sum.py

```
# aからbまでの総和を求める（for文）

print('aからbまでの総和を求めます。')
a = int(input('整数a：'))
b = int(input('整数b：'))

if a > b:
    a, b = b, a

sum = 0
for i in range(a, b + 1):
    sum += i    # sumにiを加える

print(f'{a}から{b}までの総和は{sum}です。')
```

aとbを昇順にソート

実行例

```
① aからbまでの総和を求めます。
整数a：3↵
整数b：8↵
3から8までの総和は33です。

② aからbまでの総和を求めます。
整数a：8↵
整数b：3↵
3から8までの総和は33です。
```

網かけ部では、$a \leqq b$ となるように、a と b を昇順（しょうじゅん）に**ソート**（*sort*）しています。2値のソートは、a の値が b の値より大きいときにのみ、変数 a と b の値を交換することで行います。

▶ すなわち、実行例①では交換は行われず、実行例②では交換が行われます。

a と b の2値の交換は、次に示す単一の代入文で行うのが定石です（**Column 1-8**）。

```
a, b = b, a      # aとbの値を交換
```

なお、総和を求めるアルゴリズム自体は、前のプログラムと同じです。開始値と終了値の変更に伴って、`range(1, n + 1)` が `range(a, b + 1)` に変更されています。

▶ ソート全般に関しては、第6章で学習します。

Column 1-8 2値の交換（その1）

a と b の交換が単一の代入文 “`a, b = b, a`” で実現できるのは、**Fig.1C-7** に示すように代入が行われるからです。

- 右辺の `b, a` によって、2値がパックされたタプル `(b, a)` が生成される。
- 代入実行時に、タプル `(b, a)` がアンパックされて（先頭から順に）取り出された b と a が、それぞれ a と b に代入される。

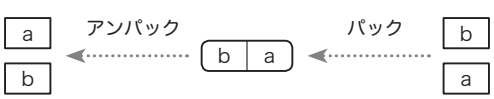


Fig.1C-7 2値の交換

繰返しの過程における条件判定（その1）

次は、aからbまでの総和を求める過程の式を表示するように仕様を変更します。**List 1-10** に示すのが、そのプログラムです。

List 1-10 chap01/sum_verbose1.py

```
# aからbまでの総和を求める（求める過程の式も表示：その１）

print('aからbまでの総和を求めます。')
a = int(input('整数a：'))
b = int(input('整数b：'))

if a > b:
    a, b = b, a

sum = 0
for i in range(a, b + 1):
    if i < b:           # 途中
        print(f'{i} + ', end='')  ←1
    else:               # 最後
        print(f'{i} = ', end='')  ←2
    sum += i            # sumにiを加える

print(sum)
```

- 繰返しはn回。
- if文の判定はn回。

実行例

aからbまでの総和を求めます。

① 整数a：3↵
整数b：3↵
3 = 3

② 整数a：3↵
整数b：4↵
3 + 4 = 7

③ 整数a：3↵
整数b：7↵
3 + 4 + 5 + 6 + 7 = 25

まずは実行しましょう。加算する数値がn個のとき、表示する＋記号はn − 1個です。

▶ たとえば、実行例③の“3 + 4 + 5 + 6 + 7 = 25”では、加算する数値は5個で、表示する＋記号は4個です（加算する数値の個数nは、b − a + 1で得られます）。

本プログラムのfor文が、iの値をaからbまでインクリメントすることは、前のプログラムと同じです。

そのfor文のループ本体内の網かけ部では、if文によって表示内容を変えています。

1 途中の値の表示：数値の後ろに＋を出力。　例 '3 + '、'4 + '、'5 + '、'6 + '。

2 最後の値の表示：数値の後ろに＝を出力。　例 '7 = '。

このような実装は、好ましくありません。たとえばaが1で、bが10000であるとします。そうすると、for文の繰返しは10,000回行われます。最初の9,999回は、判定式i < bが成立してif節内の1が実行されます。判定式が成立せずelse節内の2が実行されるのは1回だけです。最後に1回だけ実行すべき2のために、10,000回もの判定を行うわけです。

ある特定の回だけに成立することが既知であるにもかかわらず、繰返しのたびに条件判定を行うのは、明らかに無駄です。

＊

iがbと等しいときを“特別扱い”したほうがよいことが分かりました。そのように書き直したのが、右ページのList 1-11のプログラムです。

List 1-11 chap01/sum_verbose2.py

```
# aからbまでの総和を求める（求める過程の式も表示：その２）

print('aからbまでの総和を求めます。')
a = int(input('整数a：'))
b = int(input('整数b：'))

if a > b:
    a, b = b, a

sum = 0
for i in range(a, b):
    print(f'{i} + ', end='')          ←1
    sum += i          # sumにiを加える

print(f'{b} = ', end='')
sum += b              # sumにbを加える  ←2

print(sum)
```

- 繰返しは n−1 回。
- if 文の判定は 0 回。

実行例

List 1-10と同じ実行結果が得られます。

本プログラムでは、表示を２ステップで行います。

1 途中の値の表示：for 文によって、a から b - 1 までの値の後ろに + を付けて出力。

2 最後の値の表示：b の値の後ろに = を付けて出力。

繰返しの回数が n 回から n - 1 回に減少するとともに、if 文による判定回数が n 回から 0 回になりました。

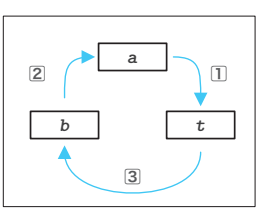
▶ ただし、繰返しの回数の１回の減少分は、追加された2の実行によって、相殺されます。

Column 1-9 ２値の交換（その２）

２値 a, b の交換を、単一の代入文 “a, b = b, a” で行えることを知らなければ、以下のように、まわりくどく実現せざるを得ません。

① a の値を t に保存しておく。
② b の値を a に代入する。
③ t に保存していた最初の a の値を b に代入する。

なお、２値のソートを含むソートの実装については、**Column 6-4**（p.219）でも学習します。



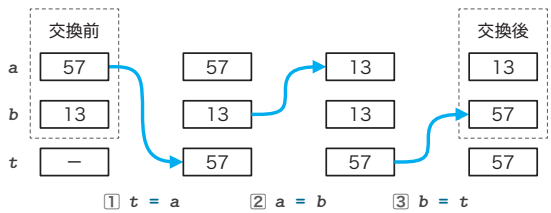


Fig.1C-8 作業用の変数を利用した２値の交換

繰返しの過程における条件判定（その2）

次に作るのは、指定された個数の記号文字を（途中で改行せずに）連続表示するプログラムです。なお、表示は+と-を交互に行うものとします。

それが、**List 1-12** のプログラムです。

List 1-12 chap01/alternative1.py

```
# 記号文字+と-を交互に表示（その１）

print('記号文字+と-を交互に表示します。')
n = int(input('全部で何個：'))

for i in range(n):
    if i % 2:                   # 奇数
        print('-', end='')
    else:                       # 偶数
        print('+', end='')
print()
```

- 繰返しはn回。
- 除算はn回。
- if文の判定はn回。

実行例

```
記号文字+と-を交互に表示します。
全部で何個：12↵
+-+-+-+-+-+-
```

for文で変数iの値を0, 1, 2, …, n - 1とインクリメントする過程で、以下のように表示を行います。

- iが奇数であれば（2で割った余りが0でなければ）：'-'を出力する。
- iが偶数であれば：'+'を出力する。

このプログラムには、大きく二つの欠点があります。

1 繰返しのたびにif文の判定を行う

for文による繰返しのたびに、if文が実行されます。そのため、iが奇数であるかどうかの判定は、全部でn回行われます（nが50000であれば5万回行われます）。

2 変更に対して柔軟に対応しにくい

本プログラムでは、カウンタ用変数iの値を0からn - 1までインクリメントしています。もし変数iのインクリメントを、（0からではなく）1からnまでとするのであれば、プログラム中のfor文を、次のように変更しなければなりません（'chap01/alternative1a.py'）。

```
for i in range(1, n + 1):
    if i % 2:                   # 奇数
        print('+', end='')
    else:                       # 偶数
        print('-', end='')
```

すなわち、ループ本体であるif文の変更も余儀なくされます（二つのprint関数の呼出しの順序を入れかえなければなりません）。

*

問題点を解決しましょう。**List 1-13** に示すのが、そのプログラムです。

List 1-13 chap01/alternative2.py

```
# 記号文字+と-を交互に表示（その２）

print('記号文字+と-を交互に表示します。')
n = int(input('全部で何個：'))

for _ in range(n // 2):                    ①
    print('+-', end='')

if n % 2:                                  ②
    print('+', end='')
print()
```

- 繰返しは n // 2 回。
- 除算は 2 回。
- if 文の判定は 1 回。

実行例

List 1-12と同じ実行結果が得られます。

主要部は二つのステップで構成されます。**Fig.1-14** を見ながら理解しましょう。

① n // 2 個の '+-' を出力

for 文は '+-' の出力を n // 2 回行います。出力回数は、たとえば n が 12 であれば6回、n が 15 であれば7回です。そのため、n が偶数であれば、このステップのみで表示が完了します。

なお、繰返しのカウンタ用の変数名を _ としています。**1個の下線文字 _ の変数名は、その変数の値をループ本体で使用しないことを、プログラムの読み手に伝えます。**

▶ ループ本体で変数 _ の値を取り出したり使ったりすることは可能です。

② n が奇数のときのみ '+' を出力

n が奇数のときに、最後の '+' を出力します。これで、n が奇数のときの表示が完了します。

本プログラムは、繰返しのたびに if 文による判定を行う必要がありません。そのため、if 文の判定が②での1回だけとなっています。

さらに、除算の回数も減っています。①での n // 2 と、②での n % 2 の合計2回のみです。

▶ カウンタ用変数の開始値を 0 ではなく 1 に変更するのも柔軟に対応できます。①の for 文を以下のように変更するだけです（'chap01/alternative2a.py'：変更は range 関数に与える引数を変更するだけであって、ループ本体の変更は不要です）。

```
for _ in range(1, n // 2 + 1):
    print('+-', end='')
```

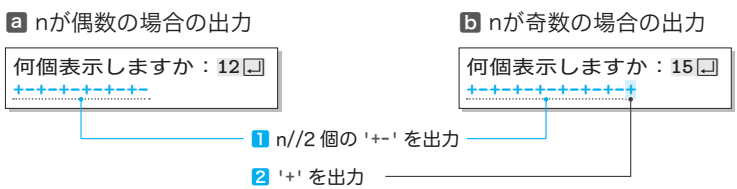


Fig.1-14 記号文字 + と – を交互に n 個表示

繰返しの過程における条件判定（その3）

次に作るのは、記号文字 * を n 個表示するプログラムです。ただし、w 個表示するごとに改行するものとします。

それが、**List 1-14** のプログラムです。

List 1-14 chap01/print_stars1.py

```
# n個の記号文字*をw個ごとに改行しながら表示（その１）

print('記号文字*を表示します。')
n = int(input('全部で何個：'))
w = int(input('何個ごとに改行：'))

for i in range(n):
    print('*', end='')
    if i % w == w - 1:
        print()              # 改行

if n % w:
    print()                  # 改行
```

- 繰返しは n 回。
- if 文の判定は n + 1 回。

実行例

```
記号文字*を表示します。
全部で何個：14↵
何個ごとに改行：5↵
*****
*****
****
```

変数 i の値を 0, 1, 2, … とインクリメントしながら記号文字 '*' を出力します。改行を行うのは、以下の2箇所です。

1 for 文による繰返しの過程で、記号文字を出力した際に、変数 i の値を w で割った余りが w - 1 のときに改行します。**Fig.1-15** に示すように、改行文字が出力されるのは、w が 5 であれば、i の値が 4, 9, 14, … のときです。

2 図 a のように、n が w の倍数であれば、最後に出力した * の後の“最後の改行”は完了しています。一方、図 b のように、n が w の倍数でなければ、最後の改行はまだ行われていません。そこで、n が w の倍数でないときにのみ改行を行います。

本プログラムは、for 文による繰返しのたびに if 文による判定が行われるため、それほど効率がよくない、という欠点があります。問題点を解消したのが、**List 1-15** のプログラムです。

Fig.1-15 n個の記号文字 * をw個ごとに改行しながら表示（その1）

List 1-15 chap01/print_stars2.py

```
# n個の記号文字*をw個ごとに改行しながら表示（その２）

print('記号文字*を表示します。')
n = int(input('全部で何個：'))
w = int(input('何個ごとに改行：'))

for _ in range(n // w):                    ←1
    print('*' * w)

rest = n % w
if rest:                                  ←2
    print('*' * rest)
```

- 繰返しは n // w 回。
- if 文の判定は 1 回。

実行例

List 1-14と同じ実行結果が得られます。

大きく二つのステップで構成されています。**Fig.1-16** を見ながら理解しましょう。

1 w個の '*' の出力をn // w回行う

for 文によって、w 個の '*' の出力（最後に改行を伴います）を、n // w 回繰り返します。

出力回数は、たとえば、n が 15 で w が 5 であれば '*****' の表示が3回で、n が 14 で w が 5 であれば '*****' の表示が2回です。n が w の倍数のときは、本ステップで出力が完了します。

▶ 式 '*' * w は、文字列 '*' を w 回繰り返した文字列を生成します。

2 n % w 個の '*' と改行文字を出力

n が w の倍数でないときに、残った最後の行を出力します。

n を w で割った余りを変数 rest に求め、rest 個（たとえば n が 14 で w が 5 であれば4個）の '*' を表示した上で改行文字を出力します。

▶ n が w の倍数であれば、変数 rest の値は 0 ですから、記号文字も改行文字も出力されません。

Fig.1-16 n個の記号文字 * をw個ごとに改行しながら表示（その2）

Column 1-10 なぜカウンタ用変数の名前はiやjなのか

多くのプログラマが、for 文などの繰返し文を制御するための変数として i や j を使います。

その歴史は、技術計算用のプログラミング言語 FORTRAN の初期の時代にまで遡（さかのぼ）ります。この言語では変数は原則として実数です。しかし、名前の先頭文字が I，J，…，N の変数だけは自動的に整数とみなされていました。そのため、繰返しを制御するための変数としては I，J,…を使うのが最も手軽な方法だったのです。

正の値の読込み

1 から n までの総和を求める **List 1-8** のプログラム（p.16）に戻ります。このプログラムを実行して、*n* に対して負の値である **-5** を入力してみましょう。次のように表示されます。

```
1 から -5 までの総和は 0 です。
```

これは、数学的に不正である以前に、感覚的にもおかしいものです。

そもそも、このプログラムでは、*n* に読み込む値を**正の値**に限定すべきです。そのように改良したプログラムを **List 1-16** に示します。

List 1-16 chap01/sum1ton_positive.py

```
# 1からnまでの総和を求める（nに正の整数値を読み込む）

print('1からnまでの総和を求めます。')

while True:
    n = int(input('nの値：'))
    if n > 0:
        break

sum = 0
i = 1

for i in range(1, n + 1):
    sum += i    # sumにiを加える
    i += 1      # iに1を加える
print(f'1から{n}までの総和は{sum}です。')
```

n が 0 より大きくなるまで繰り返す

実行例

```
1からnまでの総和を求めます。
nの値：-6↵
nの値：0↵
nの値：10↵
1から10までの総和は55です。
```

0 以下であれば再読込み

無限ループと break 文

読み込んだ値を n に代入する文が、網かけ部の **while** 文の中に入っています。その **while** 文の判定式は、単なる **True** です。これは真ですから、**while** 文は無限に繰り返されます。このような繰返しは、**無限ループ**と呼ばれます。

＊

キーボードから整数値を読み込んだ後は、*n* が正であるかどうかを、**if** 文で判定します。判定が成立したときに実行しているのが、**break 文**（*break statement*）です。

繰返し文中で break 文が実行されると、その繰返し文の実行が強制的に終了します。その結果、プログラムの流れは無限ループから脱出します。

▶ break 文の働きは、**Fig.1-18**（p.27）に示しています。

なお、読み込んだ整数値 n が 0 以下であれば、**break** 文は実行されないため、**while** 文の実行が再び繰り返されます（「n の値：」と入力を促して、キーボードから整数値の読込みを行います）。

そのため、**while** 文が終了したときの n の値は正になっています。

多くのプログラミング言語では、ループ本体を実行した後に、繰返しを続けるかどうかの判定を行う、**後判定繰返し**を実現する繰返し文が、`do` ～ `while` 文、あるいは、`repeat` ～ `until` 文などとして提供されます。

Python では後判定繰返しのための繰返し文が提供されないため、前判定繰返し文（`while` 文あるいは `for` 文）と `break` 文を組み合わせる必要があります。

*

Fig.1-17 に示すのが、プログラム網かけ部のフローチャートです。

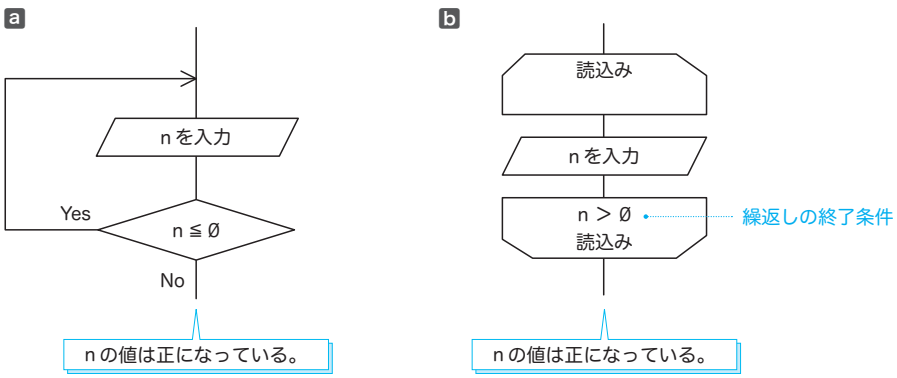


Fig.1-17　正の値の読込み

二つのフローチャートは、本質的には同じです。もっとも、繰返しの終了条件を下側のループ端に書く図**b**は、**前判定繰返しとの見分けがつきにくい**ため、図**a**が好まれます。

Column 1-11　for 文終了時のカウンタ用変数の値

List 1-7（p.14）で学習したように、頭部が `while i <= n:` となっている `while` 文が終了したときのカウンタ用変数 `i` の値は、`n` ではなく `n + 1` です。

一方、頭部が `for i in range(1, n + 1):` となっている `for` 文で実現した **List 1-8**（p.16）のプログラムでは、`for` 文が終了したときのカウンタ用変数 `i` の値は、`n` です。

一般に、`for i in range(a, b):` という `for` 文は、`[a, a + 1, a + 2, …, b - 1]` というイテラブルオブジェクトを生成し、そこから1個ずつ値を `i` に取り出して繰返しを行います。そのため、`for` 文が終了したときの `i` の値は `b` ではなく、`b - 1` となるのです。

念のため、まとめましょう（いずれも、ループ本体が最後に実行されるときの `i` の値は `n` です）。

```
while i <= n:                    繰返し終了時の i の値は n + 1。
for i in range(開始値 , n + 1):  繰返し終了時の i の値は n。
```

辺と面積が整数値である長方形

次に作るのは、辺も面積も整数である長方形の辺の長さを列挙するプログラムです。なお、与えられた面積に対して辺の長さを列挙し、短辺と長辺は区別しないものとします。

たとえば、長方形の面積として 32 が与えられると、辺の長さとして、「1 と 32」、「2 と 16」、「4 と 8」を列挙します（すなわち、「2 と 16」を列挙すれば、「16 と 2」は不要です）。

List 1-17 に示すのが、そのプログラムです。

List 1-17 chap01/rectangle.py

```
# 縦横が整数で面積がareaの長方形の辺の長さを列挙

area = int(input('面積は：'))

for i in range(1, area + 1):
    if i * i > area: break
    if area % i : continue
    print(f'{i}×{area // i}')
```

実行例

```
面積は：32↵
1×32
2×16
4×8
```

『長方形』と『辺の長さ』という言葉を使いましたが、実は、本プログラムが行っているのは、**約数**（*divisor*）の列挙です。実行例では、32 の約数を列挙しています。

プログラムの `for` 文では、カウンタ用変数 `i` の値を 1 から `area` までインクリメントしながら、次のことを行います（変数 `i` の値は、長方形の短辺の長さに相当します）。

① `i * i` が `area` を超えると `for` 文を強制終了する

`i * i` が面積 `area` を超えると、カウンタ用変数 `i` の値が、長方形の短辺ではなく、長辺となってしまうからです。実行例では、`i` が 6 になった段階で、（`6 * 6` すなわち 36 が面積 32 を超えるため）`for` 文の繰返しを、`break` 文によって強制終了します。

② `area` が `i` で割り切れなければ `for` 文の次のステップへと進む

面積 `area` が `i` で割れ切れなければ、`i` は整数の辺（約数）ではありません。

実行例では、たとえば `i` が 3 の場合、`32 % 3` は 2 となります。3 は 32 の約数ではないため、列挙（表示）は不要です。

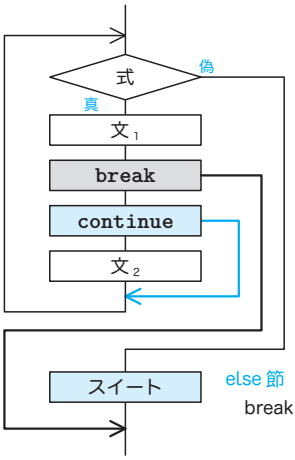
ここで利用しているのが、`break` 文と対照的な **`continue` 文**（*continue statement*）です。**Fig.1-18** に示すように、**繰返し文中で `continue` 文が実行されると、ループ本体内の後続部の実行がスキップされて、プログラムの流れが判定式へと戻ります。**

なお、図に示すように、`while` 文や `for` 文などの繰返し文の末尾には **`else` 節**を置けます。繰返し文の実行が、`break` 文によって強制終了することなく（つつがなく）終了したときにのみ、`else` 節内のスイートが実行されます。

③ 辺の長さの表示

変数 `i` の値と、`area // i` の値を、短辺および長辺の長さとして表示します。

```
while 式:
    文1
    break
    continue
    文2
else:
    スイート
```



break 文
　else 節を含めて while 文を強制終了する

continue 文
　スイート内の後続部の実行をスキップする

else 節
　break が実行されなかったときのみ実行される

※ 本図は、while 文を例に示しています。
　break 文と continue 文の働きは for 文でも同様です。

Fig.1-18　while 文と break 文と continue 文

次に、else 節を伴う for 文のプログラムを作りましょう。**List 1-18** に示すのが、そのプログラムです。

List 1-18　chap01/for_else.py

```
# 10～99の乱数をn個生成（13が生成されたら中断）

import random

n = int(input('乱数は何個：'))

for _ in range(n):
    r = random.randint(10, 99)
    print(r, end=' ')
    if r == 13:
        print('\n事情により中断します。')
        break
else:
    print('\n乱数生成終了。')
```

実行例

```
① 乱数は何個：5↵
87 82 48 83 62
乱数生成終了。
② 乱数は何個：5↵
39 72 86 13
事情により中断します。
```

乱数として 13 が生成された

for 文の繰返しによって、2桁の整数（10 ～ 99）の乱数を n 個生成します。

その過程で、13 が生成された場合は、『事情により中断します。』と表示して、for 文の繰返しを break 文によって強制的に中断します。そのため、else 節が実行されることはありません。

なお、一度も 13 が生成されなかった場合は、繰返し終了後に else 節が実行されて、『乱数生成終了。』と表示されます。

▶　乱数を生成する random.randint 関数については、**Column 1-13**（p.31）で学習します。

繰返しのスキップと複数の range の走査

for 文の繰返しの過程で、ある特定の条件のときにのみ処理を行う必要がない、ということがあります。たとえば、1 から 12 までを“8 をスキップして”繰り返したい、といったケースです。

List 1-19 に示すのが、そのプログラムです。繰返しを行う for 文の中で、i が 8 になったときに continue 文を実行することで、スキップを行っています。

List 1-19 chap01/skip1.py

```
# 1から12までを8をスキップして繰り返す（その１）

for i in range(1, 13):
    if i == 8:
        continue
    print(i, end=' ')
print()
```

実行結果

```
1 2 3 4 5 6 7 9 10 11 12
```

本プログラムは、continue 文の例題として入門用テキストで示されることがありますが、このような実装は好ましくない、ということを知っておきましょう。というのも、スキップすべきかどうかの判定にコストがかかるからです。もし、10 万回の繰返しの途中で 1 回だけスキップするのであれば、その 1 回だけのために、10 万回もの判定が行われます。

もちろん、for 文の繰返しの中で行う処理の実行時に、スキップすべき値が決定する（たとえば、キーボードから読み込む、あるいは、乱数で決定する、など）、もしくは、変化するのであれば、このような、if 文と continue 文による方法を使わざるを得ません。

*

スキップすべき値が事前に分かっているのであれば、そのことをコードとして実現します。**List 1-20** に示すのが、プログラムの一例です。

List 1-20 chap01/skip2.py

```
# 1から12までを8をスキップして繰り返す（その２）

for i in list(range(1, 8)) + list(range(9, 13)):
    print(i, end=' ')
print()
```

実行結果

```
1 2 3 4 5 6 7 9 10 11 12
```

本プログラムでは**リスト**を利用しています。1 ～ 7 のリスト [1, 2, 3, 4, 5, 6, 7] と、9 ～ 12 のリスト [9, 10, 11, 12] を連結した、リスト [1, 2, 3, 4, 5, 6, 7, 9, 10, 11, 12] を繰返しを行う前に生成します。

for 文は、生成されたリストから 1 個ずつ取り出して繰返しを行います。そのため、ループ本体内では、無駄な判定は行われません。

▶ リストについては、次章で学習します。

Column 1-12 値比較演算子の連続適用とド・モルガンの法則

List 1C-2 に示すのは、《2桁の正の整数値》を読み込むプログラムです。

List 1C-2 chap01/2digits1.py

```
# ２桁の正の整数値（10～99）を読み込む

print('２桁の整数値を入力してください。')

while True:
    no = int(input('値は：'))
    if no >= 10 and no <= 99:
        break
print(f'読み込んだのは{no}です。')
```

実行例

```
２桁の整数値を入力してください。
値は：9↵
値は：146↵
値は：57↵
読み込んだのは57です。
```

本プログラムでは、網かけ部の判定式によって、変数 no に読み込んだ整数値が 10 以上で、かつ、99 以下であれば、while 文の繰返しを抜け出ます。読み込む値の制限のために while 文と break 文を組み合わせている点は、**List 1-16**（p.24）と同じです。

▪ 値比較演算子の連続適用

連続適用された値比較演算子は“and 結合”とみなされますので、網かけ部の判定式は、次のように、簡潔に実現すべきです（'chap01/2digits2.py'）。

```
10 <= no <= 99                    # no >= 10 and no <= 99 と同じ
```

▪ ド・モルガンの法則

網かけ部の判定式を、**論理否定演算子**である **not 演算子**を使って書きかえると、次のようになります（'chap01/2digits3.py'）。

```
not(no < 10 or no > 99)           # no >= 10 and no <= 99 と同じ
```

オリジナルの判定式 `no >= 10 and no <= 99` が、繰返し終了のための《**終了条件**》であるのに対し、上記の式 `not(no < 10 or no > 99)` は、繰返しを続けるための《**継続条件**》の**否定**です。

すなわち、**Fig.1C-9** に示すイメージです。

＊

『“各条件の否定をとって、論理積・論理和を入れかえた式”の否定』が、もとの条件と同じになることを、**ド・モルガンの法則**（*De Morgan's laws*）といいます。この法則を一般的に示すと、以下のようになります。

① `x and y` の論理値と `not(not x or not y)` は等しい。

② `x or y` の論理値と `not(not x and not y)` は等しい。

※論理演算子については、**Column 8-2**（p.290）でも詳しく学習します。

```
while True:                          同じ          while True:
    # 終了条件（noは２桁）                          # 継続条件（noは２桁でない）の否定
    if no >= 10 and no <= 99:                       if not(no < 10 or no > 99):
        break                                           break
```

Fig.1C-9 繰返しの継続条件と終了条件

構造化プログラミング

単一の入り口点と単一の出口点とをもつ構成要素だけを用いて、階層的に配置してプログラムを構成する手法を、**構造化プログラミング**（*structured programming*）といいます。構造化プログラミングでは、順次、選択、繰返しの3種類の制御の流れを利用します。

▶ 構造化プログラミングは、**整構造プログラミング**とも呼ばれます。

多重ループ

本節のここまでのプログラムは、単純な繰返しを行うものでした。繰返しの中で繰返しを行うこともできます。

そのような繰返しは、ループの入れ子の深さに応じて、**2重ループ**、**3重ループ**、… と呼ばれます。もちろん、その総称は、**多重ループ**です。

九九の表

2重ループを用いたアルゴリズムの例として、《九九の表》を表示するプログラムを学習しましょう。**List 1-21** に示すのが、そのプログラムです。

List 1-21 chap01/multiplication_table.py

```
# 九九の表を表示

print('-' * 27)
for i in range(1, 10):
    for j in range(1, 10):
        print(f'{i * j:3}', end='')
    print()
print('-' * 27)
```

行ループ

列ループ

実行結果

```
---------------------------
  1  2  3  4  5  6  7  8  9
  2  4  6  8 10 12 14 16 18
  3  6  9 12 15 18 21 24 27
  4  8 12 16 20 24 28 32 36
  5 10 15 20 25 30 35 40 45
  6 12 18 24 30 36 42 48 54
  7 14 21 28 35 42 49 56 63
  8 16 24 32 40 48 56 64 72
  9 18 27 36 45 54 63 72 81
---------------------------
```

九九の表示を行う網かけ部のフローチャートを **Fig.1-19** に示しています。なお、右側の図は、変数 *i* と *j* の値の変化を表したものです。

外側の for 文（行ループ）では、変数 *i* の値を 1 から 9 までインクリメントします。その繰返しは、表の1行目、2行目、… 9行目に対応します。すなわち、**縦方向の繰返し**です。

その各行で実行される**内側の for 文（列ループ）**は、変数 *j* の値を 1 から 9 までインクリメントします。これは、各行における**横方向の繰返し**です。

変数 *i* の値を 1 から 9 まで増やす《**行ループ**》は9回繰り返されます。その各繰返しで、変数 *j* の値を 1 から 9 まで増やす《**列ループ**》が9回繰り返されます。《**列ループ**》終了後の改行の出力は、次の行へと進むための準備です。

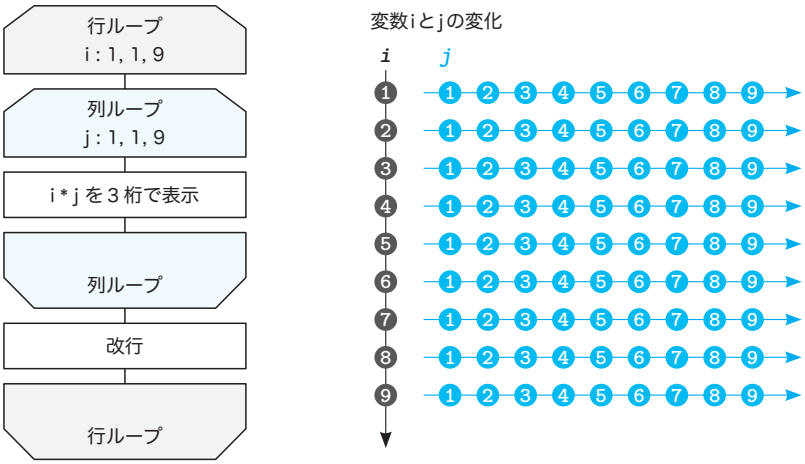


Fig.1-19　九九の表を表示するフローチャート

そのため、この２重ループでは、次のように処理が行われます。

- *i* が 1 のとき：*j* を 1 ⇨ 9 とインクリメントしながら 3 桁で 1 * *j* を表示して改行
- *i* が 2 のとき：*j* を 1 ⇨ 9 とインクリメントしながら 3 桁で 2 * *j* を表示して改行
- *i* が 3 のとき：*j* を 1 ⇨ 9 とインクリメントしながら 3 桁で 3 * *j* を表示して改行

　　… 中略 …

- *i* が 9 のとき：*j* を 1 ⇨ 9 とインクリメントしながら 3 桁で 9 * *j* を表示して改行

Column 1-13　乱数を生成する random.randint 関数

List 1-18（p.27）では、`random` モジュールに含まれる **`randint` 関数**を利用しました。

その **`random.randint(a, b)`** は、`a` 以上 `b` 以下の乱数を生成して（`a` 以上 `b` 以下の整数値の中から無作為に１個の整数値を抽出して）、その値を返却します（**Fig.1C-10**）。

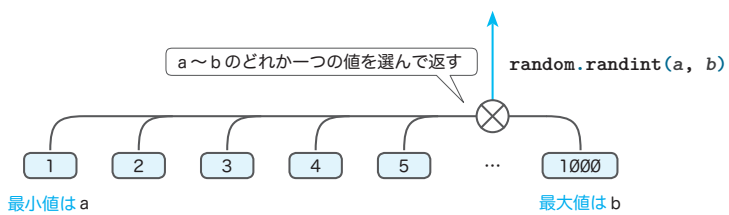


Fig.1C-10　random.randint 関数による乱数の生成

直角三角形の表示

2重ループを応用すると、記号文字を並べて三角形や四角形などの図形の表示が行えます。**List 1-22** に示すのは、左下側が直角の三角形を表示するプログラムです。

List 1-22 chap01/triangle_lb.py

```
# 左下側が直角の二等辺三角形を表示

print('左下直角の二等辺三角形')
n = int(input('短辺の長さ：'))

for i in range(n):
    for j in range(i + 1):
        print('*', end='')
    print()
```

行ループ

列ループ

実行例

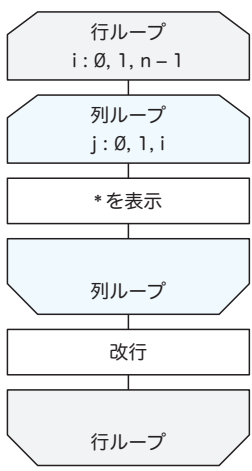
```
左下直角の二等辺三角形
短辺の長さ：5
*
**
***
****
*****
```

直角三角形の表示を行う網かけ部のフローチャートを **Fig.1-20** に示しています。右側の図は、変数 i と j の値の変化を表したものです。

実行例のように、n の値が 5 である場合を例にとって、処理がどのように行われるかを考えていきましょう。

外側の for 文（行ループ）では、変数 i の値を 0 から n - 1 すなわち 4 までインクリメントします。これは、三角形の各行に対応する**縦方向の繰返し**です。

その各行で実行される**内側の for 文（列ループ）**は、変数 j の値を 0 から i までインクリメントしながら表示を行います。これは各行における**横方向の繰返し**です。



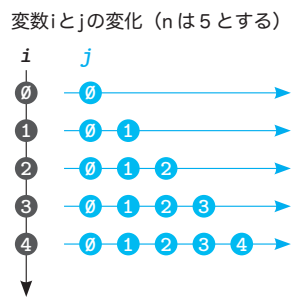


Fig.1-20 左下側が直角の二等辺三角形を表示するフローチャート

そのため、この2重ループでは、次のように処理が行われます。

- *i* が 0 のとき：*j* を 0 ⇨ 0 とインクリメントしながら '*' を表示して改行　　*
- *i* が 1 のとき：*j* を 0 ⇨ 1 とインクリメントしながら '*' を表示して改行　　**
- *i* が 2 のとき：*j* を 0 ⇨ 2 とインクリメントしながら '*' を表示して改行　　***
- *i* が 3 のとき：*j* を 0 ⇨ 3 とインクリメントしながら '*' を表示して改行　　****
- *i* が 4 のとき：*j* を 0 ⇨ 4 とインクリメントしながら '*' を表示して改行　　*****

三角形を上から第 0 行～第 *n* - 1 行とすると、第 *i* 行目に *i* + 1 個の '*' を表示して、最終行である第 *n* - 1 行目には *n* 個の '*' を表示するわけです。

＊

次は、右下側が直角の二等辺三角形を表示するプログラムを作ります。**List 1-23** に示すのが、そのプログラムです。

List 1-23　chap01/triangle_rb.py

```
# 右下側が直角の二等辺三角形を表示

print('右下直角の二等辺三角形')
n = int(input('短辺の長さ：'))

for i in range(n):
    for _ in range(n - i - 1):
        print(' ', end='')
    for _ in range(i + 1):
        print('*', end='')
    print()
```

実行例

```
右下直角の二等辺三角形
短辺の長さ：5
    *
   **
  ***
 ****
*****
```

先ほどのプログラムよりも複雑です。というのも、記号文字 * に先立って、適切な個数のスペースの出力が必要だからです。そのため、for 文の中には、二つの for 文が入っています。

- **1番目の for 文**：*n* - *i* - 1 個の空白文字 ' ' を表示
- **2番目の for 文**：　*i* + 1 個の記号文字 '*' を表示

どの行においても、空白文字と記号文字 * の個数の合計は *n* です。

＊

本プログラムでは、内側の二つの for 文のカウンタ用の変数名を、1個の下線文字 _ としています（このスタイルは p.21 の **List 1-13** で学習しました）。

▶　変数名を下線とすることによって、ループ本体の中でカウンタ用変数の値を使わないことを、プログラムの読み手に伝えるのでした。

なお、左ページの **List 1-22** のカウンタ *j* の名前も _ にできます（'chap01/triangle_lb2.py'）。

Column 1-14 Pythonの変数について

Pythonでは、データ、関数、クラス、モジュール、パッケージなど、ありとあらゆるものが**オブジェクト**（*object*）です。オブジェクトは、**型**（*type*）をもつとともに、**記憶域**（*storage*）を占有します。

すべてがオブジェクトであるPythonでは、変数も独特です。Pythonの変数は、値を保持しません。

たとえば、`x = 17`という代入の結果、『変数`x`が17という値を保持する』ことにはならないのです。

『変数とは、値を格納する《箱》のようなものである。』というたとえが使われることがありますが、まったく当てはまりません。

Pythonの変数とオブジェクトを少し大まかに説明すると、次のようになります（厳密ではありません）。

変数は、オブジェクトを参照する、すなわち、オブジェクトに結び付けられた名前にすぎない。
すべてのオブジェクトは、記憶域を占有して型をもつだけでなく、他のオブジェクトと識別できるようにするための識別番号（アイデンティティー）（*identity*）**すなわち同一性をもっている。**

基本対話モード（インタラクティブシェル）で確認しましょう。

```
>>> n = 17↵
>>> id(17)↵
140711199888704      ← 17の識別番号
>>> id(n)↵
140711199888704      ← n の識別番号（17の識別番号と同じ）
```

注意：表示される値は、環境などによって異なります。これ以降も同様です。

変数`n`に17を代入した後で、組込み関数である**`id`関数**を2回呼び出しています。`id`関数は、オブジェクトに固有の値である《識別番号（同一性）》を返却する関数です。

Pythonの代入では、**Fig.1C-11** aのような値のコピーは行われません。

図bに示すように、まず値17の`int`型オブジェクトが存在していて、そのオブジェクトを参照するように、`n`という名前を**結び付ける**（*bind*）だけです。

識別番号（同一性）が代入される結果、整数リテラル17の識別番号と、変数`n`の識別番号が同じ値となるのです。このことは、**“17と`n`は同一である”**と表現されます。

あえて《箱》という表現を使うのであれば、17という`int`型のオブジェクトが箱です。変数は、値を格納する箱ではありません。変数`n`は、物理的なオブジェクト（箱）である`int`型の17と結び付いた、単なる名前にすぎません。

なお、`n`の値を17以外の値に更新すると、その値をもつオブジェクトが生成され、それを参照するように、`n`の参照先が更新されます。当然、`n`の識別番号は更新されます（**Column 2-1**：p.46）。

それでは、変数が名前にすぎないことを、**List 1C-3**のプログラムで確認しましょう。

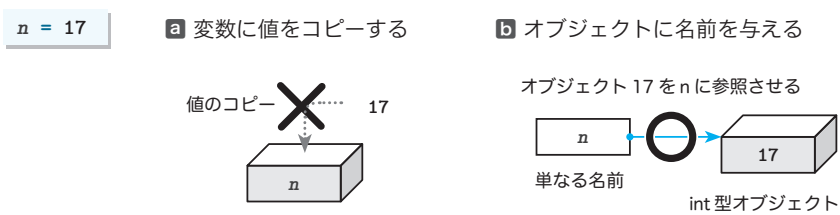


Fig.1C-11 変数への値の代入

List 1C-3 object_function.py

```
# 関数内外で定義された変数とオブジェクト
# オブジェクトと変数の識別番号を表示
n = 1               # 広域変数（関数内外で利用できる）
def put_id():
    x = 1           # 局所変数（関数内でのみ利用できる）
    print(f'id(x) = {id(x)}')
print(f'id(1) = {id(1)}')
print(f'id(n) = {id(n)}')
put_id()
```

実行結果一例

```
id(1) = 140736956818064
id(n) = 140736956818064
id(x) = 140736956818064
```

このプログラムでは、二つの変数が定義されています。

変数 n は、（すべての）関数の外で定義された、プログラム全体で利用できる**広域変数**です。

もう一つの変数 x は、関数 put_id の中で定義された、関数の中でのみ利用できる**局所変数**です。

Fig.1C-12 に示すように、いずれの変数も、int 型オブジェクト 1 を参照する名前にすぎないことが実行結果からも確認できます。

というのも、1、変数 n、変数 x の識別番号が、すべて同じ値となっているからです。

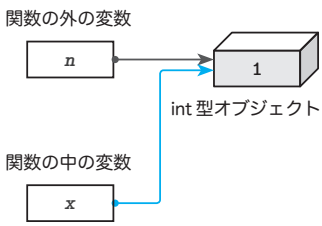


Fig.1C-12 関数内外の変数

＊

C言語などの言語では、関数の中で定義された局所変数は、関数の実行開始などのタイミングで生成されて、関数の実行終了などのタイミングで破棄されるのが一般的です。これは、**自動記憶域期間（自動記憶寿命）**と呼ばれます。また、すべての関数の外で定義された変数は、プログラムの実行を通じて、最初から最後まで存在します。これは、**静的記憶域期間（静的記憶寿命）**と呼ばれます。

本プログラムから、Python には、そのような概念は存在しない、ということが分かります。1 という整数オブジェクトは、関数 put_id の実行とは無関係に存在し続けるからです。

Python では、関数の実行開始や終了に伴って、オブジェクトが生成されたり、破棄されたりすることはありません。記憶域期間（記憶寿命）という概念が Python に存在し得る余地は、まったくありません。

さて、変数の値が変わるだけで別のオブジェクトが生成されることから、たとえば、for 文で 1 から 100 まで繰り返すだけで、100 個のオブジェクトが生成されます。**List 1C-4** で確認しましょう。

List 1C-4 for.py

```
# 1から100まで繰返して表示

for i in range(1, 101):
    print(f'i = {i:3}  id(i) = {id(i)}')
```

実行結果一例

```
i =   1  id(i) = 140706589794960
i =   2  id(i) = 140706589794992
i =   3  id(i) = 140706589795024
i =   4  id(i) = 140706589795056
   … 以下省略 …
```

1 から 100 までを並べたイテラブルオブジェクトが生成され、変数 i に1個ずつ取り出されます（**Column 1-11**：p.25）。

そのため、変数 i の値と識別番号の両方が更新されていく（100 個のオブジェクトを参照するように、変数 i の参照先が更新される）ことが、実行結果からも分かります。

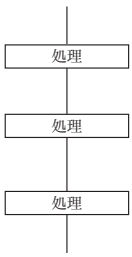
章末問題

各章の章末に示しているのは、基本情報技術者試験（旧・第2種情報処理技術者試験）で出題された問題の一部です。章末問題の解答は、p.337 に示しています。

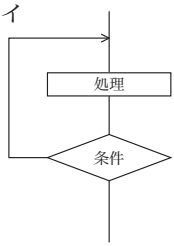
▪ 平成9年度（1997年度）秋期 午前 問37

次のプログラムの制御構造のうち、選択構造はどれか。

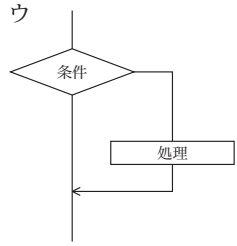
ア



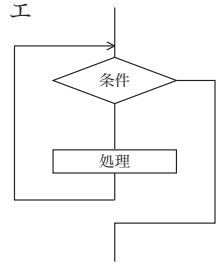
イ



ウ



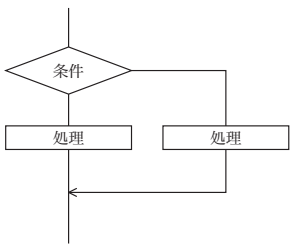
エ



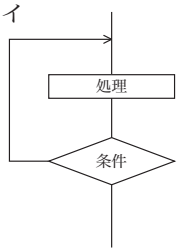
▪ 平成18年度（2006年度）春期 午前 問36

プログラムの制御構造のうち、while 型の繰返し構造はどれか。

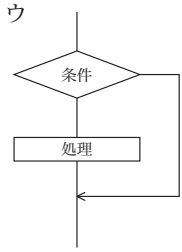
ア



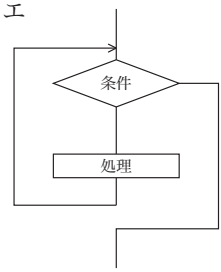
イ



ウ



エ



▪ 平成16年度（2004年度）秋期 午前 問41

プログラムの制御構造に関する記述のうち、適切なものはどれか。

ア “後判定繰返し”は、繰返し処理の先頭で終了条件の判定を行う。

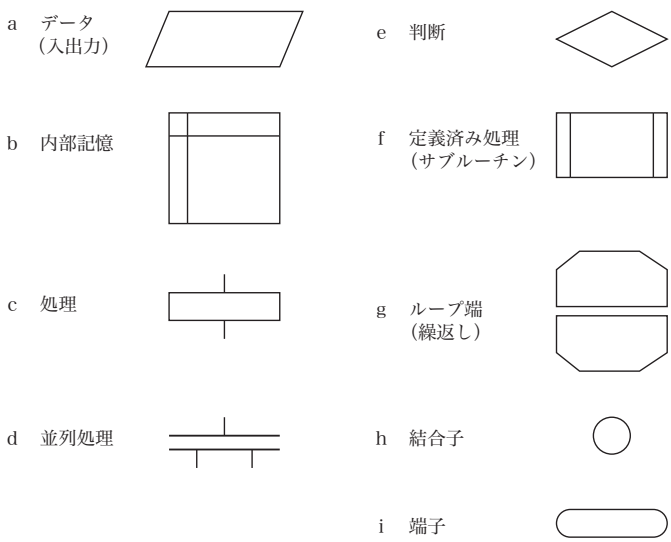
イ “双岐選択”は、前の処理に戻るか、次の処理に進むかを選択する。

ウ “多岐選択”は、二つ以上の処理を並列に行う。

エ “前判定繰返し”は、繰返し処理の本体を1回も実行しないことがある。

▪ 平成6年度(1994年度)秋期 午前 問41

整構造プログラミング（構造化プログラミング）における基本3構造と呼ばれるものに、最も密接な関係のある流れ図記号の組合せはどれか。

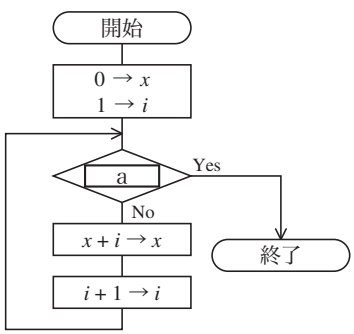


ア　a，b，c　　イ　a，b，d　　ウ　c，e，g

エ　d，e，g　　オ　f，h，i

▪ 平成12年度(2000年度)春期 午前 問16

次の流れ図は、1からN（$N \geqq 1$）までの整数の総和（$1 + 2 + \cdots + N$）を求め、結果を変数xに入れるアルゴリズムを示している。流れ図中のaに当てはまる式はどれか。



ア　$i = N$　　イ　$i < N$　　ウ　$i > N$　　エ　$x > N$