11章: 探索アルゴリズム

アルゴリズム

アルゴリズムとは、問題の解を求めるための処理や手順を定めたものである。アルゴリズムの例として、計算の過程を明示的に書き出しながら行う計算方法である筆算が挙げられる。例えば、2桁の2数の加算を筆算で行うには、以下のようなアルゴリズムにより計算する。

1. 加算する2数を、それぞれの1の位と10の位が縦に揃うように、上下に並べて書く。
2. 2数の1の位を足し合わせる。得られた数の1の位を、和の1の位の場所に書く。得られた数の10の位を、和の10の位の場所に小書きする。
3. 2数の10の位と、2.で小書きした10の位の値を足し合わせる。得られた数の10の位を、和の1の位の場所に書く。得られた数の100の位を、和の100の位の場所に書く。

このようにアルゴリズムは明確な手順として定義され、定められた手順に従うことで機械的に目的の解を得られるようになっている。日本産業規格（JIS）では、アルゴリズムは「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成る集合」と定義される。

アルゴリズムは問題の解法の手順をまとめたものであるため、特定のプログラミング言語やコンピュータのアーキテクチャによらず適用可能である。また、世の中の典型的な問題には、それを解決するための良く知られたアルゴリズムが存在している。そのため、アルゴリズムを学ぶことは、広く応用可能な優れた解法を知ることにつながる。本章では、アルゴリズムの代表例の一つとして、データの集合から目的のデータを探し出すための手法である探索アルゴリズムを解説する。

アルゴリズムの性能

ある問題に対して、解を与えるアルゴリズムは唯一ではなく複数考えられるが、その性能はアルゴリズムによって異なる。選択するアルゴリズムによって、わずかな時間やリソースで済むこともあれば、膨大な時間やリソースが必要になることもあるため、アルゴリズムを使用するときはその性能を評価し、適切なアルゴリズムを選択しなければならない。アルゴリズムの世界では、その良し悪しにより何百万倍もの性能差が出ることも珍しくなく、アルゴリズムがプログラムの性能の大部分を左右することが往々にしてある。特に、大規模なデータを扱ったり、大量の処理を行ったりするような問題においては、アルゴリズムの性能に対する理解が不可欠である。

アルゴリズムの性能は主に、計算に要する時間を表す「時間計算量」（time complexity）と、計算に必要とする記憶領域を表す「空間計算量」（space complexity）により評価される。

時間計算量

アルゴリズムは問題を解くための抽象化された手順であり、特定のプログラミング言語やコンピュータアーキテクチャなどの実行環境に依存しないものであるため、その性能を表す指標も実行環境によらない汎用的な尺度である必要がある。アルゴリズムの計算に要する時間を表すことを考えたとき、実際のプログラムの実行時間を計測するような方法を採ってしまうと、そのプログラムの実行環境により差が出てしまい、汎用的な尺度にならない。そこで、アルゴリズムをステップごとに分解し、ステップごとにかかる時間は実行環境によって決まる定数であるとみなして、各ステップの実行回数の総和をとることでアルゴリズムの計算時間の尺度とする。このようにして表される尺度を、時間計算量と呼ぶ。

アルゴリズム中の各ステップの実行回数は、処理の対象となる入力データによって左右されることが多い。例えば、要素の集合で表されるデータの各要素に対して処理を行うようなアルゴリズムでは、要素の数が増えるに伴って各ステップの実行回数も増加するほか、データ中の要素の値によって処理が変わる場合はそれに伴って各ステップの実行回数も変化する。そのため、アルゴリズムの時間計算量は、入力データのサイズ n の関数として表したうえで、そのアルゴリズムにおいて最も時間がかかるような入力データを対象としたときの時間計算量である「最悪時間計算量」（worst-case time complexity）や、同一のサイズのデータが与えられたときの平均的な時間計算量である「平均時間計算量」（average-case time complexity）を用いるのが一般的である。

アルゴリズムの時間計算量を入力データのサイズ n の関数として表したとき、$f(n) = n^2 + n$ のように、n が含まれる項が複数存在することがある。この関数においてn が変化したとき、n が十分に大きければ、時間計算量 $f(n)$ の変化量は n^2 によって支配的に決定される。例えば、入力データのサイズ n が10倍になったとき、n^2 は100倍になるのに対し、n は10倍にしかならず、$f(n)$ の増加量は110倍となるため n^2 の増加量で近似できる。このように、アルゴリズムの時間計算量は、関数の式に含まれる項のうち増加の度合いが最大の項による影響を最も大きく受ける。そういった項を主要項（leading term）といい、時間計算量を表すときは主要項のみに注目することが多い。また、$f(n) = 2n^2$ のように主要項に係数が含まれる場合もあるが、n が十分に大きいとき、 n の変化に伴う $f(n)$ の変化量と比べて係数の影響は無視できるほどに小さいため、主要項の中でも係数を除いた部分が重要となる。

主要項のみに注目して時間計算量を表す記法として、「O記法」（O-notation）または「オーダー記法」（order notation）と呼ばれる記法が一般によく用いられる。時間計算量を表す関数 $f(n)$ の主要項の係数を除いたものを $g(n)$ としたとき、$f(n)$ をO記法で $O(g(n))$ と表す。例

えば、主要項が n^2 の関数を O記法で表すと $O(n^2)$ となる。つまり、時間計算量が $O(n^2)$ で表されるとき、そのアルゴリズムの計算時間はデータのサイズ n の2乗で近似されることを意味する。

アルゴリズムの時間計算量をO記法で表すと、一般的なアルゴリズムであれば、ほとんどの場合は以下のいずれかになる。

- $O(1)$、$O(\log n)$：時間計算量がこれらの式で表されるとき、n が増加してもほとんど時間計算量は増加しないため、データのサイズが大きくても十分に高速に動作する。
- $O(n)$、$O(n \log n)$：時間計算量は n に対してほぼ線形に増加するため、データのサイズが比較的大きくても現実的な速度で動作する。
- $O(n^2)$：n の増加に対して時間計算量が大きく増加するため、比較的小さなデータサイズの場合に限り実用性がある。
- $O(n^3)$、$O(2^n)$：n の増加に対して時間計算量が非常に急速に増加するため、かなり小さなデータサイズの場合に限り実用性がある。

このように、時間計算量をO記法で表すことで、そのアルゴリズムの計算時間の特性を容易に把握できる。

空間計算量

アルゴリズムの性能を評価するとき、計算時間も重要であるが、アルゴリズムの計算に要する記憶領域についても同時に考える必要がある。高速に動作するアルゴリズムであっても、膨大な記憶領域が必要となる場合、現実的には動作が困難になる可能性がある。ここまで述べてきた時間計算量と同様に、アルゴリズムが使用する記憶領域に対しても、汎用的な尺度として空間計算量と呼ばれるものを用いる。

空間計算量は、アルゴリズムの計算に必要となる記憶領域を表したもので、時間計算量と同様に、一般的に入力データのサイズ n の関数として表される。O記法の用法についても時間計算量と同様で、空間計算量の関数 $f(n)$ において主要項の係数を除いた部分が $g(n)$ のとき、空間計算量はO記法で $O(g(n))$ と表される。空間計算量もO記法により表すのが一般的であり、これによりアルゴリズムが必要とする記憶領域の特性の把握が容易になる。

探索アルゴリズム

探索とは、特定のデータの集合から目的のデータを探し出すことである。探索には、さまざまなアルゴリズムが用意されており、それぞれに特徴がある。

線形探索というアルゴリズムでは、一つずつ要素を取り出して探索する。

整列済のデータに対しては、二分探索というアルゴリズムが使用できる。二分探索は、データを半分に分割していき探索するアルゴリズムであり、データの要素数が大きくなっても効率が良いため、高速であるとされている。

また、ハッシュテーブルというデータ構造を使用することで、探索を高速化することができる。ハッシュテーブルは、ハッシュ値を使用して、データを指定した位置に格納することで、探索を高速化するものである。

データの特性や使用するプログラムの目的によって最適なアルゴリズムは異なるため、状況に応じて適切なアルゴリズムを選択する必要がある。今回は、探索アルゴリズムの中から線形探索と二分探索について解説する。

線形探索

線形探索とは、探索アルゴリズムの一つで、あるデータを探すために、順番に要素を一つずつ調べていく探索方法である。そのため、線形探索を使用すると、データが整列されているかどうかに関わらず探索を行うことができる。

また、線形探索は単純なアルゴリズムであるため、実装が簡単である。しかし、データが大量にある場合は、探索するデータが見つかるまでにかかる時間が長くなるため効率が悪い。そのため、データが大量にある場合は、二分探索やハッシュ探索など、より効率の良い探索アルゴリズムを使用することが推奨される。

以下は、線形探索を用いてデータを探索する一般的な手順である。

1. 探索対象のデータから最初の要素を取り出す。
2. 取り出した要素が探索するデータであるかを判定する。
3. 探索するデータである場合、探索を終了する。探索するデータでない場合、次の要素を取り出す。
4. 手順3. の操作を繰り返し、探索するデータが見つかるか、データの集合を最後まで調べるまで行う。

このように線形探索では、要素を順番に一つずつ調べていくことで目的のデータを探し出す。

線形探索の時間計算量は、配列やリストの要素数がn個ある場合、最悪の場合には全ての要素を比較する必要があるため、最悪のケースと平均

のケースともに$O(n)$である。

線形探索では配列やリストをコピーするような処理は行わず、配列やリストの要素を一つずつ走査するだけのため、空間計算量は$O(1)$である。

前述の線形探索の手順はPythonを用いて以下のように記述できる。

```
def sequential_search(data, target):
    # データの中から一つずつの要素を取り出す
    for element in data:
        # 取り出した要素が探索するデータであるかを判定する
        if element == target:
            # 探索するデータである場合、探索を終了しTrueを返す
            return True
    # データを最後まで調べて探索するデータが見つからなかった場合、Falseを返す。
    return False

# 探索対象のデータ
li = [5, 2, 4, 3, 1]
# 探索したい値
num = 4

if sequential_search(li, num):
    print("探索成功")
else:
    print("探索失敗")
```

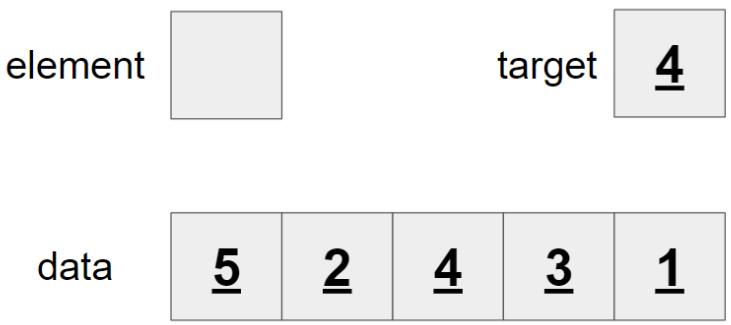
```
探索成功
```

上記のサンプルプログラムでは、リスト `li` の中に変数 `num` の値が存在するか、線形探索を行う `sequential_search()` 関数に渡すことで確認している。 `sequential_search()` 関数では、値が存在したら `True` を、存在しなければ `False` を返し、最終的に目的の値が探索対象のデータに存在すれば `探索成功` と出力し、存在しなければ `探索失敗` と出力する。

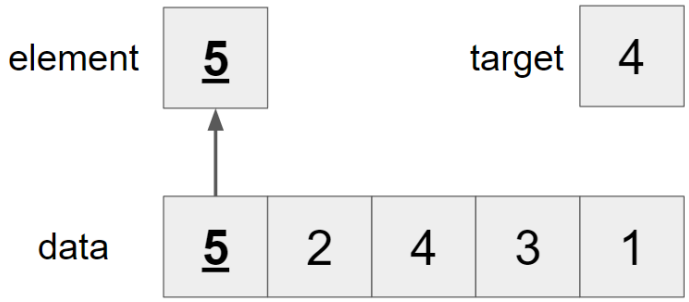
上記のサンプルプログラムでは、リスト `li` には `[5, 2, 4, 3, 1]` が格納されており、変数 `num` には `4` が格納されている。

リスト `li` の値と変数 `num` を `sequential_search()` 関数に渡した後の、 `sequential_search()` 関数の中の処理を詳しく見ていこう。

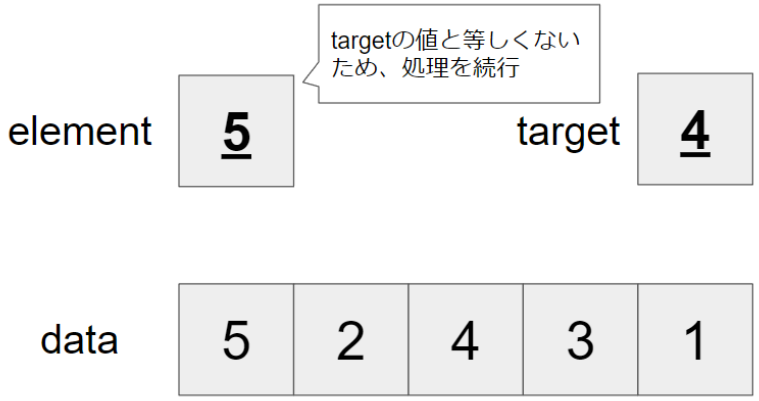
`sequential_search()` 関数の中では、探索対象のデータは `data` に、探索したい値は `target` に格納される。
`sequential_search()` 関数を呼び出す際に、第一引数にリスト `li` を、第二引数に変数 `num` を指定しているため、 `data` には `[5, 2, 4, 3, 1]` が、変数 `target` には `4` が格納されている。



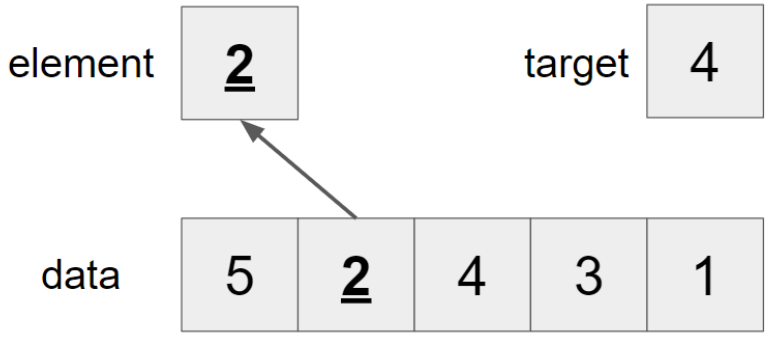
`for` 文では、 `data` に格納されている要素を先頭から一つずつ取り出して、 `element` に格納し処理を行うため、まず、 `data` の先頭の要素である `5` を `element` に格納する。



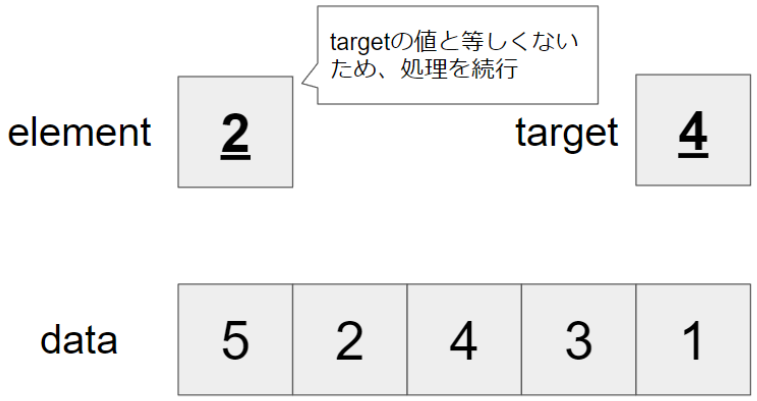
続いて、 `element` に格納されている値が、 `target` に格納されている値と等しいか判定する。 `element` には `5` が、 `target` には `4` が格納されており、異なる値であるため、処理を続ける。



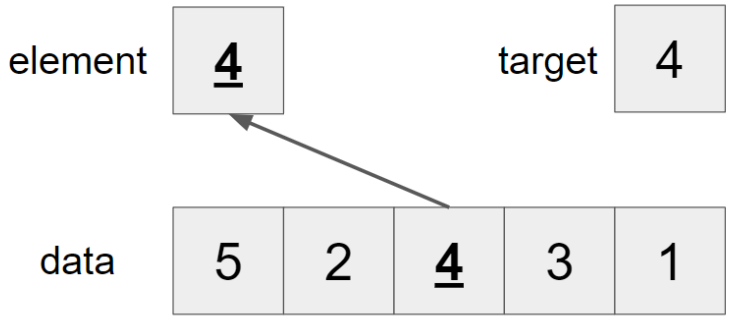
`for` 文の1回目の処理が終了し、 `element` に、 `data` の次の要素である `2` を格納する。



続いて、 `element` に格納されている値が、 `target` に格納されている値と等しいか判定する。 `element` には `2` が、 `target` には `4` が格納されており、異なる値であるため、処理を続ける。



`for` 文の2回目の処理が終了し、 `element` に、 `data` の次の要素である `4` を格納する。



続いて、 `element` に格納されている値が、 `target` に格納されている値と等しいか判定する。elementには `4` が、targetには `4` が格納されており、等しい値であるため、処理を終了し、 `True` を返す。

以上で、 `sequential_search()` 関数は終了し、 `sequential_search()` 関数の実行結果として、 `True` が返ってくるため、 `if` 文の条件式が真となり、 `探索成功` が出力される。

二分探索

二分探索は探索アルゴリズムの一つで、データを探すために、探索範囲の中央のデータと目的のデータの大小を比較して、目的のデータが探索範囲の前半分にあるか後ろ半分にあるかを判定し、探索する範囲を絞り込んでいく探索方法である。

二分探索を行うためには、データが整列されている必要がある。二分探索は、線形探索と比較して、特にデータが大量にある場合に効率が良い。データを半分に分けて、探索する範囲を絞り込むことで、探索するデータが見つかるまでにかかる時間を短縮することができる。ただし、二分探索のアルゴリズムは、線形探索よりも実装が複雑である。また、元々データが整列されていない場合は、データを整列する処理が別途必要になる。

以下は、二分探索を用いてデータを探索する一般的な手順である。

1. 探索範囲のデータの中央にある要素を取り出す。
2. 取り出した要素が探索するデータであるかを判定する。
3. 取り出した要素が探索するデータである場合は、探索を終了する。取り出した要素が探索するデータよりも小さい場合は、探索する範囲を取り出した要素よりも大きい側に絞り込む。 取り出した要素が探索するデータよりも大きい場合は、探索する範囲を取り出した要素よりも小さい側に絞り込む。
4. 手順1. から3. の手順を、探索する範囲が1つになるまで繰り返す。
5. 探索する範囲が1つになった場合でも、探索するデータが見つからなかった場合、探索を終了する。

このように二分探索では、探索範囲を半分に絞り込んでいき目的のデータを探し出す。

二分探索の時間計算量は、最悪のケースと平均のケースともに$O(\log n)$である。探索を行うたびに、探索する範囲を半分にすることで、探索するデータが見つかるまでにかかる時間を短縮することができるためである。

二分探索では特に配列やリストをコピーするような処理は行わず、探索した範囲や取り出した要素を一時的に記憶するための領域しか使用しないため、空間計算量は$O(1)$である。

前述の二分探索の手順はPythonを用いて以下のように記述できる。

```python
def binary_search(data, target):
    # 探索する範囲を表す変数leftとrightを設定する
    left = 0
    right = len(data) - 1

    # leftがrightより大きくなるまで探索を続ける
    while left <= right:
        # 探索する範囲の中央の位置を取得する
        mid = (left + right) // 2
        # 探索する範囲の中央の位置の要素を判定する
        if data[mid] == target:
            # 探索するデータと等しい場合、探索を終了しTrueを返す
            return True
        elif data[mid] < target:
            # 探索するデータよりも小さい場合、探索する範囲を取り出した要素よりも大きい側に絞り込む
            left = mid + 1
        else:
            # 探索するデータよりも大きい場合、探索する範囲を取り出した要素よりも小さい側に絞り込む
            right = mid - 1

    # 探索するデータが見つからなかった場合、Falseを返す
    return False

# 探索対象のデータ
li = [1, 2, 3, 4, 5, 6, 7]
# 探索したい値
num = 2

if binary_search(li, num):
    print("探索成功")
else:
    print("探索失敗")
```

```
探索成功
```

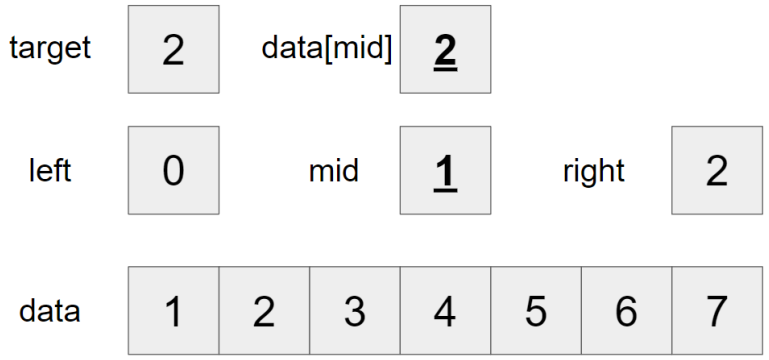
上記のサンプルプログラムでは、リスト `li` の中に変数 `num` の値が存在するか、二分探索を行う `binary_search()` 関数に渡すことで確認している。 `binary_search()` 関数では、値が存在したら `True` を、存在しなければ `False` を返し、最終的に目的の値が探索対象のデータに存在すれば `探索成功` と出力し、存在しなければ `探索失敗` と出力する。

上記のサンプルプログラムでは、リスト `li` には `[1, 2, 3, 4, 5, 6, 7]` と数字が整列されて格納されており、変数 `num` には `2` が格納されている。

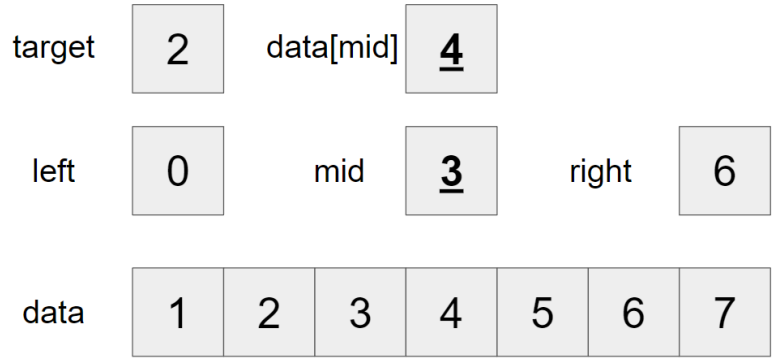
リスト `li` の値と変数 `num` を `binary_search()` 関数に渡した後の、 `binary_search()` 関数の中の処理を詳しく見ていこう。

`binary_search()` 関数の中では、探索対象のデータは `data` に、探索したい値は `target` に格納される。 `binary_search()` 関数を呼び出す際に、第一引数にリスト `li` を、第二引数に変数 `num` を指定しているため、 `data` には `[1, 2, 3, 4, 5, 6, 7]` が、変数 `target` には `2` が格納されている。

`left` では探索範囲の先頭の位置を、 `right` では探索範囲の末尾の位置をそれぞれ持つ。初めの探索範囲は、 `data` 全体であるため、 `left` には、 `data` の先頭を指すインデックスである `0` を、 `right` には `data` の末尾を指すインデックスである `dataの長さ - 1` をそれぞれ代入する。 `data` の長さは `7` であるため、 `right` には `6` が格納されている。

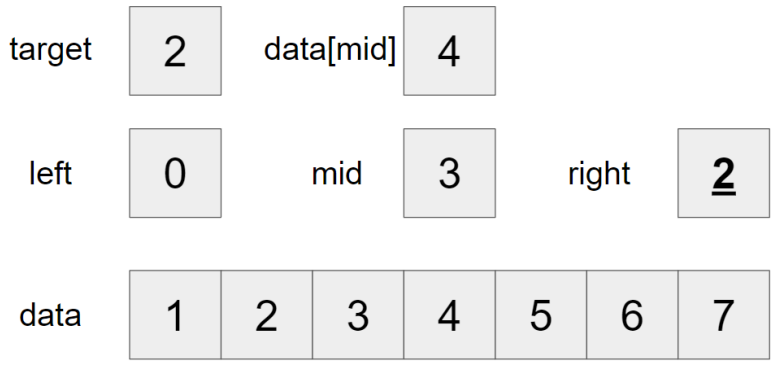


続いて、 `while` 文を使用して、 `left` が `right` よりも大きくなるまで繰り返し処理を行う。 `while` 文の中で、探索する範囲の中央の位置を取得するために、 `mid` を宣言し、 `left` と `right` を足した値を2で割った商を代入する。 `left` は `0` 、 `right` は `6` であるため、2で割ると `3` になる。そのため、 `mid` には、 `3` を代入する。 `mid` には `3` が格納されているため、 `data[mid]` の値は `4` である。

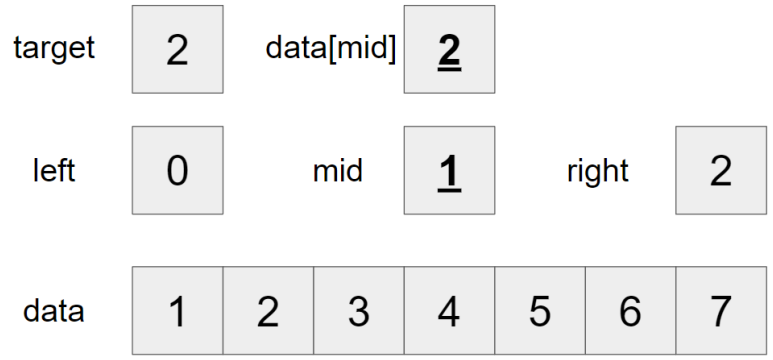


続く `if` 文では、探索する範囲の中央の位置の要素である `data[mid]` が `target` と等しいか判定する。 `data[mid]` が `target` と等しい場合、探索成功と判断し `True` を返して探索処理を終了する。 `data[mid]` が `target` よりも小さい場合、探索する範囲を `mid` の位置よりも末尾側に探索範囲を絞り込む。 `data[mid]` が `target` よりも大きい場合、探索する範囲を `mid` の位置よりも先頭側に探索範囲を絞り込む。

`data[mid]` の値は `4` で、 `target` の値 `2` よりも大きいため、 `right = mid - 1` を実行し、探索する範囲を `mid` の位置よりも先頭側に探索範囲を絞り込む。 `mid` には `3` が格納されているため、 `right` には新たに `2` が格納される。



新しい探索範囲の中の中央の位置を `mid` に再度代入する。 `left` は `0` 、 `right` は `2` であるため、2で割ると `1` になる。そのため、 `mid` には、 `1` を代入する。 `mid` には `1` が格納されているため、 `data[mid]` の値は `2` である。



改めて、 `data[mid]` が `target` と等しいか判定する。 `data[mid]` の値が `2` で、 `target` の値 `2` と等しいため、処理を終了し、 `True` を返す。

以上で、 `binary_search()` 関数は終了し、 `binary_search()` 関数の実行結果として、 `True` が返ってくるため、 `if` 文の条件式が真となり、 `探索成功` が出力される。

問題1

問題文

異なる整数を含む数列Aと、整数Kが与えられる。 **線形探索**を用いてAからKと等しい要素を検索して、検索過程を出力する `sequential_search()` 関数を作成せよ。

制約

- $0 \leq a, K \leq 100 \, (a \in A)$
- Aの要素は互いに異なる。
- 入力は全て整数である。

入力

入力は次の形式で標準入力から与えられる。

> $A_0 \ A_1 \ ... \ A_{N-1}$ （Nは正の整数） K

出力

AからKと等しい要素を見つけるまで、Aの要素とKを比較するたびに、空白区切りで要素の位置（0始まり）とAの要素の値を1行に出力する。なお、AからKと等しい要素を見つけた際も出力する必要がある。最後に、Aの値をKの中に見つけることができれば `探索成功` と、見つけることができなければ `探索失敗` と出力する。

入力例1

```
10 20 30 40 50
40
```

出力例1

```
0 10
1 20
2 30
3 40
探索成功
```

入力例2

```
10 20 30 40 50 60 70 80
62
```

出力例2

```
0 10
1 20
2 30
3 40
4 50
5 60
6 70
探索失敗
```

入力例3

```
50 60 70 80 10 30 20 40
20
```

出力例3

```
0 50
1 60
2 70
3 80
4 10
5 30
6 20
探索成功
```

解答の雛形

```python
def sequential_search(data, target):
    # 以下のpassを削除してから解答を入力
    pass


# 探索対象のデータ
A = [int(x) for x in input().split()]
# 探索したい値
K = int(input())

if sequential_search(A, K):
    print("探索成功")
else:
    print("探索失敗")
```

問題2

問題文

異なる整数を含む数列Aと、整数Kが与えられる。 **線形探索**を用いてAからKと等しい要素を検索して、比較した回数を出力する `sequential_search()` 関数を作成せよ。

制約

- $0 \leq a, K \leq 100\,(a \in A)$
- Aの要素は互いに異なる。
- 入力は全て整数である。

入力

入力は次の形式で標準入力から与えられる。

> $A_0\ A_1\ ...\ A_{N-1}$ （Nは正の整数） K

出力

AからKと等しい要素を見つけるまでに、Aの要素とKを比較した回数を一行目に出力する。なお、AからKと等しい要素を見つけた際も比較した回数としてカウントする。二行目に、Aの値をKの中に見つけることができれば `探索成功` と、見つけることができなければ `探索失敗` と出力する。

入力例1

```
10 20 30 40 50
40
```

出力例1

```
4
探索成功
```

入力例2

```
10 20 30 40 50 60 70 80
62
```

出力例2

```
8
探索失敗
```

入力例3

```
50 60 70 80 10 30 20 40
20
```

出力例3

```
7
探索成功
```

解答の雛形

```
def sequential_search(data, target):
    # 以下のpassを削除してから解答を入力
    pass


# 探索対象のデータ
A = [int(x) for x in input().split()]
# 探索したい値
K = int(input())

if sequential_search(A, K):
    print("探索成功")
else:
    print("探索失敗")
```

問題3

問題文

異なる整数を含む数列Aと、整数Kが与えられる。 **二分探索**を用いてAからKと等しい要素を検索して、検索過程を出力する `binary_search()` 関数を作成せよ。なお、二分探索において偶数個の要素から中央の位置を求める際は、添字が小さい方（切り捨て除算）の要素を中央とすること。

制約

- $0 \leq a, K \leq 100\,(a \in A)$
- Aの要素は互いに異なる。
- Aの要素は昇順に整列されている。
- 入力は全て整数である。

入力

入力は次の形式で標準入力から与えられる。

> A_0 A_1 ... A_{N-1} （Nは正の整数） K

出力

AからKと等しい要素を見つけるまで、Aの要素とKを比較するたびに、空白区切りで要素の位置（0始まり）とAの要素の値を1行に出力する。なお、AからKと等しい要素を見つけた際も出力する必要がある。最後に、Aの値をKの中に見つけることができれば `探索成功` と、見つけることができなければ `探索失敗` と出力する。

入力例1

```
10 20 30 40 50
40
```

出力例1

```
2 30
3 40
探索成功
```

入力例2

```
10 20 30 40 50 60 70 80
62
```

出力例2

```
3 40
5 60
6 70
探索失敗
```

解答の雛形

```python
def binary_search(data, target):
    # 以下のpassを削除してから解答を入力
    pass


# 探索対象のデータ
A = [int(x) for x in input().split()]
# 探索したい値
K = int(input())

if binary_search(A, K):
    print("探索成功")
else:
    print("探索失敗")
```

問題4

問題文

異なる整数を含む数列Aと、整数Kが与えられる。 **二分探索**を用いてAからKと等しい要素を検索して、比較した回数を出力する `binary_search()` 関数を作成せよ。なお、二分探索において偶数個の要素から中央の位置を求める際は、添字が小さい方（切り捨て除算）の要素を中央とすること。

制約

- $0 \leq a, K \leq 100\,(a \in A)$
- Aの要素は互いに異なる。
- Aの要素は昇順に整列されている。
- 入力は全て整数である。

入力

入力は次の形式で標準入力から与えられる。

> $A_0\ A_1\ ...\ A_{N-1}$ （Nは正の整数） K

出力

AからKと等しい要素を見つけるまでに、Aの要素とKを比較した回数を一行目に出力する。なお、AからKと等しい要素を見つけた際も比較した回数としてカウントする。二行目に、Aの値をKの中に見つけることができれば `探索成功` と、見つけることができなければ `探索失敗` と出力する。

入力例1

```
10 20 30 40 50
40
```

出力例1

```
2
探索成功
```

入力例2

```
10 20 30 40 50 60 70 80
62
```

出力例2

```
3
探索失敗
```

解答の雛形

```python
def binary_search(data, target):
    # 以下のpassを削除してから解答を入力
    pass


# 探索対象のデータ
A = [int(x) for x in input().split()]
# 探索したい値
K = int(input())

if binary_search(A, K):
    print("探索成功")
else:
    print("探索失敗")
```

12章: ソートアルゴリズム

ソートの基本

ソートとは、データを特定の順序に並べることである。データを特定の順序に並べることにより、データを効率的に検索したり、分析したりすることができるようになる。

データをソートすることで得られるメリットとして、次のようなことが挙げられる。

- データがより見やすくなる。
- データがより検索しやすくなる。

リスト内のデータを並べ替えることで、データをより見やすくすることができる。例えば、学生のテストの結果を成績順に並べ替えることで、成績の高低を一目で確認しやすくなる。

また、リスト内のデータを並べ替えることで、データを検索しやすくすることができる。例えば、学生の名前をあいうえお順に並び替えることで、特定の学生の名前を見つけやすくなる。

データをソートするときに使用する順序は、昇順または降順のいずれかである。ソートする際には、昇順または降順のどちらかを、選択してデータを並びかえることになる。

昇順とは、データを小さいものから大きいものの順番に並べることを指す。例えば、学生の成績を昇順に並べると、最低点から最高点の順番に並ぶことになる。以下は、昇順にデータが並んでいるリストの例である。インデックス `0` に一番小さい値が格納されており、インデックスが大きくなるにつれて、格納されている値も大きくなっている。

昇順のリストの例

インデックス	0	1	2	3	4
値	20	35	50	70	80

降順とは、データを大きいものから小さいものの順番に並べることを指す。例えば、学生の成績を降順に並べると、最高点から最低点の順番に並ぶことになる。以下は、降順にデータが並んでいるリストの例である。インデックス `0` に一番大きい値が格納されており、インデックスが大きくなるにつれて、格納されている値は小さくなっていく。

降順のリストの例

インデックス	0	1	2	3	4
値	80	70	50	35	20

昇順と降順のどちらを選択するかは、目的やニーズによって異なる。

データをソートする方法として、さまざまなアルゴリズムが用意されており、それぞれに特徴がある。一般的なアルゴリズムとしては、バブルソート、選択ソート、挿入ソート、マージソート、クイックソートなどがある。それぞれのアルゴリズムは、データのサイズや特性、使用する言語などによって、最適なものが異なるため、状況に応じて適切なアルゴリズムを選択する必要がある。

今回は、ソートアルゴリズムの中からバブルソート、選択ソート、挿入ソート、バブルソートについて解説する。

バブルソート

バブルソートは、簡単なソートアルゴリズムの一つである。隣り合った2つのデータを比較して、順番を入れ替えることを繰り返して、データの並び替えを行う。バブルソートはソートアルゴリズムとしての効率は良くないが、直感的に分かりやすいアルゴリズムであるため、まず初めに紹介する。

以下は、バブルソートを用いてデータを昇順にソートする一般的な手順である。

1. リストの末尾から順番に、隣り合った要素を２つずつ比較する。
2. 前の要素が後ろの要素よりも大きい場合は、2つの要素を入れ替える。
3. リスト内の全ての要素を比較し入れ替えを終えると、先頭の要素は最小の要素になるため、比較の対象から除外し、リストの末尾から再度比較し入れ替えを行う。
4. 上記のステップを繰り返し、リスト内の要素を全てソートし終えるまで続ける。

バブルソートの時間計算量は、リストの要素数がnのとき、最悪の場合には、要素を比較する回数は$n(n-1)/2$回になるため、$O(n^2)$となる。最良の場合には、すべての要素が既に昇順に並んでいる場合であり、この場合には、要素を比較する回数は$n-1$回になるため、$O(n)$となる。バブルソートの平均の時間計算量は、$O(n^2)$である。

バブルソートでは、ソートする要素を格納する分の領域があればソートができるので、空間計算量は$O(n)$である。また、要素を交換するために、一時的に要素を保存するための変数が必要であるため、バブルソート自体で補助的に必要となる空間計算量は$O(1)$である。

前述のバブルソートの手順はPythonを用いて以下のように記述できる。

```python
def bubble_sort(data):
    # リストの要素数を取得
    length = len(data)
    # 一度内側のfor文を繰り返すごとに先頭から一つずつ要素をソート対象から外すループ
    for i in range(length - 1):
        # 末尾から隣り合った要素を比較し、左側の要素が大きければ交換する処理
        for j in range(length - 1, i, -1):
            if data[j - 1] > data[j]:
                tmp = data[j]
                data[j] = data[j - 1]
                data[j - 1] = tmp
    # 並び替えたリストを返す
    return data

# 使用例
li = [4, 1, 3, 2]
sorted_li = bubble_sort(li)
print(sorted_li)
```

```
[1, 2, 3, 4]
```

上記のサンプルプログラムは、リストの要素をバブルソートを使って昇順に並び替える例である。このプログラムでは、バブルソートを行う `bubble_sort()` 関数を定義している。この関数では、引数として与えられたリストを昇順に並べ替える処理を行い、並べ替えた後のリストを返している。

まず、 `bubble_sort()` 関数では、 `length` を定義し、引数として与えられたリストの要素数を代入する。次に、 `for` 文を使用して、要素を比較する回数分のループを行う。この外側の `for` 文の中では、内側の `for` 文を使用して、要素を比較する処理を行う。この内側の `for` 文では、隣り合った要素を比較して、左側の要素が大きければ、隣り合った要素を交換する処理を行う。

上記のサンプルプログラムでは、リスト `li` には `[4, 1, 3, 2]` が格納されている。

リスト `li` の値を `bubble_sort()` 関数に渡した後の、 `bubble_sort()` 関数の中の処理を詳しく見ていこう。

まず、 `length = len(data)` で、 `length` に `data` に格納されているリストの要素数を代入している。 `bubble_sort()` 関数を呼び出すときに引数として、 `li` を指定しているため、 `data` には、 `[4, 1, 3, 2]` が格納されている。 `[4, 1, 3, 2]` の要素数は `4` であるため、 `length` には `4` が代入される。

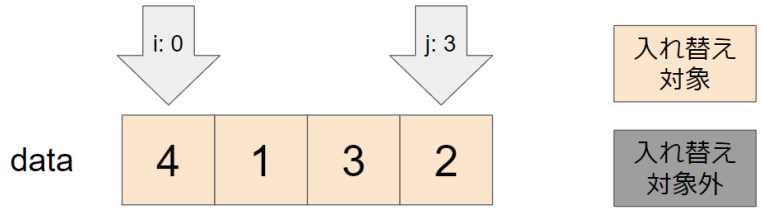
外側の `for` 文で、 `i` に `0` から `length - 1` を順番に格納していく。まず、 `i` には `0` を格納する。 `i` は比較する範囲の先頭を指す。

内側の `for` 文で、 `j` に `length - 1` から `i` より1つ大きい数まで `1` ずつ減らした値を順番に格納していく。

内側の `for` 文をJavaに置き換えると、以下のようなイメージである。

```
for(int j = length - 1; j > i; j--){
    <処理>
}
```

まず、 `j` には `length - 1` 、すなわち `3` を格納する。 `j` は比較する範囲の末尾を指す。



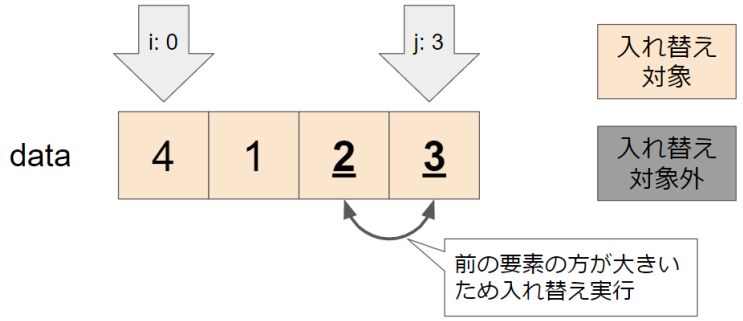
末尾から隣り合った要素を比較して、前の要素（ `data[j - 1]` ）が後ろの要素（ `data[j]` ）よりも大きければ、以下の処理で、値の入れ替えを行う。 `tmp` は値の入れ替えために一時的に使用する変数である。

```
tmp = data[j]
data[j] = data[j - 1]
data[j - 1] = tmp
```

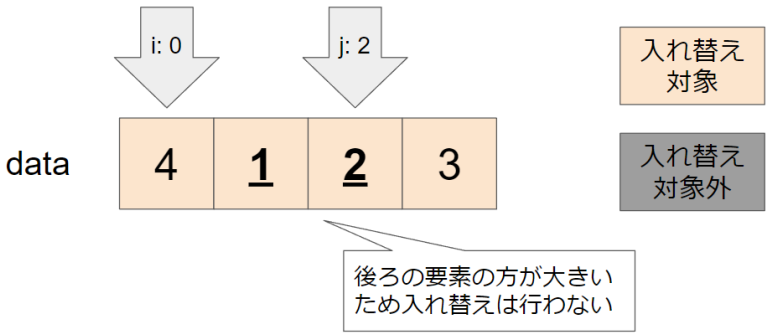
補足ではあるが、Pythonでは以下のように記述しても、値を入れ替えることができる。

```
data[j], data[j - 1] = data[j - 1], data[j]
```

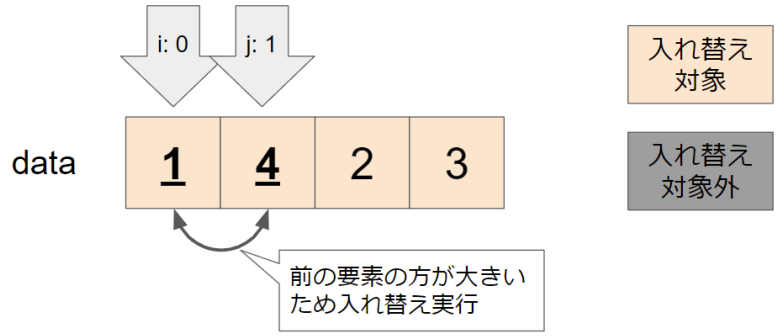
末尾から順番に隣り合った要素を比較するため、まずは `data[2]` （ `3` ）と `data[3]` （ `2` ）を比較する。前の要素（ `data[2]` ）の方が後ろの要素（ `data[3]` ）よりも大きいため、値を入れ替える。



続いて、 `j` に `2` が格納され、 `data[1]` （ `1` ）と `data[2]` （ `2` ）を比較する。後ろの要素（ `data[2]` ）の方が前の要素（ `data[1]` ）よりも大きいため、入れ替えは行わない。



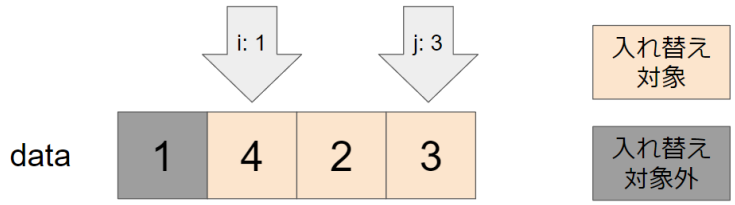
続いて、jに `1` が格納され、 `data[0]` （ `4` ）と `data[1]` （ `1` ）を比較する。前の要素（ `data[0]` ）の方が後ろの要素（ `data[1]` ）よりも大きいため、値を入れ替える。



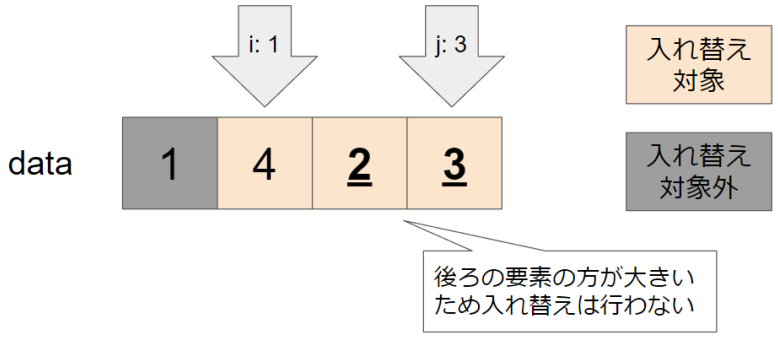
ここまでの処理で、リストの中で一番小さい値 `1` が先頭に格納された。

内側の `for` 文の処理が一通り終わったため、外側の `for` 文の更新処理を行う。 `i` に `1` を格納する。 `i` に `1` を格納することで、ソート済みの要素である `data[0]` を比較範囲から取り除くことができる。

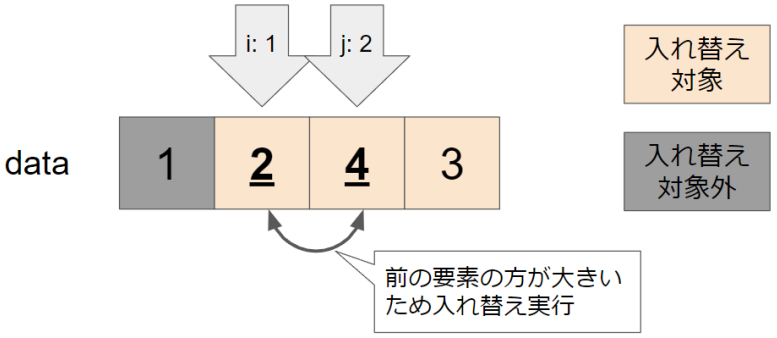
`i` に `1` が格納された状態で再度内側の `for` 文を実行する。



同様に末尾から、隣り合った要素を比較してする。 `data[2]` （ `2` ）と `data[3]` （ `3` ）を比較する。後ろの要素（ `data[3]` ）の方が前の要素（ `data[2]` ）よりも大きいため、入れ替えは行わない。



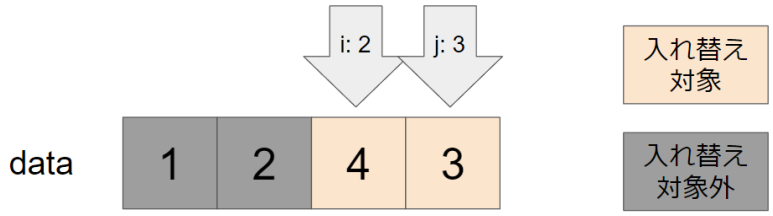
続いて、 `j` に `2` が格納され、 `data[1]` （ `4` ）と `data[2]` （ `2` ）を比較する。前の要素（ `data[1]` ）の方が後ろの要素（ `data[2]` ）よりも大きいため、値を入れ替える。



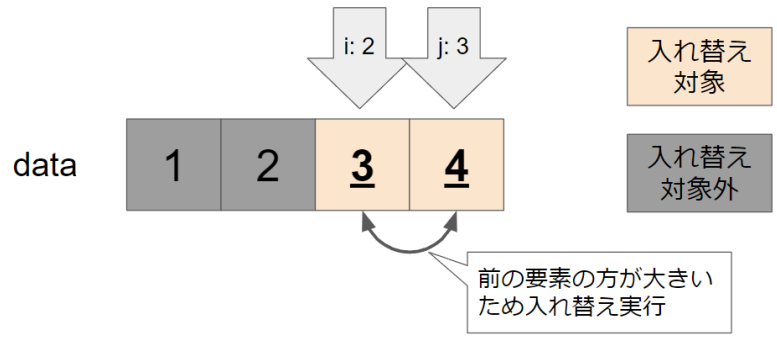
ここまでの処理で、比較範囲の中で一番小さい値 `2` が比較範囲の中の先頭に格納された。

内側の `for` 文の処理が一通り終わったため、外側の `for` 文の更新処理を行う。 `i` に `2` を格納する。 `i` に `2` を格納することで、ソート済みの要素である `data[0]` と `data[1]` を比較範囲から取り除くことができる。

`i` に `2` が格納された状態で再度内側の `for` 文を実行する。



同様に末尾から、隣り合った要素を比較してする。 `data[2]` （ `4` ）と `data[3]` （ `3` ）を比較する。前の要素（ `data[2]` ）の方が後ろの要素（ `data[3]` ）よりも大きいため、値を入れ替える。



内側の `for` 文も終了し、外側の `for` 文も終了したため、バブルソートを用いて昇順に並び替えた `data` を返す。

最終的に、 `sorted_li` には、リスト `li` が昇順に並び替えられた結果のリストが代入される。そして、 `print()` 関数を使用して、並び替えられたリストを出力している。

選択ソート

選択ソートは、簡単なソートアルゴリズムの一つである。昇順にソートする場合は、与えられたデータの中から最小値を選び、それをデータの先頭に挿入することを繰り返す。降順にソートする場合は、与えられたデータの中から最大値を選び、それをデータの先頭に挿入することを繰り返す。

以下は、選択ソートを用いてデータを昇順にソートする一般的な手順である。

1. データのソートする範囲の中の最小値を探す。
2. 最小値を見つけたら、ソートする範囲の先頭と交換する。
3. ソートする範囲の先頭を、取り除き、新しいソートする範囲を定める。
4. 手順1. から3. をソートする範囲が二つになるまで繰り返す。

選択ソートの時間計算量は、最悪のケース、平均のケースともに$O(n^2)$である。つまり、リストの要素数が増えるにつれて、計算量が指数関数的に増加する。そのため、データの要素数が大きい場合には、他のアルゴリズムを用いることが望ましい。

選択ソートでは、ソートする要素を格納する分の領域があればソートができるので、空間計算量は$O(n)$である。また、要素を交換するために、一時的に要素を保存するための変数が必要であるため、選択ソート自体で補助的に必要となる空間計算量は$O(1)$である。

前述の選択ソートの手順はPythonを用いて以下のように記述できる。

```python
def selection_sort(data):
    # リストの要素数を取得
    length = len(data)
    # リストの先頭から最小値を探す
    for i in range(length - 1):
        # ソート範囲の先頭の位置を、最小の値が格納されている位置と仮置きする
        min_index = i
        for j in range(i + 1, length):
            if data[j] < data[min_index]:
                # 比較して小さい値が入っている方の位置をmin_indexに保持する
                min_index = j

        # 最小値を見つけたら、それをリストの先頭と交換する
        if(i != min_index):
            data[i], data[min_index] = data[min_index], data[i]
    # 並び替えたリストを返す
    return data

# 使用例
li = [4, 1, 3, 2]
sorted_li = selection_sort(li)
print(sorted_li)
```

```
[1, 2, 3, 4]
```

上記のサンプルプログラムは、リストの要素を選択ソートを使って昇順に並び替える例である。 このプログラムでは、選択ソートを行う `selection_sort()` 関数を定義している。この関数では、引数として与えられたリストを昇順に並べ替える処理を行い、並べ替えた後のリストを返している。

`selection_sort()` 関数では、 `length` を定義し、引数として与えられたリストの要素数を代入する。次に、 `for` 文を使用して、リストの先頭から最小値を探している。そのため、 `for` 文では、 `range(length - 1)` を指定している。外側の `for` 文の `i` の値を `0` から `1` ずつ増やすことで、ソート範囲を狭めている。ソート範囲の中の最小の位置を探す上で、ソート範囲の先頭の位置と仮置きする `min_index` を定義している。

次に、内側の `for` 文を使用して、ソート範囲の第二要素から最小値を探している。このとき、探索する範囲は、 `range(i + 1, length)` である。内側 `for` 文内では、 `if` 文を使用して、比較して小さい値が入っている方の位置を `min_index` に保持する。

内側の `for` 文の処理が一通り終われば、ソート範囲の中の最小値の位置が見つかるため、それをソート範囲の先頭と交換する。

最終的に、 `selection_sort()` 関数は並び替えたリストを呼び出し元に返す。

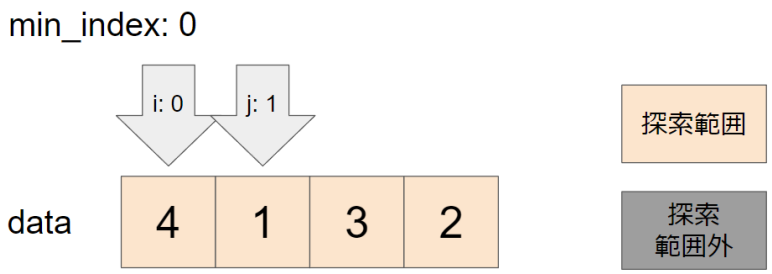
上記のサンプルプログラムでは、リスト `li` には `[4, 1, 3, 2]` が格納されている。

リスト `li` の値を `selection_sort()` 関数に渡した後の、 `selection_sort()` 関数の中の処理を詳しく見ていこう。

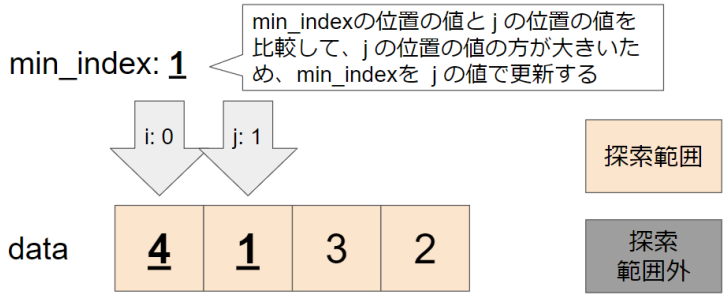
まず、 `length = len(data)` で、 `length` に `data` に格納されているリストの要素数を代入している。 `selection_sort()` 関数を呼び出すときに引数として、 `li` を指定しているため、 `data` には、 `[4, 1, 3, 2]` が格納されている。 `[4, 1, 3, 2]` の要素数は `4` であるため、 `length` には `4` が代入される。

外側の `for` 文で、 `i` に `0` を代入する。 `min_index` に、 `i` を格納する。 `i` は `0` なので、 `min_index` には `0` が格納されている。

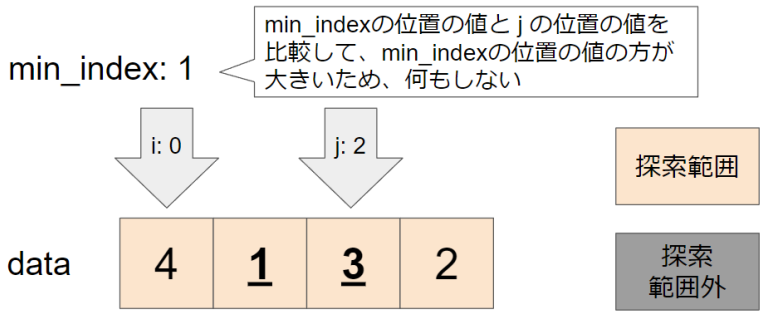
内側の `for` 文で、 `j` に `i + 1` から代入していくことで、 `i` が指す次の要素から順番に参照していき、 `min_index` の位置の要素を比較していくことで、まずはリスト全体の中の最小を探索していく。初め、 `j` には、 `1` が格納されている。



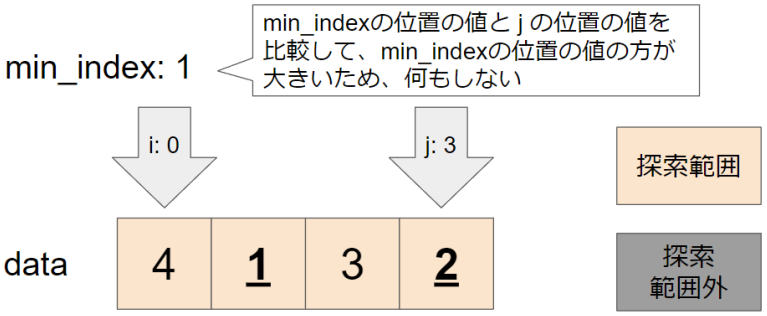
`if` 文で、 `data[j] < data[min_index]` を評価する。 `data[1]` は `1` であり、 `data[0]` は `4` である。 `1 < 4` であるので条件式は真となり、 `min_index` を `j` の値 `1` に更新する。 `min_index` には `1` が格納される。



`j` に `2` が代入され、再度 `if` 文で、 `data[j] < data[min_index]` を評価する。 `data[2]` は `3` であり、 `data[1]` は `1` である。 `3 > 1` であるので条件式は偽となり、 `min_index` は変更されない。 `min_index` は `1` のままである。

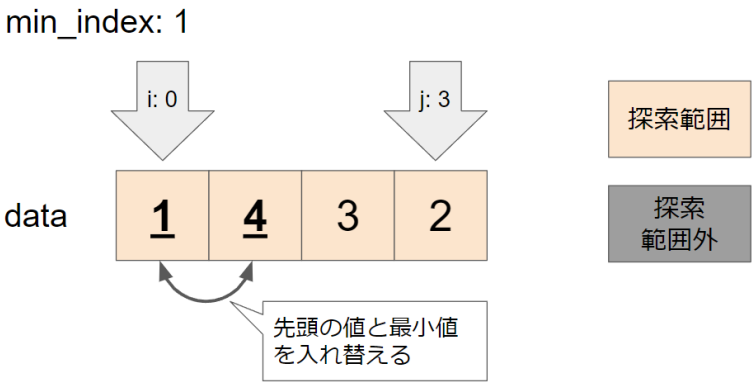


`j` に `3` が代入され、再度 `if` 文で、 `data[j] < data[min_index]` を評価する。 `data[3]` は `2` であり、 `data[1]` は `1` である。 `2 > 1` であるので条件式は偽となり、 `min_index` は変更されない。 `min_index` は `1` のままである。



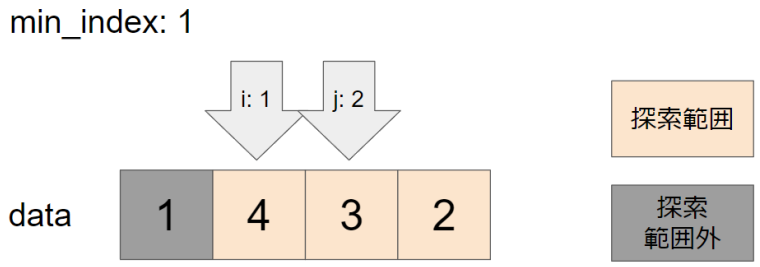
以上の操作で、 `data` のインデックス `0` から `3` までの範囲の最小値の位置が `min_index` （ `1` ）に格納される。

最小値の位置を見つけたら、それをソート範囲の先頭と交換する。 `data[i]` には、 `data[0]` が、 `data[min_index]` には、 `data[1]` が該当する。そのため、 `data[0]` と `data[1]` が交換され、 `data` は `[1, 4, 3, 2]` になる。

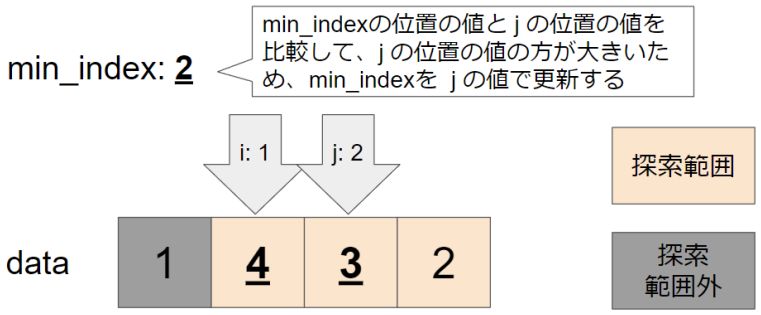


外側の `for` 文に戻り、 `i` には `1` が代入される。これによって、最小値の探索範囲が狭まる。 `min_index` に、 `i` を格納する。 `i` は `1` なので、 `min_index` には `1` が格納されている。

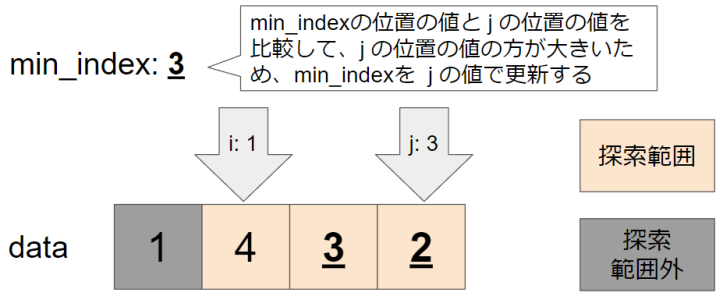
内側の `for` 文に進み、 `j` には、 `i + 1` の結果である `2` が格納される。



`if` 文で、 `data[j] < data[min_index]` を評価する。 `data[2]` は `3` であり、 `data[1]` は `4` である。 `3 < 4` であるので条件式は真となり、 `min_index` を `j` の値 `2` に更新する。 `min_index` は `2` が格納される。

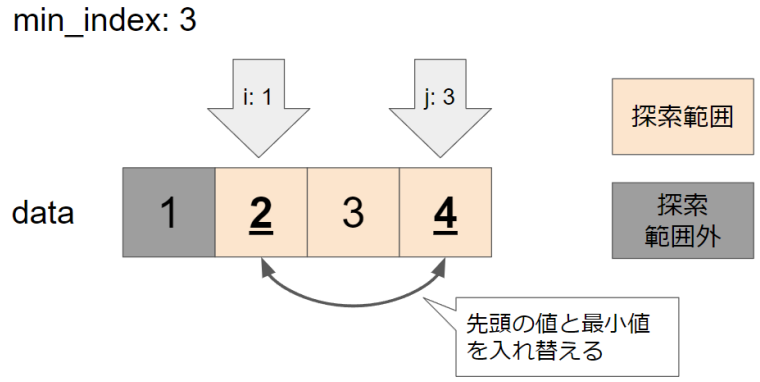


`j` に `3` が代入され、再度 `if` 文で、 `data[j] < data[min_index]` を評価する。 `data[3]` は `2` であり、 `data[2]` は `3` である。 `2 < 3` であるので条件式は真となり、 `min_index` を `j` の値 `3` に更新する。 `min_index` は `3` が格納される。



以上の操作で、 `data` のインデックス `1` から `3` までの範囲の最小値の位置が `min_index` （ `3` ）に格納される。

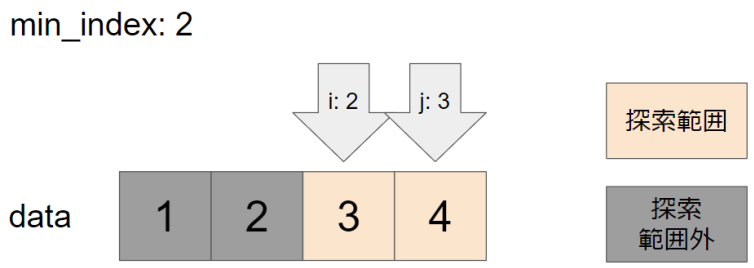
最小値の位置を見つけたら、それをソート範囲の先頭と交換する。 `data[i]` には、 `data[1]` が、 `data[min_index]` には、 `data[3]` が該当する。そのため、 `data[1]` と `data[3]` が交換され、 `data` は `[1, 2, 3, 4]` になる。



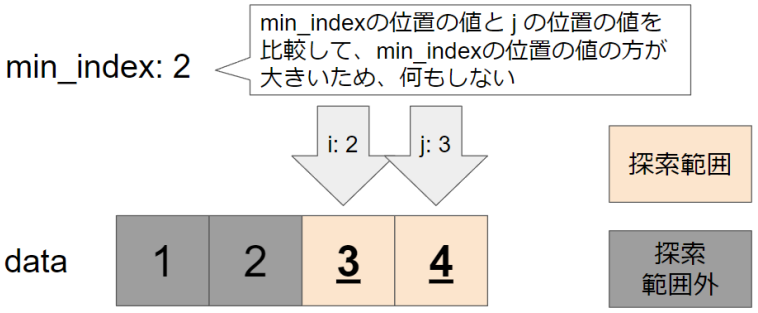
外側の `for` 文に戻り、 `i` には `2` が代入される。これによって、最小値の探索範囲が狭まる。 `min_index` に、 `i` を格納する。 `i` は `2`

なので、 `min_index` には `2` が格納されている。

内側の `for` 文に進み、 `j` には、 `i + 1` の結果である `3` が格納される。

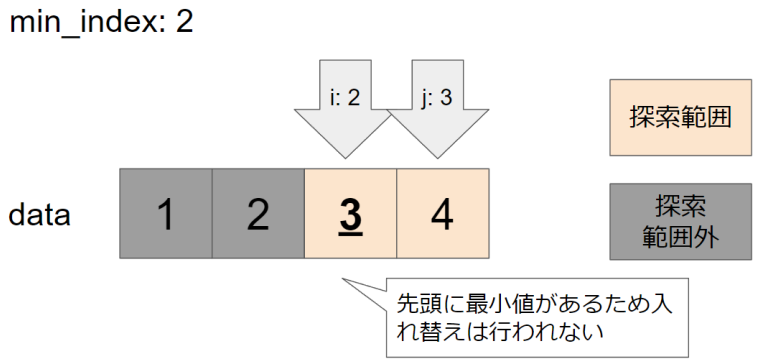


`if` 文で、 `data[j] < data[min_index]` を評価する。 `data[3]` は `4` であり、 `data[2]` は `3` である。 `4 > 3` であるので条件式は偽となり、 `min_index` は変更されない。 `min_index` は `2` のままである。



以上の操作で、 `data` のインデックス `2` から `3` までの範囲の最小値の位置が `min_index` （ `2` ）に格納される。

最小値の位置を見つけたら、それをソート範囲の先頭と交換する。 `data[i]` には、 `data[2]` が、 `data[min_index]` には、 `data[2]` が該当する。 `i` と `min_index` がともに `2` であるため、交換は実行されず、 `data` は `[1, 2, 3, 4]` のままである。



以上の操作で、内側の `for` 文も終了し、外側の `for` 文も終了したため、選択ソートを用いて昇順に並び替えた `data` を返す。

最終的に、 `sorted_li` には、リスト `li` が昇順に並び替えられた結果のリストが代入される。そして、 `print()` 関数を使用して、並び替えられたリストを出力している。

挿入ソート

挿入ソートは、簡単なソートアルゴリズムの一つである。整列済みのデータに要素を追加して、その要素を適切な位置に挿入することを繰り返し、ソートを行う。

以下は、選択ソートを用いてデータを昇順にソートする一般的な手順である。

1. 並び替えたいデータの先頭を整列済みの要素とみなし、次の要素を整列済みの部分の位置に挿入しようとする。
2. 挿入方法は、整列済みの部分の末尾から順番に比較していき、追加する要素の方が小さければ入れ替えていく。
3. 手順2.を追加する要素が先頭になるか、比較して追加する要素の方が大きいと判定されるまで繰り返す。
4. 新しい整列済みの部分ができるため、新しい整列済みの部分の末尾の次の要素を新しい追加の要素として、手順2.に戻る。
5. 手順2.から4.を、追加の要素が並び替えたいデータ全体の末尾の要素になるまで繰り返す。

挿入ソートの時間計算量は、最悪のケース、平均のケースともに$O(n^2)$である。つまり、リストの要素数が増えるにつれて、計算量が指数関数的に増加する。そのため、データの要素数が大きい場合には、他のアルゴリズムを用いることが望ましい。

挿入ソートでは、ソートする要素を格納する分の領域があればソートができるので、空間計算量は$O(n)$である。また、要素を交換するために、一時的に要素を保存するための変数が必要であるため、挿入ソート自体で補助的に必要となる空間計算量は$O(1)$である。

前述の挿入ソートの手順はPythonを用いて以下のように記述できる。

```python
def insertion_sort(data):
     # リストの要素数を取得
    length = len(data)
    # 整列されている部分に追加する要素の位置をfor文で指す
    for i in range(1, length):
        j = i
        # 追加する要素を整列済みの部分の末尾から比較して、追加要素が小さければ入れ替える
        # 比較は、追加する要素が先頭になるか、入れ替えできなくなるまで続ける
        while j > 0 and data[j-1] > data[j]:
            data[j-1], data[j] = data[j], data[j-1]
            j -= 1
    # 並び替えたリストを返す
    return data

# 使用例
li = [4, 1, 3, 2]
sorted_li = insertion_sort(li)
print(sorted_li)
```

```
[1, 2, 3, 4]
```

上記のサンプルプログラムは、リストの要素を挿入ソートを使って昇順に並び替える例である。 このプログラムでは、挿入ソートを行う `insertion_sort()` 関数を定義している。この関数では、引数として与えられたリストを昇順に並べ替える処理を行い、並べ替えた後のリストを返している。

`insertion_sort()` 関数では、 `length` を定義し、引数として与えられたリストの要素数を代入する。次に、 `for` 文を使用して、並び替えたいデータの中の整列済みの部分に追加する要素の位置を指している。

`j` に `i` の値を代入し、 `while` 文では、 `j` を使い、追加要素を整列済みの部分の中で、整列が保たれる位置に挿入する。

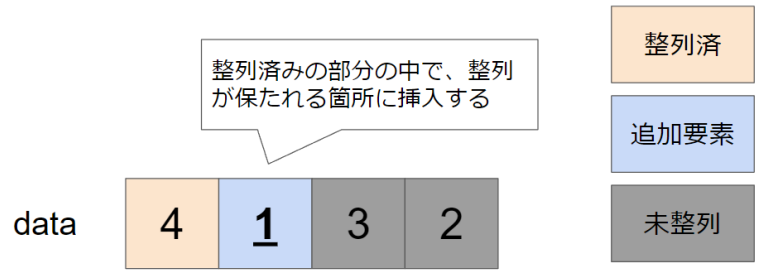
最終的に、 `insertion_sort()` 関数は並び替えたリストを呼び出し元に返す。

上記のサンプルプログラムでは、リスト `li` には `[4, 1, 3, 2]` が格納されている。

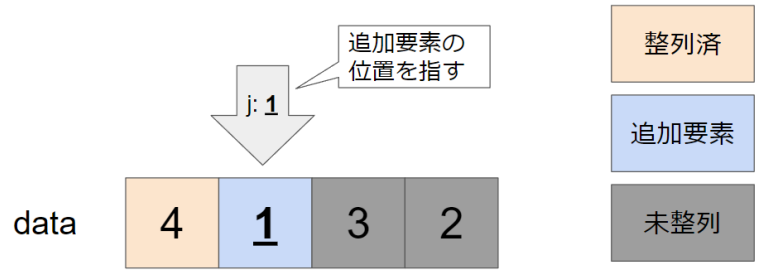
リスト `li` の値を `insertion_sort()` 関数に渡した後の、 `insertion_sort()` 関数の中の処理を詳しく見ていこう。

まず、 `length = len(data)` で、 `length` に `data` に格納されているリストの要素数を代入している。 `insertion_sort()` 関数を呼び出すときに引数として、 `li` を指定しているため、 `data` には、 `[4, 1, 3, 2]` が格納されている。 `[4, 1, 3, 2]` の要素数は `4` であるため、 `length` には `4` が代入される。

`for` 文では、 `i` に `1` から `length` より `1` 小さい値である `3` までが順番に代入される。 `i` は並び替えたいデータの中の整列済みの部分に追加する要素の位置を指している。 `data` の先頭の要素である `data[0]` （ `4` ）は整列済みと判断できるため、先頭の要素に、先頭の次の要素である `data[1]` （ `1` ）を挿入する場合を考える。



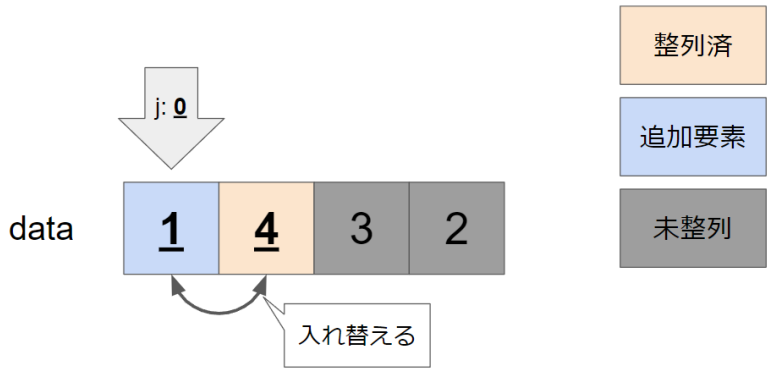
`j` に `i` を代入するため、 `j` には `1` が格納される。後続の処理で入れ替えを行うことで追加要素の位置がリストの中で変化する。 `j` は追加要素の位置を常に指し示す変数である。



`while` 文では、 `j > 0 and data[j-1] > data[j]` を評価する。 `data[j-1] > data[j]` では、追加要素の前にある要素（ `data[j-1]` ）と追加要素（ `data[j]` ）を比較して、前にある要素の方が大きければ入れ替えを行うため、入れ替えを行うことができるかを判定している。 `j > 0` では、追加要素が整列済みの部分の先頭には存在せず、比較する要素があることを判定している。 `j > 0` と `data[j-1] > data[j]` の両方を満たしていると、追加要素と整列済みの部分の中で適切な位置にあるとはいえず、入れ替えを行う必要があるため、 `and` でこの二つの条件式を繋げている。入れ替えを行うことができなければ、整列済みの部分に対して追加要素が整列を保つような適切な位置に挿入されたと判断できるため、新しい追加要素を挿入する処理に移る。

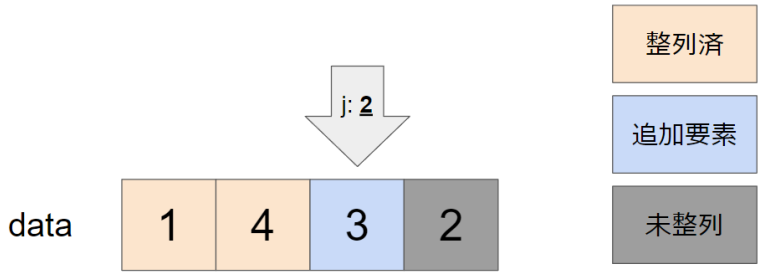
`j > 0 and data[j-1] > data[j]` を評価する。 `j` には `1` が格納されており、 `data[0]` には `4` が、 `data[1]` には `1` が格納されているため、 `j > 0` と `data[j-1] > data[j]` はともに真と評価できる。そのため、 `j > 0 and data[j-1] > data[j]` は真と評価できるため、 `while` 文の中の処理を行う。

`while` 文の中の処理では、 `data[0]` と `data[1]` の値を入れ替えるため、 `data[0]` には `1` が、 `data[1]` には `4` が格納される。 `j` は `1` 引くため、 `0` が格納される。



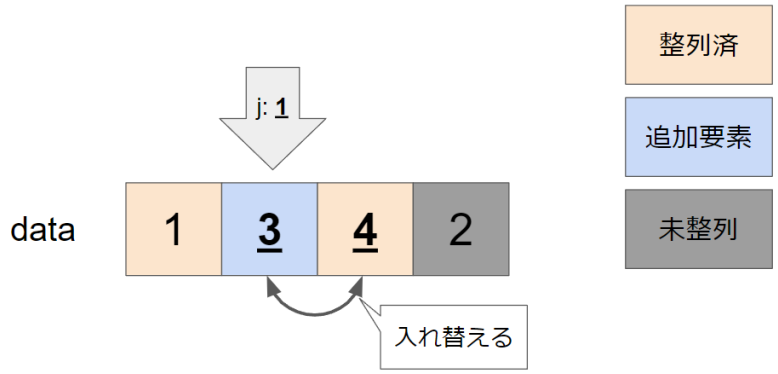
`while` 文の中の処理が終わったため、改めて、 `j > 0 and data[j-1] > data[j]` を評価する。 `j` は `0` であるため、 `j > 0` は偽となる。そのため、 `j > 0 and data[j-1] > data[j]` は偽と評価され、 `while` 文の処理は終了する。

`for` 文に戻り、 `i` に `2` を代入する。 `for` 文の中の処理に移り、 `j` に `i` を代入するため、 `j` には `2` が格納される。



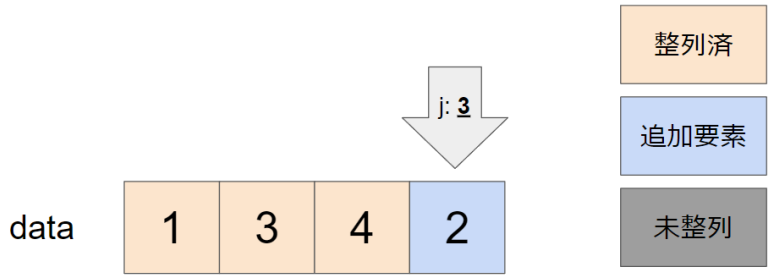
`while` 文で、 `j > 0 and data[j-1] > data[j]` を評価する。 `j` には `2` が格納されており、 `data[1]` には `4` が、 `data[2]` には `3` が格納されているため、 `j > 0` と `data[j-1] > data[j]` はともに真と評価できる。そのため、 `j > 0 and data[j-1] > data[j]` は真と評価できるため、 `while` 文の中の処理を行う。

`while` 文の中の処理では、 `data[1]` と `data[2]` の値を入れ替えるため、 `data[1]` には `3` が、 `data[2]` には `4` が格納される。 `j` は `1` 引くため、 `1` が格納される。



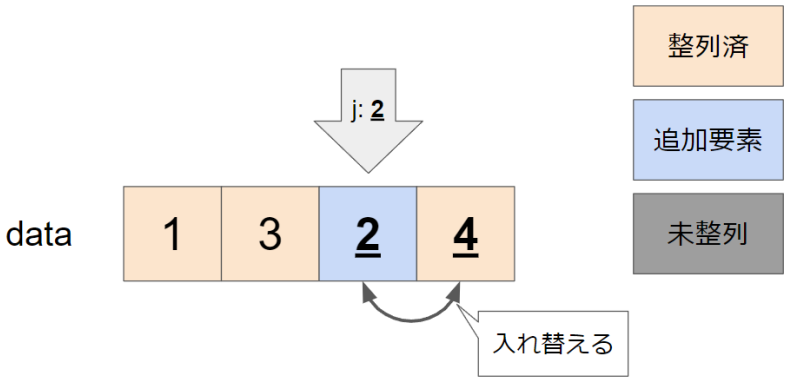
`while` 文の中の処理が終わったため、改めて、 `j > 0 and data[j-1] > data[j]` を評価する。 `j` は `1` であるため、 `j > 0` は真と評価できる。ただし、 `data[j-1] > data[j]` は、 `data[0]` には `1` が、 `data[1]` には `3` が格納されているため、偽と評価される。そのため、 `j > 0 and data[j-1] > data[j]` は偽と評価され、 `while` 文の処理は終了する。

`for` 文に戻り、 `i` に `3` を代入する。 `for` 文の中の処理に移り、 `j` に `i` を代入するため、 `j` には `3` が格納される。



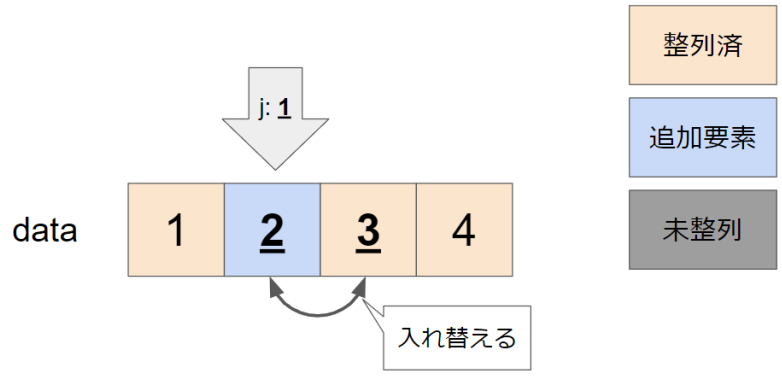
`while` 文で、 `j > 0 and data[j-1] > data[j]` を評価する。 `j` には `3` が格納されており、 `data[2]` には `4` が、 `data[3]` には `2` が格納されているため、 `j > 0` と `data[j-1] > data[j]` はともに真と評価できる。そのため、 `j > 0 and data[j-1] > data[j]` は真と評価できるため、 `while` 文の中の処理を行う。

`while` 文の中の処理では、 `data[2]` と `data[3]` の値を入れ替えるため、 `data[2]` には `2` が、 `data[3]` には `4` が格納される。 `j` は `1` 引くため、 `2` が格納される。



`while` 文の中の処理が終わったため、改めて、 `j > 0 and data[j-1] > data[j]` を評価する。 `j` には `2` が格納されており、 `data[1]` には `3` が、 `data[2]` には `2` が格納されているため、 `j > 0` と `data[j-1] > data[j]` はともに真と評価できる。そのため、 `j > 0 and data[j-1] > data[j]` は真と評価できるため、 `while` 文の中の処理を行う。

`while` 文の中の処理では、 `data[1]` と `data[2]` の値を入れ替えるため、 `data[1]` には `2` が、 `data[2]` には `3` が格納される。 `j` は `1` 引くため、 `1` が格納される。



`while` 文の中の処理が終わったため、改めて、 `j > 0 and data[j-1] > data[j]` を評価する。 `j` は `1` であるため、 `j > 0` は真と評価できる。ただし、 `data[j-1] > data[j]` は、 `data[0]` には `1` が、 `data[1]` には `2` が格納されているため、偽と評価される。そのため、 `j > 0 and data[j-1] > data[j]` は偽と評価され、 `while` 文の処理は終了する。

以上の操作で、 `for` 文の処理も終了するため、挿入ソートを用いて昇順に並び替えた `data` を返す。

最終的に、 `sorted_li` には、リスト `li` が昇順に並び替えられた結果のリストが代入される。そして、 `print()` 関数を使用して、並び替えられたリストを出力している。

マージソート

マージソートは、高速なソートアルゴリズムの一つである。データを2分割していく操作を、要素数が1になるまで繰り返していき、細分化し終えたら要素同士を整列しながら戻していくことでソートを行うアルゴリズムである。データを2分割していく操作には再帰を用いる。

以下は、マージソートを用いてデータを昇順にソートする一般的な手順である。

1. 再帰処理を用いてデータを二分割をする。
2. 分割後のデータの要素数が1つであれば、再帰処理を終え、1つになった要素の値を返す。要素数が2つ以上であれば、二分割を続ける。
3. 二分割する再帰処理の両方から値が返ってきたら、返ってきた2つの要素の列を比較して昇順に整列しながら1つのデータに統合する。その後、統合したデータを返す。
4. 手順1. から3. を、整列したいデータ全ての要素が最終的に1つのデータとして再度統合されるまで繰り返す。

マージソートの時間計算量は、最悪のケースと平均のケースでともに、$O(n \log n)$である。

空間計算量は、データを二分割していく処理に追加のメモリが必要になるため、$O(n)$である。

前述のマージソートの手順をPythonを用いて以下のように記述できる。なお、マージソートの理論的な空間計算量は$O(n)$だが、以下の実装では分かりやすさのためにリストの分割時にデータのコピーを余分に行っているため、空間計算量が$O(n \log n)$となっている。

```python
def merge_sort(data):
    # リストのサイズが1以下の場合はそのまま返す
    if len(data) <= 1:
        return data

    # リストを二分割する
    mid = len(data) // 2
    left = data[:mid]
    right = data[mid:]

    # 再帰的に処理を行い、二分割したリストをさらに分割する
    left = merge_sort(left)
    right = merge_sort(right)

    # 細分化されたリストをマージする
    return merge(left, right)

def merge(left, right):
    # 整列しながらマージした結果を格納するリスト
    result = []
    # 二つのリストを比較しながら要素を追加していく
    while left and right:
        if left[0] < right[0]:
            result.append(left.pop(0))
        else:
            result.append(right.pop(0))
    # 残った要素を追加する
    if left:
        result.extend(left)
    if right:
        result.extend(right)
    return result

# 使用例
li = [5, 2, 4, 6, 1, 3]
sorted_li = merge_sort(li)
print(sorted_li)
```

```
[1, 2, 3, 4, 5, 6]
```

上記のサンプルプログラムは、リストの要素をマージソートを使って昇順に並び替える例である。 このプログラムでは、マージソートを行う `merge_sort()` 関数を定義している。この関数では、引数として与えられたリストを昇順に並べ替える処理を行い、並べ替えた後のリストを返している。また、二つのデータを昇順に整列しながらマージする `merge()` 関数も定義しており、関数 `merge_sort` の中で呼び出している。

`merge_sort()` 関数では、リストの要素数が `1` 以下の場合、そのまま返して、再帰処理を終了する。リストの要素数が二つ以上の場合は、`left` と `rigth` に二分割する。分けた `left` と `rigth` に対して、それぞれを引数にとって `merge_sort()` 関数を再帰的に呼び出し、結果を `left` と `rigth` に格納し直す。格納し直した `left` と `rigth` を引数に `merge()` 関数を呼び出し、実行結果を返す。

`merge()` 関数では、 `left` と `rigth` を整列しながらマージした結果を格納する `result` を用意する。 `while` 文では、 `left` と `rigth` のどちらかの要素がなくなるまで、 `left` と `rigth` の先頭を比較し、小さい方を取り出し `result` の末尾から追加していく。最後に、 `left` と `rigth` のうち残った要素を `result` の末尾に追加し、 `result` を返す。

上記のサンプルプログラムでは、リスト `li` には `[5, 2, 4, 6, 1, 3]` が格納されている。

リスト `li` の値を `merge_sort()` 関数に渡した後の、 `merge_sort()` 関数の中の処理を詳しく見ていこう。

ここでは、引数に `li` を渡して、呼び出した `merge_sort()` 関数を1回目の `merge_sort()` 関数の呼び出しとする。

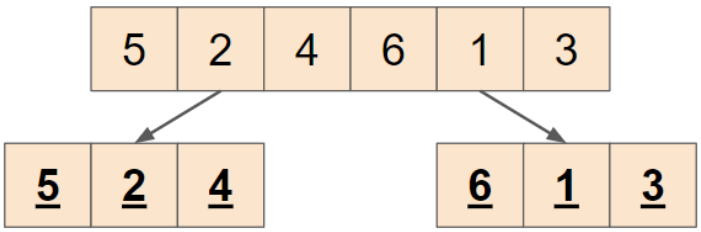
まず、 `data` の要素が `1` 以下であるかどうかを判定する。 `data` の要素数は `6` であるため、条件を満たさず、後続の処理に進む。

`data` を二分割する。以下の処理で `data` を二分割する位置を求める。

```
1  mid = len(data) // 2
```

`mid` には `3` が格納される。 `mid` を用いて、 `data` の左半分を `left` に、右半分を `right` に格納する。

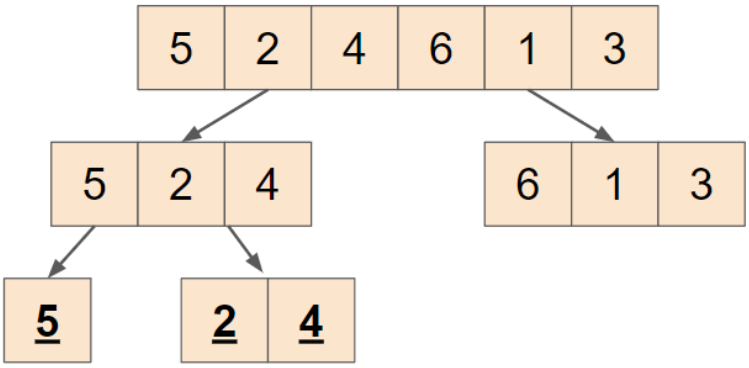
`left` には `[5, 2, 4]` が、 `right` には `[6, 1, 3]` が格納される。



`left` （ `[5, 2, 4]` ）、 `right` （ `[6, 1, 3]` ）それぞれを引数に、再帰的に `merge_sort()` 関数を呼び出す、

2回目の `merge_sort()` 関数の呼び出しを行い、 `left` （ `[5, 2, 4]` ）を再帰的に処理していく。

2回目の `merge_sort()` 関数の呼び出しでは、 `data` （ `[5, 2, 4]` ）もまた二分割される。2回目の `merge_sort()` 関数の呼び出しの中で、 `left` には `[5]` 、 `right` には `[2, 4]` が格納される。



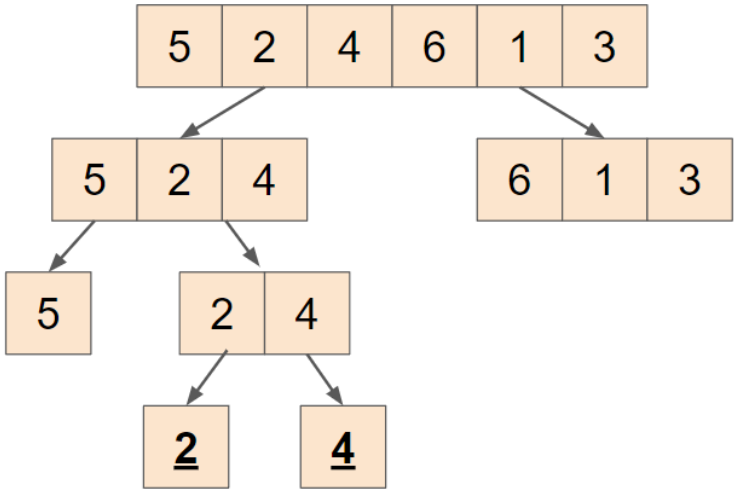
`left` （ `[5]` ）を引数に、3回目の `merge_sort()` 関数の呼び出しを行う。

3回目の `merge_sort()` 関数の呼び出しでは、 `data` （ `[5]` ）の要素数が `1` 以下であるため、再帰処理が終了し、 `[5]` を返す。

2回目の `merge_sort()` 関数の呼び出しの中で、 `left` （ `[5]` ）を引数に `merge_sort()` 関数に呼び出した結果、 `left` には再度 `[5]` が格納される。

2回目の `merge_sort()` 関数の呼び出しの中で、 `right` （ `[2, 4]` ）を引数に、4回目の `merge_sort()` 関数の呼び出しを行う。

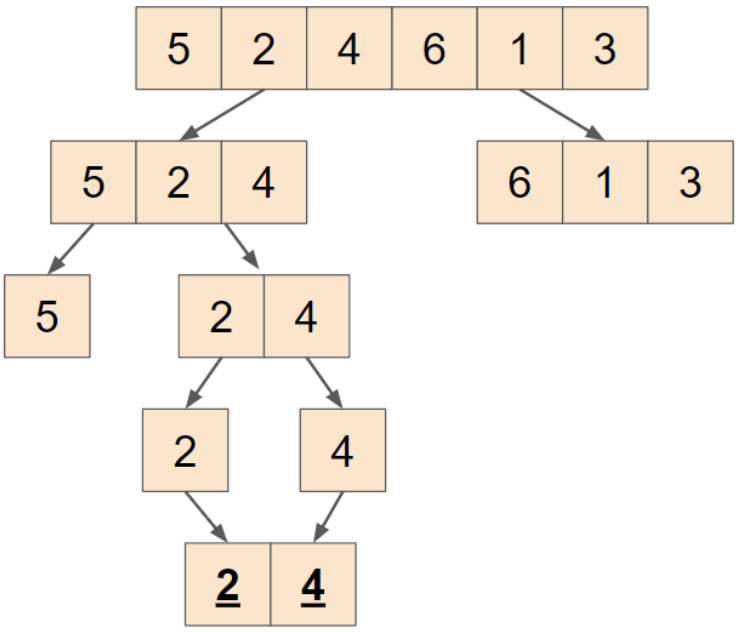
4回目の `merge_sort()` 関数の呼び出しでは、 `data` （ `[2, 4]` ）もまた二分割される。4回目の `merge_sort()` 関数の呼び出しの中で、 `left` には `[2]` 、 `right` には `[4]` が格納される。



4回目の `merge_sort()` 関数の呼び出しで、 `left` （ `[2]` ）を引数に、5回目の `merge_sort()` 関数の呼び出しを、 `right` （ `[4]` ）を引数に、6回目の `merge_sort()` 関数の呼び出しを行う。それぞれ要素数が `1` 以下であるため、引数にとった要素がそのまま返され、 `left` には `[2]` が、 `right` には `[4]` が改めて格納される。

4回目の `merge_sort()` 関数の呼び出しで全ての再帰呼び出しが終わったため、次の処理に進み、 `left` （ `[2]` ）と `right` （ `[4]` ）それぞれを引数にとって、 `merge()` 関数を実行する。

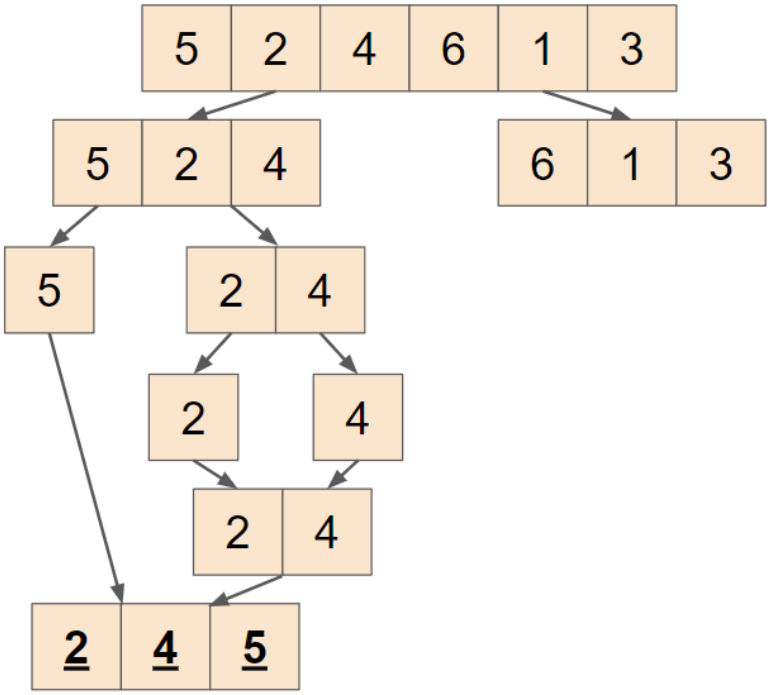
`merge()` 関数では、引数にとった二つの要素を整列しながらマージして返す。 `merge()` 関数は結果として、 `[2, 4]` を返す。



`merge_sort()` 関数は、 `merge()` 関数の戻り値をそのまま返すため、4回目の `merge_sort()` 関数の呼び出しは `[2, 4]` を返す。

4回目の `merge_sort()` 関数の呼び出しは、2回目の `merge_sort()` 関数の呼び出しの中で、 `right` （ `[2, 4]` ）を引数に `merge_sort()` 関数を呼び出した際の処理であるため、2回目の `merge_sort()` 関数の呼び出しで全ての再帰呼び出しが終わり、次の処理に進む。 `left` （ `[5]` ）と `right` （ `[2, 4]` ）それぞれを引数にとって、 `merge()` 関数を実行する。 `merge()` 関数は、 `left` （ `[5]` ）と `right` （ `[2, 4]` ）を整列しながらマージするため、 `[2, 4, 5]` を返す。

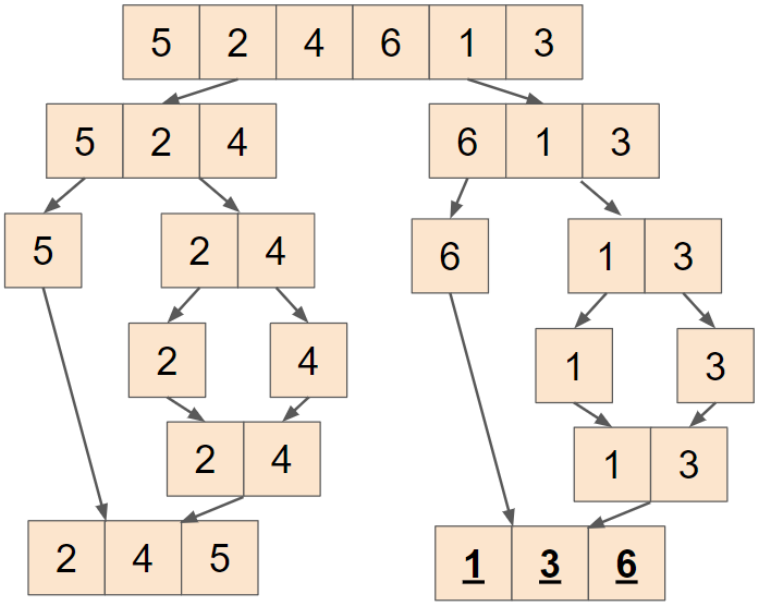
`merge_sort()` 関数は、 `merge()` 関数の戻り値をそのまま返すため、2回目の `merge_sort()` 関数の呼び出しは `[2, 4, 5]` を返す。



2回目の `merge_sort()` 関数の呼び出しは、1回目の `merge_sort()` 関数の呼び出しの中で、 `left` （ `[5, 2, 4]` ）を引数に `merge_sort()` 関数を呼び出した際の処理であるため、 `left` には、 `[2, 4, 5]` が再度格納される。

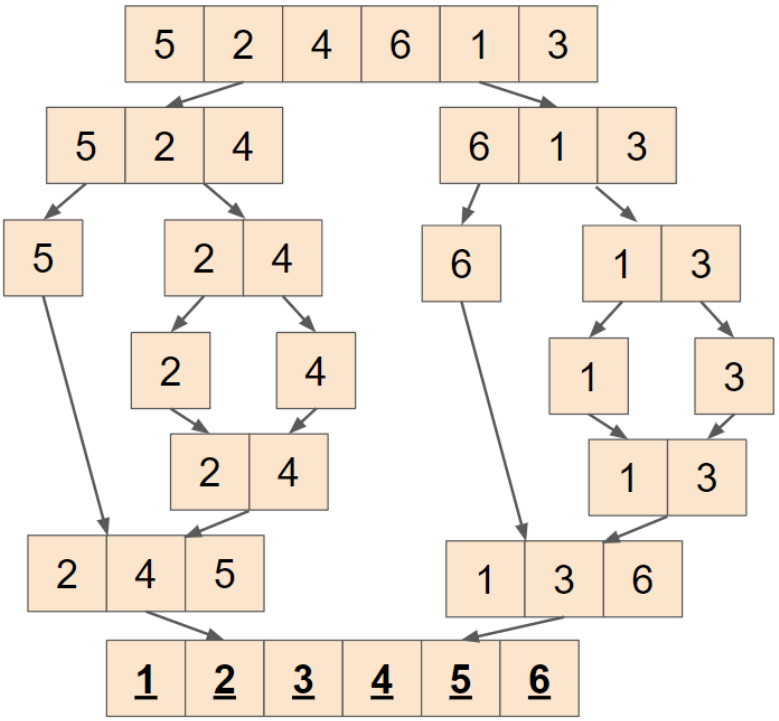
1回目の `merge_sort()` 関数の呼び出しの中で、 `left` （ `[5, 2, 4]` ）を引数にに取った再帰呼び出しが終わったため、続く処理として、 `right` （ `[6, 1, 3]` ）を引数にに取って `merge_sort()` 関数を呼び出す再帰処理を実行する。

`right` （ `[6, 1, 3]` ）を引数にに取った再帰呼び出しも、 `left` （ `[5, 2, 4]` ）を引数にに取った再帰呼び出しと同様の処理が実行されるため、1回目の `merge_sort()` 関数の呼び出しの中で、 `right` には `[1, 3, 6]` が再度格納される。



1回目の `merge_sort()` 関数の呼び出しで全ての再帰呼び出しが終わり、次の処理に進む。 `left` （ `[2, 4, 5]` ）と `right` （ `[1, 3, 6]` ）それぞれを引数にとって、 `merge()` 関数を実行する。 `merge()` 関数は、 `left` （ `[2, 4, 5]` ）と `right` （ `[1, 3, 6]` ）を整列しながらマージするため、 `[1, 2, 3, 4, 5, 6]` を返す。

`merge_sort()` 関数は、 `merge()` 関数の戻り値をそのまま返すため、1回目の `merge_sort()` 関数の呼び出しは `[1, 2, 3, 4, 5, 6]` を返す。



最終的に、 `sorted_li` には、リスト `li` が昇順に並び替えられた結果のリストが代入される。そして、 `print()` 関数を使用して、並び替えられたリストを出力している。

問題1

問題文

互いに異なる整数を含む数列Aが与えられる。 **バブルソート**を用いてAを**昇順**に整列し、その過程を出力する `bubble_sort()` 関数を作成せよ。なお、ソートする際は、末尾の要素を前方に移動させること。前方の要素を末尾に移動させてはいけない。

制約

- $0 \leq a \leq 100\ (a \in A)$
- Aの要素は互いに異なる。
- 入力は全て整数である。

入力

入力は次の形式で標準入力から与えられます。

> $A_0\ A_1\ ...\ A_{N-1}$ （Nは正の整数）

出力

隣り合った要素を入れ換える操作が発生するたびに、その操作後のAをリストの形で1行に出力する。

入力例1

```
1 8 4 3 7 6 5 2 1
```

出力例1

```
[8, 4, 3, 7, 6, 5, 1, 2]
[8, 4, 3, 7, 6, 1, 5, 2]
[8, 4, 3, 7, 1, 6, 5, 2]
[8, 4, 3, 1, 7, 6, 5, 2]
[8, 4, 1, 3, 7, 6, 5, 2]
[8, 1, 4, 3, 7, 6, 5, 2]
[1, 8, 4, 3, 7, 6, 5, 2]
[1, 8, 4, 3, 7, 6, 2, 5]
[1, 8, 4, 3, 7, 2, 6, 5]
[1, 8, 4, 3, 2, 7, 6, 5]
[1, 8, 4, 2, 3, 7, 6, 5]
[1, 8, 2, 4, 3, 7, 6, 5]
[1, 2, 8, 4, 3, 7, 6, 5]
[1, 2, 8, 4, 3, 7, 5, 6]
[1, 2, 8, 4, 3, 5, 7, 6]
[1, 2, 8, 3, 4, 5, 7, 6]
[1, 2, 3, 8, 4, 5, 7, 6]
[1, 2, 3, 8, 4, 5, 6, 7]
[1, 2, 3, 4, 8, 5, 6, 7]
[1, 2, 3, 4, 5, 8, 6, 7]
[1, 2, 3, 4, 5, 6, 8, 7]
[1, 2, 3, 4, 5, 6, 7, 8]
```

入力例2

```
1 2 3 4 5 6 7 8 9 10
```

出力例2

```
1
```

入れ換える操作が1回も発生しないので、何も出力してはいけない。

解答の雛形

```
def bubble_sort(data):
    # 以下のpassを削除してから解答を入力
    pass


# ソート対象のデータ
A = [int(x) for x in input().split()]
bubble_sort(A)
```

問題2

問題文

互いに異なる整数を含む数列Aが与えられる。 **バブルソート**を用いてAを**降順**に整列し、その過程を出力する `bubble_sort()` 関数を作成せよ。なお、ソートする際は、末尾の要素を前方に移動させること。前方の要素を末尾に移動させてはいけない。

制約

- $0 \leq a \leq 100\,(a \in A)$
- Aの要素は互いに異なる。
- 入力は全て整数である。

入力

入力は次の形式で標準入力から与えられます。

> $A_0\ A_1\ ...\ A_{N-1}$ （Nは正の整数）

出力

隣り合った要素を入れ換える操作が発生するたびに、その操作後のAをリストの形で1行に出力する。

入力例1

```
8 4 3 7 6 5 2 1
```

出力例1

```
[8, 4, 7, 3, 6, 5, 2, 1]
[8, 7, 4, 3, 6, 5, 2, 1]
[8, 7, 4, 6, 3, 5, 2, 1]
[8, 7, 6, 4, 3, 5, 2, 1]
[8, 7, 6, 4, 5, 3, 2, 1]
[8, 7, 6, 5, 4, 3, 2, 1]
```

入力例2

```
10 9 8 7 6 5 4 3 2 1
```

出力例2

```
1
```

入れ換える操作が1回も発生しないので、何も出力してはいけない。

解答の雛形

```python
def bubble_sort(data):
    # 以下のpassを削除してから解答を入力
    pass


# ソート対象のデータ
A = [int(x) for x in input().split()]
bubble_sort(A)
```

問題3

問題文

互いに異なる整数を含む数列Aが与えられる。 **選択ソート**を用いてAを**昇順**に整列し、その過程を出力する `selection_sort()` 関数を作成せよ。なお、ソートする際は、**最小の要素を前方に移動**させること。最大の要素を後方に移動させてはいけない。

制約

- $0 \leq a \leq 100$ $(a \in A)$
- Aの要素は互いに異なる。
- 入力は全て整数である。

入力

入力は次の形式で標準入力から与えられます。

> A_0 A_1 ... A_{N-1} （Nは正の整数）

出力

未整列部分の最小要素を未整列部分の先頭と入れ換える操作が発生するたびに、その操作後のAをリストの形で1行に出力する。

入力例1

```
8 4 3 7 6 5 2 1
```

出力例1

```
[1, 4, 3, 7, 6, 5, 2, 8]
[1, 2, 3, 7, 6, 5, 4, 8]
[1, 2, 3, 4, 6, 5, 7, 8]
[1, 2, 3, 4, 5, 6, 7, 8]
```

入力例2

```
1 2 3 4 5 6 7 8 9 10
```

出力例2

```

```

入れ換える操作が1回も発生しないので、何も出力してはいけない。

解答の雛形

```python
def selection_sort(data):
    # 以下のpassを削除してから解答を入力
    pass


# ソート対象のデータ
A = [int(x) for x in input().split()]
selection_sort(A)
```

問題4

問題文

互いに異なる整数を含む数列Aが与えられる。 **選択ソート**を用いてAを**降順**に整列し、その過程を出力する `selection_sort()` 関数を作成せよ。なお、ソートする際は、**最大の要素を前方に移動**させること。最小の要素を後方に移動させてはいけない。

制約

- $0 \leq a \leq 100\ (a \in A)$
- Aの要素は互いに異なる。
- 入力は全て整数である。

入力

入力は次の形式で標準入力から与えられます。

> $A_0\ A_1\ ...\ A_{N-1}$ （Nは正の整数）

出力

追加要素を整列部分の適切な位置への配置が行われるたびに、その操作後のAをリストの形で1行に出力する。

入力例1

```
1 8 4 3 7 6 5 2 1
```

出力例1

```
1 [8, 7, 3, 4, 6, 5, 2, 1]
2 [8, 7, 6, 4, 3, 5, 2, 1]
3 [8, 7, 6, 5, 3, 4, 2, 1]
4 [8, 7, 6, 5, 4, 3, 2, 1]
```

入力例2

```
1 10 9 8 7 6 5 4 3 2 1
```

出力例2

```
1
```

入れ換える操作が1回も発生しないので、何も出力してはいけない。

解答の雛形

```
def selection_sort(data):
    # 以下のpassを削除してから解答を入力
    pass


# ソート対象のデータ
A = [int(x) for x in input().split()]
selection_sort(A)
```

問題5

問題文

互いに異なる整数を含む数列Aが与えられる。 **挿入ソート**を用いてAを**昇順**に整列し、その過程を出力する `insertion_sort()` 関数を作成せよ。なお、ソートする際は、**要素を前方に挿入**すること。要素を後方に挿入してはいけない。

制約

- $0 \leq a \leq 100\ (a \in A)$
- Aの要素は互いに異なる。
- 入力は全て整数である。

入力

入力は次の形式で標準入力から与えられます。

> $A_0\ A_1\ ...\ A_{N-1}$ （Nは正の整数）

出力

未整列部分の最小要素を整列済み部分に挿入する操作が発生するたびに、その操作後のAをリストの形で1行に出力する。

入力例1

```
1 8 4 3 7 6 5 2 1
```

出力例1

```
1 [4, 8, 3, 7, 6, 5, 2, 1]
2 [3, 4, 8, 7, 6, 5, 2, 1]
3 [3, 4, 7, 8, 6, 5, 2, 1]
4 [3, 4, 6, 7, 8, 5, 2, 1]
5 [3, 4, 5, 6, 7, 8, 2, 1]
6 [2, 3, 4, 5, 6, 7, 8, 1]
```

```
7 [1, 2, 3, 4, 5, 6, 7, 8]
```

入力例2

```
1 1 2 3 4 5 6 7 8 9 10
```

出力例2

```
1 [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
2 [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
3 [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
4 [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
5 [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
6 [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
7 [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
8 [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
9 [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

解答の雛形

```
def insertion_sort(data):
    # 以下のpassを削除してから解答を入力
    pass


# ソート対象のデータ
A = [int(x) for x in input().split()]
insertion_sort(A)
```

問題6

問題文

互いに異なる整数を含む数列Aが与えられる。 **挿入ソート**を用いてAを**降順**に整列し、その過程を出力する `insertion_sort()` 関数を作成せよ。なお、ソートする際は、**要素を前方に挿入**すること。要素を後方に挿入してはいけない。

制約

- $0 \leq a \leq 100 \, (a \in A)$
- Aの要素は互いに異なる。
- 入力は全て整数である。

入力

入力は次の形式で標準入力から与えられます。

> $A_0 \ A_1 \ ... \ A_{N-1}$ （Nは正の整数）

出力

追加要素を整列部分の適切な位置への配置が行われるたびに、その操作後のAをリストの形で1行に出力する。

入力例1

```
8 4 3 7 6 5 2 1
```

出力例1

```
[8, 4, 3, 7, 6, 5, 2, 1]
[8, 4, 3, 7, 6, 5, 2, 1]
[8, 7, 4, 3, 6, 5, 2, 1]
[8, 7, 6, 4, 3, 5, 2, 1]
[8, 7, 6, 5, 4, 3, 2, 1]
[8, 7, 6, 5, 4, 3, 2, 1]
[8, 7, 6, 5, 4, 3, 2, 1]
```

入力例2

```
10 9 8 7 6 5 4 3 2 1
```

出力例2

```
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

解答の雛形

```python
def insertion_sort(data):
    # 以下のpassを削除してから解答を入力
    pass


# ソート対象のデータ
A = [int(x) for x in input().split()]
insertion_sort(A)
```

問題7

問題文

互いに異なる整数を含む数列Aが与えられる。 **マージソート**を用いてAを**昇順**に整列し、その過程を出力する `merge_sort()` 関数と分割した区間を整列しながらマージを行う `merge()` 関数を作成せよ。

`merge_sort()` 関数は以下の制約を満たすものとします。

- 数列の対象区間を分割するときには、中央（番号の端数を切り捨て）で二等分すること。
- 数列の対象区間の要素数が 1 以下の場合は、何も処理をせずスキップすること。
- 分割された各区間を再び分割する（再帰処理する）ときには、数列の先頭側（左側）の区間を先に処理すること。
- 分割した区間を再度マージする際には、 `merge()` 関数を使用すること。

制約

- $0 \leq a \leq 100\,(a \in A)$
- Aの要素は互いに異なる。
- 入力は全て整数である。

入力

入力は次の形式で標準入力から与えられます。

> $A_0\ A_1\ ...\ A_{N-1}$ （Nは正の整数）

出力

区間をマージするたびに、新しく作られたリストを1行で出力する。

入力例1

```
8 4 3 7 6 5 2 1
```

出力例1

```
[4, 8]
[3, 7]
[3, 4, 7, 8]
[5, 6]
[1, 2]
[1, 2, 5, 6]
[1, 2, 3, 4, 5, 6, 7, 8]
```

入力例2

```
1 2 3 4 5 6 7 8 9 10
```

出力例2

```
[1, 2]
[4, 5]
[3, 4, 5]
[1, 2, 3, 4, 5]
[6, 7]
[9, 10]
[8, 9, 10]
[6, 7, 8, 9, 10]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

解答の雛形

```
def merge_sort(data):
    # 以下のpassを削除してから解答を入力
    pass


def merge(left, right):
    # 以下のpassを削除してから解答を入力
    pass


# ソート対象のデータ
A = [int(x) for x in input().split()]
merge_sort(A)
```

問題8

問題文

互いに異なる整数を含む数列Aが与えられる。 **マージソート**を用いてAを**降順**に整列し、その過程を出力する `merge_sort()` 関数と分割した区間を整列しながらマージを行う `merge()` 関数を作成せよ。

`merge_sort()` 関数は以下の制約を満たすものとします。

- 数列の対象区間を分割するときには、中央（番号の端数を切り捨て）で二等分すること。
- 数列の対象区間の要素数が 1 以下の場合は、何も処理をせずスキップすること。
- 分割された各区間を再び分割する（再帰処理する）ときには、数列の先頭側（左側）の区間を先に処理すること。
- 分割した区間を再度マージする際には、 `merge()` 関数を使用すること。

制約

- $0 \leq a \leq 100 \, (a \in A)$
- Aの要素は互いに異なる。
- 入力は全て整数である。

入力

入力は次の形式で標準入力から与えられます。

> $A_0 \ A_1 \ ... \ A_{N-1}$ （Nは正の整数）

出力

区間をマージするたびに、新しく作られたリストを1行で出力する。

入力例1

```
1 8 4 3 7 6 5 2 1
```

出力例1

```
[8, 4]
[7, 3]
[8, 7, 4, 3]
[6, 5]
[2, 1]
[6, 5, 2, 1]
[8, 7, 6, 5, 4, 3, 2, 1]
```

入力例2

```
10 9 8 7 6 5 4 3 2 1
```

出力例2

```
[10, 9]
[7, 6]
[8, 7, 6]
[10, 9, 8, 7, 6]
[5, 4]
[2, 1]
[3, 2, 1]
[5, 4, 3, 2, 1]
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

解答の雛形

```python
def merge_sort(data):
    # 以下のpassを削除してから解答を入力
    pass


def merge(left, right):
    # 以下のpassを削除してから解答を入力
    pass


# ソート対象のデータ
A = [int(x) for x in input().split()]
merge_sort(A)
```

13章: アプリケーション開発（1）

アプリケーション開発の流れ

13章から15章にかけて、これまで学習した知識を使いコンソールアプリケーションの開発を行う。

各章では以下の内容を扱う。

- 13章では、制御構文・関数・コレクションを用いてコンソールアプリケーションを開発する。与えられた仕様を実現する過程を通じて、仕様に対して適切なプログラムを記述する力を養う。
- 14章では、クラスを用いて、13章で開発したコンソールアプリケーションを改造する。制御構文や関数のみで書かれたプログラムをクラスを用いたプログラムに変える過程を通じて、オブジェクト指向プログラミングへの理解を行う。
- 15章では、継承を用いて、14章で開発したコンソールアプリケーションを改造する。実践的な継承の利用を通じて、オブジェクト指向プログラミングへのより深い理解を行う。

コンソールアプリケーション

コンソールアプリケーションは、Windowsにおけるコマンドプロンプトや、macOSにおけるターミナルなどのようなテキストベースのインターフェースで動作するアプリケーションである。コンソールアプリケーションでは、テキストインターフェースを通して文字列の入力や出力を行う。

コンソールアプリケーションは、高度な画像やグラフィカルなインターフェースを必要としない処理や、サーバにおいてバックグラウンドで動作する処理などによく使われる。また、データのインポートやエクスポート、バッチ処理などにも使われる。

コンソールアプリケーションを作成するには、言語によって異なるが、標準入力や標準出力、コマンドライン引数を処理する機能を使用することが多い。言語やプラットフォームによって異なるが、コンソールアプリケーションを作成するためのフレームワークやライブラリも存在する。

ここまで講義で取り扱ったプログラムは全てテキストインターフェース上で動作するため、これらもまたコンソールアプリケーションである。たとえば、2つの数字を標準入力から与え、与えた数字を足した結果を標準出力に出力するプログラムもコンソールアプリケーションの一つである。

今回開発するのは、標準入力・標準出力を使い、入力した内容によって盤面が変化するゲーム形式のコンソールアプリケーションである。

ゲームの雛形プログラム

ゲームを実現する方法の一つに無限ループを使った方法がある。無限ループは、明示的に処理を打ち切らない限りは繰り返し処理がいつまでも続くようなループである。Pythonにおいて無限ループでゲームを実現するには、次のように実装する。

```
while True:
    # ゲームのロジックを記述する
```

上記のように、`while True:`とすることで無限ループを実現できる。この無限ループの中で、ゲームに必要な処理を記述していく。

無限ループ内では、ゲームが終了するなどの特定の条件が満たされた場合に、`break`や`return`を使用してループを抜けるようにしておくことが推奨される。特に、ユーザーからの入力を受け付ける場合は、特定の入力があった場合にループを抜けるようにしておくとよい。

以下に、今回の開発の雛形となるゲームのプログラムを示す。

```
while True:
    # ユーザーからの入力を受け付ける
    line = input()
    # 入力された文字列が"end"の場合は、無限ループを終了する
    if line == "end":
        print("終了")
        break
    # 入力された文字列を出力する
```

```
    print(line)
```

```
hogehoge
hogehoge
end
終了
```

上記のプログラムは、無限ループの中でユーザーからの入力を受け付け、 `end` という文字列を受け取った場合は `終了` と出力し、 `break` を使って無限ループを終了する。 `end` ではない文字列を受け取った場合は、入力された文字列を出力する。

以下が、上記のプログラムの動作の詳細である。

1. `while True:` で無限ループを開始する。
2. `input()` 関数でユーザーからの入力を受け付ける。
3. 入力された文字列が `end` の場合は、 `終了` と出力し無限ループを終了する。そうでない場合は、入力された文字列を出力し、手順2.の処理に戻る。

このように、上記のプログラムでは、無限ループ内でユーザーからの入力を受け付け、特定の文字列を受け取った場合に無限ループを終了する。

このプログラムをベースにしてゲームを開発する。

ゲームの仕様

今回開発するゲームとして、縦5マス横5マスのフィールド上で、駒を配置してプレイヤーが敵を取り除くゲームを実装する。

ゲームの仕様は以下である。

- フィールドは縦5マス横5マスであり、プレイヤーの駒がいるマスは `S` 、敵がいるマスは `*` 、それ以外のマスには `.` が表示される。
- プレイヤーの駒の初期配置はフィールドの中心。敵の初期配置はフィールドの左上隅として、ゲーム開始時に以下のように出力される。駒と敵はそれぞれ縦方向の座標と横方向の座標を変数として持ち、フィールドの出力時に使用する。フィールドの左上隅の座標が横方向 `0` 、縦方向 `0` の座標、フィールドの右下隅が横方向 `4` 、縦方向 `4` の座標である。

```
*....
.....
..S..
.....
.....
```

- プレイヤーは自分の駒 `S` を移動できる。移動するには移動先の座標を入力する。横方向の座標と縦方向の座標はそれぞれ `0` から `4` の数字で入力できる。駒の座標は `1 2` のように `横方向の座標 縦方向の座標` の順で標準入力から与える。これを繰り返し、敵がフィールド上からなくなるか、プレイヤーが `end` と入力するまで続けられる。

たとえば、 `1 2` と与えると以下のようなフィールドの状態になる。

```
*....
.....
.S...
.....
.....
```

- 駒が移動するたびに、現在のフィールドの状態を表示する。
- 敵がフィールド上からなくなり、ゲームが終了した場合、 `ゲームクリア！` のメッセージが表示される。プレイヤーが `end` と入力してゲームが終了した場合、 `ゲームを強制終了` と表示される。

雛形プログラムを元に上記の仕様を満たすようなプログラムを記述していく。

二次元リスト

まず、以下の仕様を実現するために、二次元リストを使うという考えもあるため、知識として、二次元リストについて説明する。

- フィールドは縦5マス横5マスであり、プレイヤーの駒がいるマスは `S` 、敵がいるマスは `*` 、それ以外のマスには `.` が表示される。

Pythonでは、リストを使用することで、複数のデータを1つの変数に格納することができる。リスト内にさらにリストを格納することで、二次元リストを作成することができる。

二次元リストを使用すると、行列やマス目のようなデータを扱うことができる。例えば、画像データを表す場合や、EXCELシートのようなデータを扱う場合などによく使われる。

二次元リストの例を以下に示す。

```
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]
```

上記の例では、 `matrix` という二次元リストが3つのリストから構成されている。

二次元リストの要素にアクセスするには、2つのインデックスを使用する。例えば、matrixの上から2行目、左から3列目の要素にアクセスするには、以下のようにする。

```
element = matrix[1][2]
```

上記の例では、 `element` には `6` が格納される。

たとえば、縦に5個、横に5個 `.` を出力するプログラムは以下のように記述できる。

```
field = [
    [".", ".", ".", ".", "."],
    [".", ".", ".", ".", "."],
    [".", ".", ".", ".", "."],
    [".", ".", ".", ".", "."],
    [".", ".", ".", ".", "."]
]

for line in field:
    print(line[0]+line[1]+line[2]+line[3]+line[4])
```

```
.....
.....
.....
.....
.....
```

上記のサンプルプログラムでは、 `field` に `.` を五つ並べたリスト（ `[".", ".", ".", ".", "."]` ）を、五つ並べた二次元リストである。二次元リストを `for` 文にかけることで、二次元リストのリストを一つずつ取り出すことができる。サンプルプログラムの `for` 文の処理で、 `field` の格納されている、リスト（ `[".", ".", ".", ".", "."]` ）を一つずつ取り出し、 `line` に格納し、それぞれの要素をインデックスで参照して、出力している。

ただ、上記のサンプルプログラムを元に今回の仕様を満たそうとした場合、 `field` に格納されている二次元リストを移動の度に書き換えを行

わなければならなくなり、実装が複雑になる。また、出力方法もインデックスを一つずつ参照する方法では、もし `field` に格納されている二次元リストが縦9マス横9マスや、縦19マス横19マスなど変わった場合に、都度出力の処理も変更しなければならず、修正範囲が多くなってしまうという問題がある。そのため、今回の実装では、フィールドの出力のたびに、表示するフィールドのリストを生成し、出力をするという実装を行う。

以下が、表示するフィールドのリストを生成し出力するプログラムの例である。

```
# リストを作成する
field = []
# y座標を表す変数を0から4までループ
for y in range(5):
    # x座標を表す変数を0から4までループ
    for x in range(5):
        # フィールドのマスを"."で表す
        field.append(".")
    # x座標を表す変数が4まで到達したら改行文字を追加する
    field.append("\n")
# 作成したリストを文字列に変換して表示する
print("".join(field))
```

```
.....
.....
.....
.....
.....
```

上記のサンプルプログラムでは、 `for` 文による二重ループを使い、縦5マス横5マスのフィールドを表すリストを生成し、最後に生成したリストを文字列に変換して表示している。

最初に、 `field` を空のリストとして定義している。次に、外側の `for` 文で `range()` 関数を使って5回ループさせる。次に、内側の `for` 文で `range()` 関数を使って5回ループさせる。内側の `for` 文では、 `.` という文字を `field` に追加し内側の `for` 文のループ処理が全て終わるたびに、 `field` に改行文字である `\n` を追加する。

二重ループの処理を終えると、 `field` には以下のようなリストが生成される。

```
['.', '.', '.', '.', '.', '\n', '.', '.', '.', '.', '.', '\n', '.', '.', '.', '.', '.', '\n', '.', '.', '.', '.', '.'
```

最後に、 `join()` 関数を使用して `field` を文字列に変換し、それを表示している。 `join()` 関数を使うことで、リストに格納されている要素を連結することができる。 `<連結時に間に挿入する文字列>.join(<連結したい文字列のリスト>)` で使うことができる。 `"".join(field)` では、 `field` の中のリストの要素を全て連結し、連結するとき文字列の間には何も文字列を挟まないという処理を行っている。

結果として、縦5マス横5マスのフィールドが表される文字列が表示される。

プログラムの実装1

ここからはゲームの雛形プログラムを元に、仕様を満たすプログラムを実装していく。

まず、以下の仕様を満たすプログラムを考えてみよう。

- フィールドは縦5マス横5マスであり、プレイヤーの駒がいるマスは `S` 、敵がいるマスは `*` 、それ以外のマスには `.` が表示される。
- プレイヤーの駒の初期配置はフィールドの中心。敵の初期配置はフィールドの左上隅として、ゲーム開始時に以下のように出力される。駒と敵はそれぞれ縦方向の座標と横方向の座標を変数として持ち、フィールドの出力時に使用する。フィールドの左上隅の座標が横方向 `0` 、縦方向 `0` の座標、フィールドの右下隅が横方向 `4` 、縦方向 `4` の座標である。

```
*....
```

```
.....
..S..
.....
.....
```

仕様から、プレイヤーの駒と敵の縦方向の座標（y軸）と横方向の座標（x軸）をそれぞれ変数として、まず定義する必要があることがわかる。また、駒と敵の座標の初期値は、駒はフィールドの中心のため、縦方向の座標（y軸）と横方向の座標（x軸）はそれぞれ `2` と `2` 、敵はフィールドの左上隅のため、縦方向の座標（y軸）と横方向の座標（x軸）はそれぞれ `0` と `0` であることがわかる。

また、フィールド上のプレイヤーの駒がいるマスに `S` を、敵がいるマスに `*` を表示するため、フィールドを表示するリストを生成する際に、それぞれの座標の位置の文字をリストに追加するタイミングで `.` ではなく、 `S` や `*` をリストに追加する必要がある。

雛形となるプログラムに、上記の仕様を追加したプログラムが以下である。

```python
# 座標を表す変数を定義
player_x, player_y = 2, 2
enemy_x, enemy_y = 0, 0

# リストを作成する
field = []
# 縦方向の座標（y座標）を表す変数を0から4までループ
for y in range(5):
    # 横方向の座標（x座標）を表す変数を0から4までループ
    for x in range(5):
        # 駒が配置されている座標を"S"で表す
        if x == player_x and y == player_y:
            field.append("S")
        # 敵が配置されている座標を"*"で表す
        elif x == enemy_x and y == enemy_y:
            field.append("*")
        # それ以外の位置は"."で表す
        else:
            field.append(".")
    # x座標を表す変数が4まで到達したら改行する
    field.append("\n")
# 作成したリストを文字列に変換して表示する
print("".join(field))

while True:
    # ユーザーからの入力を受け付ける
    line = input()
    # 入力された文字列が"end"の場合は、無限ループを終了する
    if line == "end":
        print("終了")
        break
    # 入力された文字列を出力する
    print(line)
```

```
*....
.....
..S..
.....
.....

end
終了
```

上記のプログラムは、ゲームの雛形プログラムに、プレイヤーの駒と敵の初期配置を行ったフィールドを表示するプログラムを追加したものである。

初期配置の表示は一度しか行わないため、無限ループを実行する前に、初期配置の表示のを行うプログラムを記述している。

最初に、プレイヤーの駒の座標として `player_x` と `player_y` を、敵の座標として `enemy_x` と `enemy_y` を定義している。初期値とし

て `player_x` と `player_y` にはそれぞれに `2` を、 `enemy_x` と `enemy_y` にはそれぞれ `0` を代入している。

続いて、フィールドの表示に使用する `field` を、空のリストとして定義する。

続いて、 `y` を `0` から `4` までの5つの整数でループさせる。続いて、 `x` を `0` から `4` までの5つの整数でもう1つのループを設定する。それぞれのループで、以下の処理を実行する。

`x` と `y` が `player_x` と `player_y` にそれぞれ等しい場合、 `field` に `S` を追加する。 `x` と `y` が `enemy_x` と `enemy_y` にそれぞれ等しい場合、 `field` に `*` を追加する。それ以外の場合、 `field` に `.` を追加する。 `x` が `4` に到達するたびに、 `field` に改行文字 `\n` を追加する。

最後に、 `join()` 関数を使用して `field` を文字列に変換し、表示する。結果として、縦5マス横5マスのフィールドが表される文字列が表示される。このフィールドには、 `S` でプレイヤーの駒が、 `*` で敵の駒が表される。それ以外のマスは、 `.` で表される。

以上の実装を行うことで以下の仕様を満たすことができる。

- フィールドは縦5マス横5マスであり、プレイヤーの駒がいるマスは `S` 、敵がいるマスは `*` 、それ以外のマスには `.` が表示される。
- プレイヤーの駒の初期配置はフィールドの中心。敵の初期配置はフィールドの左上隅として、ゲーム開始時に以下のように出力される。駒と敵はそれぞれ縦方向の座標と横方向の座標を変数と持ち、フィールドの出力時に使用する。フィールドの左上隅の座標が横方向 `0` 、縦方向 `0` の座標、フィールドの右下隅が横方向 `4` 、縦方向 `4` の座標である。

```
*....
.....
..S..
.....
.....
```

プログラムの実装2

続いて、以下の仕様を満たすプログラムを考えてみよう。

- プレイヤーは自分の駒 `S` を移動できる。移動するには移動先の座標を入力する。横方向の座標と縦方向の座標はそれぞれ `0` から `4` の数字で入力できる。駒の座標は `1 2` のように横方向の座標 縦方向の座標の順で標準入力から与える。これを繰り返し、敵がフィールド上からなくなるか、プレイヤーが `end` と入力するまで続けられる。

仕様から、以下のような処理を追加する必要があることがわかる。

- 無限ループの中で入力した文字列を判定し、 `end` であれば無限ループを抜ける処理。
- 入力された文字列が `end` 以外であれば、二つに分割し、一つ目の数字を横方向の座標、二つ目の数字を縦方向の座標として、プレイヤーの駒の座標を更新する処理。
- 更新後のプレイヤーの駒の座標が敵の座標と一致すれば、無限ループを抜ける処理。

以上の処理を実装したプログラムが以下である。この時点で、雛形プログラムに記述されていた、入力した文字列を表示する処理は取り除いている。

```python
# 座標を表す変数を定義
player_x, player_y = 2, 2
enemy_x, enemy_y = 0, 0

# リストを作成する
field = []
# 縦方向の座標（y座標）を表す変数を0から4までループ
for y in range(5):
    # 横方向の座標（x座標）を表す変数を0から4までループ
    for x in range(5):
        # 駒が配置されている座標を"S"で表す
        if x == player_x and y == player_y:
            field.append("S")
```

```
        # 敵が配置されている座標を"*"で表す
        elif x == enemy_x and y == enemy_y:
            field.append("*")
        # それ以外の位置は"."で表す
        else:
            field.append(".")
    # x座標を表す変数が4まで到達したら改行する
    field.append("\n")
# 作成したリストを文字列に変換して表示する
print("".join(field))

while True:
    # ユーザーからの入力を受け付ける
    line = input()
    # 入力された文字列が"end"の場合は、無限ループを終了する
    if line == "end":
        print("終了")
        break
    # 入力された座標を使用して、駒オブジェクトを配置する
    player_x, player_y = map(int, line.split())

    # ゲーム終了判定
    if enemy_x == player_x and enemy_y == player_y:
        break
```

```
*....
.....
..S..
.....
.....

1 1
0 0
```

上記のプログラムは、前節のプログラムに、本節で実装する仕様を追加したプログラムである。ユーザーの入力を判断して、ゲームを終了したり、プレイヤーの駒の座標を更新する処理を行うため、無限ループの中で実装している。

それぞれの処理の実装方法について解説する。

- 無限ループの中で入力した文字列を判定し、`end`であれば無限ループを抜ける処理。

上記の処理は雛形プログラムにあった処理をそのまま利用している。

- 入力された文字列が`end`以外であれば、二つに分割し、一つ目の数字を横方向の座標、二つ目の数字を縦方向の座標として、プレイヤーの駒の座標を更新する処理。

上記の処理は、`end`が入力されたときに無限ループを抜ける処理の次に、以下の処理を記述することで実装している。

```
player_x, player_y = map(int, line.split())
```

この処理では、まず、`split()`関数を使って、入力を半角スペース区切りで分割し、リストを作っている。たとえば、入力が`1 2`の場合、`line.split()`で`["1", "2"]`を生成している。

続いて、`map()`関数を使い、生成したリストのそれぞれの要素を整数型に直し、`player_x`と`player_y`にそれぞれ代入している。

以上の処理を行うことで、たとえば、入力が`1 2`の場合、`player_x`には`1`が、`player_y`には`2`が代入される。

`map()`関数を使わない場合は、以下のような処理で置き換えることもできる。

```
tmpList = line.split()
```

```
player_x = int(tmpList[0])
player_y = int(tmpList[1])
```

- 更新後のプレイヤーの駒の座標が敵の座標と一致すれば、無限ループを抜ける処理。

上記の処理は、駒の座標の更新処理を行った次に、以下の処理を記述することで実装している。

```
if enemy_x == player_x and enemy_y == player_y:
    break
```

駒の座標を更新した後に、 `enemy_x` と `player_x` の値が等しく、かつ、 `enemy_y` と `player_y` の値が等しい場合、駒と敵の座標が一致していると考えられるため、 `break` で無限ループを抜けている。

プログラムの実装3

続いて、以下の仕様を満たすプログラムを考えてみよう。

- 駒が移動するたびに、現在のフィールドの状態を表示する。

仕様から、初期配置の表示に使用した以下のプログラムをそのまま、駒の座標を更新したあとに追加すればよいことがわかる。

```
field = []
for y in range(5):
    for x in range(5):
        if x == player_x and y == player_y:
            field.append("S")
        elif x == enemy_x and y == enemy_y:
            field.append("*")
        else:
            field.append(".")
     field.append("\n")
print("".join(field))
```

ただし、同様の処理を複数回行うときには、共通する処理を関数にまとめて、必要な箇所で関数を呼び出すようにした方がよい。

共通する処理を関数にして、仕様を満たすような実装したプログラムが以下である。

```
# 座標を表す変数を定義
player_x, player_y = 2, 2
enemy_x, enemy_y = 0, 0

# 敵や自分の駒が配置されている座標を表示する関数
def show_field():
    # リストを作成する
    field = []
    # 縦方向の座標（y座標）を表す変数を0から4までループ
    for y in range(5):
        # 横方向の座標（x座標）を表す変数を0から4までループ
        for x in range(5):
            # 自分の駒が配置されている座標を"S"で表す
            if x == player_x and y == player_y:
                field.append("S")
            # 敵が配置されている座標を"*"で表す
            elif x == enemy_x and y == enemy_y:
                field.append("*")
            # それ以外の位置は"."で表す
            else:
                field.append(".")
        # x座標を表す変数が4まで到達したら改行する
        field.append("\n")
    # 作成したリストを文字列に変換して表示する
```

```
    print("".join(field))


# フィールドの状態を表示
show_field()

while True:
    # 駒オブジェクトの配置位置を指定する
    line = input()
    # 入力された文字列が"end"の場合は、無限ループを終了する
    if line == "end":
        print("終了")
        break
    # 入力された座標を使用して、駒オブジェクトを配置する
    player_x, player_y = map(int, line.split())

    # フィールドの状態を表示
    show_field()

    # ゲーム終了判定
    if enemy_x == player_x and enemy_y == player_y:
        break
```

上記のプログラムは、前節のプログラムに、本節で実装する仕様を追加したプログラムである。フィールドの状態を表示する処理を `show_field()` という関数にまとめ、無限ループに入る前とプレイヤーの駒の座標を更新した後で、フ `show_field()` 関数を呼び出しているプログラムである。

無限ループに入る前に、 `show_field()` 関数を呼び出すことで、初期配置後のフィールドの状態を表示できる。無限ループ内でプレイヤーの駒の座標を更新したあとに、 `show_field()` 関数を呼び出すことで、更新後の駒の座標を反映したフィールドの状態を表示できる。

プログラムの実装4

最後に、以下の仕様を満たすプログラムを考えてみよう。

- 敵がフィールド上からなくなり、ゲームが終了した場合、 `ゲームクリア！` のメッセージが表示される。プレイヤーが `end` と入力してゲームが終了した場合、 `ゲームを強制終了` と表示される。

この仕様を満たすには、 `print("ゲームクリア！")` という処理と、 `print("ゲームを強制終了")` という処理を、それぞれ、駒の座標と敵の座標が一致したときと `end` が入力されたときの `if` 文内に追加すれば満たすことができる。

全ての仕様を満たしたプログラムが以下である。

```
# 座標を表す変数を定義
player_x, player_y = 2, 2
enemy_x, enemy_y = 0, 0

# 敵や自分の駒が配置されている座標を表示する関数
def show_field():
    # リストを作成する
    field = []
    # 縦方向の座標（y座標）を表す変数を0から4までループ
    for y in range(5):
        # 横方向の座標（x座標）を表す変数を0から4までループ
        for x in range(5):
            # 自分の駒が配置されている座標を"S"で表す
            if x == player_x and y == player_y:
                field.append("S")
            # 敵が配置されている座標を"*"で表す
            elif x == enemy_x and y == enemy_y:
                field.append("*")
            # それ以外の位置は"."で表す
            else:
                field.append(".")
        # x座標を表す変数が4まで到達したら改行する
        field.append("\n")
    # 作成したリストを文字列に変換して表示する
```

```
    print("".join(field))


# フィールドの状態を表示
show_field()

while True:
    # 駒オブジェクトの配置位置を指定する
    line = input()
    # 入力された文字列が"end"の場合は、無限ループを終了する
    if line == "end":
        print("ゲームを強制終了")
        break
    # 入力された座標を使用して、駒オブジェクトを配置する
    player_x, player_y = map(int, line.split())

    # フィールドの状態を表示
    show_field()

    # ゲーム終了判定
    if enemy_x == player_x and enemy_y == player_y:
        print("ゲームクリア！")
        break
```

```
*....
.....
..S..
.....
.....

1 1
*....
.S...
.....
.....
.....

0 0
S....
.....
.....
.....
.....

ゲームクリア！
```

上記のプログラムについて解説する。

このプログラムは、縦5マス横5マスのフィールド上にプレイヤーの駒と敵を配置し、駒を移動して敵を取るゲームを実装している。

まず、プレイヤーの駒の座標を表す `player_x` と `player_y` を宣言して、それぞれに初期の座標である `2` を代入している。また、敵の座標を表す `enemy_x` と `enemy_y` を宣言して、それぞれに初期の座標である `0` を代入している。

続いて、 `show_field()` 関数が定義している。この関数は、フィールドの状態を表示するために使用する。フィールドは、縦5マス横5マスで表される。各マスには、 `S` （プレイヤーの駒が配置されているマス）、 `*` （敵が配置されているマス）、 `.` （駒が配置されていないマス）のいずれかで表される。

`show_field()` 関数の中の処理では、まず、空のリスト `field` を作成する。次に、縦方向の座標（y座標）を表す変数を `0` から `4` までループさせる。そして、横方向の座標（x座標）を表す変数を `0` から `4` までループさせる。

このとき、 `field` に以下の内容を追加する。

- プレイヤーの駒が配置されている座標には `S`
- 敵が配置されている座標には `*`

- それ以外の座標には `.`

最後に、 `field` を文字列に変換して表示する。

`show_field()` 関数のあとには、フィールドの初期状態を表示するために、 `show_field()` 関数を呼び出す。

続いて、無限ループが開始する。

無限ループの中の処理では、まず、駒オブジェクトの配置位置を指定する入力を受け付ける。入力された文字列は `line` に格納する。 `line` が `end` の場合、無限ループを終了し、ゲームを強制終了する。

そうでない場合、入力された座標を使用して駒の座標を更新する。

その後、駒の更新した座標を表示するため、 `show_field()` 関数を呼び出し、フィールドの状態を表示する。

最後に、駒が敵と同じ座標に配置されているかどうかを判定し、敵と同じ座標に配置されている場合は、ゲームクリアと表示し、無限ループを終了する。

そうでない場合は、無限ループが続けられ、駒の座標を指定する入力を受け付ける状態に戻る。

以上が、仕様を全て満たすプログラムの解説である。

追加のゲーム仕様

ここまで作ったゲームでは、駒を自由な位置に移動できるため、最初に `0 0` を入力すれば、ゲームが終わってしまう。

すぐゲームが終わってしまう問題を回避するために、駒の移動に関する仕様を以下に変更する。

- 駒は一回の操作で上下左右のいずれかに1マス移動できる。標準入力から移動方向を上下左右で与えられると、受け取った方向に1マス移動する。 `up` と受け取ると上に1マス、 `down` と受け取ると下に1マス、 `left` と受け取ると左に1マス、 `right` と受け取ると右に1マス移動する。それ以外の文字列を受け取ると、フィールドの状態は表示せず、 `無効な方向です` と表示され、再度方向を入力できるようになる。

前節までに開発したプログラムを元に上記の仕様を満たすようなプログラムを記述していく。

プログラムの実装5

駒の移動の仕様が変更されたため、駒の移動に関する以下の処理を削除し、追加の仕様を満たす処理を追加する必要がある。

```
1 player_x, player_y = map(int, line.split())
```

入力された文字列に応じて、 `player_x` と `player_y` の値を増減することで、上下左右への駒の移動を実現できる。 `up` と入力されたら、 `player_y` の値を1つ減らし、 `down` と入力されたら、 `player_y` の値を1つ増やす。 `left` と入力されたら、 `player_x` の値を1つ減らし、 `right` と入力されたら、 `player_x` の値を1つ増やすことで、駒を移動できる。

また、無効な文字列が入力された場合、 `continue` を使うことで後続の処理を一度スキップして、再度ループに戻ることで、フィールドの状態を表示せず、再度入力を受け付けることができる。

以下が、追加の仕様を実現したプログラムである。入力した値を付け取っていた `line` は、方向を意味する英単語である `direction` に入れ替えている。

```
1  # 座標を表す変数を定義
2  player_x, player_y = 2, 2
3  enemy_x, enemy_y = 0, 0
4
5  # 敵や自分の駒が配置されている座標を表示する関数
6  def show_field():
7      # リストを作成する
8      field = []
```

```
    # 縦方向の座標（y座標）を表す変数を0から4までループ
    for y in range(5):
        # 横方向の座標（x座標）を表す変数を0から4までループ
        for x in range(5):
            # 自分の駒が配置されている座標を"S"で表す
            if x == player_x and y == player_y:
                field.append("S")
            # 敵が配置されている座標を"*"で表す
            elif x == enemy_x and y == enemy_y:
                field.append("*")
            # それ以外の位置は"."で表す
            else:
                field.append(".")
        # x座標を表す変数が4まで到達したら改行する
        field.append("\n")
    # 作成したリストを文字列に変換して表示する
    print("".join(field))


# フィールドの状態を表示
show_field()

while True:
    # 駒を移動する方向を入力する
    direction = input()
    # 入力された文字列が"end"の場合は、無限ループを終了する
    if direction == "end":
        print("ゲームを強制終了")
        break
    # 入力された方向に駒を移動する
    if direction == "up":
        player_y -= 1
    elif direction == "down":
        player_y += 1
    elif direction == "left":
        player_x -= 1
    elif direction == "right":
        player_x += 1
    else:
        print("無効な方向です")
        continue

    # フィールドの状態を表示
    show_field()

    # ゲーム終了判定
    if enemy_x == player_x and enemy_y == player_y:
        print("ゲームクリア！")
        break
```

```
*....
.....
..S..
.....
.....

up
*....
..S..
.....
.....
.....

上
無効な方向です
up
*.S..
.....
.....
.....
```

```
.....

left
*S...
.....
.....
.....
.....

left
S....
.....
.....
.....
.....

ゲームクリア！
```

上記のプログラムの修正した以下の箇所について、詳しく解説する。

```
# 入力された方向に駒を移動する
if direction == "up":
    player_y -= 1
elif direction == "down":
    player_y += 1
elif direction == "left":
    player_x -= 1
elif direction == "right":
    player_x += 1
else:
    print("無効な方向です")
    continue
```

`direction` には、ユーザーが入力した文字列が格納されている。 `direction` が `up` の場合は、 `player_y` を1減らすことで駒を上に1マス移動する。 `direction` が `down` の場合は、 `player_y` を1増やすことで駒を下に1マス移動する。 `direction` が `left` の場合は、 `player_x` を1減らすことで駒を左に1マス移動する。 `direction` が `right` の場合は、 `player_x` を1増やすことで駒を右に1マス移動する。

上記の四つ以外の文字列が `direction` に格納されていた場合、 `無効な方向です` と出力し、 `continue` を実行する。 `continue` を実行することで、以降に記述されている、 `show_field()` 関数の呼び出しや、ゲームの終了判定をスキップし、無限ループの最初の処理に戻る。 `continue` を使うことによって、 `show_field()` 関数の呼び出しを飛ばすことができるため、無効な方向が入力された場合には、フィールドの状態を表示せずに、文字列の入力を行うことができる。

以上が、追加の仕様を実装したプログラムの解説である。

問題1

問題文

二次元リストが格納された変数 `li` がある。添字を指定することで、二次元リストの中の値 `17` を出力するプログラムを記述せよ。

制約

- 追記するプログラムでは、 `li` の中に格納されている値(8 など)を直接記載せず、 `li` の要素を添字で参照すること。

出力

```
17
```

解答の雛形

```
li = [
    [1, 2, 3, 4, 5],
    [6, 7, 8, 9, 10],
    [11, 12, 13, 14, 15],
    [16, 17, 18, 19, 20],
    [21, 22, 23, 24, 25],
]

# ここに解答を入力
```

問題2

問題文

このプログラムは、テキストの13章で取り扱った、縦5マス横5マスのフィールド上に駒を配置して、駒の移動先の座標をプレイヤーが指定することで、敵を取り除くゲームである。

横方向の座標Xと縦方向の座標Yを入力して、プレイヤーの駒 `S` を敵 `*` の位置に移動すればゲームクリアとなりゲームを終了する。

フィールドの左上隅の座標が横方向0、縦方向0の座標、フィールドの右下隅が横方向4、縦方向4の座標である。

また、 `end` と入力されると、 `ゲームを強制終了` と表示して、ゲームを終了する。

ゲームの初期配置は以下である。

```
*....
.....
..S..
.....
.....
```

このプログラムに対して処理の追加や修正を行い、フィールド上でのプレイヤーの駒の表示を `S` から `P` に変更せよ。

制約

- $0 \leq X \leq 4$
- $0 \leq Y \leq 4$
- XとYは整数である。

入力

駒を移動する場合は、横方向の座標Xと縦方向の座標Yを1行で以下のように入力する。

```
X Y
X Y
...
```

また、ゲームを強制終了する場合は、最後に `end` を1行で入力する。

```
end
```

出力

入力した値に基づいて、以下のような駒を移動した盤面が表示される。

```
*....
.....
..P..
.....
.....
```

ゲームをクリアしたら、最後に `ゲームクリア！` と表示される。

ゲームを強制終了したら、最後に `ゲームを強制終了` と表示される。

入力例1

```
0 0
```

出力例1

```
*....
.....
..P..
.....
.....

P....
.....
.....
.....
.....

ゲームクリア！
```

入力例2

```
0 1
0 0
```

出力例2

```
*....
.....
..P..
.....
.....

*....
P....
.....
.....
.....

```

```
P....
.....
.....
.....
.....

ゲームクリア！
```

解答の雛形

```
player_x, player_y = 2, 2
enemy_x, enemy_y = 0, 0


def show_field():
    field = []
    for y in range(5):
        for x in range(5):
            if x == player_x and y == player_y:
                field.append("S")
            elif x == enemy_x and y == enemy_y:
                field.append("*")
            else:
                field.append(".")
        field.append("\n")
    print("".join(field))


show_field()

while True:
    line = input()
    if line == "end":
        print("ゲームを強制終了")
        break
    player_x, player_y = map(int, line.split())

    show_field()

    if enemy_x == player_x and enemy_y == player_y:
        print("ゲームクリア！")
        break
```

問題3

問題文

問題2と同様、このプログラムは、テキストの13章で取り扱った、縦5マス横5マスのフィールド上に駒を配置して、駒の移動先の座標をプレイヤーが指定することで、敵を取り除くゲームである。

横方向の座標Xと縦方向の座標Yを入力して、プレイヤーの駒 `S` を敵 `*` の位置に移動すればゲームクリアとなりゲームを終了する。

フィールドの左上隅の座標が横方向0、縦方向0の座標、フィールドの右下隅が横方向4、縦方向4の座標である。

また、 `end` と入力されると、 `ゲームを強制終了` と表示して、ゲームを終了する。

ゲームの初期配置は以下である。

```
*....
.....
```

```
..S..
.....
.....
```

このプログラムに対して処理の追加や修正を行い、XかY$に `0` から `4` までの数字以外の値が入力されたら、 `0から4までの数字を入力してください` と表示し再入力を促す処理を追加せよ。

制約

- $-10 \leq X \leq 10$
- $-10 \leq Y \leq 10$
- XとYは整数である。

入力

入力は次の形式で与えられる。

駒を移動する場合は、横方向の座標Xと縦方向の座標Yを1行で以下のように入力する。

```
X Y
X Y
...
```

また、ゲームを強制終了する場合は、最後に `end` を1行で入力する。

```
end
```

出力

入力した値に基づいて、以下のような駒を移動した盤面が表示される。

```
*....
.....
..S..
.....
.....
```

ゲームをクリアしたら、最後に `ゲームクリア！` と表示される。

ゲームを強制終了したら、最後に `ゲームを強制終了` と表示される。

XかY$に `0` から `4` までの数字以外の値が入力されたら、 `0から4までの数字を入力してください` と表示される。

入力例1

```
0 5
-1 0
0 0
```

出力例1

```
*....
.....
```

```
..S..
.....
.....

0から4までの数字を入力してください
0から4までの数字を入力してください
S....
.....
.....
.....
.....

ゲームクリア！
```

入力例2

```
4 4
0 0
```

出力例2

```
*....
.....
..S..
.....
.....

*....
.....
.....
.....
....S

S....
.....
.....
.....
.....

ゲームクリア！
```

解答の雛形

```
player_x, player_y = 2, 2
enemy_x, enemy_y = 0, 0


def show_field():
    field = []
    for y in range(5):
        for x in range(5):
            if x == player_x and y == player_y:
                field.append("S")
            elif x == enemy_x and y == enemy_y:
                field.append("*")
            else:
                field.append(".")
        field.append("\n")
    print("".join(field))


show_field()
```

```
while True:
    line = input()
    if line == "end":
        print("ゲームを強制終了")
        break
    player_x, player_y = map(int, line.split())

    show_field()

    if enemy_x == player_x and enemy_y == player_y:
        print("ゲームクリア！")
        break
```

問題4

問題文

このプログラムは、テキストの13章で取り扱った、縦5マス横5マスのフィールド上に、駒を配置して、駒の移動する方向をプレイヤーが指定することで、駒を移動し、敵を取り除くゲームである。

`up` 、 `down` 、 `left` 、 `right` のいずれかを入力しプレイヤーの駒 `S` を移動させて、駒 `S` を敵 `*` の位置に移動すればゲームクリアとなりゲームを終了する。

フィールドの左上隅の座標が横方向0、縦方向0の座標、フィールドの右下隅が横方向4、縦方向4の座標である。

また、 `end` と入力されると、 `ゲームを強制終了` と表示して、ゲームを終了する。

ゲームの初期配置は以下である。

```
1 *....
2 .....
3 ..S..
4 .....
5 .....
```

このプログラムに対して処理の追加や修正を行い、駒の座標が盤外にでる（横方向もしくは縦方向の座標が0から4以までの数字以外の値になる）ような方向が入力されたら、 `駒が盤外に出るため再度方向を入力してください` と表示し再入力を促す処理を追加せよ。

例えば、駒を移動して以下のような配置のなったあとに、 `up` と指定すると、縦方向の座標が-1となり、駒が盤外に出てしまうため、 `駒が盤外に出るため再度方向を入力してください` と表示し再入力を促す。

```
1 *.S..
2 .....
3 .....
4 .....
5 .....
```

入力

入力は次の形式で与えられる。

駒を移動する場合は、移動方向 `DIRECTION` を1行で入力する。

```
DIRECTION
DIRECTION
...
```

移動方向 `DIRECTION` は、 `up` 、 `down` 、 `left` 、 `right` のいずれかの値が与えられる。

また、ゲームを強制終了する場合は、最後に `end` を1行で入力する。

```
end
```

出力

入力した方向に基づいて、以下のような駒を移動した盤面が表示される。

```
*....
.....
..S..
.....
.....
```

移動方向 `DIRECTION` として `up` 、 `down` 、 `left` 、 `right` 以外の文字列が入力されたら、 `無効な方向です` と表示される。

ゲームをクリアしたら、最後に `ゲームクリア！` と表示される。

ゲームを強制終了したら、最後に `ゲームを強制終了` と表示される。

駒が盤外にでるような方向が入力された、 `駒が盤外に出るため再度方向を入力してください` と表示され、直前の盤面が表示される。

入力例1

```
up
up
up
left
left
```

出力例1

```
*....
.....
..S..
.....
.....

*....
..S..
.....
.....
.....

*.S..
.....
.....
.....
.....

駒が盤外に出るため再度方向を入力してください
```

```
*.S..
.....
.....
.....
.....

*S...
.....
.....
.....
.....

S....
.....
.....
.....
.....

ゲームクリア！
```

入力例2

```
left
left
left
left
up
up
```

出力例2

```
*....
.....
..S..
.....
.....

*....
.....
.S...
.....
.....

*....
.....
S....
.....
.....

駒が盤外に出るため再度方向を入力してください
*....
.....
S....
.....
.....

駒が盤外に出るため再度方向を入力してください
*....
.....
S....
.....
.....

*....
S....
.....
```

```
.....
.....

S....
.....
.....
.....
.....

ゲームクリア！
```

解答の雛形

```
player_x, player_y = 2, 2
enemy_x, enemy_y = 0, 0


def show_field():
    field = []
    for y in range(5):
        for x in range(5):
            if x == player_x and y == player_y:
                field.append("S")
            elif x == enemy_x and y == enemy_y:
                field.append("*")
            else:
                field.append(".")
        field.append("\n")
    print("".join(field))


show_field()

while True:
    direction = input()
    if direction == "end":
        print("ゲームを強制終了")
        break
    if direction == "up":
        player_y -= 1
    elif direction == "down":
        player_y += 1
    elif direction == "left":
        player_x -= 1
    elif direction == "right":
        player_x += 1
    else:
        print("無効な方向です")
        continue

    show_field()

    if enemy_x == player_x and enemy_y == player_y:
        print("ゲームクリア！")
        break
```

問題5

問題文

問題4と同様、このプログラムは、テキストの13章で取り扱った、縦5マス横5マスのフィールド上に、駒を配置して、駒の移動する方向をプレイヤーが指定することで、駒を移動し、敵を取り除くゲームである。

`up` 、 `down` 、 `left` 、 `right` のいずれかを入力しプレイヤーの駒 `S` を移動させて、駒 `S` を敵 `*` の位置に移動すればゲームクリアとなりゲームを終了する。

フィールドの左上隅の座標が横方向0、縦方向0の座標、フィールドの右下隅が横方向4、縦方向4の座標である。

また、 `end` と入力されると、 `ゲームを強制終了` と表示して、ゲームを終了する。

ゲームの初期配置は以下である。

```
1 *....
2 .....
3 ..S..
4 .....
5 .....
```

このプログラムに対して処理の追加や修正を行い、駒の座標が盤外にでる（横方向もしくは縦方向の座標が0から4以までの数字以外の値になる）ような方向を入力されたら、盤面の反対方向に移動する処理を追加せよ。

例えば、以下の状況を考える。

```
1 *...S
2 .....
3 .....
4 .....
5 .....
```

上記の状況で、 `up` と指定すると、以下のように盤面の反対方向（盤面の一番下）に移動する。

```
1 *....
2 .....
3 .....
4 .....
5 ....S
```

続けて、上記の状況で、 `right` と入力すると、以下のように盤面の反対方向（盤面の一番右）に移動する。

```
1 *....
2 .....
3 .....
4 .....
5 S....
```

入力

入力は次の形式で与えられる。

駒を移動する場合は、移動方向 `DIRECTION` を1行で入力する。

```
1 DIRECTION
2 DIRECTION
3 ...
```

移動方向 `DIRECTION` は、 `up` 、 `down` 、 `left` 、 `right` のいずれかの値が与えられる。

また、ゲームを強制終了する場合は、 `end` を1行で入力する。

```
end
```

出力

入力した方向に基づいて、以下のような駒を移動した盤面が表示される。

```
*....
.....
..S..
.....
.....
```

移動方向 `DIRECTION` として `up` 、 `down` 、 `left` 、 `right` 以外の文字列が入力されたら、 `無効な方向です` と表示される。

ゲームをクリアしたら、最後に `ゲームクリア！` と表示される。

ゲームを強制終了したら、最後に `ゲームを強制終了` と表示される。

入力例1

```
up
up
up
left
left
down
```

出力例1

```
*....
.....
..S..
.....
.....

*....
..S..
.....
.....
.....

*.S..
.....
.....
.....
.....

*....
.....
.....
.....
..S..

*....
.....
.....
.....
.S...

*....
.....
```

```
.....
.....
S....

S....
.....
.....
.....
.....

ゲームクリア！
```

入力例2

```
left
left
left
up
up
right
```

出力例2

```
*....
.....
..S..
.....
.....

*....
.....
.S...
.....
.....

*....
.....
S....
.....
.....

*....
.....
....S
.....
.....

*....
....S
.....
.....
.....

*...S
.....
.....
.....
.....

S....
.....
.....
.....
.....

ゲームクリア！
```

解答の雛形

```
player_x, player_y = 2, 2
enemy_x, enemy_y = 0, 0


def show_field():
    field = []
    for y in range(5):
        for x in range(5):
            if x == player_x and y == player_y:
                field.append("S")
            elif x == enemy_x and y == enemy_y:
                field.append("*")
            else:
                field.append(".")
        field.append("\n")
    print("".join(field))


show_field()

while True:
    direction = input()
```

```
    if direction == "end":
        print("ゲームを強制終了")
        break
    if direction == "up":
        player_y -= 1
    elif direction == "down":
        player_y += 1
    elif direction == "left":
        player_x -= 1
    elif direction == "right":
        player_x += 1
    else:
        print("無効な方向です")
        continue

    show_field()

    if enemy_x == player_x and enemy_y == player_y:
        print("ゲームクリア！")
        break
```

14章: アプリケーション開発（2）

本章の流れ

本章では、前章まで開発したゲームに追加の実装をする過程を通して、オブジェクト指向についての理解を深める。

前章までに開発したゲームでは、プレイヤーの駒と敵がそれぞれ一つずつであったため、コレクションや制御構文、関数のみを用いた実装でも支障なく開発することができた。

本章では、まずプレイヤーの駒と敵の数を増やす実装を、前章と同様コレクションや制御構文、関数のみを用いて実装する。

プレイヤーの駒と敵が複数存在する場合に、コレクションや制御構文、関数のみを用いた実装をすることの問題点について考え、その問題を解決するような実装について検討していく。

追加のゲーム仕様

前章で開発したゲームは、移動できるプレイヤーの駒一つに敵も一つというシンプルな作りだった。今回は、プレイヤーの駒や敵を複数配置するような実装を行い、より複雑なゲームを開発する。

前章で実装したゲームに、以下の追加の仕様を満たすような実装を行う。

- プレイヤーの駒を二つ配置する。一つ目の駒の座標の初期値は縦方向に `3` 、横方向に `3` である。二つ目の駒の座標の初期値は縦方向に `4` 、横方向に `4` である。
- 敵を三つ配置する。一つ目の敵の座標の初期値は縦方向に `0` 、横方向に `0` である。二つ目の敵の座標の初期値は縦方向に `1` 、横方向に `1` である。三つ目の敵の座標の初期値は縦方向に `2` 、横方向に `2` である。また、敵は座標以外に倒されているかの判定結果を真偽値で持つ。 `Ture` であれば倒されている状態、 `False` であれば倒されていない状態を表す。初期値は三つとも倒されていないため、 `False` である。
- ゲーム開始時のフィールドの状態は以下のように表示される。

```
*....
.*...
..*..
...S.
....S
```

- 駒の移動する方向を入力して、プレイヤーの駒を交互に動かす。例えば、初めに `up` と入力すると、一つ目の駒が1マス上に移動し、続けて `up` を入力すると、二つ目の駒が1マス上に移動する。三回目に `up` と入力すると、一つ目の駒が1マス上に移動する。
- 無効な方向を入力されたら、 `ValueError` の例外を発生させて、 `無効な方向です` と表示し、強制終了する。
- プレイヤーの駒が、敵と同じ座標に移動したら、敵を倒したとみなして、以降、倒された敵はフィールドの上に表示せず、倒された敵がいた場所にプレイヤーの駒がいない場合は `.` を表示する。
- 全ての敵を倒したらゲームクリアとなり、 `ゲームクリア！` と出力し、ゲームを終了する。

前章までに開発したゲームを元に上記の仕様を満たすようなプログラムを記述していく。

前章までに開発したゲームは以下である。このプログラムでは、まず、プレイヤーの駒と敵を一つずつ配置する。配置した駒を上下左右に動かして、駒が敵と同じ座標になればゲームクリアとなる。

```
# 座標を表す変数を定義
player_x, player_y = 2, 2
enemy_x, enemy_y = 0, 0

# 敵や自分の駒が配置されている座標を表示する関数
def show_field():
    # リストを作成する
```

```
    field = []
    # 縦方向の座標（y座標）を表す変数を0から4までループ
    for y in range(5):
        # 横方向の座標（x座標）を表す変数を0から4までループ
        for x in range(5):
            # 自分の駒が配置されている座標を"S"で表す
            if x == player_x and y == player_y:
                field.append("S")
            # 敵が配置されている座標を"*"で表す
            elif x == enemy_x and y == enemy_y:
                field.append("*")
            # それ以外の位置は"."で表す
            else:
                field.append(".")
        # x座標を表す変数が4まで到達したら改行する
        field.append("\n")
    # 作成したリストを文字列に変換して表示する
    print("".join(field))


# フィールドの状態を表示
show_field()

while True:
    # 駒を移動する方向を入力する
    direction = input()
    # 入力された文字列が"end"の場合は、無限ループを終了する
    if direction == "end":
        print("ゲームを強制終了")
        break
    # 入力された方向に駒を移動する
    if direction == "up":
        player_y -= 1
    elif direction == "down":
        player_y += 1
    elif direction == "left":
        player_x -= 1
    elif direction == "right":
        player_x += 1
    else:
        print("無効な方向です")
        continue

    # フィールドの状態を表示
    show_field()

    # ゲーム終了判定
    if enemy_x == player_x and enemy_y == player_y:
        print("ゲームクリア！")
        break
```

プログラムの実装6

追加の仕様を満たすゲームを開発する。

まず、以下の仕様を満たすプログラムを考えてみよう。

- プレイヤーの駒を二つ配置する。一つ目の駒の座標の初期値は縦方向に `3` 、横方向に `3` である。二つ目の駒の座標の初期値は縦方向に `4` 、横方向に `4` である。
- 敵を三つ配置する。一つ目の敵の座標の初期値は縦方向に `0` 、横方向に `0` である。二つ目の敵の座標の初期値は縦方向に `1` 、横方向に `1` である。三つ目の敵の座標の初期値は縦方向に `2` 、横方向に `2` である。また、敵は座標以外に倒されているかの判定結果を真偽値で持つ。 `Ture` であれば倒されている状態、 `False` であれば倒されていない状態を表す。初期値は三つとも倒されていないため、 `False` である。
- ゲーム開始時のフィールドの状態は以下のように表示される。

```
*....
.*...
..*..
...S.
....S
```

前章までのプログラムでは、プレイヤーの駒と敵がそれぞれ一つずつであったため、それぞれの座標を持つ変数を用意すればよかった。今回は、駒が二つ、敵は三つ必要なため、以下のように各座標を定義する方法が考えられる。

```
player_x1, player_y1 = 3, 3
player_x2, player_y2 = 4, 4
enemy_x1, enemy_y1 = 0, 0
enemy_x2, enemy_y2 = 1, 1
enemy_x3, enemy_y3 = 2, 2
```

ただ、上記のような方法で定義すると、フィールド状態の表示の処理で、各変数をそれぞれ確認するような形で一つずつ処理を記述しなければならなくなる。また、駒や敵が増えるたびに、フィールド状態の表示の処理も新しく追加した駒や敵の座標についての処理を追加しなければならず、修正箇所が増えて、修正漏れなどが発生しやすくなり、不具合の原因になる。

以上の理由から、各座標をまとめて処理するため、以下のようにリストで座標を持つと良い。

```
player_x = [3, 4]
player_y = [3, 4]
enemy_x = [0, 1, 2]
enemy_y = [0, 1, 2]
```

リストで駒ごと、敵ごとに縦方向の座標のリスト、横方向の座標のリストを持つことで、`for`文でまとめて処理を行うことができるため、駒や敵が増えたときの修正が変数一つ一つで座標を持つ場合と比べて、少なくて済む。

上記のリストでは、同じインデックスの要素で、駒や敵の縦方向の座標と横方向の座標を表現している。たとえば、一つ目の駒の座標は、`player_x[0]`（`3`）と`player_y[0]`（`3`）で表現されている。二つ目の敵の座標は、`enemy_x[1]`（`1`）と`enemy_y[1]`（`1`）で表現されている。

同様に、敵が倒されているかの判定結果（フラグ）もリストで持つ。

```
enemy_defeated = [False, False, False]
```

上記のリストでは、一つ目の敵の倒されているかどうかの判定を`enemy_defeated[0]`に持ち、二つ目の敵の倒されているかどうかの判定を`enemy_defeated[1]`に持ち、三つ目の敵の倒されているかどうかの判定を`enemy_defeated[2]`に持っている。もし、三つ目の敵が倒された場合には、`enemy_defeated[2]`の値を`True`に変更する。

駒と敵の座標をリストで持ったことで、以下のようにリストを使った形でフィールドの表示処理を変更する必要がある。

```
# 駒と敵の座標からフィールドの表示するセルの文字を返す関数
def get_cell(x, y):
    # 敵が配置されている座標を"*"で表す
    for i in range(len(enemy_x)):
        if x == enemy_x[i] and y == enemy_y[i] and not enemy_defeated[i]:
            return "*"
    # プレイヤーの駒が配置されている座標を"S"で表す
    for i in range(len(player_x)):
        if x == player_x[i] and y == player_y[i]:
            return "S"
     # それ以外の位置は"."で表す
     return "."

```

```
14
15 # フィールドを表示する関数
16 def show_field():
17     # リストを作成する
18     field = []
19     # 縦方向の座標（y座標）を表す変数を0から4までループ
20     for y in range(5):
21         # 横方向の座標（x座標）を表す変数を0から4までループ
22         for x in range(5):
23             cell = get_cell(x, y)
24             field.append(cell)
25         # x座標を表す変数が4まで到達したら改行する
26         field.append("\n")
27     # 作成したリストを文字列に変換して表示する
28     print("".join(field))
```

上記の関数は、前章で作成した `show_field()` 関数を修正したものである。

前章で作成した `show_field()` 関数では、内側の `for` 文の処理で、駒や敵の座標から、 `*` か `S` か `.` のどの文字を追加すべきか判定して、 `field` に追加していたが、座標から `*` と `S` と `.` のどれを追加すべきか判定する処理を `get_cell()` 関数に分けている。

`get_cell()` 関数では、縦方向と横方向の座標を受け取り、駒と敵が受け取った座標に存在するかどうか確認し、駒が存在すれば `S` を、倒されていない敵が存在すれば `*` を、どちらも存在しなければ `.` を返す。

続いて、以下の仕様を満たすプログラムを考えてみよう。

- 駒の移動する方向を入力して、プレイヤーの駒を交互に動かす。例えば、初めに `up` と入力すると、一つ目の駒が1マス上に移動し、続けて `up` を入力すると、二つ目の駒が1マス上に移動する。三回目に `up` と入力すると、一つ目の駒が1マス上に移動する。
- 無効な方向を入力されたら、 `ValueError` の例外を発生させて、 `無効な方向です` と表示し、強制終了する。

駒を交互に動かすために、 `player_x` と `player_y` のどちらを動かすターンであるか、と考えることができる。また、駒を動かすことで値が変わるのは、 `player_x` か `player_y` のリストの要素であるため、ターンは駒のリストが持つインデックスである `0` か `1` で表現できればよい。

ターンを表現するには、まず無限ループの外側で以下のように `turn` を宣言する。

```
1 turn = 0
```

初めは、一つ目の駒を動かしたいので、 `0` を代入する。

また、駒が移動したらターンが変わるので、ターンの移り変わりは無限ループの中で以下のように処理を記述することで実装できる。

```
1 turn = (turn + 1) % 2
```

上記のように記述することで、 `turn` に `0` と `1` を交互に代入できる。

例外処理に関する仕様について、無限ループ内にあった駒の移動に関する処理を `move_player()` 関数としてまとめて、仕様を満たすような修正を行ったものが以下である。

```
1 # 駒を移動する関数
2 def move_player(direction):
3     if direction == "up":
4         player_y[turn] -= 1
5     elif direction == "down":
6         player_y[turn] += 1
7     elif direction == "left":
8         player_x[turn] -= 1
9     elif direction == "right":
```

```
        player_x[turn] += 1
    else:
        raise ValueError("無効な方向です")
```

`move_player()` 関数では、引数に駒を進めたい方向を表す文字列を受け取り、文字列の内容に応じて、 `player_x` か `player_y` のリストの要素の値を変更している。

元となるプログラムの無効な方向を入力した際の処理を、

```
print("無効な方向です")
continue
```

から、

```
raise ValueError("無効な方向です")
```

に変更し、例外を発生させている。

続いて、以下の仕様を満たすプログラムを考えてみよう。

- プレイヤーの駒が、敵と同じ座標に移動したら、敵を倒したとみなして、以降、倒された敵はフィールドの上に表示せず、倒された敵がいた場所にプレイヤーの駒がいない場合は `.` を表示する。

上記の仕様を満たすためには、プレイヤーの駒を移動したのちに、移動後の座標に倒されていない敵が存在するか確認し、存在すれば、その敵の `enemy_defeated` の値を `True` に変更する処理を必要がある。

以下のように記述することで上記の処理を実装する。

```
for i in range(len(enemy_x)):
    # 駒が倒されていない敵と同一座標に到達したら、その敵を倒したことにする
    if (
        player_x[turn] == enemy_x[i]
        and player_y[turn] == enemy_y[i]
        and not enemy_defeated[i]
    ):
        enemy_defeated[i] = True
```

`for` 文で、 `enemy_x` と `enemy_y` と `enemy_defeated` の要素全てを参照し、直前に移動した駒の横方向の座標と縦方向の座標が一致して、かつ `enemy_defeated` が `False` であれば、敵を倒したと判定し、 `enemy_defeated` の値を `True` に変更している。

仕様に記述されている `倒された敵がいた場所にプレイヤーの駒がいない場合は . を表示する。` の箇所については、 `get_cell()` 関数内の敵が配置されているか判定する条件式に、 `x == enemy_x[i] and y == enemy_y[i] and not enemy_defeated[i]` というように、 `not enemy_defeated[i]` と倒されていないかの判定結果を含めることで実現している。

最後に、以下の仕様を満たすプログラムを考えてみよう。

- 全ての敵を倒したらゲームクリアとなり、 `ゲームクリア！` と出力し、ゲームを終了する。

上記の仕様は、 `enemy_defeated` の要素が全て `True` であれば、敵を全て倒したと判断できるため、無限ループの中に以下のように記述することで実現できる。

```
# 全部の敵を倒したらクリア
if all(enemy_defeated):
    print("ゲームクリア！")
    break
```

`all()` 関数は、引数に指定された全ての要素が `True` であれば `True` を返す。そのため、 `enemy_defeated` の要素が全て `True` であれば、 `if` 文の条件式が真となり、 `if` 文の中の処理を実行する。

`if` 文の中の処理では、 `print()` 関数で `ゲームクリア！` と出力したあと、 `break` を実行することで、無限ループを抜ける。

以上の処理を全てまとめたプログラムが以下である。

```
# 二つの駒と三つ敵の座標をそれぞれ表すリスト
player_x = [3, 4]
player_y = [3, 4]
enemy_x = [0, 1, 2]
enemy_y = [0, 1, 2]
# 敵が倒されたかどうかを表すフラグ
enemy_defeated = [False, False, False]
# 動かす駒を表す変数
turn = 0


# 駒と敵の座標からフィールドの表示するマスの文字を返す関数
def get_cell(x, y):
    # 敵が配置されている座標を"*"で表す
    for i in range(len(enemy_x)):
        if x == enemy_x[i] and y == enemy_y[i] and not enemy_defeated[i]:
            return "*"
    # プレイヤーの駒が配置されている座標を"S"で表す
    for i in range(len(player_x)):
        if x == player_x[i] and y == player_y[i]:
            return "S"
    # それ以外の位置は"."で表す
    return "."


# フィールドを表示する関数
def show_field():
    # リストを作成する
    field = []
    # 縦方向の座標（y座標）を表す変数を0から4までループ
    for y in range(5):
        # 横方向の座標（x座標）を表す変数を0から4までループ
        for x in range(5):
            cell = get_cell(x, y)
            field.append(cell)
        # x座標を表す変数が4まで到達したら改行する
        field.append("\n")
    # 作成したリストを文字列に変換して表示する
    print("".join(field))

# 駒を移動する関数
def move_player(direction):
    if direction == "up":
        player_y[turn] -= 1
    elif direction == "down":
        player_y[turn] += 1
    elif direction == "left":
        player_x[turn] -= 1
    elif direction == "right":
        player_x[turn] += 1
    else:
        raise ValueError("無効な方向です")


# フィールドの状態を表示
```

```
show_field()

while True:
    # 駒を移動する方向を入力する
    direction = input()
    # 入力された文字列が"end"の場合は、無限ループを終了する
    if direction == "end":
        print("ゲームを強制終了")
        break
    # 入力された方向に駒を移動する
    move_player(direction)

    for i in range(len(enemy_x)):
        # 駒が倒されていない敵と同一座標に到達したら、その敵を倒したことにする
        if (
            player_x[turn] == enemy_x[i]
            and player_y[turn] == enemy_y[i]
            and not enemy_defeated[i]
        ):
            enemy_defeated[i] = True

    # フィールドの状態を表示
    show_field()

    # 全部の敵を倒したらクリア
    if all(enemy_defeated):
        print("ゲームクリア！")
        break

    # 駒を交互に移動させる
    turn = (turn + 1) % 2
```

```
*....
.*...
..*..
...S.
....S

up
*....
.*...
..*S.
.....
....S

up
*....
.*...
..*S.
....S
.....

left
*....
.*...
..S..
....S
.....

left
*....
.*...
..S..
...S.
.....

up
*....
.*S..
.....
```

```
...S.
.....

up
*....
.*S..
...S.
.....
.....

left
*....
.S...
...S.
.....
.....

left
*....
.S...
..S..
.....
.....

up
*S...
.....
..S..
.....
.....

up
*S...
..S..
.....
.....
.....

left
S....
..S..
.....
.....
.....

ゲームクリア！
```

上記のプログラムでは、まず、 `player_x` と `player_y` に駒の座標を、 `enemy_x` と `enemy_y` に敵の座標を宣言している。 `player_x[0]` と `player_x[0]` には一つ目の駒の座標を、 `player_x[1]` と `player_x[1]` には二つ目の駒の座標を宣言している。 `enemy_x[0]` と `enemy_y[0]` には一つ目の敵の座標を、 `enemy_x[1]` と `enemy_y[1]` には二つ目の敵の座標を、 `enemy_x[2]` と `enemy_y[2]` には三つ目の駒の座標を宣言している。

`enemy_defeated` には三つの敵が倒されているかどうかのフラグを宣言している。

`turn` で、動かす駒を指定している。初めは一つ目の駒を動かすため、 `0` を代入している。

続いて、 `get_cell()` 関数を定義している。この関数は、指定された横方向の座標と縦方向の座標におけるマスの内容を取得するものである。引数として横方向と縦方向それぞれの座標を受け取り、指定された位置にプレイヤーの駒がいる場合は `S` を、倒されていない敵がいる場合は `*` を、それ以外の場合は `.` を返す。

続いて、 `show_field()` 関数を定義している。この関数は、ゲームのフィールドを表示するものである。フィールドは、縦5マス横5マスで表される。各マスを `get_cell()` 関数を使用して取得し、文字列として連結して表示する。

続いて、 `move_player()` 関数を定義している。この関数は、プレイヤーの駒を指定された方向に移動するものである。引数として、上下左右（ `up` 、 `down` 、 `left` 、 `right` ）のいずれかの方向を受け取る。指定された方向に応じて、駒の座標を更新する。また、引数が `up` 、 `down` 、 `left` 、 `right` のいずれでもない場合、例外として `ValueError` を発生させる。

ここまでが使用する変数や関数の定義である。

メインの処理では、最初に、`show_field()`関数を呼び出して、ゲーム開始時のフィールドの状態を表示する。その後、無限ループが開始する。

無限ループ内では、まず、プレイヤーの移動方向を入力する。`end`と入力された場合、ゲームを強制終了する。`up`、`down`、`left`、`right`が入力された場合、`move_player()`関数を使用して駒を移動する。

駒を移動した後、駒が倒されていない敵の位置にいるかどうかを確認する。駒が倒されていない敵の位置にいる場合、駒と座標が一致している敵は倒され、`enemy_defeated`で倒された敵に該当する要素を`True`にする。その後、`show_field()`関数を使用してフィールドを再度表示する。

最後に、すべての敵が倒されているかどうかを確認する。すべての敵が倒されている場合、ゲームクリアとなり、無限ループを抜ける。そうでない場合、`turn`を切り替える。`turn`は`0`であれば`1`に、`1`であれば`0`に切り替わる。

前節の実装の問題点

前節までで、複数の駒や敵を配置して動かすことができるゲームが完成した。

個人開発で、これ以上このゲームに対して追加の仕様を付け加えることがないのであれば、完成としてもよい。ただ、一般的なアプリケーション開発は、チーム開発であり、かつ、一度プログラムが完成したら終わりではなく、継続して不具合の修正や、追加仕様の実装などを行っていく。

継続的にゲーム改修していく観点と、チーム開発をする観点で、前節の実装にはそれぞれ問題点がある。

継続的にゲーム改修していくという観点では以下のような問題がある。

- 新しく駒や敵を追加する場合に該当するリスト全てを更新しなければならず、そのため、修正漏れのリスクがある。
- 駒や敵の情報がプログラム上の様々なところ（リストや`move_player`関数、`get_cell()`関数の内部など）にあるため、初期配置やフィールド上での表示、移動方法などの駒や敵に関して変更を加えたい場合、該当する部分をプログラム全体の中から見つけて、変更しなければならない。
- 他の箇所にも言えることだが、特に`get_cell()`関数は、フィールドに対する操作と駒や敵に関する情報が入り組んでいるため、`get_cell()`関数に行った修正が他にも影響しないか十分に検討・検証する必要がある。

また、チーム開発をするという観点では以下のような問題がある。

- 駒や敵の座標、状態、フィールド上での表示（`S`や`*`）が複数のリストや表示の処理の中に散らばっているため、駒に関連する情報と敵に関連する情報を、プログラムから把握するのが難しい。
- フィールドに対する操作や駒に対する操作が散らばっているため、処理を一つずつ追っていかないと、プログラムの全体像を理解するのが難しい。特に`get_cell()`関数は、フィールドに対する操作に駒や敵の表示情報（`S`や`*`）が内包されているため、プログラム内のコメントや事前のゲームの仕様を知らなければ、`S`と`*`がそれぞれプレイヤーの駒と敵を表していることが分かりづらい。

以上の問題点を整理すると、以下にまとめることができる。

- プレイヤーの駒、敵、フィールド、それぞれの概念に関する情報や操作がプログラム全体に散らばっており、また、各概念に関する操作や情報が一つの関数にまとまっている箇所もあるため、改修を行うにはプログラム全体を理解して、改修に必要な箇所を特定し、プログラムを直す必要がある。また、初めて見る人にとってプログラムだけでは、どのようなゲームか、またどのような実装でゲームが実現されているか、わかりづらい。

前節の実装の問題点は、クラスを用いて、オブジェクト指向プログラミングに基づいて実装を行うことで解決できる。

オブジェクト指向プログラミングに基づいた実装のメリット

オブジェクト指向プログラミングに基づいた実装のメリットは、一般的に以下のようなものが挙げられる。

- クラスに同様の処理をまとめることができるため、駒や敵など必要な概念が増えるたびにクラスを定義しインスタンスを生成することで、概念に関する処理を実行できる。

- 概念ごと（今回の場合は、プレイヤーの駒や敵など）にクラスを作成しデータ属性やメソッドをまとめることで、プログラムが読みやすくなり、理解しやすくなる。
- 概念ごとにデータ属性やメソッドがまとまっているため、変更箇所を特定しやすく、変更が行いやすい。

前節で挙げた問題点は、概念ごとに情報や操作がプログラム上に散らばっているため、理解もしづらく、修正しづらいということであったため、上記に挙げるメリットから、オブジェクト指向プログラミングに基づいた実装によって問題を解決できることがわかる。

オブジェクト指向プログラミングに基づいた実装

ここまでに開発したゲームをオブジェクト指向プログラミングに基づいた実装に直してみよう。

まず、必要なクラスを考える。

このゲームでは、登場してきた概念として、プレイヤーの駒、敵、フィールドの三つがある。

プレイヤーの駒、敵、フィールドの三つの概念をクラスとして以下のような形で定義できる。無限ループでは、ユーザーの入力と、`end`と入力したら無限ループを終了する処理のみ記述している。

```
# プレイヤーの駒のクラス
class Player:
    pass


# 敵のクラス
class Enemy:
    pass


# フィールドのクラス
class Field:
    pass


while True:
    direction = input()
    if direction == "end":
        print("ゲームを強制終了")
        break
```

```
end
ゲームを強制終了
```

上記のプログラムでは、プレイヤーの駒のクラスとして`Player`を、敵のクラスとして`Enemy`を、フィールドのクラスとして`Field`を定義している。

まだ、クラスの中に何も処理を記述していないうえ、インスタンスを生成していないので、上記のプログラムでは、`end`と入力したら、`ゲームを強制終了`を出力し、プログラムが終了するのみのプログラムである。

これまでの実装の中で以下の処理から、プレイヤーの駒は初期値として縦方向の座標と横方向の座標を、敵は初期値として縦方向の座標と横方向の座標と倒されたかどうかのフラグを持つことが分かる。

```
# 二つの駒と三つ敵の座標をそれぞれ表すリスト
player_x = [3, 4]
player_y = [3, 4]
enemy_x = [0, 1, 2]
enemy_y = [0, 1, 2]
# 敵が倒されたかどうかを表すフラグ
enemy_defeated = [False, False, False]
```

オブジェクト指向プログラミングに基づいた実装では、駒や敵は作るのにインスタンスを生成して、使用する。生成したインスタンスに対して初期値を与えるには、インスタンスを生成したタイミングで処理が実行される `__init__()` メソッドをクラスに定義し、その中で、初期値となるデータ属性を定義すればよい。

また、敵の倒されたかどうかのフラグは、敵が作られたタイミングでは常に `False` であるが、駒と敵の座標は、常に一定ではなく、インスタンス生成時に指定する必要がある。そのため、 `Player` クラスと `Enemy` クラスで定義する `__init__()` は、縦方向の座標と横方向の座標を受け取る必要がある。

ここまでの説明を実装したプログラムが以下である。以下のプログラムでは、プレイヤーの駒と敵のインスタンスが正しく作られているか無限ループに入る直前に確認する処理を記述している。

```
# プレイヤーの駒のクラス
class Player:
    def __init__(self, x, y):
        # 初期配置の座標
        self.x = x
        self.y = y


# 敵のクラス
class Enemy:
    def __init__(self, x, y):
        # 初期配置の座標
        self.x = x
        self.y = y
        # 倒されたかどうかのフラグ
        self.defeated = False


# フィールドのクラス
class Field:
    pass

# 正しくインスタンスを生成できているか確認するための一時的な処理
player = Player(3, 3)
enemy = Enemy(0, 0)
print(player.x)
print(player.y)
print(enemy.x)
print(enemy.y)
print(enemy.defeated)


while True:
    direction = input()
    if direction == "end":
        print("ゲームを強制終了")
        break
```

```
3
3
0
0
False
end
ゲームを強制終了
```

上記のプログラムでは、 `Player` クラスと `Enemy` クラスのそれぞれに、 `__init__()` メソッドを定義している。 `__init__()` メソッドには、横方向の座標と縦方向の座標を受け取れるように引数を定義しているため、 `Player` クラスと `Enemy` クラスのインスタンスを生成する際には、引数を指定する。

上記のプログラムでは、横方向の座標と縦方向の座標が `3, 3` の位置にプレイヤーの駒を、 `0, 0` の位置に敵を配置するようにインスタンスを生成している。インスタンスを生成後、それぞれのデータ属性が正しく定義されているか `print()` 関数を使って確認している。出力結果

から、正しくインスタンスを生成されていることがわかる。

続いて、オブジェクト指向プログラミングに基づかない実装の中での `move_player()` 関数について、オブジェクト指向プログラミングにおいては、どのように扱うべきか考えてみよう。

`move_player()` 関数は、呼び出されるとプレイヤーの駒の座標を更新する関数である。各概念ごとにクラスに分けたことにより、プレイヤーの駒の座標は `Player` クラスのインスタンスが持つようになる。オブジェクト指向プログラミングでは、インスタンスのデータ属性を操作するようなメソッドはできるだけ、操作するデータ属性を持つインスタンスに持たせた方がよいため、 `move_player()` 関数の処理は、 `Player` クラスのメソッドとするのがよいと考えられる。また、 `move_player()` 関数はプレイヤーの駒の操作に関する処理なので、関連した処理を同一クラスにまとめるといった観点でも、 `Player` クラスに `move_player()` 関数の処理をメソッドとして持たせるのがよいと考えられる。

以下のプログラムは、 `Player` クラスに `move_player()` 関数の処理をメソッドとして持たせたものである。 `Player` クラスのインスタンスに関するメソッドであることが自明であるため、メソッド名は `move` としている。また、無限ループの直前に、 `move()` メソッドが正しく記述できているか確認する処理を記述している。

```python
# プレイヤーの駒のクラス
class Player:
    def __init__(self, x, y):
        # 初期配置の座標
        self.x = x
        self.y = y

    # プレイヤーの駒の移動を行うメソッド
    def move(self, direction):
        if direction == "up":
            self.y -= 1
        elif direction == "down":
            self.y += 1
        elif direction == "left":
            self.x -= 1
        elif direction == "right":
            self.x += 1
        else:
            raise ValueError("無効な方向です")


# 敵のクラス
class Enemy:
    def __init__(self, x, y):
        # 初期配置の座標
        self.x = x
        self.y = y
        # 倒されたかどうかのフラグ
        self.defeated = False


# フィールドのクラス
class Field:
    pass

# moveメソッドを正しく記述できているか確認するための一時的な処理
player = Player(3, 3)
print(player.x)
print(player.y)
player.move("up")
player.move("right")
print(player.x)
print(player.y)



while True:
    direction = input()
    if direction == "end":
        print("ゲームを強制終了")
        break
```

```
3
3
4
2
end
ゲームを強制終了
```

上記のプログラムでは、`Player` クラスに、`move()` メソッドを定義している。`move()` メソッドには、移動する方向を受け取る引数を定義しているため、`move()` メソッドを呼び出す際には、引数を一つ指定する。

上記のプログラムでは、横方向の座標と縦方向の座標が `3, 3` の位置にプレイヤーの駒を生成して、`move()` メソッドを2回呼び出している。出力結果から、駒の座標が指定した方向に合わせて意図通りに変更されていることがわかるため、正しく `move()` メソッドを記述できていることがわかる。

続いては、プレイヤーの駒と敵のフィールド上での表示について、オブジェクト指向プログラミングではどのように実装するのがよいか考えてみよう。

オブジェクト指向プログラミングに基づかないの実装の中では、プレイヤーの駒と敵のフィールド上での表示（ `S` と `*` ）は、 `get_cell()` 関数に記述されていた。しかし、プレイヤーの駒と敵それぞれフィールド上ではどのような表示になるかということは、プレイヤーの駒と敵ごとの情報であるため、それぞれ `Player` クラスと `Enemy` クラスそれぞれに持つのが良いと考えられる。

表示する文字をデータ属性として定義する方法もあるが、プレイヤーの駒や敵のフィールド上での表示は、各インスタンスの文字表現と考えることができるため、 `__str__()` メソッドという特殊メソッドを使う方法を今回紹介する。

`__str__()` メソッドは、インスタンスを `print()` 関数や `str()` 関数の引数に指定した際に呼び出されるメソッドである。

以下は、 `__str__()` メソッドを使ったサンプルプログラムである。

`Sample` クラスのインスタンスを `samsple` に代入して、 `print()` 関数と `str()` 関数の引数に指定している。 `print()` 関数の引数に指定した際には、 `__str__()` メソッドが呼び出され、 `sample message` という文字列を返すため、 `print(samsple)` の実行結果として、 `sample message` を出力する。

`str()` 関数の引数に指定した際にも、 `__str__()` メソッドが呼び出され、 `sample message` という文字列を返すため、 `text = str(sample)` の実行結果として、 `sample message` という文字列を、変数 `text` に代入している。

```
class Sample:
    def __str__(self):
        return "sample message"


sample  = Sample()
print(sample)
text = str(sample)
print(text)
```

```
sample message
sample message
```

`__str__()` メソッドを使用して、プレイヤーの駒と敵のフィールド上での表示を呼び出す処理を追加したプログラムが以下である。プレイヤーの駒は `S` 、敵は `*` としてフィールド上で表示されるため、 `Player` クラスには `S` を返す `__str__()` メソッドを、 `Enemy` クラスには `*` を返す `__str__()` メソッドを追加した。

```
# プレイヤーの駒のクラス
class Player:
    def __init__(self, x, y):
        # 初期配置の座標
        self.x = x
        self.y = y

    # フィールド上での表示
    def __str__(self):
        return "S"


    # プレイヤーの駒の移動を行うメソッド
    def move(self, direction):
        if direction == "up":
            self.y -= 1
        elif direction == "down":
            self.y += 1
        elif direction == "left":
            self.x -= 1
        elif direction == "right":
            self.x += 1
        else:
```

```
            raise ValueError("無効な方向です")


# 敵のクラス
class Enemy:
    def __init__(self, x, y):
        # 初期配置の座標
        self.x = x
        self.y = y
        # 倒されたかどうかのフラグ
        self.defeated = False

    # フィールド上での表示
    def __str__(self):
        return "*"


# フィールドのクラス
class Field:
    pass

# __str__メソッドを正しく記述できているか確認するための一時的な処理
player = Player(3, 3)
enemy = Enemy(0, 0)
print(player)
print(enemy)


while True:
    direction = input()
    if direction == "end":
        print("ゲームを強制終了")
        break
```

```
S
*
end
ゲームを強制終了
```

ゲームの仕様では、敵は倒されていなければフィールド上で `*` と表示されるが、倒されていると `.` と表示される。

`Enemy` クラスの `__str__()` メソッドでは、データ属性 `defeated` の値によって、 `*` を返すか、 `.` を返すかを変える必要がある。`defeated` の値が真のときに `.` を返し、偽のときに `*` を返す処理を追加したプログラムが以下である。

```
# プレイヤーの駒のクラス
class Player:
    def __init__(self, x, y):
        # 初期配置の座標
        self.x = x
        self.y = y

    # フィールド上での表示
    def __str__(self):
        return "S"


    # プレイヤーの駒の移動を行うメソッド
    def move(self, direction):
        if direction == "up":
            self.y -= 1
        elif direction == "down":
            self.y += 1
        elif direction == "left":
            self.x -= 1
        elif direction == "right":
            self.x += 1
        else:
```

```
            raise ValueError("無効な方向です")


# 敵のクラス
class Enemy:
    def __init__(self, x, y):
        # 初期配置の座標
        self.x = x
        self.y = y
        # 倒されたかどうかのフラグ
        self.defeated = False

    # フィールド上での表示
    def __str__(self):
        message = "*"
        if self.defeated:
            message = "."
        return message


# フィールドのクラス
class Field:
    pass

# __str__メソッドを正しく記述できているか確認するための一時的な処理
player = Player(3, 3)
enemy = Enemy(0, 0)
print(player)
print(enemy)
enemy.defeated = True
print(enemy)

while True:
    direction = input()
    if direction == "end":
        print("ゲームを強制終了")
        break
```

```
S
*
.
end
ゲームを強制終了
```

`Enemy` クラスの `__str__()` メソッドの処理は三項演算子を使った処理に書き換えることができる。

Pythonでは、三項演算子は以下のように記述できる。

```
<条件式が真のときに評価する式> if <条件式> else <条件式が偽のときに評価する式>
```

そのため、 `Enemy` クラスの、

```
def __str__(self):
    message = "*"
    if self.defeated:
        message = "."
    return message
```

の処理を三項演算子を使うことで、以下のように書き換えることができる。

```
def __str__(self):
```

```
2       return "*" if not self.defeated else "."
```

上記の処理では、条件式が `not self.defeated` であるため、 `self.defeated` が偽であれば条件式は真となり `*` を返し、真であれば条件式は偽となり `.` を返す。

今後のプログラムでは、三項演算子を使った処理で記述していく。

続いて、 `show_field()` 関数と `get_cell()` 関数で行っている処理をメソッドとして持つような `Field` クラスを定義していく。

`Field` クラスはゲームのフィールドに関する情報や処理を持つクラスとして定義する。まず、 `Field` クラスが持つデータ属性について考える。

フィールドが持つ情報として、まず、フィールドの大きさが考えられる。また、フィールド上に、プレイヤーの駒や敵を配置するため、プレイヤーの駒や敵もフィールドが持っていると考えられる。ただし、フィールドのどの位置に駒や敵がいるかは、駒と敵は自身の座標を持っているため、フィールド自体がそれぞれの駒や敵の座標を持つ必要はない。

以下は、 `Field` クラスに `__init__()` メソッドを追加したプログラムである。 `Field` クラスのインスタンス生成時に、プレイヤーの駒として `Player` クラスのインスタンスを、敵として `Enemy` クラスのインスタンスを引数として指定して、それぞれ `players` と `enemies` というデータ属性に格納している。また、大きさは縦5マス横5マスであるため、 `size` というデータ属性に `5` を格納している。

無限ループの外側で、ゲームに必要なインスタンスの作成を行う処理も追加している。ゲームに必要な `Player` クラスと `Enemy` クラスのインスタンスをリストとして定義し、 `Field` クラスのインスタンスを作成するときにそれぞれ引数として指定している。

```
1  # プレイヤーの駒のクラス
2  class Player:
3      def __init__(self, x, y):
4          # 初期配置の座標
5          self.x = x
6          self.y = y
7
8      # フィールド上での表示
9      def __str__(self):
10         return "S"
11
12
13     # プレイヤーの駒の移動を行うメソッド
14     def move(self, direction):
15         if direction == "up":
16             self.y -= 1
17         elif direction == "down":
18             self.y += 1
19         elif direction == "left":
20             self.x -= 1
21         elif direction == "right":
22             self.x += 1
23         else:
24             raise ValueError("無効な方向です")
25
26
27 # 敵のクラス
28 class Enemy:
29     def __init__(self, x, y):
30         # 初期配置の座標
31         self.x = x
32         self.y = y
33         # 倒されたかどうかのフラグ
34         self.defeated = False
35
36     # フィールド上での表示
37     def __str__(self):
38         return "*" if not self.defeated else "."
39
40
41 # フィールドのクラス
42 class Field:
```

```
    def __init__(self, players, enemies):
        # フィールド上に配置するプレイヤーの駒と敵
        self.players = players
        self.enemies = enemies
        # フィールドの大きさ
        self.size = 5


# ゲームに必要なインスタンスの作成
players = [Player(3, 3), Player(4, 4)]
enemies = [Enemy(0, 0), Enemy(1, 1), Enemy(2, 2)]
field = Field(enemies, players)

while True:
    direction = input()
    if direction == "end":
        print("ゲームを強制終了")
        break
```

```
end
ゲームを強制終了
```

続いて、 `show_field()` 関数と `get_cell()` 関数での処理を、オブジェクト指向プログラミングに基づいた実装をする場合について考える。

`show_field()` 関数と `get_cell()` 関数での処理は、フィールドの状態に関連した処理であるため、 `Field` クラスのメソッドとすることが適切と考えられる。

以下は、 `Field` クラスに `show_field()` 関数での処理を担う `show()` メソッドと `get_cell()` 関数での処理を担う `get_cell()` メソッドを追加したプログラムである。 `show()` メソッドは、 `Field` クラスのメソッドであることが自明であるため、メソッド名から `_field` という文字列を取り除いている。

`show()` メソッドでは、フィールドの状態を出力する処理を行っている。 `show_field()` 関数で `range(5)` となっていた部分を `range(self.size)` とすることで、データ属性 `size` に格納されている値のマスだけフィールドを定義できる。各座標にどのような文字を出力すべきかは、 `show_field()` 関数と同様、 `get_cell()` メソッドを呼び出して確認している。

`get_cell()` メソッドでは、引数に指定された座標にプレイヤーの駒がいるのか、倒されていない敵がいるのか、何もいないかで異なる文字を返す処理を行っている。倒されていない敵がいないか、また、プレイヤーの駒がいないかの判断には、 `Field` クラスのインスタンスが持つ、 `Player` クラスのインスタンスのデータ属性 `x` と `y` や、 `Enemy` クラスのインスタンスのデータ属性 `x` と `y` を使っている。また、 `show()` メソッドに返す文字を、 `str(player)` や `str(enemy)` として、 `Player` クラスや `Enemy` クラスが持つ `__str__()` メソッドを実行することで、 `Field` クラス自体はプレイヤーの駒や敵のフィールド上での文字表現を持たせなくすることができる。

`Field` クラスのインスタンス生成後に、 `show()` メソッドを呼び出すことで、フィールドの初期状態を出力する処理も追加している。

```
# プレイヤーの駒のクラス
class Player:
    def __init__(self, x, y):
        # 初期配置の座標
        self.x = x
        self.y = y

    # フィールド上での表示
    def __str__(self):
        return "S"


    # プレイヤーの駒の移動を行うメソッド
    def move(self, direction):
        if direction == "up":
            self.y -= 1
        elif direction == "down":
            self.y += 1
```

```
        elif direction == "left":
            self.x -= 1
        elif direction == "right":
            self.x += 1
        else:
            raise ValueError("無効な方向です")


# 敵のクラス
class Enemy:
    def __init__(self, x, y):
        # 初期配置の座標
        self.x = x
        self.y = y
        # 倒されたかどうかのフラグ
        self.defeated = False

    # フィールド上での表示
    def __str__(self):
        return "*" if not self.defeated else "."


# フィールドのクラス
class Field:
    def __init__(self, players, enemies):
        # フィールド上に配置するプレイヤーの駒と敵
        self.players = players
        self.enemies = enemies
        # フィールドの大きさ
        self.size = 5

    # 指定した座標のマスの文字を返すメソッド
    def get_cell(self, x, y):
        for player in self.players:
            if x == player.x and y == player.y:
                return str(player)
        for enemy in self.enemies:
            if x == enemy.x and y == enemy.y:
                return str(enemy)
        return "."

    # フィールドの状態を出力するメソッド
    def show(self):
        field = []
        for y in range(self.size):
            for x in range(self.size):
                cell = self.get_cell(x, y)
                field.append(cell)
            field.append("\n")
        print("".join(field))


# ゲームに必要なインスタンスの作成
players = [Player(3, 3), Player(4, 4)]
enemies = [Enemy(0, 0), Enemy(1, 1), Enemy(2, 2)]
field = Field(players, enemies)

# フィールドの初期状態の表示
field.show()

while True:
    direction = input()
    if direction == "end":
        print("ゲームを強制終了")
        break
```

```
*....
.*...
..*..
...S.
```

```
....S

end
ゲームを強制終了
```

クラスの定義については以上となる。

最後に、ゲームとして必要な処理を追加したプログラムが以下である。

動かす駒を表す変数として `turn` を定義して、 `0` を格納している。

無限ループ内で、ユーザーの入力と `end` と入力されたかどうかの判断を行った後に、 `Player` クラスのインスタンスの `move` メソッドを呼び出して、プレイヤーの駒を移動している。

```
field.players[turn].move(direction)
```

上記の処理では、 `Field` クラスのインスタンスが持っている `Player` クラスのインスタンスを指定して、 `move()` メソッドを呼び出している。

続く以下の処理で、移動した駒と敵の座標が一致していないか確認し、一致していれば、倒したと判定し、座標が一致していた敵のインスタンスのデータ属性 `defeated` を `True` にする。座標が一致するか確認する際には、駒も敵も `Field` クラスのインスタンスが持っている `Player` クラスのインスタンスと `Enemy` クラスのインスタンスを使用する。

```
for enemy in field.enemies:
    if field.players[turn].x == enemy.x and field.players[turn].y == enemy.y:
        enemy.defeated = True
```

敵を倒したか判定した後、以下のように `show()` メソッドを呼び出して、フィールドの最新の状態を表示する。

```
field.show()
```

その後、全ての敵を倒したか判定し、全ての敵を倒していれば無限ループを抜けてゲームを終了する処理を行う。まず、 `is_game_clear` というゲームクリアか判定する変数を定義し、 `True` を格納する。 `Field` クラスのインスタンスが持つ全ての `Enemy` クラスのインスタンスの `defeated` が `True` であれば、全ての敵を倒したと判定できる。逆に、一つでも `defeated` が `False` であれば、ゲームはクリアしたと判定できないため、 `Field` クラスのインスタンスが持つ全ての `Enemy` クラスのインスタンスの `defeated` を `for` 文を使って一つずつ確認し、一つでも `False` であれば、 `is_game_clear` に `False` を代入する。もし、全ての `Enemy` クラスのインスタンスの `defeated` が `True` であれば、 `is_game_clear` は `True` のまま `for` 文の処理を終えるため、その後の `if` 文で `ゲームクリア！` と出力し、無限ループを抜け、ゲームを終了する。

```
is_game_clear = True
for enemy in field.enemies:
    if not enemy.defeated:
        is_game_clear = False
        break
if is_game_clear:
    print("ゲームクリア！")
    break
```

最後に、ゲームが終了しなければ、次の駒を動かすため、以下の処理で `turn` の値を更新する。

```
turn = (turn + 1) % len(players)
```

```python
# プレイヤーの駒のクラス
class Player:
    def __init__(self, x, y):
        # 初期配置の座標
        self.x = x
        self.y = y

    # フィールド上での表示
    def __str__(self):
        return "S"

    # プレイヤーの駒の移動を行うメソッド
    def move(self, direction):
        if direction == "up":
            self.y -= 1
        elif direction == "down":
            self.y += 1
        elif direction == "left":
            self.x -= 1
        elif direction == "right":
            self.x += 1
        else:
            raise ValueError("無効な方向です")


# 敵のクラス
class Enemy:
    def __init__(self, x, y):
        # 初期配置の座標
        self.x = x
        self.y = y
        # 倒されたかどうかのフラグ
        self.defeated = False

    # フィールド上での表示
    def __str__(self):
        return "*" if not self.defeated else "."


# フィールドのクラス
class Field:
    def __init__(self, players, enemies):
        # フィールド上に配置するプレイヤーの駒と敵
        self.players = players
        self.enemies = enemies
        # フィールドの大きさ
        self.size = 5

    # 指定した座標のマスの文字を返すメソッド
    def get_cell(self, x, y):
        for player in self.players:
            if x == player.x and y == player.y:
                return str(player)
        for enemy in self.enemies:
            if x == enemy.x and y == enemy.y:
                return str(enemy)
        return "."

    # フィールドの状態を出力するメソッド
    def show(self):
        field = []
        for y in range(self.size):
            for x in range(self.size):
                cell = self.get_cell(x, y)
                field.append(cell)
            field.append("\n")
        print("".join(field))


# ゲームに必要なインスタンスの作成
players = [Player(3, 3), Player(4, 4)]
```

```
enemies = [Enemy(0, 0), Enemy(1, 1), Enemy(2, 2)]
field = Field(players, enemies)

# フィールドの初期状態の表示
field.show()

# 動かす駒を表す変数
turn = 0

while True:
    # 駒を移動する方向を入力する
    direction = input()
    # 入力された文字列が"end"の場合は、無限ループを終了する
    if direction == "end":
        print("ゲームを強制終了")
        break
    # 入力された方向に駒を移動する
    field.players[turn].move(direction)

    for enemy in field.enemies:
        # 駒が倒されていない敵と同一座標に到達したら、その敵を倒したことにする
        if field.players[turn].x == enemy.x and field.players[turn].y == enemy.y:
            enemy.defeated = True

    # フィールドの状態を表示
    field.show()

    # 全部の敵を倒したらクリア
    is_game_clear = True
    for enemy in field.enemies:
        if not enemy.defeated:
            is_game_clear = False
            break
    if is_game_clear:
        print("ゲームクリア！")
        break

    # 駒を交互に移動させる処理
    turn = (turn + 1) % len(players)
```

```
*....
.*...
..*..
...S.
....S

up
*....
.*...
..*S.
.....
....S

left
*....
.*...
..*S.
.....
...S.

left
*....
.*...
..S..
.....
...S.

up
*....
.*...
```

```
..S..
...S.
.....

up
*....
.*S..
.....
...S.
.....

up
*....
.*S..
...S.
.....
.....

left
*....
.S...
...S.
.....
.....

left
*....
.S...
..S..
.....
.....

up
*S...
.....
..S..
.....
.....

up
*S...
..S..
.....
.....
.....

left
S....
..S..
.....
.....
.....

ゲームクリア！
```

ここまでに開発したゲームをオブジェクト指向プログラミングに基づいた実装に修正する作業は以上である。

実際に、オブジェクト指向プログラミングに基づいた実装にすることによって、問題点が解決されたか一つずつ確認していこう。

オブジェクト指向プログラミングに基づかない実装では、以下のような問題点が挙げられていた。

1. プレイヤーの駒、敵、フィールドの各概念に関する情報と操作が、プログラム上に散らばっていたり、複数の概念が関連した関数があるため、プログラムを理解しづらい。
2. プレイヤーの駒、敵、フィールドの各概念に関する情報と操作が、プログラム上に散らばっていたり、複数の概念が関連した関数があるため、修正が行いづらい。

オブジェクト指向プログラミングに基づいた実装を行ったことで、実際に上記の問題点が解決されたかみてみよう。

一つ目の問題点については、プレイヤーの駒、敵、フィールドの各概念ごとにクラスを作成し、それぞれに関連したデータ属性と操作をまとめたことで、各概念ごとに持つ情報や操作が簡単にわかるようになった。

二つ目の問題点については、プレイヤーの駒、敵、フィールドの各概念ごとにクラスを作成し、それぞれに関連したデータ属性と操作をまとめたことで、変更箇所を特定し、修正を行いやすくなった。また、クラスにしたことで、プレイヤーの駒や敵を増やしたいときにはインスタンスを生成すれば、簡単に増やすことができるようになった。

具体的な挙げた問題点についてそれぞれ解決されたかみてみよう。

- 新しく駒や敵を追加する場合に該当するリスト全てを更新しなければならず、そのため、修正漏れのリスクがある。

新しく敵を一つ増やしたい場合、修正前は、以下のように `enemy_x` と `enemy_y` と `enemy_defeated` を宣言している箇所のそれぞれに修正を行う必要があった。

```
enemy_x = [0, 1, 2, 1]
enemy_y = [0, 1, 2, 0]
enemy_defeated = [False, False, False, False]
```

修正後は以下のように、 `enemies` を宣言している一か所に対して、以下のように `Enemy` クラスのインスタンスを追加するだけで簡単に増やすことができる。

```
enemies = [Enemy(0, 0), Enemy(1, 1), Enemy(2, 2), Enemy(1, 0)]
```

- 駒や敵の情報がプログラム上の様々なところ（リストや `move_player` 関数、 `get_cell()` 関数の内部など）にあるため、初期配置やフィールド上での表示、移動方法などの駒や敵に関して変更を加えたい場合、該当する部分をプログラム全体の中から見つけて、変更しなければならない。

プレイヤーの駒のフィールド上での表示を `S` から `P` に変更したい場合を考えてみよう。

修正前では、プログラム全体を見て、まず `S` を表示するような箇所を見つける必要があった。また、プレイヤーの駒の表示を指定する処理を行っている関数は、 `get_cell()` 関数ではあるため、駒のフィールド上での表示を `S` から `P` に変更するには `get_cell()` 関数を以下のように修正する必要があるが、 `get_cell()` 関数の呼び出し元をたどっていき、修正しても問題ないか確かめる必要がある。

```
def get_cell(x, y):
    for i in range(len(enemy_x)):
        if x == enemy_x[i] and y == enemy_y[i] and not enemy_defeated[i]:
            return "*"
    for i in range(len(player_x)):
        if x == player_x[i] and y == player_y[i]:
            return "P"
    return "."
```

修正後は、プレイヤーの駒を表す `Player` クラスに着目すると、 `__init__()` メソッドで定義されているデータ属性から、プレイヤーの駒は縦方向の座標と横方向の座標を持つことがわかる。また、 `__str__()` メソッドから、プレイヤーの駒のフィールド上での表示は `S` であることがわかる。また、 `move()` メソッドから、プレイヤーの駒は上下左右に1マスずつ動くことができることがわかる。

プレイヤーの駒のフィールドの表示を `S` から、 `P` に変更したい場合は、 `Player` クラスにある `__str__()` メソッドでフィールド上での表示を表しているとすぐにわかるため、以下のように修正することで駒の表示を変えることができる。

```
def __str__(self):
    return "P"
```

- 他の箇所にも言えることだが、特に `get_cell()` 関数は、フィールドに対する操作と駒や敵に関する情報が入り組んでいるため、 `get_cell()` 関数に行った修正が他にも影響しないか十分に検討・検証する必要がある。

- フィールドに対する操作や駒に対する操作が散らばっているため、処理を一つずつ追っていかないと、プログラムの全体像を理解するのが難しい。特に `get_cell()` 関数は、フィールドに対する操作に駒や敵の表示情報（ `S` や `*` ）が内包されているため、プログラム内のコメントや事前のゲームの仕様を知らなければ、 `S` と `*` がそれぞれプレイヤーの駒と敵を表していることが分かりづらい。
- 駒や敵の座標、状態、フィールド上での表示（ `S` や `*` ）が複数のリストや表示の処理の中に散らばっているため、駒に関連する情報と敵に関連する情報を、プログラムから把握するのが難しい。

上記の三つの問題が解決されたか、確認しよう。

まず、敵の情報について注目してみよう。敵は情報として、座標と倒されていないかのフラグ、フィールド上での表示（ `*` ）を持っている。

修正前は、プログラム全体を確認なければ、以下の箇所に敵に関する情報があるとわからない。

```
enemy_x = [0, 1, 2]
enemy_y = [0, 1, 2]
enemy_defeated = [False, False, False]

def get_cell(x, y):
    for i in range(len(enemy_x)):
        if x == enemy_x[i] and y == enemy_y[i] and not enemy_defeated[i]:
            return "*"
    for i in range(len(player_x)):
         if x == player_x[i] and y == player_y[i]:
             return "S"
     return "."
```

修正後であれば、 `Enemy` を確認するだけで敵に関する情報がわかる。

```
class Enemy:
    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.defeated = False

    def __str__(self):
        return "*" if not self.defeated else "."
```

続いて、フィールドの各マスに表示すべき文字を返す関数である `get_cell()` 関数がどのように変わったか実際に見てみよう。

修正前の `get_cell()` 関数は、フィールドに関する関数であるが、処理の中で、駒や敵が持つべき情報である、 `*` や `S` が直接記述されていたため、 `*` や `S` がどんな役割を果たすか分かりづらく、修正を行いづらかった。

```
def get_cell(x, y):
    for i in range(len(enemy_x)):
        if x == enemy_x[i] and y == enemy_y[i] and not enemy_defeated[i]:
            return "*"
    for i in range(len(player_x)):
        if x == player_x[i] and y == player_y[i]:
            return "S"
    return "."
```

修正後は、フィールドの操作である `get_cel()` メソッドから、 `*` や `S` の表記が消えて、代わりに `str(enemy)` と `str(player)` が記述されることで、駒や敵に関連する情報は対象のインスタンスから取得できるようになった。これによって、 `get_cel()` では、フィールドに関連した処理のみが残り、駒や敵に関する情報を分離でき、駒や敵に関しての修正が `get_cel()` にまで影響するか修正前ほど考える必要がなくなった。たとえば、駒の表示を `S` から `P` に修正したい場合は、修正前であれば、駒に関する情報であるにもかかわらずフィールドの操作に関連する `get_cell()` 関数を修正する必要があり、他に影響する箇所がないか確認する必要があった。修正後は、 `Player` クラスの `__str__` メソッドを修正すれば表示の変更ができ、 `get_cell()` メソッドを修正する必要はなくなった。

```
def get_cell(self, x, y):
    for player in self.players:
        if x == player.x and y == player.y:
            return str(player)
    for enemy in self.enemies:
        if x == enemy.x and y == enemy.y:
            return str(enemy)
    return "."
```

以上のように、それぞれ問題に対してもオブジェクト指向プログラミングに基づいた実装を行うことで解決したことがわかる。

問題1

問題文

このプログラムは、テキストの14章で取り扱った、縦5マス横5マスのフィールド上に、複数の駒を配置して、駒の移動する方向をプレイヤーが指定することで、駒を移動し、敵を取り除くゲームである。

`up`、`down`、`left`、`right`のいずれかを入力しプレイヤーの駒`S`を移動させて、駒`S`を敵`*`の位置に移動すれば敵`*`を倒すことができる。全ての敵`*`を倒せば、ゲームクリアとなり、ゲームを終了する。

フィールドの左上隅の座標が横方向0、縦方向0の座標、フィールドの右下隅が横方向4、縦方向4の座標である。

また、`end`と入力されると、ゲームを強制終了と表示して、ゲームを終了する。

修正前のゲームの初期配置は以下である。

```
*....
.*...
..*..
...S.
....S
```

このプログラムに対して処理の追加や修正を行い、横方向に3、縦方向に0の座標と、横方向に4、縦方向に3の座標の座標に敵`*`を追加せよ。ただし、処理の追加や修正を行い際に、クラスは使用しないこと。

敵を追加したあとのゲームの初期配置は以下である。

```
*..*.
.*...
..*..
...S*
....S
```

制約

- `up`、`down`、`left`、`right`、`end`以外の文字列を入力しない。
- 駒の座標が盤外にでる（横方向もしくは縦方向の座標が0から4以までの数字以外の値になる）ような方向を入力しない。
- このプログラムに対しての処理の追加や修正を行う際に、クラスを使用しない。

入力

入力は次の形式で与えられる。

駒を移動する場合は、移動方向 `DIRECTION` を1行ずつ入力する。

```
DIRECTION
DIRECTION
...
```

移動方向 `DIRECTION` は、 `up` 、 `down` 、 `left` 、 `right` のいずれかの値が与えられる。

また、ゲームを強制終了する場合は、最後に `end` を1行で入力する。

```
end
```

出力

方向を入力するごとに、入力した方向に基づいて、以下のような駒を移動した後の盤面が表示される。

```
*..*.
.*...
..*..
...S*
....S
```

ゲームをクリアしたら、最後に `ゲームクリア！` と表示される。

ゲームを強制終了したら、最後に `ゲームを強制終了` と表示される。

入力例1

```
up
up
left
up
up
up
left
up
up
 left
 left
```

出力例1

```
*..*.
.*...
..*..
...S*
....S

*..*.
.*...
..*S.
....*
....S

*..*.
.*...
..*S.
....S
```

```
.....

*..*.
.*...
..S..
....S
.....

*..*.
.*...
..S.S
.....
.....

*..*.
.*S..
....S
.....
.....

*..*.
.*S.S
.....
.....
.....

*..*.
.S..S
.....
.....
.....

*..*S
.S...
.....
.....
.....

*S.*S
.....
.....
.....
.....

*S.S.
.....
.....
.....
.....

S..S.
.....
.....
.....
.....

ゲームクリア！
```

入力例2

```
end
```

出力例2

```
*..*.
.*...
```

```
..*..
...S*
....S

ゲームを強制終了
```

解答の雛形

```python
# 二つの駒と三つ敵の座標をそれぞれ表すリスト
player_x = [3, 4]
player_y = [3, 4]
enemy_x = [0, 1, 2]
enemy_y = [0, 1, 2]
# 敵が倒されたかどうかを表すフラグ
enemy_defeated = [False, False, False]
# 動かす駒を表す変数
turn = 0


# 駒と敵の座標からフィールドの表示するマスの文字を返す関数
def get_cell(x, y):
    # 敵が配置されている座標を'*'で表す
    for i in range(len(enemy_x)):
        if x == enemy_x[i] and y == enemy_y[i] and not enemy_defeated[i]:
            return "*"
    # プレイヤーの駒が配置されている座標を'S'で表す
    for i in range(len(player_x)):
        if x == player_x[i] and y == player_y[i]:
            return "S"
    # それ以外の位置は'.'で表す
    return "."


# フィールドを表示する関数
def show_field():
    # リストを作成する
    field = []
    # 縦方向の座標（y座標）を表す変数を0から4までループ
    for y in range(5):
        # 横方向の座標（x座標）を表す変数を0から4までループ
        for x in range(5):
            cell = get_cell(x, y)
            field.append(cell)
        # x座標を表す変数が4まで到達したら改行する
        field.append("\n")
    # 作成した配列を文字列に変換して表示する
    print("".join(field))


# 駒を移動する関数
def move_player(direction):
    if direction == "up":
        player_y[turn] -= 1
    elif direction == "down":
        player_y[turn] += 1
    elif direction == "left":
        player_x[turn] -= 1
    elif direction == "right":
        player_x[turn] += 1
    else:
        raise ValueError("無効な方向です")


# フィールドの状態を表示
show_field()

while True:
    # 駒を移動する方向を入力する
    direction = input()
```

```
    # 入力された文字列が'end'の場合は、無限ループを終了する
    if direction == "end":
        print("ゲームを強制終了")
        break
    # 入力された方向に駒を移動する
    move_player(direction)

    for i in range(len(enemy_x)):
        # 駒が倒されていない敵と同一座標に到達したら、その敵を倒したことにする
        if (
            player_x[turn] == enemy_x[i]
            and player_y[turn] == enemy_y[i]
            and not enemy_defeated[i]
        ):
            enemy_defeated[i] = True

    # フィールドの状態を表示
    show_field()

    # 全部の敵を倒したらクリア
    if all(enemy_defeated):
        print("ゲームクリア！")
        break

    # 駒を交互に移動させる
    turn = (turn + 1) % 2
```

問題2

問題文

本問で提示されているプログラムは、問題1と異なる記述がされているが、同様の挙動をするプログラムである。問題にある仕様を満たすための処理の追記や修正を行った後の挙動も同様である。

問題1ではクラスを使わないプログラムが記述されており、本問ではクラスを使ったプログラムが記述されている。クラスを用いることで、クラスを用いなかった問題1と比べて、プログラムの分かりやすさや処理の追加・修正のしやすさが、どのように変わるか体験してみよう。

以下、問題の本文である。

このプログラムは、テキストの14章で取り扱った、縦5マス横5マスのフィールド上に、複数の駒を配置して、駒の移動する方向をプレイヤーが指定することで、駒を移動し、敵を取り除くゲームである。

`up`、`down`、`left`、`right`のいずれかを入力しプレイヤーの駒`S`を移動させて、駒`S`を敵`*`の位置に移動すれば敵`*`を倒すことができる。全ての敵`*`を倒せば、ゲームクリアとなり、ゲームを終了する。

フィールドの左上隅の座標が横方向0、縦方向0の座標、フィールドの右下隅が横方向4、縦方向4の座標である。

また、`end`と入力されると、ゲームを強制終了と表示して、ゲームを終了する。

修正前のゲームの初期配置は以下である。

```
1 *....
2 .*...
3 ..*..
4 ...S.
5 ....S
```

このプログラムに対して処理の追加や修正を行い、横方向に3、縦方向に0の座標と、横方向に4、縦方向に3の座標の座標に敵`*`を追加せよ。

敵を追加したあとのゲームの初期配置は以下である。

```
*..*.
.*...
..*..
...S*
....S
```

制約

- `up` 、 `down` 、 `left` 、 `right` 、 `end` 以外の文字列を入力しない。
- 駒の座標が盤外にでる（横方向もしくは縦方向の座標が0から4以までの数字以外の値になる）ような方向を入力しない。

入力

入力は次の形式で与えられる。

駒を移動する場合は、移動方向 `DIRECTION` を1行ずつ入力する。

```
DIRECTION
DIRECTION
...
```

移動方向 `DIRECTION` は、 `up` 、 `down` 、 `left` 、 `right` のいずれかの値が与えられる。

また、ゲームを強制終了する場合は、最後に `end` を1行で入力する。

```
end
```

出力

方向を入力するごとに、入力した方向に基づいて、以下のような駒を移動した後の盤面が表示される。

```
*..*.
.*...
..*..
...S*
....S
```

ゲームをクリアしたら、最後に `ゲームクリア！` と表示される。

ゲームを強制終了したら、最後に `ゲームを強制終了` と表示される。

入力例1

```
up
up
left
up
up
up
left
up
up
left
```

```
left
```

出力例1

```
*..*.
.*...
..*..
...S*
....S

*..*.
.*...
..*S.
....*
....S

*..*.
.*...
..*S.
....S
.....

*..*.
.*...
..S..
....S
.....

*..*.
.*...
..S.S
.....
.....

*..*.
.*S..
....S
.....
.....

*..*.
.*S.S
.....
.....
.....

*..*.
.S..S
.....
.....
.....

*..*S
.S...
.....
.....
.....

*S.*S
.....
.....
.....
.....

*S.S.
.....
.....
.....
.....
```

```
S..S.
.....
.....
.....
.....

ゲームクリア！
```

入力例2

```
end
```

出力例2

```
*..*.
.*...
..*..
...S*
....S

ゲームを強制終了
```

解答の雛形

```python
# プレイヤーの駒のクラス
class Player:
    def __init__(self, x, y):
        # 初期配置の座標
        self.x = x
        self.y = y

    # フィールド上での表示
    def __str__(self):
        return "S"

    # プレイヤーの駒の移動を行うメソッド
    def move(self, direction):
        if direction == "up":
            self.y -= 1
        elif direction == "down":
            self.y += 1
        elif direction == "left":
            self.x -= 1
        elif direction == "right":
            self.x += 1
        else:
            raise ValueError("無効な方向です")


# 敵のクラス
class Enemy:
    def __init__(self, x, y):
        # 初期配置の座標
        self.x = x
        self.y = y
        # 倒されたかどうかのフラグ
        self.defeated = False

    # フィールド上での表示
    def __str__(self):
        return "*" if not self.defeated else "."
```

```
# フィールドのクラス
class Field:
    def __init__(self, players, enemies):
        # フィールド上に配置するプレイヤーの駒と敵
        self.players = players
        self.enemies = enemies
        # フィールドの大きさ
        self.size = 5

    # 指定した座標のマスの文字を返すメソッド
    def get_cell(self, x, y):
        for player in self.players:
            if x == player.x and y == player.y:
                return str(player)
        for enemy in self.enemies:
            if x == enemy.x and y == enemy.y:
                return str(enemy)
        return "."

    # フィールドの状態を出力するメソッド
    def show(self):
        field = []
        for y in range(self.size):
            for x in range(self.size):
                cell = self.get_cell(x, y)
                field.append(cell)
            field.append("\n")
        print("".join(field))


# ゲームに必要なインスタンスの作成
players = [Player(3, 3), Player(4, 4)]
enemies = [Enemy(0, 0), Enemy(1, 1), Enemy(2, 2)]
field = Field(players, enemies)

# フィールドの初期状態の表示
field.show()

# 動かす駒を表す変数
turn = 0

while True:
    # 駒を移動する方向を入力する
    direction = input()
    # 入力された文字列が'end'の場合は、無限ループを終了する
    if direction == "end":
        print("ゲームを強制終了")
        break
    # 入力された方向に駒を移動する
    field.players[turn].move(direction)

    for enemy in field.enemies:
        # 駒が倒されていない敵と同一座標に到達したら、その敵を倒したことにする
        if field.players[turn].x == enemy.x and field.players[turn].y == enemy.y:
            enemy.defeated = True

    # フィールドの状態を表示
    field.show()

    # 全部の敵を倒したらクリア
    is_game_clear = True
    for enemy in field.enemies:
        if not enemy.defeated:
            is_game_clear = False
            break
    if is_game_clear:
        print("ゲームクリア！")
        break

    # 駒を交互に移動させる処理
    turn = (turn + 1) % len(players)
```

問題3

問題文

このプログラムは、テキストの14章で取り扱った、縦5マス横5マスのフィールド上に、複数の駒を配置して、駒の移動する方向をプレイヤーが指定することで、駒を移動し、敵を取り除くゲームである。

`up` 、 `down` 、 `left` 、 `right` のいずれかを入力しプレイヤーの駒 `S` を移動させて、駒 `S` を敵 `*` の位置に移動すれば敵 `*` を倒すことができる。全ての敵 `*` を倒せば、ゲームクリアとなり、ゲームを終了する。

フィールドの左上隅の座標が横方向0、縦方向0の座標、フィールドの右下隅が横方向4、縦方向4の座標である。

また、 `end` と入力されると、ゲームを強制終了と表示して、ゲームを終了する。

修正前のゲームの初期配置は以下である。

```
1 *....
2 .*...
3 ..*..
4 ...S.
5 ....S
```

このプログラムでは、倒された敵がいるマスは、 `.` と何もないフィールドと同様文字で表されている。このプログラムに対して処理の追加や修正を行い、倒された敵がいるマスには `.` ではなく `_` を表示せよ。ただし、このプログラムに対しての処理の追加や修正を行う際に、クラスを使用しないこと。

一例として、処理の追加や修正を行った後、ゲームをクリアした時点でのフィールドは以下のように表示される。

```
1 S..S.
2 ._...
3 .._..
4 .....
5 .....
```

制約

- `up` 、 `down` 、 `left` 、 `right` 、 `end` 以外の文字列を入力しない。
- 駒の座標が盤外にでる（横方向もしくは縦方向の座標が0から4以までの数字以外の値になる）ような方向を入力しない。
- このプログラムに対しての処理の追加や修正を行う際に、クラスを使用しない。

入力

入力は次の形式で与えられる。

駒を移動する場合は、移動方向 `DIRECTION` を1行ずつ入力する。

```
1 DIRECTION
2 DIRECTION
3 ...
```

移動方向 `DIRECTION` は、 `up` 、 `down` 、 `left` 、 `right` のいずれかの値が与えられる。

また、ゲームを強制終了する場合は、最後に `end` を1行で入力する。

```
end
```

出力

方向を入力するごとに、入力した方向に基づいて、以下のような駒を移動した後の盤面が表示される。

```
*....
.*...
..*..
...S.
....S
```

また、プレイヤーの駒がいなく倒された敵がいるマスには、 `_` と表示される。

ゲームをクリアしたら、最後に `ゲームクリア！` と表示される。

ゲームを強制終了したら、最後に `ゲームを強制終了` と表示される。

入力例1

```
up
up
left
up
up
up
left
up
up
left
left
```

出力例1

```
*....
.*...
..*..
...S.
....S

*....
.*...
..*S.
.....
....S

*....
.*...
..*S.
....S
.....

*....
.*...
..S..
....S
.....

*....
.*...
..S.S
```

```
.....
.....

*....
.*S..
..._.S
.....
.....

*....
.*S.S
..._..
.....
.....

*....
.S..S
..._..
.....
.....

*...S
.S...
..._..
.....
.....

*S..S
._...
..._..
.....
.....

*S.S.
._...
..._..
.....
.....

S..S.
._...
..._..
.....
.....

ゲームクリア！
```

入力例2

```
end
```

出力例2

```
*....
.*...
..*..
...S.
....S

ゲームを強制終了
```

解答の雛形

```
# 二つの駒と三つ敵の座標をそれぞれ表すリスト
player_x = [3, 4]
player_y = [3, 4]
enemy_x = [0, 1, 2]
enemy_y = [0, 1, 2]
# 敵が倒されたかどうかを表すフラグ
enemy_defeated = [False, False, False]
# 動かす駒を表す変数
turn = 0


# 駒と敵の座標からフィールドの表示するマスの文字を返す関数
def get_cell(x, y):
    # プレイヤーの駒が配置されている座標を'S'で表す
    for i in range(len(player_x)):
        if x == player_x[i] and y == player_y[i]:
            return "S"
    # 敵が配置されている座標を'*'で表す
    for i in range(len(enemy_x)):
        if x == enemy_x[i] and y == enemy_y[i] and not enemy_defeated[i]:
            return "*"
    # それ以外の位置は'.'で表す
    return "."


# フィールドを表示する関数
def show_field():
    # リストを作成する
    field = []
    # 縦方向の座標（y座標）を表す変数を0から4までループ
    for y in range(5):
        # 横方向の座標（x座標）を表す変数を0から4までループ
        for x in range(5):
            cell = get_cell(x, y)
            field.append(cell)
        # x座標を表す変数が4まで到達したら改行する
        field.append("\n")
    # 作成した配列を文字列に変換して表示する
    print("".join(field))


# 駒を移動する関数
def move_player(direction):
    if direction == "up":
        player_y[turn] -= 1
    elif direction == "down":
        player_y[turn] += 1
    elif direction == "left":
        player_x[turn] -= 1
    elif direction == "right":
        player_x[turn] += 1
    else:
        raise ValueError("無効な方向です")


# フィールドの状態を表示
show_field()

while True:
    # 駒を移動する方向を入力する
    direction = input()
    # 入力された文字列が'end'の場合は、無限ループを終了する
    if direction == "end":
        print("ゲームを強制終了")
        break
    # 入力された方向に駒を移動する
    move_player(direction)

    for i in range(len(enemy_x)):
        # 駒が倒されていない敵と同一座標に到達したら、その敵を倒したことにする
        if (
```

```
            player_x[turn] == enemy_x[i]
            and player_y[turn] == enemy_y[i]
            and not enemy_defeated[i]
        ):
            enemy_defeated[i] = True

    # フィールドの状態を表示
    show_field()

    # 全部の敵を倒したらクリア
    if all(enemy_defeated):
        print("ゲームクリア！")
        break

    # 駒を交互に移動させる
    turn = (turn + 1) % 2
```

問題4

問題文

本問で提示されているプログラムは、問題3と異なる記述がされているが、同様の挙動をするプログラムである。問題にある仕様を満たすための処理の追記や修正を行った後の挙動も同様である。

問題3ではクラスを使わないプログラムが記述されており、本問ではクラスを使ったプログラムが記述されている。クラスを用いることで、クラスを用いなかった問題3と比べて、プログラムの分かりやすさや処理の追加・修正のしやすさが、どのように変わるか体験してみよう。

以下、問題の本文である。

このプログラムは、テキストの14章で取り扱った、縦5マス横5マスのフィールド上に、複数の駒を配置して、駒の移動する方向をプレイヤーが指定することで、駒を移動し、敵を取り除くゲームである。

`up`、`down`、`left`、`right` のいずれかを入力しプレイヤーの駒 `S` を移動させて、駒 `S` を敵 `*` の位置に移動すれば敵 `*` を倒すことができる。全ての敵 `*` を倒せば、ゲームクリアとなり、ゲームを終了する。

フィールドの左上隅の座標が横方向0、縦方向0の座標、フィールドの右下隅が横方向4、縦方向4の座標である。

また、`end` と入力されると、ゲームを強制終了と表示して、ゲームを終了する。

修正前のゲームの初期配置は以下である。

```
1 *....
2 .*...
3 ..*..
4 ...S.
5 ....S
```

このプログラムでは、倒された敵がいるマスは、`.` と何もないフィールドと同様文字で表されている。このプログラムに対して処理の追加や修正を行い、倒された敵がいるマスには `.` ではなく `_` を表示せよ。

一例として、処理の追加や修正を行った後、ゲームをクリアした時点でのフィールドは以下のように表示される。

```
S..S.
._...
..._..
.....
.....
```

制約

- `up` 、 `down` 、 `left` 、 `right` 、 `end` 以外の文字列を入力しない。
- 駒の座標が盤外にでる（横方向もしくは縦方向の座標が0から4以までの数字以外の値になる）ような方向を入力しない。

入力

入力は次の形式で与えられる。

駒を移動する場合は、移動方向 `DIRECTION` を1行ずつ入力する。

```
DIRECTION
DIRECTION
...
```

移動方向 `DIRECTION` は、 `up` 、 `down` 、 `left` 、 `right` のいずれかの値が与えられる。

また、ゲームを強制終了する場合は、最後に `end` を1行で入力する。

```
end
```

出力

方向を入力するごとに、入力した方向に基づいて、以下のような駒を移動した後の盤面が表示される。

```
*....
 .*...
 ..*..
 ...S.
 ....S
```

また、プレイヤーの駒がいなく倒された敵がいるマスには、 `_` と表示される。

ゲームをクリアしたら、最後に `ゲームクリア！` と表示される。

ゲームを強制終了したら、最後に `ゲームを強制終了` と表示される。

入力例1

```
up
up
left
up
up
up
left
up
up
 left
```

```
left
```

出力例1

```
*....
.*...
..*..
...S.
....S

*....
.*...
..*S.
.....
....S

*....
.*...
..*S.
....S
.....

*....
.*...
..S..
....S
.....

*....
.*...
..S.S
.....
.....

*....
.*S..
.._.S
.....
.....

*....
.*S.S
.._..
.....
.....

*....
.S..S
.._..
.....
.....

*...S
.S...
.._..
.....
.....

*S..S
._...
.._..
.....
.....

*S.S.
._...
.._..
.....
.....
```

```
66
67 S..S.
68 ._...
69 .._..
70 .....
71 .....
72
73 ゲームクリア！
```

入力例2

```
1 end
```

出力例2

```
1 *....
2 .*...
3 ..*..
4 ...S.
5 ....S
6
7 ゲームを強制終了
```

解答の雛形

```python
# プレイヤーの駒のクラス
class Player:
    def __init__(self, x, y):
        # 初期配置の座標
        self.x = x
        self.y = y

    # フィールド上での表示
    def __str__(self):
        return "S"

    # プレイヤーの駒の移動を行うメソッド
    def move(self, direction):
        if direction == "up":
            self.y -= 1
        elif direction == "down":
            self.y += 1
        elif direction == "left":
            self.x -= 1
        elif direction == "right":
            self.x += 1
        else:
            raise ValueError("無効な方向です")


# 敵のクラス
class Enemy:
    def __init__(self, x, y):
        # 初期配置の座標
        self.x = x
        self.y = y
        # 倒されたかどうかのフラグ
        self.defeated = False

    # フィールド上での表示
    def __str__(self):
        return "*" if not self.defeated else "."
```

```
# フィールドのクラス
class Field:
    def __init__(self, players, enemies):
        # フィールド上に配置するプレイヤーの駒と敵
        self.players = players
        self.enemies = enemies
        # フィールドの大きさ
        self.size = 5

    # 指定した座標のマスの文字を返すメソッド
    def get_cell(self, x, y):
        for player in self.players:
            if x == player.x and y == player.y:
                return str(player)
        for enemy in self.enemies:
            if x == enemy.x and y == enemy.y:
                return str(enemy)
        return "."

    # フィールドの状態を出力するメソッド
    def show(self):
        field = []
        for y in range(self.size):
            for x in range(self.size):
                cell = self.get_cell(x, y)
                field.append(cell)
            field.append("\n")
        print("".join(field))


# ゲームに必要なインスタンスの作成
players = [Player(3, 3), Player(4, 4)]
enemies = [Enemy(0, 0), Enemy(1, 1), Enemy(2, 2)]
field = Field(players, enemies)

# フィールドの初期状態の表示
field.show()

# 動かす駒を表す変数
turn = 0

while True:
    # 駒を移動する方向を入力する
    direction = input()
    # 入力された文字列が'end'の場合は、無限ループを終了する
    if direction == "end":
        print("ゲームを強制終了")
        break
    # 入力された方向に駒を移動する
    field.players[turn].move(direction)

    for enemy in field.enemies:
        # 駒が倒されていない敵と同一座標に到達したら、その敵を倒したことにする
        if field.players[turn].x == enemy.x and field.players[turn].y == enemy.y:
            enemy.defeated = True

    # フィールドの状態を表示
    field.show()

    # 全部の敵を倒したらクリア
    is_game_clear = True
    for enemy in field.enemies:
        if not enemy.defeated:
            is_game_clear = False
            break
    if is_game_clear:
        print("ゲームクリア！")
        break

    # 駒を交互に移動させる処理
    turn = (turn + 1) % len(players)
```

問題5

問題文

このプログラムは、テキストの14章で取り扱った、縦5マス横5マスのフィールド上に、複数の駒を配置して、駒の移動する方向をプレイヤーが指定することで、駒を移動し、敵を取り除くゲームである。

`up`、`down`、`left`、`right`のいずれかを入力しプレイヤーの駒`S`を移動させて、駒`S`を敵`*`の位置に移動すれば敵`*`を倒すことができる。全ての敵`*`を倒せば、ゲームクリアとなり、ゲームを終了する。

フィールドの左上隅の座標が横方向0、縦方向0の座標、フィールドの右下隅が横方向4、縦方向4の座標である。

また、`end`と入力されると、ゲームを強制終了と表示して、ゲームを終了する。

修正前のゲームの初期配置は以下である。

```
*....
.*...
..*..
...S.
....S
```

このプログラムに対して処理の追加や修正を行い、横方向と縦方向の座標が3の駒には`A`と、横方向と縦方向の座標が4の駒には`B`という名前を与える。加えて、敵`*`を倒すごとに得点として1点を駒に与える。そして、ゲームをクリアした際に、駒`A`と`B`それぞれ名前と得点を表示するようにせよ。ただし、このプログラムに対しての処理の追加や修正を行う際に、クラスを使用しないこと。

一例として、処理の追加や修正を行った後、ゲームを行い、駒`A`が2体、駒`B`が1体敵を倒して、ゲームをクリアした際には、最後に以下のように表示される。

```
ゲームクリア！
Aは2点
Bは1点
```

制約

- `up`、`down`、`left`、`right`、`end`以外の文字列を入力しない。
- 駒の座標が盤外にでる（横方向もしくは縦方向の座標が0から4以までの数字以外の値になる）ような方向を入力しない。
- このプログラムに対しての処理の追加や修正を行う際に、クラスを使用しない。

入力

入力は次の形式で与えられる。

駒を移動する場合は、移動方向`DIRECTION`を1行ずつ入力する。

```
DIRECTION
DIRECTION
...
```

移動方向`DIRECTION`は、`up`、`down`、`left`、`right`のいずれかの値が与えられる。

また、ゲームを強制終了する場合は、最後に`end`を1行で入力する。

```
end
```

出力

方向を入力するごとに、入力した方向に基づいて、以下のような駒を移動した後の盤面が表示される。

```
*....
.*...
..*..
...S.
....S
```

ゲームをクリアしたら、最後に以下のように `ゲームクリア！` という文字列と、駒 `A` の得点Nと駒 `B` の得点Mが以下のように1行ずつ表示される。

```
ゲームクリア！
Aは$N$点
Bは$M$点
```

ゲームを強制終了したら、最後に `ゲームを強制終了` と表示される。

入力例1

```
up
up
up
up
left
left
left
left
up
 up
 left
```

出力例1

```
*....
.*...
..*..
...S.
....S

*....
.*...
..*S.
.....
....S

*....
.*...
..*S.
....S
.....

*....
.*.S.
..*..
```

```
....S
.....

*....
.*.S.
..*.S
.....
.....

*....
.*S..
..*.S
.....
.....

*....
.*S..
..*S.
.....
.....

*....
.S...
..*S.
.....
.....

*....
.S...
..S..
.....
.....

*S...
.....
..S..
.....
.....

*S...
..S..
.....
.....
.....

S....
..S..
.....
.....
.....

ゲームクリア！
Aは2点
Bは1点
```

入力例2

```
end
```

出力例2

```
*....
.*...
..*..
...S.
....S
```

```
6
7  ゲームを強制終了
```

解答の雛形

```
# 二つの駒と三つ敵の座標をそれぞれ表すリスト
player_x = [3, 4]
player_y = [3, 4]
enemy_x = [0, 1, 2]
enemy_y = [0, 1, 2]
# 敵が倒されたかどうかを表すフラグ
enemy_defeated = [False, False, False]
# 動かす駒を表す変数
turn = 0


# 駒と敵の座標からフィールドの表示するマスの文字を返す関数
def get_cell(x, y):
    # 敵が配置されている座標を'*'で表す
    for i in range(len(enemy_x)):
        if x == enemy_x[i] and y == enemy_y[i] and not enemy_defeated[i]:
            return "*"
    # プレイヤーの駒が配置されている座標を'S'で表す
    for i in range(len(player_x)):
        if x == player_x[i] and y == player_y[i]:
            return "S"
    # それ以外の位置は'.'で表す
    return "."


# フィールドを表示する関数
def show_field():
    # リストを作成する
    field = []
    # 縦方向の座標（y座標）を表す変数を0から4までループ
    for y in range(5):
        # 横方向の座標（x座標）を表す変数を0から4までループ
        for x in range(5):
            cell = get_cell(x, y)
            field.append(cell)
        # x座標を表す変数が4まで到達したら改行する
        field.append("\n")
    # 作成した配列を文字列に変換して表示する
    print("".join(field))


# 駒を移動する関数
def move_player(direction):
    if direction == "up":
        player_y[turn] -= 1
    elif direction == "down":
        player_y[turn] += 1
    elif direction == "left":
        player_x[turn] -= 1
    elif direction == "right":
        player_x[turn] += 1
    else:
        raise ValueError("無効な方向です")


# フィールドの状態を表示
show_field()

while True:
    # 駒を移動する方向を入力する
    direction = input()
    # 入力された文字列が'end'の場合は、無限ループを終了する
    if direction == "end":
        print("ゲームを強制終了")
```

```
        break
    # 入力された方向に駒を移動する
    move_player(direction)

    for i in range(len(enemy_x)):
        # 駒が倒されていない敵と同一座標に到達したら、その敵を倒したことにする
        if (
            player_x[turn] == enemy_x[i]
            and player_y[turn] == enemy_y[i]
            and not enemy_defeated[i]
        ):
            enemy_defeated[i] = True

    # フィールドの状態を表示
    show_field()

    # 全部の敵を倒したらクリア
    if all(enemy_defeated):
        print("ゲームクリア！")
        break

    # 駒を交互に移動させる
    turn = (turn + 1) % 2
```

問題6

問題文

本問で提示されているプログラムは、問題5と異なる記述がされているが、同様の挙動をするプログラムである。問題にある仕様を満たすための処理の追記や修正を行った後の挙動も同様である。

問題5ではクラスを使わないプログラムが記述されており、本問ではクラスを使ったプログラムが記述されている。クラスを用いることで、クラスを用いなかった問題5と比べて、プログラムの分かりやすさや処理の追加・修正のしやすさが、どのように変わるか体験してみよう。

以下、問題の本文である。

このプログラムは、テキストの14章で取り扱った、縦5マス横5マスのフィールド上に、複数の駒を配置して、駒の移動する方向をプレイヤーが指定することで、駒を移動し、敵を取り除くゲームである。

`up`、`down`、`left`、`right`のいずれかを入力しプレイヤーの駒`S`を移動させて、駒`S`を敵`*`の位置に移動すれば敵`*`を倒すことができる。全ての敵`*`を倒せば、ゲームクリアとなり、ゲームを終了する。

フィールドの左上隅の座標が横方向0、縦方向0の座標、フィールドの右下隅が横方向4、縦方向4の座標である。

また、`end`と入力されると、ゲームを強制終了と表示して、ゲームを終了する。

修正前のゲームの初期配置は以下である。

```
1 *....
2 .*...
3 ..*..
4 ...S.
5 ....S
```

このプログラムに対して処理の追加や修正を行い、横方向と縦方向の座標が3の駒には`A`と、横方向と縦方向の座標が4の駒には`B`という名前を与える。加えて、敵`*`を倒すごとに得点として1点を駒に与える。そして、ゲームをクリアした際に、駒`A`と`B`それぞれ名前と得点を表示するようにせよ。

一例として、処理の追加や修正を行った後、ゲームを行い、駒 `A` が2体、駒 `B` が1体敵を倒して、ゲームをクリアした際には、最後に以下のように表示される。

```
ゲームクリア！
Aは2点
Bは1点
```

制約

- `up` 、 `down` 、 `left` 、 `right` 、 `end` 以外の文字列を入力しない。
- 駒の座標が盤外にでる（横方向もしくは縦方向の座標が0から4以までの数字以外の値になる）ような方向を入力しない。

入力

入力は次の形式で与えられる。

駒を移動する場合は、移動方向 `DIRECTION` を1行ずつ入力する。

```
DIRECTION
DIRECTION
...
```

移動方向 `DIRECTION` は、 `up` 、 `down` 、 `left` 、 `right` のいずれかの値が与えられる。

また、ゲームを強制終了する場合は、最後に `end` を1行で入力する。

```
end
```

出力

方向を入力するごとに、入力した方向に基づいて、以下のような駒を移動した後の盤面が表示される。

```
*....
.*...
..*..
...S.
....S
```

ゲームをクリアしたら、最後に以下のように `ゲームクリア！` という文字列と、駒 `A` の得点Nと駒 `B` の得点Mが以下のように1行ずつ表示される。

```
ゲームクリア！
Aは$N$点
Bは$M$点
```

ゲームを強制終了したら、最後に `ゲームを強制終了` と表示される。

入力例1

```
up
up
up
```

```
up
left
left
left
left
up
 up
 left
```

出力例1

```
*....
.*...
..*..
...S.
....S

*....
.*...
..*S.
.....
....S

*....
.*...
..*S.
....S
.....

*....
.*.S.
..*..
....S
.....

*....
.*.S.
..*.S
.....
.....

*....
.*S..
..*.S
.....
.....

*....
.*S..
..*S.
.....
.....

*....
.S...
..*S.
.....
.....

*....
.S...
..S..
.....
.....

*S...
.....
..S..
.....
```

```
.....

*S...
..S..
.....
.....
.....

S....
 ..S..
.....
.....
.....

ゲームクリア！
Aは2点
Bは1点
```

入力例2

```
end
```

出力例2

```
*....
 .*...
..*..
...S.
....S

ゲームを強制終了
```

解答の雛形

```
# プレイヤーの駒のクラス
class Player:
    def __init__(self, x, y):
        # 初期配置の座標
        self.x = x
        self.y = y

    # フィールド上での表示
    def __str__(self):
        return "S"

    # プレイヤーの駒の移動を行うメソッド
    def move(self, direction):
        if direction == "up":
            self.y -= 1
        elif direction == "down":
            self.y += 1
        elif direction == "left":
            self.x -= 1
        elif direction == "right":
            self.x += 1
        else:
            raise ValueError("無効な方向です")


# 敵のクラス
class Enemy:
    def __init__(self, x, y):
        # 初期配置の座標
```

```
        self.x = x
        self.y = y
        # 倒されたかどうかのフラグ
        self.defeated = False

    # フィールド上での表示
    def __str__(self):
        return "*" if not self.defeated else "."


# フィールドのクラス
class Field:
    def __init__(self, players, enemies):
        # フィールド上に配置するプレイヤーの駒と敵
        self.players = players
        self.enemies = enemies
        # フィールドの大きさ
        self.size = 5

    # 指定した座標のマスの文字を返すメソッド
    def get_cell(self, x, y):
        for player in self.players:
            if x == player.x and y == player.y:
                return str(player)
        for enemy in self.enemies:
            if x == enemy.x and y == enemy.y:
                return str(enemy)
        return "."

    # フィールドの状態を出力するメソッド
    def show(self):
        field = []
        for y in range(self.size):
            for x in range(self.size):
                cell = self.get_cell(x, y)
                field.append(cell)
            field.append("\n")
        print("".join(field))


# ゲームに必要なインスタンスの作成
players = [Player(3, 3), Player(4, 4)]
enemies = [Enemy(0, 0), Enemy(1, 1), Enemy(2, 2)]
field = Field(players, enemies)

# フィールドの初期状態の表示
field.show()

# 動かす駒を表す変数
turn = 0

while True:
    # 駒を移動する方向を入力する
    direction = input()
    # 入力された文字列が'end'の場合は、無限ループを終了する
    if direction == "end":
        print("ゲームを強制終了")
        break
    # 入力された方向に駒を移動する
    field.players[turn].move(direction)

    for enemy in field.enemies:
        # 駒が倒されていない敵と同一座標に到達したら、その敵を倒したことにする
        if field.players[turn].x == enemy.x and field.players[turn].y == enemy.y:
            enemy.defeated = True

    # フィールドの状態を表示
    field.show()

    # 全部の敵を倒したらクリア
    is_game_clear = True
    for enemy in field.enemies:
```

```
            if not enemy.defeated:
                is_game_clear = False
                break
        if is_game_clear:
            print("ゲームクリア！")
            break

        # 駒を交互に移動させる処理
        turn = (turn + 1) % len(players)
```

15章: アプリケーション開発（3）

本章の流れ

本章では、14章までに開発したゲームに追加の実装をする過程を通して、継承の実用的な使い方について学ぶ。

また、実装する過程で、プログラムに対してリファクタリングという作業を行い、より品質の高いプログラムを作成する方法についても学ぶ。

リファクタリング

リファクタリングとは、修正前と同じ動作をするように外部から見た振る舞いを保ちつつ、理解や修正が簡単にできるようにプログラムの内部構造を修正することである。

今回実施していた開発の中ではわかりやすいリファクタリングの例としては、13章で行ったフィールドの状態の表示を行う処理を `show_field()` 関数として抽出した実装がある。

以下は、フィールドの状態の表示を行う処理を `show_field()` 関数としてまとめなかった場合の例である。

```
# 座標を表す変数を定義
player_x, player_y = 2, 2
enemy_x, enemy_y = 0, 0

# フィールドの状態を表示
# リストを作成する
field = []
# 縦方向の座標（y座標）を表す変数を0から4までループ
for y in range(5):
    # 横方向の座標（x座標）を表す変数を0から4までループ
    for x in range(5):
        # 自分の駒が配置されている座標を"S"で表す
        if x == player_x and y == player_y:
            field.append("S")
        # 敵が配置されている座標を"*"で表す
        elif x == enemy_x and y == enemy_y:
            field.append("*")
        # それ以外の位置は"."で表す
        else:
            field.append(".")
    # x座標を表す変数が4まで到達したら改行する
    field.append("\n")
# 作成したリストを文字列に変換して表示する
print("".join(field))



while True:
    # 駒オブジェクトの配置位置を指定する
    line = input()
    # 入力された文字列が"end"の場合は、無限ループを終了する
    if line == "end":
        print("終了")
        break
    # 入力された座標を使用して、駒オブジェクトを配置する
    player_x, player_y = map(int, line.split())

    # フィールドの状態を表示
    # リストを作成する
    field = []
    # 縦方向の座標（y座標）を表す変数を0から4までループ
    for y in range(5):
        # 横方向の座標（x座標）を表す変数を0から4までループ
        for x in range(5):
            # 自分の駒が配置されている座標を"S"で表す
            if x == player_x and y == player_y:
                field.append("S")
            # 敵が配置されている座標を"*"で表す
```

```
            elif x == enemy_x and y == enemy_y:
                field.append("*")
            # それ以外の位置は"."で表す
            else:
                field.append(".")
        # x座標を表す変数が4まで到達したら改行する
        field.append("\n")
    # 作成したリストを文字列に変換して表示する
    print("".join(field))



    # ゲーム終了判定
    if enemy_x == player_x and enemy_y == player_y:
        break
```

```
*....
.....
..S..
.....
.....

1 0
*S...
.....
.....
.....

end
終了
```

以下は、上記のプログラムに対して、リファクタリングを行い、フィールドの状態の表示を行う処理を `show_field()` 関数としてまとめた場合の例である。

```
# 座標を表す変数を定義
player_x, player_y = 2, 2
enemy_x, enemy_y = 0, 0

# 敵や自分の駒が配置されている座標を表示する関数
def show_field():
    # リストを作成する
    field = []
    # 縦方向の座標（y座標）を表す変数を0から4までループ
    for y in range(5):
        # 横方向の座標（x座標）を表す変数を0から4までループ
        for x in range(5):
            # 自分の駒が配置されている座標を"S"で表す
            if x == player_x and y == player_y:
                field.append("S")
            # 敵が配置されている座標を"*"で表す
            elif x == enemy_x and y == enemy_y:
                field.append("*")
            # それ以外の位置は"."で表す
            else:
                field.append(".")
        # x座標を表す変数が4まで到達したら改行する
        field.append("\n")
    # 作成したリストを文字列に変換して表示する
    print("".join(field))


# フィールドの状態を表示
show_field()

while True:
    # 駒オブジェクトの配置位置を指定する
```

```
    line = input()
    # 入力された文字列が"end"の場合は、無限ループを終了する
    if line == "end":
        print("終了")
        break
    # 入力された座標を使用して、駒オブジェクトを配置する
    player_x, player_y = map(int, line.split())

    # フィールドの状態を表示
    show_field()

    # ゲーム終了判定
    if enemy_x == player_x and enemy_y == player_y:
        break
```

```
*....
.....
..S..
.....
.....

1 0
*S...
.....
.....
.....

end
終了
```

リファクタリングを行うことで、以下のようなメリットがある。

- 重複するコードを書かなくて済む
- プログラムを理解しやすくなる
- バグを見つけやすくなる
- 開発の速度が上がる

上に挙げたメリットが実際にあるか、フィールドの状態の表示を行う処理を `show_field()` 関数にまとめなかった例と、まとめた例を比較して確かめてみよう。

まずは、重複するコードを書かなくて済むか確認する。

重複したコードを減らすことは、開発において重要なことの一つである。なぜなら、プログラムを修正する場合、重複したコード全てに対して正しく修正を行う必要があるが、重複したコードを排除すれば、修正箇所を一か所に限定でき、修正のためのコストや修正漏れのリスクを大きく減らすことができるからである。

フィールドの状態の表示を行う処理を `show_field()` 関数にまとめなかった例では、フィールドの状態の表示を行いたい処理を追加しようとするたびに、複数行にわたるプログラムを追加しなければならなず、何度も重複したコードを書かなくてはならない。

フィールドの状態の表示を行う処理を `show_field()` 関数にまとめた例では、フィールドの状態の表示を行いたい処理を追加したければ、`show_field()` 関数を記述すればよいため、重複コードを書かなくても済む。

以上から、リファクタリングを行うことで重複したコードを書かなくて済むことがわかる。

続いて、プログラムが理解しやすくなっているか確認する。

プログラムの理解のしやすさは、チーム開発において特に重要である。実際に開発する場合、複数人数で開発することが多く、自分が書いたプログラムを他人が見たり、他人が書いたプログラムを自分が見たり、さらには過去に自分が書いたプログラムを読み解かなければ場面が多々ある。プログラムが理解しやすければ、既存のプログラムをすぐに理解でき、機能の追加やバグの発見を行いやすくなり、スムーズに開発できるようになる。

フィールドの状態の表示を行う処理を `show_field()` 関数にまとめなかった例では、フィールドの状態の表示を行う処理が無限ループの外と中で一回ずつ登場する。ただ、初見では、同じ処理を行っているかパッと見ただけではわからないため、無限ループの外と中で行われているフィールドの状態の表示を行う処理を一行ずつ確認して初めて同じ処理が行われているということがわかる。そのため、理解するのに時間がかかる。

フィールドの状態の表示を行う処理を `show_field()` 関数にまとめた例では、 `show_field()` 関数の処理を確認して、フィールドの状態の表示を行う処理を実施する関数であると理解する。一度 `show_field()` 関数について理解すれば、 `show_field()` 関数が呼び出されている箇所（この例では、無限ループの外と中で一か所ずつ）を確認するだけで、いつフィールドの状態の表示を行うか理解できる。

以上から、リファクタリングを行うことでプログラムが理解しやすくなったことがわかる。

続いて、バグを見つけやすくなるか確認する。

例に挙げたプログラムでは、プレイヤーの駒をフィールド上では `S` と表示するが、本来は `P` と表示しなければならなかったとする。プログラムの開発者以外の人が、この表示が仕様と異なるバグを見つけて修正するまでの流れをみてみよう。

フィールドの状態の表示を行う処理を `show_field()` 関数にまとめなかった例では、まず、フィールドの状態を表示する処理を行っている箇所をプログラム全体から特定する必要がある。上記の例では、無限ループの外と中に一か所ずつ合計二か所、フィールドの状態の表示に関わる処理があるため、この二か所に対して修正を行う。

フィールドの状態の表示を行う処理を `show_field()` 関数にまとめた例では、 `show_field` という名称から、 `show_field()` 関数がフィールドの状態の表示に関連する処理が記述されている箇所だとあたりとつけることができる。そして、 `show_field()` 関数に記述されている処理の一か所のみに対して修正を行う。

今回の例では、まとめなかった例では二か所、まとめた例では一か所の修正を大きな違いはなかったが、同様の処理を関数にまとめなかった場合には、該当する処理を全て特定し、その全てに対して修正をかけなければいけない。リファクタリングを行い、同様の処理を一つの関数にまとめた場合には、関数にまとめた処理にバグがあれば、関数の中の処理だけ修正すればよいため、発見もしやすく修正を行いやすい。

以上から、リファクタリングを行うことでバグを見つけやすくなることがわかる。

最後に、開発の速度が上がるか確認する。

たとえば、開発を進める中で新たにフィールドの状態を表示する必要が生じたとする。 `show_field()` 関数にまとめなかった例では、フィールドの状態の表示を行う処理をコピー&ペースすることで対応できる。 `show_field()` 関数にまとめた例では `show_field()` と記述することで対応できる。

開発を進めるだけでは、作業量はほぼ同じで開発速度は変わらないように見える。しかし、まとめなかった例では、複数行にわたる処理が新しく追加されたことで、よりプログラムが理解しづらくなり、かつバグを発見しづらくなったのに対して、まとめた例では、 `show_field()` 関数を新たに呼び出しているだけなので、プログラムの理解のしづらさやバグの発見しづらさは修正前と後でほぼ変わらない。

実際のソフトウェア開発の現場では、例に挙げたプログラムよりもはるかに多い記述量のプログラムを扱う。さらに、仕様の変更があったり、追加の機能開発があったり、バグの修正があったりとすでに開発したプログラムに対して修正を行う機会も多い。また、ソフトウェア開発は複数人で開発することが多いため、自分が書いたプログラムを他の人が修正したり、自分も他の人が書いたプログラムを修正したりすることがある。さらには、自分か過去に書いたプログラムを修正することもある。他人が書いたプログラムを直したり、自身が過去に書いたプログラムを直したりする中で、理解しやすくバグも見つけやすいプログラムであれば、新しく理解したり直したりする時間が短くなるため、開発の速度が上がる。

以上から、リファクタリングを行うことで開発の速度が上がることがわかる。

追加のゲームの仕様

前章で開発したゲームでは、一種類の駒を複数個配置して動かしていくものであった。今回は、異なる動きをする複数種類の駒を複数個配置するような実装を行い、より複雑なゲームを開発する。

前章で実装したゲームに、以下の追加の仕様を満たすような実装を行う。

- 二種類の駒を用意する。一種類は `Normal Player` と呼び、前章までのプレイヤーの駒と同じように上下左右に一マスずつ移動する。もう一種類は `Jump Player` と呼び、上下左右にニマスずつ移動する。

- `Normal Player` のフィールド上での表示は `N` 、 `Jump Player` のフィールド上での表示は `J` である。
- ゲーム開始時に、 `Normal Player` を二つ、 `Jump Player` を一つ配置する。一つ目の `Normal Player` の座標の初期値は縦方向に `3` 、横方向に `3` である。二つ目の `Normal Player` の座標の初期値は縦方向に `4` 、横方向に `4` である。 `Jump Player` の座標の初期値は縦方向に `4` 、横方向に `2` である。フィールドの初期状態は以下となる。

```
*....
.*...
..*..
...N.
..J.N
```

- 駒は、一つ目の `Normal Player` 、二つ目の `Normal Player` 、 `Jump Player` の順で動かす。

前章までに開発したゲームを元に上記の仕様を満たすようなプログラムを記述していく。

前章までに開発したゲームは以下である。このプログラムでは、まず、同じ動きをするプレイヤーの駒を二つ、敵を三つ配置する。配置した駒を上下左右に動かして、駒が敵と同じ座標になればゲームクリアとなる。

```python
# プレイヤーの駒のクラス
class Player:
    def __init__(self, x, y):
        # 初期配置の座標
        self.x = x
        self.y = y

    # フィールド上での表示
    def __str__(self):
        return "S"

    # プレイヤーの駒の移動を行うメソッド
    def move(self, direction):
        if direction == "up":
            self.y -= 1
        elif direction == "down":
            self.y += 1
        elif direction == "left":
            self.x -= 1
        elif direction == "right":
            self.x += 1
        else:
            raise ValueError("無効な方向です")


# 敵のクラス
class Enemy:
    def __init__(self, x, y):
        # 初期配置の座標
        self.x = x
        self.y = y
        # 倒されたかどうかのフラグ
        self.defeated = False

    # フィールド上での表示
    def __str__(self):
        return "*" if not self.defeated else "."


# フィールドのクラス
class Field:
    def __init__(self, players, enemies):
        # フィールド上に配置するプレイヤーの駒と敵
        self.players = players
        self.enemies = enemies
        # フィールドの大きさ
        self.size = 5
```

```
    # 指定した座標のマスの文字を返すメソッド
    def get_cell(self, x, y):
        for player in self.players:
            if x == player.x and y == player.y:
                return str(player)
        for enemy in self.enemies:
            if x == enemy.x and y == enemy.y:
                return str(enemy)
        return "."

    # フィールドの状態を出力するメソッド
    def show(self):
        field = []
        for y in range(self.size):
            for x in range(self.size):
                cell = self.get_cell(x, y)
                field.append(cell)
            field.append("\n")
        print("".join(field))


# ゲームに必要なインスタンスの作成
players = [Player(3, 3), Player(4, 4)]
enemies = [Enemy(0, 0), Enemy(1, 1), Enemy(2, 2)]
field = Field(players, enemies)

# フィールドの初期状態の表示
field.show()

# 動かす駒を表す変数
turn = 0

while True:
    # 駒を移動する方向を入力する
    direction = input()
    # 入力された文字列が"end"の場合は、無限ループを終了する
    if direction == "end":
        print("ゲームを強制終了")
        break
    # 入力された方向に駒を移動する
    field.players[turn].move(direction)

    for enemy in field.enemies:
        # 駒が倒されていない敵と同一座標に到達したら、その敵を倒したことにする
        if field.players[turn].x == enemy.x and field.players[turn].y == enemy.y:
            enemy.defeated = True

    # フィールドの状態を表示
    field.show()

    # 全部の敵を倒したらクリア
    is_game_clear = True
    for enemy in field.enemies:
        if not enemy.defeated:
            is_game_clear = False
            break
    if is_game_clear:
        print("ゲームクリア！")
        break

    # 駒を交互に移動させる処理
    turn = (turn + 1) % len(players)
```

```
*....
 .*...
 ..*..
 ...S.
 ....S

```

```
up
*....
.*...
..*S.
.....
....S

up
*....
.*...
..*S.
....S
.....

left
*....
.*...
..S..
....S
.....

left
*....
.*...
..S..
...S.
.....

up
*....
.*S..
.....
...S.
.....

up
*....
.*S..
...S.
.....
.....

left
*....
.S...
...S.
.....
.....

left
*....
.S...
..S..
.....
.....

up
*S...
.....
..S..
.....
.....

up
*S...
..S..
.....
.....
.....

left
S....
```

```
..S..
.....
.....
.....

ゲームクリア！
```

追加仕様の実装

まずは仕様を満たすように、仕様通りに開発を進める。

仕様として `Normal Player` と `Jump Player` の二種類の駒が登場するため、これまでの `Player` クラスを削除して、新たに `NormalPlayer` クラスと `JumpPlayer` クラスを作成する。

`NormalPlayer` クラスと `JumpPlayer` クラスは、 `Player` クラスと同様のデータ属性とメソッドを持つ。違いとしては、 `__str__()` メソッドで返す値が、 `NormalPlayer` クラスでは `N` 、 `JumpPlayer` クラスでは `J` である。また、 `Jump Player` は上下左右にニマスずつ移動するため、 `JumpPlayer` クラスの `move()` メソッドで、データ属性 `x` と `y` に対して `2` ずつ加算/減算を行っている。

```
# Normal Playerのクラス
class NormalPlayer:
    def __init__(self, x, y):
        # 初期配置の座標
        self.x = x
        self.y = y

    # フィールド上での表示
    def __str__(self):
         return "N"

     # プレイヤーの駒の移動を行うメソッド
     def move(self, direction):
         if direction == "up":
             self.y -= 1
         elif direction == "down":
             self.y += 1
         elif direction == "left":
             self.x -= 1
         elif direction == "right":
             self.x += 1
         else:
             raise ValueError("無効な方向です")


# Jump Playerのクラス
class JumpPlayer:
    def __init__(self, x, y):
        # 初期配置の座標
        self.x = x
        self.y = y

    # フィールド上での表示
    def __str__(self):
        return "J"

    # プレイヤーの駒の移動を行うメソッド
    def move(self, direction):
        if direction == "up":
            self.y -= 2
        elif direction == "down":
            self.y += 2
        elif direction == "left":
            self.x -= 2
        elif direction == "right":
            self.x += 2
        else:
            raise ValueError("無効な方向です")
```

また、`NormalPlayer`クラスと`JumpPlayer`クラスのそれぞれに対してインスタンスを作成する必要があるため、プレイヤーの駒のインスタンス作成の処理を、

```
players = [Player(3, 3), Player(4, 4)]
```

から、

```
players = [NormalPlayer(3, 3), NormalPlayer(4, 4), JumpPlayer(2, 4)]
```

に変更する。

上記の修正を行ったプログラムが以下である。クラスを新しく定義した点と、プレイヤーの駒のインスタンス生成の処理以外に、修正箇所はない。

```
# Normal Playerのクラス
class NormalPlayer:
    def __init__(self, x, y):
        # 初期配置の座標
        self.x = x
        self.y = y

    # フィールド上での表示
    def __str__(self):
        return "N"

    # プレイヤーの駒の移動を行うメソッド
    def move(self, direction):
        if direction == "up":
            self.y -= 1
        elif direction == "down":
            self.y += 1
        elif direction == "left":
            self.x -= 1
        elif direction == "right":
            self.x += 1
        else:
            raise ValueError("無効な方向です")


# Jump Playerのクラス
class JumpPlayer:
    def __init__(self, x, y):
        # 初期配置の座標
        self.x = x
        self.y = y

    # フィールド上での表示
    def __str__(self):
        return "J"

    # プレイヤーの駒の移動を行うメソッド
    def move(self, direction):
        if direction == "up":
            self.y -= 2
        elif direction == "down":
            self.y += 2
        elif direction == "left":
            self.x -= 2
        elif direction == "right":
            self.x += 2
        else:
            raise ValueError("無効な方向です")


```

```python
# 敵のクラス
class Enemy:
    def __init__(self, x, y):
        # 初期配置の座標
        self.x = x
        self.y = y
        # 倒されたかどうかのフラグ
        self.defeated = False

    # フィールド上での表示
    def __str__(self):
        return "*" if not self.defeated else "."


# フィールドのクラス
class Field:
    def __init__(self, players, enemies):
        # フィールド上に配置するプレイヤーの駒と敵
        self.players = players
        self.enemies = enemies
        # フィールドの大きさ
        self.size = 5

    # 指定した座標のマスの文字を返すメソッド
    def get_cell(self, x, y):
        for player in self.players:
            if x == player.x and y == player.y:
                return str(player)
        for enemy in self.enemies:
            if x == enemy.x and y == enemy.y:
                return str(enemy)
        return "."

    # フィールドの状態を出力するメソッド
    def show(self):
        field = []
        for y in range(self.size):
            for x in range(self.size):
                cell = self.get_cell(x, y)
                field.append(cell)
            field.append("\n")
        print("".join(field))


# ゲームに必要なインスタンスの作成
players = [NormalPlayer(3, 3), NormalPlayer(4, 4), JumpPlayer(2, 4)]
enemies = [Enemy(0, 0), Enemy(1, 1), Enemy(2, 2)]
field = Field(players, enemies)

# フィールドの初期状態の表示
field.show()

# 動かす駒を表す変数
turn = 0

while True:
    # 駒を移動する方向を入力する
    direction = input()
    # 入力された文字列が"end"の場合は、無限ループを終了する
    if direction == "end":
        print("ゲームを強制終了")
        break
    # 入力された方向に駒を移動する
    field.players[turn].move(direction)

    for enemy in field.enemies:
        # 駒が倒されていない敵と同一座標に到達したら、その敵を倒したことにする
        if field.players[turn].x == enemy.x and field.players[turn].y == enemy.y:
            enemy.defeated = True

    # フィールドの状態を表示
    field.show()
```

```

    # 全部の敵を倒したらクリア
    is_game_clear = True
    for enemy in field.enemies:
        if not enemy.defeated:
            is_game_clear = False
            break
    if is_game_clear:
        print("ゲームクリア！")
        break

    # 駒を交互に移動させる処理
    turn = (turn + 1) % len(players)
```

```
*....
.*...
..*..
...N.
..J.N

up
*....
.*...
..*N.
.....
..J.N

up
*....
.*...
..*N.
....N
..J..

up
*....
.*...
..JN.
....N
.....

up
*....
.*.N.
..J..
....N
.....

up
*....
.*.N.
..J.N
.....
.....

up
*.J..
.*.N.
....N
.....
.....

left
*.J..
.*N..
....N
.....
.....

left
```

```
*.J..
.*N..
...N.
.....
.....

left
J....
.*N..
...N.
.....
.....

left
J....
.N...
...N.
.....
.....

ゲームクリア！
```

追加仕様の実装へのリファクタリング1

`NormalPlayer` クラスと `JumpPlayer` クラスを定義すると、この二つのクラスは、 `__str__()` メソッドの戻り値と `move()` メソッドでデータ属性 `x` と `y` に対して加算/減算する値以外は全て同じ構造になっていることがわかる。

概念としても `Normal Player` と `Jump Player` は、フィールド上での表示と動き方が異なるだけで、ともにプレイヤーの駒である。

現在、 `NormalPlayer` クラスと `JumpPlayer` クラスが別々で定義されているため、プログラム上からは `NormalPlayer` クラスと `JumpPlayer` クラスが同様にプレイヤーの駒であるということがわかりづらい。

プレイヤーの駒の概念として `Player` クラスを作り、 `NormalPlayer` クラスと `JumpPlayer` クラスに継承する形で前節のプログラムをリファクタリングする。

プレイヤーの駒は、フィールド上での座標、フィールド上での表示文字を共通して持つ。また、駒は移動するためのメソッドも共通して持つと考えられる。よって、 `Player` クラスは以下のように定義できる。

`__init__()` メソッドで、座標を表すデータ属性 `x` と `y` の初期化を行う。フィールド上での表示を返すメソッドとして、 `__str__()` メソッドを用意する。移動を行うメソッドとして、 `move()` メソッドを用意する。 `__str__()` メソッドと `move()` メソッドは駒の種類によって変わるため、 `Player` クラスでは処理を記述しない。

```
class Player:
    def __init__(self, x, y):
        # 初期配置の座標
        self.x = x
        self.y = y

    # フィールド上での表示
    def __str__(self):
        # 具体的な処理は子クラスで定義
         pass

     # プレイヤーの駒の移動を行うメソッド
     def move(self, direction):
         # 具体的な処理は子クラスで定義
         pass
```

`Player` クラスを継承する形で、 `NormalPlayer` クラスと `JumpPlayer` クラスを作成すると以下のようにプログラムを修正できる。

`Player` クラスで `__init__()` メソッドを定義しており、子クラスで処理を変えたり追加する必要がないため、 `NormalPlayer` クラスと `JumpPlayer` クラスには `__init__()` メソッドは記述しない。 `__str__()` メソッドと `move()` メソッドは、子クラス側（ `NormalPlayer` クラスと `JumpPlayer` クラス）でオーバーライドして、具体的な処理を記述している。

```python
# Playerのクラス（プレイヤーの駒の親クラス）
class Player:
    def __init__(self, x, y):
        # 初期配置の座標
        self.x = x
        self.y = y

    # フィールド上での表示
    def __str__(self):
        # 具体的な処理は子クラスで定義
        pass

    # プレイヤーの駒の移動を行うメソッド
    def move(self, direction):
        # 具体的な処理は子クラスで定義
        pass


# Normal Playerのクラス
class NormalPlayer(Player):
    # フィールド上での表示
    def __str__(self):
        return "N"

    # プレイヤーの駒の移動を行うメソッド
    def move(self, direction):
        if direction == "up":
            self.y -= 1
        elif direction == "down":
            self.y += 1
        elif direction == "left":
            self.x -= 1
        elif direction == "right":
            self.x += 1
        else:
            raise ValueError("無効な方向です")


# Jump Playerのクラス
class JumpPlayer(Player):
    # フィールド上での表示
    def __str__(self):
        return "J"

    # プレイヤーの駒の移動を行うメソッド
    def move(self, direction):
        if direction == "up":
            self.y -= 2
        elif direction == "down":
            self.y += 2
        elif direction == "left":
            self.x -= 2
        elif direction == "right":
            self.x += 2
        else:
            raise ValueError("無効な方向です")


# 敵のクラス
class Enemy:
    def __init__(self, x, y):
        # 初期配置の座標
        self.x = x
        self.y = y
        # 倒されたかどうかのフラグ
        self.defeated = False

    # フィールド上での表示
    def __str__(self):
        return "*" if not self.defeated else "."

```

```python

# フィールドのクラス
class Field:
    def __init__(self, players, enemies):
        # フィールド上に配置するプレイヤーの駒と敵
        self.players = players
        self.enemies = enemies
        # フィールドの大きさ
        self.size = 5

    # 指定した座標のマスの文字を返すメソッド
    def get_cell(self, x, y):
        for player in self.players:
            if x == player.x and y == player.y:
                return str(player)
        for enemy in self.enemies:
            if x == enemy.x and y == enemy.y:
                return str(enemy)
        return "."

    # フィールドの状態を出力するメソッド
    def show(self):
        field = []
        for y in range(self.size):
            for x in range(self.size):
                cell = self.get_cell(x, y)
                field.append(cell)
            field.append("\n")
        print("".join(field))


# ゲームに必要なインスタンスの作成
players = [NormalPlayer(3, 3), NormalPlayer(4, 4), JumpPlayer(2, 4)]
enemies = [Enemy(0, 0), Enemy(1, 1), Enemy(2, 2)]
field = Field(players, enemies)

# フィールドの初期状態の表示
field.show()

# 動かす駒を表す変数
turn = 0

while True:
    # 駒を移動する方向を入力する
    direction = input()
    # 入力された文字列が"end"の場合は、無限ループを終了する
    if direction == "end":
        print("ゲームを強制終了")
        break
    # 入力された方向に駒を移動する
    field.players[turn].move(direction)

    for enemy in field.enemies:
        # 駒が倒されていない敵と同一座標に到達したら、その敵を倒したことにする
        if field.players[turn].x == enemy.x and field.players[turn].y == enemy.y:
            enemy.defeated = True

    # フィールドの状態を表示
    field.show()

    # 全部の敵を倒したらクリア
    is_game_clear = True
    for enemy in field.enemies:
        if not enemy.defeated:
            is_game_clear = False
            break
    if is_game_clear:
        print("ゲームクリア！")
        break

    # 駒を交互に移動させる処理
    turn = (turn + 1) % len(players)
```

```
*....
.*...
..*..
...N.
..J.N

up
*....
.*...
 ..*N.
 .....
 ..J.N

 up
 *....
 .*...
 ..*N.
 ....N
 ..J..

 up
 *....
 .*...
 ..JN.
 ....N
 .....

 up
 *....
 .*.N.
 ..J..
 ....N
 .....

 up
 *....
 .*.N.
 ..J.N
 .....
 .....

 up
 *.J..
 .*.N.
 ....N
 .....
 .....

 left
 *.J..
 .*N..
 ....N
 .....
 .....

 left
 *.J..
 .*N..
 ...N.
 .....
 .....

 left
 J....
 .*N..
 ...N.
 .....
 .....

 left
 J....
```

```
.N...
...N.
.....
.....

ゲームクリア！
```

`Player` クラスを継承するように `NormalPlayer` クラスと `JumpPlayer` クラスを作ることで、 `NormalPlayer` クラスと `JumpPlayer` クラスがプレイヤーの駒という共通の概念の上に成り立つクラスであるとプログラム上からもわかるようになった。

`Player` クラスを継承するように `NormalPlayer` クラスと `JumpPlayer` クラスを作ったことで、さらにリファクタリングできる箇所はないかみてみよう。

`move()` メソッドに着目すると、 `NormalPlayer` クラスと `JumpPlayer` クラスでは、加算/減算する値が異なるだけで、他の処理は全く同じであることに気づく。そのため、 `move()` メソッドを `Player` クラスで持ち、加算/減算する値（移動する値）だけ、子クラスである `NormalPlayer` クラスと `JumpPlayer` クラスで定義できないか考えてみる。

移動する値は、データ属性として持つ方法やメソッドとして持つ方法が考えられる。今回は、移動する値はメソッドとして持ちリファクタリングを進める。

改めて、 `Player` クラスを以下のように定義する。 `get_move_unit()` メソッドで、移動する値を定義する。デフォルトとして、 `get_move_unit()` メソッドでは `1` を返すように定義する。 `move()` メソッドでは、駒の移動する処理を記述し、移動する値は `get_move_unit()` メソッドから呼び出すように記述する。

```python
class Player:
    def __init__(self, x, y):
        # 初期配置の座標
        self.x = x
        self.y = y

    # フィールド上での表示
    def __str__(self):
        # 具体的な処理は子クラスで定義
         pass

     # プレイヤーの駒の移動を行うメソッド
     def move(self, direction):
         if direction == "up":
             self.y -= self.get_move_unit()
         elif direction == "down":
             self.y += self.get_move_unit()
         elif direction == "left":
             self.x -= self.get_move_unit()
         elif direction == "right":
             self.x += self.get_move_unit()
         else:
             raise ValueError("無効な方向です")

     # プレイヤーの駒の移動距離を取得するメソッド
     def get_move_unit(self):
         # プレイヤーの駒のデフォルトの移動距離を1と設定
         return 1
```

`Player` クラスの定義を変更したことで、 `NormalPlayer` クラスと `JumpPlayer` クラスの定義を修正したプログラムが以下である。

`NormalPlayer` クラスは、 `Player` クラスの `__init__()` メソッド、 `move()` メソッド、 `get_move_unit()` をそのまま使うため、 `__str__()` メソッドのみオーバーライドする形で定義する。

`JumpPlayer` クラスは、 `Player` クラスの `__init__()` メソッドと `move()` メソッドをそのまま使うため、 `__str__()` メソッドと `get_move_unit()` メソッドをオーバーライドする形で定義する。

```
# Playerのクラス（プレイヤーの駒の親クラス）
```

```python
class Player:
    def __init__(self, x, y):
        # 初期配置の座標
        self.x = x
        self.y = y

    # フィールド上での表示
    def __str__(self):
        # 具体的な処理は子クラスで定義
        pass

    # プレイヤーの駒の移動を行うメソッド
    def move(self, direction):
        if direction == "up":
            self.y -= self.get_move_unit()
        elif direction == "down":
            self.y += self.get_move_unit()
        elif direction == "left":
            self.x -= self.get_move_unit()
        elif direction == "right":
            self.x += self.get_move_unit()
        else:
            raise ValueError("無効な方向です")

    # プレイヤーの駒の移動距離を取得するメソッド
    def get_move_unit(self):
        # プレイヤーの駒のデフォルトの移動距離を1と設定
        return 1


# Normal Playerのクラス
class NormalPlayer(Player):
    # フィールド上での表示
    def __str__(self):
        return "N"


# Jump Playerのクラス
class JumpPlayer(Player):
    # フィールド上での表示
    def __str__(self):
        return "J"

    # プレイヤーの駒の移動距離を取得するメソッド
    def get_move_unit(self):
        return 2


# 敵のクラス
class Enemy:
    def __init__(self, x, y):
        # 初期配置の座標
        self.x = x
        self.y = y
        # 倒されたかどうかのフラグ
        self.defeated = False

    # フィールド上での表示
    def __str__(self):
        return "*" if not self.defeated else "."


# フィールドのクラス
class Field:
    def __init__(self, players, enemies):
        # フィールド上に配置するプレイヤーの駒と敵
        self.players = players
        self.enemies = enemies
        # フィールドの大きさ
        self.size = 5

    # 指定した座標のマスの文字を返すメソッド
```

```
    def get_cell(self, x, y):
        for player in self.players:
            if x == player.x and y == player.y:
                return str(player)
        for enemy in self.enemies:
            if x == enemy.x and y == enemy.y:
                return str(enemy)
        return "."

    # フィールドの状態を出力するメソッド
    def show(self):
        field = []
        for y in range(self.size):
            for x in range(self.size):
                cell = self.get_cell(x, y)
                field.append(cell)
            field.append("\n")
        print("".join(field))


# ゲームに必要なインスタンスの作成
players = [NormalPlayer(3, 3), NormalPlayer(4, 4), JumpPlayer(2, 4)]
enemies = [Enemy(0, 0), Enemy(1, 1), Enemy(2, 2)]
field = Field(players, enemies)

# フィールドの初期状態の表示
field.show()

# 動かす駒を表す変数
turn = 0

while True:
    # 駒を移動する方向を入力する
    direction = input()
    # 入力された文字列が"end"の場合は、無限ループを終了する
    if direction == "end":
        print("ゲームを強制終了")
        break
    # 入力された方向に駒を移動する
    field.players[turn].move(direction)

    for enemy in field.enemies:
        # 駒が倒されていない敵と同一座標に到達したら、その敵を倒したことにする
        if field.players[turn].x == enemy.x and field.players[turn].y == enemy.y:
            enemy.defeated = True

    # フィールドの状態を表示
    field.show()

    # 全部の敵を倒したらクリア
    is_game_clear = True
    for enemy in field.enemies:
        if not enemy.defeated:
            is_game_clear = False
            break
    if is_game_clear:
        print("ゲームクリア！")
        break

    # 駒を交互に移動させる処理
    turn = (turn + 1) % len(players)
```

```
*....
.*...
..*..
...N.
..J.N

up
*....
```

```
.*...
..*N.
.....
..J.N

up
*....
.*...
..*N.
....N
..J..

up
*....
.*...
..JN.
....N
.....

up
*....
.*.N.
..J..
....N
.....

up
*....
.*.N.
..J.N
.....
.....

up
*.J..
.*.N.
....N
.....
.....

left
*.J..
.*N..
....N
.....
.....

left
*.J..
.*N..
...N.
.....
.....

left
J....
.*N..
...N.
.....
.....

left
J....
.N...
...N.
.....
.....

ゲームクリア！
```

上記のようなリファクタリングを行うことで、`NormalPlayer` クラスと `JumpPlayer` クラスは、フィールド上での表示と移動する値のみ

を定義すればよくなった。よって、 `NormalPlayer` クラスと `JumpPlayer` クラスを理解するには、まず `Player` クラスを見れば、共通して持つデータ属性やメソッドを理解でき、フィールド上での表示と移動する値については `NormalPlayer` クラスと `JumpPlayer` クラスを見ることでわかるため、より簡単にプログラムを理解できるようになった。

追加仕様の実装へのリファクタリング2

ここまでに修正したプログラムをさらにリファクタリングできないか考えてみよう。

これまでプレイヤーの駒と敵を別の概念として扱ってきたが、ともにフィールド上に存在するオブジェクトであると気付く。

実際に、 `Player` クラスと `Enemy` クラスは共通して、フィールド上での座標と表示文字（ `__str__()` メソッド）を持っている。

本節では、フィールド上のオブジェクトを表すクラスとして `Object` クラスを作り、 `Object` クラスを継承する形で、プレイヤーの駒と敵のクラスを再定義していく。

まず、 `Object` クラスを定義する。プレイヤーの駒と敵は共通して、フィールド上での座標と表示文字（ `__str__()` メソッド）を持っているため、 `Object` クラスは以下のように定義できると考えられる。

```
class Object:
    def __init__(self, x, y):
        # 初期配置の座標
        self.x = x
        self.y = y

    # フィールド上での表示
    def __str__(self):
        # 具体的な処理は子クラスで定義
         pass
```

`__init__()` メソッドで座標を示すデータ属性 `x` と `y` を初期化する。 `__str__()` メソッドはオブジェクトによって変わるため、 `Object` クラスでは処理を記述しない。

`Object` クラスを定義したことで `Player` クラスと `Enemy` を修正したプログラムが以下である。

`Player` クラスに定義されていた `__init__()` メソッドと `__str__()` メソッドは、親クラスである `Object` クラスが持っているため、削除した。

`Enemy` クラスでは、データ属性 `defeated` をインスタンス生成時に座標とともに定義するため、 `__init__()` メソッドをオーバーライドする。座標の初期化を行う処理は、親クラスである `Object` クラスの `__init__()` メソッドの処理をそのまま使うため、 `super()` を使い、親クラスである `Object` クラスの `__init__()` メソッドを呼び出す。その後、データ属性 `defeated` の定義を行っている。
`__str__()` メソッドもオーバーライドする形で処理を定義している。

```
# Objectのクラス
class Object:
    def __init__(self, x, y):
        # 初期配置の座標
        self.x = x
        self.y = y

    # フィールド上での表示
    def __str__(self):
        # 具体的な処理は子クラスで定義
        pass


# Playerのクラス
class Player(Object):
    # プレイヤーの駒の移動を行うメソッド
    def move(self, direction):
        if direction == "up":
            self.y -= self.get_move_unit()
        elif direction == "down":
```

```
            self.y += self.get_move_unit()
        elif direction == "left":
            self.x -= self.get_move_unit()
        elif direction == "right":
            self.x += self.get_move_unit()
        else:
            raise ValueError("無効な方向です")

    # プレイヤーの駒の移動距離を取得するメソッド
    def get_move_unit(self):
        # プレイヤーの駒のデフォルトの移動距離を1と設定
        return 1


# Normal Playerのクラス
class NormalPlayer(Player):
    # フィールド上での表示
    def __str__(self):
        return "N"


# Jump Playerのクラス
class JumpPlayer(Player):
    # フィールド上での表示
    def __str__(self):
        return "J"

    # プレイヤーの駒の移動距離を取得するメソッド
    def get_move_unit(self):
        return 2


# 敵のクラス
class Enemy(Object):
    def __init__(self, x, y):
        super().__init__(x, y)
        # 倒されたかどうかのフラグ
        self.defeated = False

    # フィールド上での表示
    def __str__(self):
        return "*" if not self.defeated else "."


# フィールドのクラス
class Field:
    def __init__(self, players, enemies):
        # フィールド上に配置するプレイヤーの駒と敵
        self.players = players
        self.enemies = enemies
        # フィールドの大きさ
        self.size = 5

    # 指定した座標のマスの文字を返すメソッド
    def get_cell(self, x, y):
        for player in self.players:
            if x == player.x and y == player.y:
                return str(player)
        for enemy in self.enemies:
            if x == enemy.x and y == enemy.y:
                return str(enemy)
        return "."

    # フィールドの状態を出力するメソッド
    def show(self):
        field = []
        for y in range(self.size):
            for x in range(self.size):
                cell = self.get_cell(x, y)
                field.append(cell)
            field.append("\n")
        print("".join(field))
```

```python

# ゲームに必要なインスタンスの作成
players = [NormalPlayer(3, 3), NormalPlayer(4, 4), JumpPlayer(2, 4)]
enemies = [Enemy(0, 0), Enemy(1, 1), Enemy(2, 2)]
field = Field(players, enemies)

# フィールドの初期状態の表示
field.show()

# 動かす駒を表す変数
turn = 0

while True:
    # 駒を移動する方向を入力する
    direction = input()
    # 入力された文字列が"end"の場合は、無限ループを終了する
    if direction == "end":
        print("ゲームを強制終了")
        break
    # 入力された方向に駒を移動する
    field.players[turn].move(direction)

    for enemy in field.enemies:
        # 駒が倒されていない敵と同一座標に到達したら、その敵を倒したことにする
        if field.players[turn].x == enemy.x and field.players[turn].y == enemy.y:
            enemy.defeated = True

    # フィールドの状態を表示
    field.show()

    # 全部の敵を倒したらクリア
    is_game_clear = True
    for enemy in field.enemies:
        if not enemy.defeated:
            is_game_clear = False
            break
    if is_game_clear:
        print("ゲームクリア！")
        break

    # 駒を交互に移動させる処理
    turn = (turn + 1) % len(players)
```

```
*....
.*...
..*..
...N.
..J.N

up
*....
.*...
..*N.
.....
..J.N

up
*....
.*...
..*N.
....N
..J..

up
*....
.*...
..JN.
....N
.....
```

```
27
28 up
29 *....
30 .*.N.
31 ..J..
32 ....N
33 .....
34
35 up
36 *....
37 .*.N.
38 ..J.N
39 .....
40 .....
41
42 up
43 *.J..
44 .*.N.
45 ....N
46 .....
47 .....
48
49 left
50 *.J..
51 .*N..
52 ....N
53 .....
54 .....
55
56 left
57 *.J..
58 .*N..
59 ...N.
60 .....
61 .....
62
63 left
64 J....
65 .*N..
66 ...N.
67 .....
68 .....
69
70 left
71 J....
72 .N...
73 ...N.
74 .....
75 .....
76
77 ゲームクリア！
```

さらにリファクタリングできる箇所はないか探してみよう。

プレイヤーの駒と敵は共通して、`Object`クラスを親に持つことから、プレイヤーの駒と敵はともにフィールド上ではオブジェクトという概念であることがわかる。そのため、以下の`Field`クラスで定義されている`get_cell()`メソッドでは、プレイヤーの駒と敵で分けて二回`for`文による処理を行っているが、プレイヤーの駒と敵がともにオブジェクトであれば、一回の`for`文による処理にまとめることはできないか考えてみる。

```
1 def get_cell(self, x, y):
2     for player in self.players:
3         if x == player.x and y == player.y:
4             return str(player)
5     for enemy in self.enemies:
6         if x == enemy.x and y == enemy.y:
7             return str(enemy)
8     return "."
```

`get_cell()` メソッドを一回の `for` 文の処理としてまとめると以下のようになる。 `self.players + self.enemies` の部分で、フィールドのインスタンスが持つ、プレイヤーの駒のインスタンスのリストと敵のインスタンスのリストを繋がている。 `for` 文を使い、繋げたリストを一つずつ取り出し指定した座標にあるか確認する。指定した座標に、プレイヤーの駒ないしは敵が存在すれば、そのインスタンスの `__str__()` メソッドを実行し、返ってきた文字を `get_cell()` メソッドは戻り値とする。

```
def get_cell(self, x, y):
    for obj in self.players + self.enemies:
        if x == obj.x and y == obj.y:
            return str(obj)
    return "."
```

上記の修正を行ったプログラムが以下である。

```
# Objectのクラス
class Object:
    def __init__(self, x, y):
        # 初期配置の座標
        self.x = x
        self.y = y

    # フィールド上での表示
    def __str__(self):
        # 具体的な処理は子クラスで定義
        pass


# Playerのクラス
class Player(Object):
    # プレイヤーの駒の移動を行うメソッド
    def move(self, direction):
        if direction == "up":
            self.y -= self.get_move_unit()
        elif direction == "down":
            self.y += self.get_move_unit()
        elif direction == "left":
            self.x -= self.get_move_unit()
        elif direction == "right":
            self.x += self.get_move_unit()
        else:
            raise ValueError("無効な方向です")

    # プレイヤーの駒の移動距離を取得するメソッド
    def get_move_unit(self):
        # プレイヤーの駒のデフォルトの移動距離を1と設定
        return 1


# Normal Playerのクラス
class NormalPlayer(Player):
    # フィールド上での表示
    def __str__(self):
        return "N"


# Jump Playerのクラス
class JumpPlayer(Player):
    # フィールド上での表示
    def __str__(self):
        return "J"

    # プレイヤーの駒の移動距離を取得するメソッド
    def get_move_unit(self):
        return 2


# 敵のクラス
class Enemy(Object):
```

```
    def __init__(self, x, y):
        super().__init__(x, y)
        # 倒されたかどうかのフラグ
        self.defeated = False

    # フィールド上での表示
    def __str__(self):
        return "*" if not self.defeated else "."


# フィールドのクラス
class Field:
    def __init__(self, players, enemies):
        # フィールド上に配置するプレイヤーの駒と敵
        self.players = players
        self.enemies = enemies
        # フィールドの大きさ
        self.size = 5

    # 指定した座標のマスの文字を返すメソッド
    def get_cell(self, x, y):
        for obj in self.players + self.enemies:
            if x == obj.x and y == obj.y:
                return str(obj)
        return "."

    # フィールドの状態を出力するメソッド
    def show(self):
        field = []
        for y in range(self.size):
            for x in range(self.size):
                cell = self.get_cell(x, y)
                field.append(cell)
            field.append("\n")
        print("".join(field))


# ゲームに必要なインスタンスの作成
players = [NormalPlayer(3, 3), NormalPlayer(4, 4), JumpPlayer(2, 4)]
enemies = [Enemy(0, 0), Enemy(1, 1), Enemy(2, 2)]
field = Field(players, enemies)

# フィールドの初期状態の表示
field.show()

# 動かす駒を表す変数
turn = 0

while True:
    # 駒を移動する方向を入力する
    direction = input()
    # 入力された文字列が"end"の場合は、無限ループを終了する
    if direction == "end":
        print("ゲームを強制終了")
        break
    # 入力された方向に駒を移動する
    field.players[turn].move(direction)

    for enemy in field.enemies:
        # 駒が倒されていない敵と同一座標に到達したら、その敵を倒したことにする
        if field.players[turn].x == enemy.x and field.players[turn].y == enemy.y:
            enemy.defeated = True

    # フィールドの状態を表示
    field.show()

    # 全部の敵を倒したらクリア
    is_game_clear = True
    for enemy in field.enemies:
        if not enemy.defeated:
            is_game_clear = False
            break
```

```
    if is_game_clear:
        print("ゲームクリア！")
        break

    # 駒を交互に移動させる処理
    turn = (turn + 1) % len(players)
```

```
*....
.*...
..*..
...N.
..J.N

up
*....
.*...
..*N.
.....
..J.N

up
*....
.*...
..*N.
....N
..J..

up
*....
.*...
..JN.
....N
.....

left
*....
.*...
..N..
....N
.....

left
*....
.*...
..N..
...N.
.....

up
*.J..
.*...
..N..
...N.
.....

up
*.J..
.*N..
.....
...N.
.....

up
*.J..
.*N..
...N.
.....
.....

left
```

```
J....
.*N..
...N.
.....
.....

left
J....
.N...
...N.
.....
.....

ゲームクリア！
```

リスト内包表記

前節までのリファクタリングで、継承を使った修正はできる限り実施した。さらにリファクタリングを進める上で、リスト内包表記について理解しないと修正できない箇所があるため、リスト内包表記について解説する。

リスト内包表記とは、既存のリストから新しいリストを生成するための表記法である。

リスト内包表記の構文は以下である。イテラブルには、リスト・タプル・集合・辞書などが該当する。

```
[<式> for <任意の変数名> in <イテラブル>]
```

たとえば、1から10までの整数の2乗のリストをリスト内包表記を使うと以下のように記述できる。

```
li = [i**2 for i in range(1, 11)]
print(li)
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

上記のサンプルプログラムでは、`range(1, 11)`というイテラブルから、`i`という変数に順番に要素が代入される。そして、`i**2`という式が評価され、その結果からなるリストが生成される。

同様の処理を、リスト内包表記を使わないで記述すると以下のようになる。

```
li = []
for i in range(1, 11):
    li.append(i**2)
print(li)
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

このようにリスト内包表記を使うと、簡潔にリストを生成できる。

また、リスト内包表記には以下のように条件式も追加できる。

```
[<式> for <任意の変数名> in <イテラブル> if <条件式>]
```

たとえば、1から10までの整数のうち、3の倍数のみを2乗した値のみで構成されるリストをリスト内包表記を使って生成するには、以下のよう

に記述できる。

```
li = [i**2 for i in range(1, 11) if i % 3 == 0]
print(li)
```

```
[9, 36, 81]
```

上記のサンプルプログラムでは、`i % 3 == 0` という条件式を使用して、3の倍数を判定している。3の倍数の場合にだけ、`i**2` という式が評価され、その結果がリストに追加されている。

条件式を追加することで、より細かく特定の条件を満たす要素だけを選択してリストを生成することができる。

追加仕様の実装へのリファクタリング3

最後に、無限ループ内でゲームの終了判定を行っている以下の処理をリファクタリングできないか考えてみる。

```
# 全部の敵を倒したらクリア
is_game_clear = True
for enemy in field.enemies:
    if not enemy.defeated:
        is_game_clear = False
        break
if is_game_clear:
    print("ゲームクリア！")
    break
```

上記の処理は、ゲームの終了判定をするために、終了判定用の変数 `is_game_clear` を定義し、`for` 文を使い、一つでも倒されていない敵がいれば `is_game_clear` に格納する値を変更し、`is_game_clear` の値が定義されたときから変わっていなければゲームの終了判定する、といったようにプログラムを丁寧に読み解かなければ理解できないような処理を行っている。

もし、`Enemy` クラスのインスタンスが持つデータ属性 `defeated` の値を全てリストで持つことができれば、`all()` 関数を使って `if` 文のみでゲームの終了を判定できる。

リスト内包表記を使うと、以下のように記述することで、`Enemy` クラスのインスタンスのデータ属性 `defeated` の全ての値を持ったリストを生成することができる。

以下のリスト内包表記では、`Field` クラスのインスタンスが持つ `Enemy` クラスのインスタンスを一つずつ取り出し、取り出した `Enemy` クラスのインスタンスのデータ属性 `defeated` をリストに追加していく処理を行っている。

```
[enemy.defeated for enemy in field.enemies]
```

リスト内包表記を使うことで、以下のようにゲームの終了判定処理を記述できる。

```
# 全部の敵を倒したらクリア
if all([enemy.defeated for enemy in field.enemies]):
    print("ゲームクリア！")
    break
```

敵が全て倒されていたらゲームが終了するといった処理が、イメージのままの形で記述されており、わかりやすくなった。

上記の修正を行ったプログラムが以下である。

```
# Objectのクラス
```

```
class Object:
    def __init__(self, x, y):
        # 初期配置の座標
        self.x = x
        self.y = y

    # フィールド上での表示
    def __str__(self):
        # 具体的な処理は子クラスで定義
        pass


# Playerのクラス
class Player(Object):
    # プレイヤーの駒の移動を行うメソッド
    def move(self, direction):
        if direction == "up":
            self.y -= self.get_move_unit()
        elif direction == "down":
            self.y += self.get_move_unit()
        elif direction == "left":
            self.x -= self.get_move_unit()
        elif direction == "right":
            self.x += self.get_move_unit()
        else:
            raise ValueError("無効な方向です")

    # プレイヤーの駒の移動距離を取得するメソッド
    def get_move_unit(self):
        # プレイヤーの駒のデフォルトの移動距離を1と設定
        return 1


# Normal Playerのクラス
class NormalPlayer(Player):
    # フィールド上での表示
    def __str__(self):
        return "N"


# Jump Playerのクラス
class JumpPlayer(Player):
    # フィールド上での表示
    def __str__(self):
        return "J"

    # プレイヤーの駒の移動距離を取得するメソッド
    def get_move_unit(self):
        return 2


# 敵のクラス
class Enemy(Object):
    def __init__(self, x, y):
        super().__init__(x, y)
        # 倒されたかどうかのフラグ
        self.defeated = False

    # フィールド上での表示
    def __str__(self):
        return "*" if not self.defeated else "."


# フィールドのクラス
class Field:
    def __init__(self, players, enemies):
        # フィールド上に配置するプレイヤーの駒と敵
        self.players = players
        self.enemies = enemies
        # フィールドの大きさ
        self.size = 5

```

```
    # 指定した座標のマスの文字を返すメソッド
    def get_cell(self, x, y):
        for obj in self.players + self.enemies:
            if x == obj.x and y == obj.y:
                return str(obj)
        return "."

    # フィールドの状態を出力するメソッド
    def show(self):
        field = []
        for y in range(self.size):
            for x in range(self.size):
                cell = self.get_cell(x, y)
                field.append(cell)
            field.append("\n")
        print("".join(field))


# ゲームに必要なインスタンスの作成
players = [NormalPlayer(3, 3), NormalPlayer(4, 4), JumpPlayer(2, 4)]
enemies = [Enemy(0, 0), Enemy(1, 1), Enemy(2, 2)]
field = Field(players, enemies)

# フィールドの初期状態の表示
field.show()

# 動かす駒を表す変数
turn = 0

while True:
    # 駒を移動する方向を入力する
    direction = input()
    # 入力された文字列が"end"の場合は、無限ループを終了する
    if direction == "end":
        print("ゲームを強制終了")
        break
    # 入力された方向に駒を移動する
    field.players[turn].move(direction)

    for enemy in field.enemies:
        # 駒が倒されていない敵と同一座標に到達したら、その敵を倒したことにする
        if field.players[turn].x == enemy.x and field.players[turn].y == enemy.y:
            enemy.defeated = True

    # フィールドの状態を表示
    field.show()

    # 全部の敵を倒したらクリア
    if all([enemy.defeated for enemy in field.enemies]):
        print("ゲームクリア！")
        break

    # 駒を交互に移動させる処理
    turn = (turn + 1) % len(players)
```

```
*....
.*...
..*..
...N.
..J.N

up
*....
.*...
..*N.
.....
..J.N

up
*....
```

```
.*...
..*N.
....N
..J..

up
*....
.*...
..JN.
....N
.....

up
*....
.*.N.
..J..
....N
.....

up
*....
.*.N.
..J.N
.....
.....

up
*.J..
.*.N.
....N
.....
.....

left
*.J..
.*N..
....N
.....
.....

left
*.J..
.*N..
...N.
.....
.....

left
J....
.*N..
...N.
.....
.....

left
J....
.N...
...N.
.....
.....

ゲームクリア！
```

修正前のプログラムと修正後のプログラムの比較

本章の最後に、リファクタリングを行ったことで、理解しやすくなり、重複する処理を書かずにプログラムを修正しやすくなったか、継承を使わずに `NormalPlayer` クラスと `JumpPlayer` クラスを定義したプログラムと最終版のプログラムに対して、新しいオブジェクトを追加する仕様を与えた場合の修正方法や修正量を比較してみよう。

以下が追加の仕様である。

- プレイヤーの駒として、新しく上下左右に3マスずつ動くことができる `Super Jump Player` を用意する。
- `Super Jump Player` のフィールド上での表示は `S` である。
- `Super Jump Player` の座標の初期値は縦方向に `1` 、横方向に `4` である。
- `Super Jump Player` は `Jump Player` のあとに動かす。
- 背景オブジェクトとして `Background` を用意する。背景オブジェクトとは何もないフィールド上の表示を `.` から変更するだけのオブジェクトで見た目が何もないマスと異なる以外、何も効果はない。
- `Background` のフィールド上での表示は `+` である。
- `Background` のは縦方向に `3` 、横方向に `1` の座標と、縦方向に `3` 、横方向に `2` の座標に配置する。フィールドの初期状態は以下となる。

```
*....
.*..S
..*..
.++N.
..J.N
```

以上の仕様と満たすプログラムにするには、どのような修正が必要かみていこう。

まず、継承を使わずに `NormalPlayer` クラスと `JumpPlayer` クラスを定義したプログラムを、上記の仕様を満たすように修正する。

`Super Jump Player` を用意するために `SuperJumpPlayer` クラスを定義する。 `NormalPlayer` クラスと `JumpPlayer` クラスと同様、 `__init__()` メソッド、 `__str__()` メソッド、 `move()` メソッドを定義する。 `__str__()` メソッドでは `S` を返すように記述し、 `move()` メソッドでは加算/減算する値を全て `3` にする。

`__init__()` メソッドや `move()` メソッドの処理は、 `NormalPlayer` クラスと `JumpPlayer` クラスが持つ `__init__()` メソッドや `move()` メソッドと同様の処理であるため、重複するプログラムを記述したことになる。

```
class SuperJumpPlayer:
    def __init__(self, x, y):
        # 初期配置の座標
        self.x = x
        self.y = y

    # フィールド上での表示
    def __str__(self):
        return "S"

     # プレイヤーの駒の移動を行うメソッド
     def move(self, direction):
         if direction == "up":
             self.y -= 3
         elif direction == "down":
             self.y += 3
         elif direction == "left":
             self.x -= 3
         elif direction == "right":
             self.x += 3
         else:
             raise ValueError("無効な方向です")
```

背景オブジェクトを用意するために `Background` クラスを定義する。背景オブジェクトはフィールド上に存在するだけなので、 `Background` クラスでは、 `__init__()` メソッド、 `__str__()` メソッドを定義する。 `__str__()` メソッドでは `+` を返すように記述する。

`__init__()` メソッドの処理は、 `NormalPlayer` クラスと `JumpPlayer` クラスが持つ `__init__()` メソッドの処理と同様であるため、重複するプログラムを記述したことになる。

```
# 背景オブジェクトのクラス
class Background:
    def __init__(self, x, y):
        # 初期配置の座標
        self.x = x
        self.y = y

    # フィールド上での表示
    def __str__(self):
         return "+"
```

`Field` クラスは背景オブジェクトを新しく持つことになるため、 `__init__()` メソッドでデータ属性 `backgrounds` を新しく定義する。また、 `get_cell()` メソッドに、背景オブジェクトの表示文字を取得する `for` 文の処理を新しく追加する。

`get_cell()` メソッドに新しく追加した `for` 文の処理は、プレイヤーの駒や敵の表示文字を取得する `for` 文の処理を同様の処理であるため、重複するプログラムを記述したことになる。

```
# フィールドのクラス
class Field:
    def __init__(self, players, enemies, backgrounds):
        # フィールド上に配置するプレイヤーの駒と敵
        self.players = players
        self.enemies = enemies
        self.backgrounds = backgrounds
        # フィールドの大きさ
        self.size = 5

    # 指定した座標のマスの文字を返すメソッド
    def get_cell(self, x, y):
        for player in self.players:
            if x == player.x and y == player.y:
                return str(player)
        for enemy in self.enemies:
            if x == enemy.x and y == enemy.y:
                return str(enemy)
        for background in self.backgrounds:
            if x == background.x and y == background.y:
                return str(background)
        return "."

    # フィールドの状態を出力するメソッド
    def show(self):
        field = []
        for y in range(self.size):
            for x in range(self.size):
                cell = self.get_cell(x, y)
                field.append(cell)
            field.append("\n")
        print("".join(field))
```

ゲームのインスタンスを作成する処理に、 `SuperJumpPlayer` クラスのインスタンスと、 `Background` クラスのインスタンスを作成する処理を追加する。

```
players = [NormalPlayer(3, 3), NormalPlayer(4, 4), JumpPlayer(2, 4), SuperJumpPlayer(4, 1)]
enemies = [Enemy(0, 0), Enemy(1, 1), Enemy(2, 2)]
backgrounds = [Background(1, 3), Background(2, 3)]
field = Field(players, enemies, backgrounds)
```

上記の修正を行ったプログラムが以下である。

```
# Normal Playerのクラス
class NormalPlayer:
```

```
    def __init__(self, x, y):
        # 初期配置の座標
        self.x = x
        self.y = y

    # フィールド上での表示
    def __str__(self):
        return "N"

    # プレイヤーの駒の移動を行うメソッド
    def move(self, direction):
        if direction == "up":
            self.y -= 1
        elif direction == "down":
            self.y += 1
        elif direction == "left":
            self.x -= 1
        elif direction == "right":
            self.x += 1
        else:
            raise ValueError("無効な方向です")


# Jump Playerのクラス
class JumpPlayer:
    def __init__(self, x, y):
        # 初期配置の座標
        self.x = x
        self.y = y

    # フィールド上での表示
    def __str__(self):
        return "J"

    # プレイヤーの駒の移動を行うメソッド
    def move(self, direction):
        if direction == "up":
            self.y -= 2
        elif direction == "down":
            self.y += 2
        elif direction == "left":
            self.x -= 2
        elif direction == "right":
            self.x += 2
        else:
            raise ValueError("無効な方向です")


# Super Jump Playerのクラス
class SuperJumpPlayer:
    def __init__(self, x, y):
        # 初期配置の座標
        self.x = x
        self.y = y

    # フィールド上での表示
    def __str__(self):
        return "S"

    # プレイヤーの駒の移動を行うメソッド
    def move(self, direction):
        if direction == "up":
            self.y -= 3
        elif direction == "down":
            self.y += 3
        elif direction == "left":
            self.x -= 3
        elif direction == "right":
            self.x += 3
        else:
            raise ValueError("無効な方向です")

```

```python

# 敵のクラス
class Enemy:
    def __init__(self, x, y):
        # 初期配置の座標
        self.x = x
        self.y = y
        # 倒されたかどうかのフラグ
        self.defeated = False

    # フィールド上での表示
    def __str__(self):
        return "*" if not self.defeated else "."


# 背景オブジェクトのクラス
class Background:
    def __init__(self, x, y):
        # 初期配置の座標
        self.x = x
        self.y = y

    # フィールド上での表示
    def __str__(self):
        return "+"


# フィールドのクラス
class Field:
    def __init__(self, players, enemies, backgrounds):
        # フィールド上に配置するプレイヤーの駒と敵
        self.players = players
        self.enemies = enemies
        self.backgrounds = backgrounds
        # フィールドの大きさ
        self.size = 5

    # 指定した座標のマスの文字を返すメソッド
    def get_cell(self, x, y):
        for player in self.players:
            if x == player.x and y == player.y:
                return str(player)
        for enemy in self.enemies:
            if x == enemy.x and y == enemy.y:
                return str(enemy)
        for background in self.backgrounds:
            if x == background.x and y == background.y:
                return str(background)
        return "."

    # フィールドの状態を出力するメソッド
    def show(self):
        field = []
        for y in range(self.size):
            for x in range(self.size):
                cell = self.get_cell(x, y)
                field.append(cell)
            field.append("\n")
        print("".join(field))


# ゲームに必要なインスタンスの作成
players = [NormalPlayer(3, 3), NormalPlayer(4, 4), JumpPlayer(2, 4), SuperJumpPlayer(4, 1)]
enemies = [Enemy(0, 0), Enemy(1, 1), Enemy(2, 2)]
backgrounds = [Background(1, 3), Background(2, 3)]
field = Field(players, enemies, backgrounds)

# フィールドの初期状態の表示
field.show()

# 動かす駒を表す変数
turn = 0
```

```
147
148 while True:
149     # 駒を移動する方向を入力する
150     direction = input()
151     # 入力された文字列が"end"の場合は、無限ループを終了する
152     if direction == "end":
153         print("ゲームを強制終了")
154         break
155     # 入力された方向に駒を移動する
156     field.players[turn].move(direction)
157
158     for enemy in field.enemies:
159         # 駒が倒されていない敵と同一座標に到達したら、その敵を倒したことにする
160         if field.players[turn].x == enemy.x and field.players[turn].y == enemy.y:
161             enemy.defeated = True
162
163     # フィールドの状態を表示
164     field.show()
165
166     # 全部の敵を倒したらクリア
167     is_game_clear = True
168     for enemy in field.enemies:
169         if not enemy.defeated:
170             is_game_clear = False
171             break
172     if is_game_clear:
173         print("ゲームクリア！")
174         break
175
176     # 駒を交互に移動させる処理
177     turn = (turn + 1) % len(players)
```

```
1  *....
2  .*..S
3  ..*..
4  .++N.
5  ..J.N
6
7  up
8  *....
9  .*..S
10 ..*N.
11 .++..
12 ..J.N
13
14 up
15 *....
16 .*..S
17 ..*N.
18 .++.N
19 ..J..
20
21 up
22 *....
23 .*..S
24 ..JN.
25 .++.N
26 .....
27
28 left
29 *....
30 .S...
31 ..JN.
32 .++.N
33 .....
34
35 up
36 *....
37 .S.N.
38 ..J..
```

```
.++.N
.....

up
*....
.S.N.
..J.N
.++..
.....

up
*.J..
.S.N.
....N
.++..
.....

down
*.J..
...N.
....N
.++..
.S...

left
*.J..
..N..
....N
.++..
.S...

left
*.J..
..N..
...N.
.++..
.S...

left
J....
..N..
...N.
.++..
.S...

ゲームクリア！
```

続いて、最終版のプログラムを、上記の仕様を満たすように修正する。

`Super Jump Player` を用意するために、 `NormalPlayer` クラスや `JumpPlayer` クラスと同様、 `Player` クラスを継承する形で `SuperJumpPlayer` クラスを定義する。 `JumpPlayer` クラスと同様、 `__str__()` メソッド、 `get_move_unit()` メソッドをオーバライドする形で定義する。 `__str__()` メソッドでは `S` を返すように記述し、 `get_move_unit()` メソッドでは `3` を返すように記述する。

継承を使ったリファクタリングをしたことで、 `__init__()` メソッドや `__move__()` メソッドといった継承元が持っているメソッドは記述しなくても済むようになった。

```python
# Super Jump Playerのクラス
class SuperJumpPlayer(Player):
    # フィールド上での表示
    def __str__(self):
        return "S"

    # プレイヤーの駒の移動距離を取得するメソッド
    def get_move_unit(self):
        return 3
```

背景オブジェクトを用意するために `Background` クラスを定義する。背景オブジェクトはフィールド上に存在するオブジェクトの一種であるため、 `Object` クラスを継承する形で定義する。背景オブジェクトはフィールド上に存在するだけなので、 `Background` クラスでは、 `__str__()` メソッドのみをオーバライドする形で定義する。 `__str__()` メソッドでは `+` を返すように記述する。

継承を使ったリファクタリングをしたことで、継承元が持っている `__init__()` メソッドを記述しなくても済むようになった。

```
# 背景オブジェクトのクラス
class Background(Object):
    # フィールド上での表示
    def __str__(self):
        return "+"
```

`Field` クラスは背景オブジェクトを新しく持つことになるため、 `__init__()` メソッドでデータ属性 `backgrounds` を新しく定義する。また、 `get_cell()` メソッドの `for` 文の処理で、プレイヤーの駒や敵のインスタンスのリストに背景オブジェクトのインスタンスのリストもつなげる処理を追加した。

リファクタリングを行ったことで、背景オブジェクトの表示文字を取得する `for` 文を新しく追加するのではなく、背景オブジェクトのインスタンスのリストを繋げる処理を記述するだけで済んだ。

```
# フィールドのクラス
class Field:
    def __init__(self, players, enemies, backgrounds):
        # フィールド上に配置するプレイヤーの駒と敵
        self.players = players
        self.enemies = enemies
        self.backgrounds = backgrounds
        # フィールドの大きさ
        self.size = 5

    # 指定した座標のマスの文字を返すメソッド
    def get_cell(self, x, y):
        for obj in self.players + self.enemies + self.backgrounds:
            if x == obj.x and y == obj.y:
                return str(obj)
        return "."

    # フィールドの状態を出力するメソッド
    def show(self):
        field = []
        for y in range(self.size):
            for x in range(self.size):
                cell = self.get_cell(x, y)
                field.append(cell)
            field.append("\n")
        print("".join(field))
```

ゲームのインスタンスを作成する処理に、 `SuperJumpPlayer` クラスのインスタンスと、 `Background` クラスのインスタンスを作成する処理を追加する。

```
players = [NormalPlayer(3, 3), NormalPlayer(4, 4), JumpPlayer(2, 4), SuperJumpPlayer(4, 1)]
enemies = [Enemy(0, 0), Enemy(1, 1), Enemy(2, 2)]
backgrounds = [Background(1, 3), Background(2, 3)]
field = Field(players, enemies, backgrounds)
```

上記の修正を行ったプログラムが以下である。

```
# Objectのクラス
class Object:
    def __init__(self, x, y):
        # 初期配置の座標
```

```
        self.x = x
        self.y = y

    # フィールド上での表示
    def __str__(self):
        # 具体的な処理は子クラスで定義
        pass


# Playerのクラス
class Player(Object):
    # プレイヤーの駒の移動を行うメソッド
    def move(self, direction):
        if direction == "up":
            self.y -= self.get_move_unit()
        elif direction == "down":
            self.y += self.get_move_unit()
        elif direction == "left":
            self.x -= self.get_move_unit()
        elif direction == "right":
            self.x += self.get_move_unit()
        else:
            raise ValueError("無効な方向です")

    # プレイヤーの駒の移動距離を取得するメソッド
    def get_move_unit(self):
        # プレイヤーの駒のデフォルトの移動距離を1と設定
        return 1


# Normal Playerのクラス
class NormalPlayer(Player):
    # フィールド上での表示
    def __str__(self):
        return "N"


# Jump Playerのクラス
class JumpPlayer(Player):
    # フィールド上での表示
    def __str__(self):
        return "J"

    # プレイヤーの駒の移動距離を取得するメソッド
    def get_move_unit(self):
        return 2


# Super Jump Playerのクラス
class SuperJumpPlayer(Player):
    # フィールド上での表示
    def __str__(self):
        return "S"

    # プレイヤーの駒の移動距離を取得するメソッド
    def get_move_unit(self):
        return 3


# 敵のクラス
class Enemy(Object):
    def __init__(self, x, y):
        super().__init__(x, y)
        # 倒されたかどうかのフラグ
        self.defeated = False

    # フィールド上での表示
    def __str__(self):
        return "*" if not self.defeated else "."


# 背景オブジェクトのクラス
```

```python
class Background(Object):
    # フィールド上での表示
    def __str__(self):
        return "+"


# フィールドのクラス
class Field:
    def __init__(self, players, enemies, backgrounds):
        # フィールド上に配置するプレイヤーの駒と敵
        self.players = players
        self.enemies = enemies
        self.backgrounds = backgrounds
        # フィールドの大きさ
        self.size = 5

    # 指定した座標のマスの文字を返すメソッド
    def get_cell(self, x, y):
        for obj in self.players + self.enemies + self.backgrounds:
            if x == obj.x and y == obj.y:
                return str(obj)
        return "."

    # フィールドの状態を出力するメソッド
    def show(self):
        field = []
        for y in range(self.size):
            for x in range(self.size):
                cell = self.get_cell(x, y)
                field.append(cell)
            field.append("\n")
        print("".join(field))


# ゲームに必要なインスタンスの作成
players = [NormalPlayer(3, 3), NormalPlayer(4, 4), JumpPlayer(2, 4), SuperJumpPlayer(4, 1)]
enemies = [Enemy(0, 0), Enemy(1, 1), Enemy(2, 2)]
backgrounds = [Background(1, 3), Background(2, 3)]
field = Field(players, enemies, backgrounds)

# フィールドの初期状態の表示
field.show()

# 動かす駒を表す変数
turn = 0

while True:
    # 駒を移動する方向を入力する
    direction = input()
    # 入力された文字列が"end"の場合は、無限ループを終了する
    if direction == "end":
        print("ゲームを強制終了")
        break
    # 入力された方向に駒を移動する
    field.players[turn].move(direction)

    for enemy in field.enemies:
        # 駒が倒されていない敵と同一座標に到達したら、その敵を倒したことにする
        if field.players[turn].x == enemy.x and field.players[turn].y == enemy.y:
            enemy.defeated = True

    # フィールドの状態を表示
    field.show()

    # 全部の敵を倒したらクリア
    if all([enemy.defeated for enemy in field.enemies]):
        print("ゲームクリア！")
        break

    # 駒を交互に移動させる処理
    turn = (turn + 1) % len(players)
```

```
*....
.*..S
..*..
.++N.
..J.N

up
*....
.*..S
..*N.
.++..
..J.N

up
*....
.*..S
..*N.
.++.N
..J..

up
*....
.*..S
..JN.
.++.N
.....

left
*....
.S...
..JN.
.++.N
.....

up
*....
.S.N.
..J..
.++.N
.....

up
*....
.S.N.
..J.N
.++..
.....

up
*.J..
.S.N.
....N
.++..
.....

down
*.J..
...N.
....N
.++..
.S...

left
*.J..
..N..
....N
.++..
.S...

left
*.J..
```

```
..N..
...N.
.++..
.S...

left
J....
..N..
...N.
.++..
.S...

ゲームクリア！
```

このように、リファクタリングを行ったことで、追加の仕様の変更が入った際に、より少ない変更かつ重複するコードを排除する形で修正を行うことができることがわかる。

問題1

問題文

空のリストが格納された変数 `li` がある。リスト内包表記を使って、1から10まで整数をそれぞれ3乗した数の中で、4の倍数のみ数を持つリストを出力するようにプログラムを修正せよ。

制約

- リスト内包表記を使用する。
- 出力するリストの中は昇順（小さい順）に並んでいる。

出力

リストに持つ数は、ぞれぞれ2の3乗（ `8` ）、4の3乗（ `64` ）、6の3乗（ `216` ）、8の3乗（ `512` ）、10の3乗（ `1000` ）、である。

```
[8, 64, 216, 512, 1000]
```

解答の雛形

```
# liに代入する値をリスト内包表記に変更して解答
li = []
print(li)
```

問題2

問題文

このプログラムは、テキストの15章で取り扱った、縦5マス横5マスのフィールド上に、複数種類の駒を配置して、駒の移動する方向をプレイヤーが指定することで、駒を移動し、敵を取り除くゲームである。

`up` 、 `down` 、 `left` 、 `right` のいずれかを入力しプレイヤーの駒 `N` ・駒 `J` を移動させて、駒を敵 `*` の位置に移動すれば敵 `*` を倒すことができる。全ての敵 `*` を倒せば、ゲームクリアとなり、ゲームを終了する。

駒 `N` は1マスずつ移動し、駒 `J` は2マスずつ移動する。

フィールドの左上隅の座標が横方向0、縦方向0の座標、フィールドの右下隅が横方向4、縦方向4の座標である。

また、 `end` と入力されると、ゲームを強制終了と表示して、ゲームを終了する。

修正前のゲームの初期配置は以下である。

```
*....
.*...
..*..
...N.
..J.N
```

このプログラムに対して処理の追加や修正を行い、縦8マス横8マスのフィールドにせよ。そして、新しく横方向と縦方向の座標が5のマスに4マスずつ移動する駒 `U` を配置せよ。駒 `U` を配置する上で、プログラムに新しく `UltraJumpPlayer` クラスを定義し、インスタンスを作成すること。ただし、このプログラムに対しての処理の追加や修正を行う際に、継承を使用しないこと。

処理の追加・修正を行った後のゲームの初期配置は以下である。

```
*.......
.*......
..*.....
...N....
..J.N...
.....U..
........
........
```

制約

- `up` 、 `down` 、 `left` 、 `right` 、 `end` 以外の文字列を入力しない。
- 駒の座標が盤外にでる（横方向もしくは縦方向の座標が0から4以までの数字以外の値になる）ような方向を入力しない。
- 移動する順番は、初期配置が横方向と縦方向の座標が3の駒 `N` 、横方向と縦方向の座標が4の駒 `N` 、駒 `J` 、駒 `U` の順である。
- 駒 `U` を配置する上で、プログラムに新しく `UltraJumpPlayer` クラスを定義し、インスタンスを作成すること。
- このプログラムに対しての処理の追加や修正を行う際に、継承を使用しない。

入力

入力は次の形式で与えられる。

駒を移動する場合は、移動方向 `DIRECTION` を1行ずつ入力する。

```
DIRECTION
DIRECTION
...
```

移動方向 `DIRECTION` は、 `up` 、 `down` 、 `left` 、 `right` のいずれかの値が与えられる。

また、ゲームを強制終了する場合は、最後に `end` を1行で入力する。

```
end
```

出力

方向を入力するごとに、入力した方向に基づいて、以下のような駒を移動した後の盤面が表示される。

```
*.......
 .*......
 ..*.....
 ...N....
 ..J.N...
 .....U..
 ........
 ........
```

ゲームをクリアしたら、最後に `ゲームクリア！` と表示される。

ゲームを強制終了したら、最後に `ゲームを強制終了` と表示される。

入力例1

```
up
up
up
up
up
up
up
left
up
 up
 left
```

出力例1

```
*.......
 .*......
 ..*.....
 ...N....
 ..J.N...
 .....U..
 ........
 ........

 *.......
  .*......
  ..*N....
  ........
  ..J.N...
  .....U..
  ........
  ........

 *.......
  .*......
  ..*N....
  ....N...
  ..J.....
  .....U..
  ........
  ........

 *.......
  .*......
  ..JN....
  ....N...
  ........
  .....U..
  ........
  ........

```

```
*.......
.*...U..
..JN....
....N...
........
........
........
........

*.......
.*.N.U..
..J.....
....N...
........
........
........
........

*.......
.*.N.U..
..J.N...
........
........
........
........
........

*.J.....
.*.N.U..
....N...
........
........
........
........
........

*.J.....
.U.N....
....N...
........
........
........
........
........

*.JN....
.U......
....N...
........
........
........
........
........

*.JN....
.U..N...
........
........
........
........
........
........

 J..N....
 .U..N...
 ........
 ........
 ........
 ........
 ........
 ........

```

```
109 ゲームクリア！
```

入力例2

```
1 end
```

出力例2

```
1 *.......
2 .*......
3 ..*.....
4 ...N....
5 ..J.N...
6 .....U..
7 ........
8 ........
9
10 ゲームを強制終了
```

解答の雛形

```python
# Normal Playerのクラス
class NormalPlayer:
    def __init__(self, x, y):
        # 初期配置の座標
        self.x = x
        self.y = y

    # フィールド上での表示
    def __str__(self):
        return "N"

    # プレイヤーの駒の移動を行うメソッド
    def move(self, direction):
        if direction == "up":
            self.y -= 1
        elif direction == "down":
            self.y += 1
        elif direction == "left":
            self.x -= 1
        elif direction == "right":
            self.x += 1
        else:
            raise ValueError("無効な方向です")


# Jump Playerのクラス
class JumpPlayer:
    def __init__(self, x, y):
        # 初期配置の座標
        self.x = x
        self.y = y

    # フィールド上での表示
    def __str__(self):
        return "J"

    # プレイヤーの駒の移動を行うメソッド
    def move(self, direction):
        if direction == "up":
            self.y -= 2
        elif direction == "down":
            self.y += 2
```

```
        elif direction == "left":
            self.x -= 2
        elif direction == "right":
            self.x += 2
        else:
            raise ValueError("無効な方向です")


# 敵のクラス
class Enemy:
    def __init__(self, x, y):
        # 初期配置の座標
        self.x = x
        self.y = y
        # 倒されたかどうかのフラグ
        self.defeated = False

    # フィールド上での表示
    def __str__(self):
        return "*" if not self.defeated else "."


# フィールドのクラス
class Field:
    def __init__(self, players, enemies):
        # フィールド上に配置するプレイヤーの駒と敵
        self.players = players
        self.enemies = enemies
        # フィールドの大きさ
        self.size = 5

    # 指定した座標のマスの文字を返すメソッド
    def get_cell(self, x, y):
        for player in self.players:
            if x == player.x and y == player.y:
                return str(player)
        for enemy in self.enemies:
            if x == enemy.x and y == enemy.y:
                return str(enemy)
        return "."

    # フィールドの状態を出力するメソッド
    def show(self):
        field = []
        for y in range(self.size):
            for x in range(self.size):
                cell = self.get_cell(x, y)
                field.append(cell)
            field.append("\n")
        print("".join(field))


# ゲームに必要なインスタンスの作成
players = [NormalPlayer(3, 3), NormalPlayer(4, 4), JumpPlayer(2, 4)]
enemies = [Enemy(0, 0), Enemy(1, 1), Enemy(2, 2)]
field = Field(players, enemies)

# フィールドの初期状態の表示
field.show()

# 動かす駒を表す変数
turn = 0

while True:
    # 駒を移動する方向を入力する
    direction = input()
    # 入力された文字列が'end'の場合は、無限ループを終了する
    if direction == "end":
        print("ゲームを強制終了")
        break
    # 入力された方向に駒を移動する
    field.players[turn].move(direction)
```

```
    for enemy in field.enemies:
        # 駒が倒されていない敵と同一座標に到達したら、その敵を倒したことにする
        if field.players[turn].x == enemy.x and field.players[turn].y == enemy.y:
            enemy.defeated = True

    # フィールドの状態を表示
    field.show()

    # 全部の敵を倒したらクリア
    is_game_clear = True
    for enemy in field.enemies:
        if not enemy.defeated:
            is_game_clear = False
            break
    if is_game_clear:
        print("ゲームクリア！")
        break

    # 駒を交互に移動させる処理
    turn = (turn + 1) % len(players)
```

問題3

問題文

本問で提示されているプログラムは、問題2と異なる記述がされているが、同様の挙動をするプログラムである。問題にある仕様を満たすための処理の追記や修正を行った後の挙動も同様である。

問題2では継承を使わないプログラムが記述されており、本問では継承を使ったプログラムが記述されている。継承を用いることで、継承を用いなかった問題2と比べて、プログラムの分かりやすさや処理の追加・修正のしやすさが、どのように変わるか体験してみよう。

以下、問題の本文である。

このプログラムは、テキストの15章で取り扱った、縦5マス横5マスのフィールド上に、複数種類の駒を配置して、駒の移動する方向をプレイヤーが指定することで、駒を移動し、敵を取り除くゲームである。

`up`、`down`、`left`、`right`のいずれかを入力しプレイヤーの駒`N`・駒`J`を移動させて、駒を敵`*`の位置に移動すれば敵`*`を倒すことができる。全ての敵`*`を倒せば、ゲームクリアとなり、ゲームを終了する。

駒`N`は1マスずつ移動し、駒`J`は2マスずつ移動する。

フィールドの左上隅の座標が横方向0、縦方向0の座標、フィールドの右下隅が横方向4、縦方向4の座標である。

また、`end`と入力されると、ゲームを強制終了と表示して、ゲームを終了する。

修正前のゲームの初期配置は以下である。

```
1 *....
2 .*...
3 ..*..
4 ...N.
5 ..J.N
```

このプログラムに対して処理の追加や修正を行い、縦8マス横8マスのフィールドにせよ。そして、新しく横方向と縦方向の座標が5のマスに4マスずつ移動する駒`U`を配置せよ。駒`U`を配置する上で、プログラムに新しく`UltraJumpPlayer`クラスを定義し、インスタンスを作成すること。

処理の追加・修正を行った後のゲームの初期配置は以下である。

```
*.......
.*......
..*.....
...N....
..J.N...
.....U..
........
........
```

制約

- `up` 、 `down` 、 `left` 、 `right` 、 `end` 以外の文字列を入力しない。
- 駒の座標が盤外にでる（横方向もしくは縦方向の座標が0から4以までの数字以外の値になる）ような方向を入力しない。
- 移動する順番は、初期配置が横方向と縦方向の座標が3の駒 `N` 、横方向と縦方向の座標が4の駒 `N` 、駒 `J` 、駒 `U` の順である。
- 駒 `U` を配置する上で、プログラムに新しく `UltraJumpPlayer` クラスを定義し、インスタンスを作成すること。

入力

入力は次の形式で与えられる。

駒を移動する場合は、移動方向 `DIRECTION` を1行ずつ入力する。

```
DIRECTION
DIRECTION
...
```

移動方向 `DIRECTION` は、 `up` 、 `down` 、 `left` 、 `right` のいずれかの値が与えられる。

また、ゲームを強制終了する場合は、最後に `end` を1行で入力する。

```
end
```

出力

方向を入力するごとに、入力した方向に基づいて、以下のような駒を移動した後の盤面が表示される。

```
*.......
.*......
..*.....
...N....
..J.N...
.....U..
........
........
```

ゲームをクリアしたら、最後に `ゲームクリア！` と表示される。

ゲームを強制終了したら、最後に `ゲームを強制終了` と表示される。

入力例1

```
up
up
```

```
up
up
up
up
up
left
up
 up
 left
```

出力例1

```
*.......
 .*......
 ..*.....
 ...N....
 ..J.N...
 .....U..
 ........
 ........

 *.......
  .*......
  ..*N....
  ........
  ..J.N...
  .....U..
  ........
  ........

 *.......
  .*......
  ..*N....
  ....N...
  ..J.....
  .....U..
  ........
  ........

 *.......
  .*......
  ..JN....
  ....N...
  ........
  .....U..
  ........
  ........

 *.......
  .*...U..
  ..JN....
  ....N...
  ........
  ........
  ........
  ........

 *.......
  .*.N.U..
  ..J.....
  ....N...
  ........
  ........
  ........
  ........

 *.......
  .*.N.U..
  ..J.N...
```

```
 ........
 ........
 ........
 ........
 ........

 *.J.....
 .*.N.U..
 ....N...
 ........
 ........
 ........
 ........
 ........

 *.J.....
 .U.N....
 ....N...
 ........
 ........
 ........
 ........
 ........

 *.JN....
 .U......
 ....N...
 ........
 ........
 ........
 ........
 ........

 *.JN....
 .U..N...
 ........
 ........
 ........
 ........
 ........
 ........

 J..N....
 .U..N...
 ........
 ........
 ........
 ........
 ........
 ........

 ゲームクリア！
```

入力例2

```
end
```

出力例2

```
*.......
.*......
..*.....
...N....
..J.N...
.....U..
........
```

```
 ........

ゲームを強制終了
```

解答の雛形

```python
# Objectのクラス
class Object:
    def __init__(self, x, y):
        # 初期配置の座標
        self.x = x
        self.y = y

    # フィールド上での表示
    def __str__(self):
        # 具体的な処理は子クラスで定義
        pass


# Playerのクラス
class Player(Object):
    # プレイヤーの駒の移動を行うメソッド
    def move(self, direction):
        if direction == "up":
            self.y -= self.get_move_unit()
        elif direction == "down":
            self.y += self.get_move_unit()
        elif direction == "left":
            self.x -= self.get_move_unit()
        elif direction == "right":
            self.x += self.get_move_unit()
        else:
            raise ValueError("無効な方向です")

    # プレイヤーの駒の移動距離を取得するメソッド
    def get_move_unit(self):
        # プレイヤーの駒のデフォルトの移動距離を1と設定
        return 1


# Normal Playerのクラス
class NormalPlayer(Player):
    # フィールド上での表示
    def __str__(self):
        return "N"


# Jump Playerのクラス
class JumpPlayer(Player):
    # フィールド上での表示
    def __str__(self):
        return "J"

    # プレイヤーの駒の移動距離を取得するメソッド
    def get_move_unit(self):
        return 2


# 敵のクラス
class Enemy(Object):
    def __init__(self, x, y):
        super().__init__(x, y)
        # 倒されたかどうかのフラグ
        self.defeated = False

    # フィールド上での表示
    def __str__(self):
        return "*" if not self.defeated else "."
```

```
# フィールドのクラス
class Field:
    def __init__(self, players, enemies):
        # フィールド上に配置するプレイヤーの駒と敵
        self.players = players
        self.enemies = enemies
        # フィールドの大きさ
        self.size = 5

    # 指定した座標のマスの文字を返すメソッド
    def get_cell(self, x, y):
        for obj in self.players + self.enemies:
            if x == obj.x and y == obj.y:
                return str(obj)
        return "."

    # フィールドの状態を出力するメソッド
    def show(self):
        field = []
        for y in range(self.size):
            for x in range(self.size):
                cell = self.get_cell(x, y)
                field.append(cell)
            field.append("\n")
        print("".join(field))


# ゲームに必要なインスタンスの作成
players = [NormalPlayer(3, 3), NormalPlayer(4, 4), JumpPlayer(2, 4)]
enemies = [Enemy(0, 0), Enemy(1, 1), Enemy(2, 2)]
field = Field(players, enemies)

# フィールドの初期状態の表示
field.show()

# 動かす駒を表す変数
turn = 0

while True:
    # 駒を移動する方向を入力する
    direction = input()
    # 入力された文字列が'end'の場合は、無限ループを終了する
    if direction == "end":
        print("ゲームを強制終了")
        break
    # 入力された方向に駒を移動する
    field.players[turn].move(direction)

    for enemy in field.enemies:
        # 駒が倒されていない敵と同一座標に到達したら、その敵を倒したことにする
        if field.players[turn].x == enemy.x and field.players[turn].y == enemy.y:
            enemy.defeated = True

    # フィールドの状態を表示
    field.show()

    # 全部の敵を倒したらクリア
    if all([enemy.defeated for enemy in field.enemies]):
        print("ゲームクリア！")
        break

    # 駒を交互に移動させる処理
    turn = (turn + 1) % len(players)
```

問題4

問題文

このプログラムは、テキストの15章で取り扱った、縦5マス横5マスのフィールド上に、複数種類の駒を配置して、駒の移動する方向をプレイヤーが指定することで、駒を移動し、敵を取り除くゲームである。

`up`、`down`、`left`、`right`のいずれかを入力しプレイヤーの駒`N`・駒`J`を移動させて、駒を敵`*`の位置に移動すれば敵`*`を倒すことができる。全ての敵`*`を倒せば、ゲームクリアとなり、ゲームを終了する。

駒`N`は1マスずつ移動し、駒`J`は2マスずつ移動する。

フィールドの左上隅の座標が横方向0、縦方向0の座標、フィールドの右下隅が横方向4、縦方向4の座標である。

また、`end`と入力されると、ゲームを強制終了と表示して、ゲームを終了する。

修正前のゲームの初期配置は以下である。

```
*....
.*...
..*..
...N.
..J.N
```

このプログラムに対して処理の追加や修正を行い、縦8マス横8マスのフィールドにせよ。そして、新しく横方向と縦方向の座標が5のマスに4マスずつ移動する駒`U`を配置せよ。駒`U`を配置する上で、プログラムに新しく`UltraJumpPlayer`クラスを定義し、インスタンスを作成すること。ただし、このプログラムに対しての処理の追加や修正を行う際に、継承を使用しないこと。

処理の追加・修正を行った後のゲームの初期配置は以下である。

```
*.......
.*......
..*.....
...N....
..J.N...
.....U..
........
........
```

さらに、駒の座標が盤外にでる（横方向もしくは縦方向の座標が0から4以までの数字以外の値になる）ような方向を入力されたら、盤面の反対方向に移動する処理を追加せよ。この仕様についても、プログラムに対しての処理の追加や修正を行う際に、継承を使用しないこと。

例えば、縦5マス横5マスのフィールド上に駒`N`が以下のように配置された状況について考える。

```
*...N
.....
.....
.....
.....
```

上記の状況で、`up`と指定すると、以下のように盤面の反対方向（盤面の一番下）に移動する。

```
*....
.....
.....
.....
....N
```

続けて、上記の状況で、 `right` と入力すると、以下のように盤面の反対方向（盤面の一番右）に移動する。

```
1 *....
2 .....
3 .....
4 .....
5 N....
```

制約

- `up` 、 `down` 、 `left` 、 `right` 、 `end` 以外の文字列を入力しない。
- 移動する順番は、初期配置が横方向と縦方向の座標が3の駒 `N` 、横方向と縦方向の座標が4の駒 `N` 、駒 `J` 、駒 `U` の順である。
- 駒 `U` を配置する上で、プログラムに新しく `UltraJumpPlayer` クラスを定義し、インスタンスを作成すること。
- このプログラムに対しての処理の追加や修正を行う際に、継承を使用しない。

入力

入力は次の形式で与えられる。

駒を移動する場合は、移動方向 `DIRECTION` を1行ずつ入力する。

```
1 DIRECTION
2 DIRECTION
3 ...
```

移動方向 `DIRECTION` は、 `up` 、 `down` 、 `left` 、 `right` のいずれかの値が与えられる。

また、ゲームを強制終了する場合は、最後に `end` を1行で入力する。

```
1 end
```

出力

方向を入力するごとに、入力した方向に基づいて、以下のような駒を移動した後の盤面が表示される。

```
1 *.......
2 .*......
3 ..*.....
4 ...N....
5 ..J.N...
6 .....U..
7 ........
8 ........
```

ゲームをクリアしたら、最後に `ゲームクリア！` と表示される。

ゲームを強制終了したら、最後に `ゲームを強制終了` と表示される。

入力例1

```
1 up
2 up
3 up
4 up
5 up
```

```
up
up
left
up
 up
 left
```

出力例1

```
*.......
 .*......
 ..*.....
 ...N....
 ..J.N...
 .....U..
 ........
 ........

 *.......
  .*......
  ..*N....
  ........
  ..J.N...
  .....U..
  ........
  ........

 *.......
  .*......
  ..*N....
  ....N...
  ..J.....
  .....U..
  ........
  ........

 *.......
  .*......
  ..JN....
  ....N...
  ........
  .....U..
  ........
  ........

 *.......
  .*...U..
  ..JN....
  ....N...
  ........
  ........
  ........
  ........

 *.......
  .*.N.U..
  ..J.....
  ....N...
  ........
  ........
  ........
  ........

 *.......
  .*.N.U..
  ..J.N...
  ........
  ........
  ........
```

```
61 ........
62 ........
63
64 *.J.....
65 .*.N.U..
66 ....N...
67 ........
68 ........
69 ........
70 ........
71 ........
72
73 *.J.....
74 .U.N....
75 ....N...
76 ........
77 ........
78 ........
79 ........
80 ........
81
82 *.JN....
83 .U......
84 ....N...
85 ........
86 ........
87 ........
88 ........
89 ........
90
91 *.JN....
92 .U..N...
93 ........
94 ........
95 ........
96 ........
97 ........
98 ........
99
100 J..N....
101 .U..N...
102 ........
103 ........
104 ........
105 ........
106 ........
107 ........
108
109 ゲームクリア！
```

入力例2

```
1 up
2 up
3 left
4 right
5 left
6 up
7 left
8 down
9 up
10 up
11 up
12 up
13 up
14 up
15 up
16 up
17 up
```

```
up
right
```

出力例2

```
*.......
.*......
..*.....
...N....
..J.N...
.....U..
........
........

*.......
.*......
..*N....
........
..J.N...
.....U..
........
........

*.......
.*......
..*N....
....N...
..J.....
.....U..
........
........

*.......
.*......
..*N....
....N...
J.......
.....U..
........
........

*.......
.*......
..*N....
....N...
J.......
.U......
........
........

*.......
.*......
..N.....
....N...
J.......
.U......
........
........

*.......
.*......
..N.N...
........
J.......
.U......
........
........

*.......
```

```
.*......
..N.N...
........
......J.
.U......
........
........

*.......
.U......
..N.N...
........
......J.
........
........
........

*.......
.UN.....
....N...
........
......J.
........
........
........

*.......
.UN.N...
........
........
......J.
........
........
........

 *.......
 .UN.N...
 ......J.
 ........
 ........
 ........
 ........
 ........

 *.......
 ..N.N...
 ......J.
 ........
 ........
 .U......
 ........
 ........

 *.N.....
 ....N...
 ......J.
 ........
 ........
 .U......
 ........
 ........

 *.N.N...
 ........
 ......J.
 ........
 ........
 .U......
 ........
 ........

 *.N.N.J.
```

```
.......
.......
.......
.......
.U.....
.......
.......

*.N.N.J.
.U.....
.......
.......
.......
.......
.......
.......

*...N.J.
.U.....
.......
.......
.......
.......
.......
..N.....

*.....J.
.U.....
.......
.......
.......
.......
.......
..N.N...

J.......
.U.....
.......
.......
.......
.......
.......
..N.N...

ゲームクリア！
```

入力例3

```
end
```

出力例3

```
*.......
.*......
..*.....
...N....
..J.N...
.....U..
.......
.......

ゲームを強制終了
```

解答の雛形

```
# Normal Playerのクラス
class NormalPlayer:
    def __init__(self, x, y):
        # 初期配置の座標
        self.x = x
        self.y = y

    # フィールド上での表示
    def __str__(self):
        return "N"

    # プレイヤーの駒の移動を行うメソッド
    def move(self, direction):
        if direction == "up":
            self.y -= 1
        elif direction == "down":
            self.y += 1
        elif direction == "left":
            self.x -= 1
        elif direction == "right":
            self.x += 1
        else:
            raise ValueError("無効な方向です")


# Jump Playerのクラス
class JumpPlayer:
    def __init__(self, x, y):
        # 初期配置の座標
        self.x = x
        self.y = y

    # フィールド上での表示
    def __str__(self):
        return "J"

    # プレイヤーの駒の移動を行うメソッド
    def move(self, direction):
        if direction == "up":
            self.y -= 2
        elif direction == "down":
            self.y += 2
        elif direction == "left":
            self.x -= 2
        elif direction == "right":
            self.x += 2
        else:
            raise ValueError("無効な方向です")


# 敵のクラス
class Enemy:
    def __init__(self, x, y):
        # 初期配置の座標
        self.x = x
        self.y = y
        # 倒されたかどうかのフラグ
        self.defeated = False

    # フィールド上での表示
    def __str__(self):
        return "*" if not self.defeated else "."


# フィールドのクラス
class Field:
    def __init__(self, players, enemies):
        # フィールド上に配置するプレイヤーの駒と敵
        self.players = players
        self.enemies = enemies
        # フィールドの大きさ
```

```
        self.size = 5

    # 指定した座標のマスの文字を返すメソッド
    def get_cell(self, x, y):
        for player in self.players:
            if x == player.x and y == player.y:
                return str(player)
        for enemy in self.enemies:
            if x == enemy.x and y == enemy.y:
                return str(enemy)
        return "."

    # フィールドの状態を出力するメソッド
    def show(self):
        field = []
        for y in range(self.size):
            for x in range(self.size):
                cell = self.get_cell(x, y)
                field.append(cell)
            field.append("\n")
        print("".join(field))


# ゲームに必要なインスタンスの作成
players = [NormalPlayer(3, 3), NormalPlayer(4, 4), JumpPlayer(2, 4)]
enemies = [Enemy(0, 0), Enemy(1, 1), Enemy(2, 2)]
field = Field(players, enemies)

# フィールドの初期状態の表示
field.show()

# 動かす駒を表す変数
turn = 0

while True:
    # 駒を移動する方向を入力する
    direction = input()
    # 入力された文字列が'end'の場合は、無限ループを終了する
    if direction == "end":
        print("ゲームを強制終了")
        break
    # 入力された方向に駒を移動する
    field.players[turn].move(direction)

    for enemy in field.enemies:
        # 駒が倒されていない敵と同一座標に到達したら、その敵を倒したことにする
        if field.players[turn].x == enemy.x and field.players[turn].y == enemy.y:
            enemy.defeated = True

    # フィールドの状態を表示
    field.show()

    # 全部の敵を倒したらクリア
    is_game_clear = True
    for enemy in field.enemies:
        if not enemy.defeated:
            is_game_clear = False
            break
    if is_game_clear:
        print("ゲームクリア！")
        break

    # 駒を交互に移動させる処理
    turn = (turn + 1) % len(players)
```

問題5

問題文

本問で提示されているプログラムは、問題4と異なる記述がされているが、同様の挙動をするプログラムである。問題にある仕様を満たすための処理の追記や修正を行った後の挙動も同様である。

問題4では継承を使わないプログラムが記述されており、本問では継承を使ったプログラムが記述されている。継承を用いることで、継承を用いなかった問題4と比べて、プログラムの分かりやすさや処理の追加・修正のしやすさが、どのように変わるか体験してみよう。

以下、問題の本文である。

このプログラムは、テキストの15章で取り扱った、縦5マス横5マスのフィールド上に、複数種類の駒を配置して、駒の移動する方向をプレイヤーが指定することで、駒を移動し、敵を取り除くゲームである。

`up`、`down`、`left`、`right`のいずれかを入力しプレイヤーの駒`N`・駒`J`を移動させて、駒を敵`*`の位置に移動すれば敵`*`を倒すことができる。全ての敵`*`を倒せば、ゲームクリアとなり、ゲームを終了する。

駒`N`は1マスずつ移動し、駒`J`は2マスずつ移動する。

フィールドの左上隅の座標が横方向0、縦方向0の座標、フィールドの右下隅が横方向4、縦方向4の座標である。

また、`end`と入力されると、ゲームを強制終了と表示して、ゲームを終了する。

修正前のゲームの初期配置は以下である。

```
*....
.*...
..*..
...N.
..J.N
```

このプログラムに対して処理の追加や修正を行い、縦8マス横8マスのフィールドにせよ。そして、新しく横方向と縦方向の座標が5のマスに4マスずつ移動する駒`U`を配置せよ。駒`U`を配置する上で、プログラムに新しく`UltraJumpPlayer`クラスを定義し、インスタンスを作成すること。

処理の追加・修正を行った後のゲームの初期配置は以下である。

```
*.......
.*......
..*.....
...N....
..J.N...
.....U..
........
........
```

さらに、駒の座標が盤外にでる（横方向もしくは縦方向の座標が0から4以までの数字以外の値になる）ような方向を入力されたら、盤面の反対方向に移動する処理を追加せよ。

例えば、縦5マス横5マスのフィールド上に駒`N`が以下のように配置された状況について考える。

```
*...N
.....
.....
.....
.....
```

上記の状況で、 `up` と指定すると、以下のように盤面の反対方向（盤面の一番下）に移動する。

```
*....
.....
.....
.....
....N
```

続けて、上記の状況で、 `right` と入力すると、以下のように盤面の反対方向（盤面の一番右）に移動する。

```
*....
.....
.....
.....
N....
```

制約

- `up` 、 `down` 、 `left` 、 `right` 、 `end` 以外の文字列を入力しない。
- 移動する順番は、初期配置が横方向と縦方向の座標が3の駒 `N` 、横方向と縦方向の座標が4の駒 `N` 、駒 `J` 、駒 `U` の順である。
- 駒 `U` を配置する上で、プログラムに新しく `UltraJumpPlayer` クラスを定義し、インスタンスを作成すること。

入力

入力は次の形式で与えられる。

駒を移動する場合は、移動方向 `DIRECTION` を1行ずつ入力する。

```
DIRECTION
DIRECTION
...
```

移動方向 `DIRECTION` は、 `up` 、 `down` 、 `left` 、 `right` のいずれかの値が与えられる。

また、ゲームを強制終了する場合は、最後に `end` を1行で入力する。

```
end
```

出力

方向を入力するごとに、入力した方向に基づいて、以下のような駒を移動した後の盤面が表示される。

```
*.......
.*......
..*.....
...N....
..J.N...
.....U..
........
........
```

ゲームをクリアしたら、最後に `ゲームクリア！` と表示される。

ゲームを強制終了したら、最後に `ゲームを強制終了` と表示される。

入力例1

```
up
up
up
up
up
up
up
left
up
 up
 left
```

出力例1

```
*.......
.*......
..*.....
...N....
..J.N...
.....U..
........
........

 *.......
 .*......
 ..*N....
 ........
 ..J.N...
 .....U..
 ........
 ........

 *.......
 .*......
 ..*N....
 ....N...
 ..J.....
 .....U..
 ........
 ........

 *.......
 .*......
 ..JN....
 ....N...
 ........
 .....U..
 ........
 ........

 *.......
 .*...U..
 ..JN....
 ....N...
 ........
 ........
 ........
 ........

 *.......
 .*.N.U..
 ..J.....
 ....N...
 ........
 ........
 ........
```

```
 ........

 *.......
 .*.N.U..
 ..J.N...
 ........
 ........
 ........
 ........
 ........

 *.J.....
 .*.N.U..
 ....N...
 ........
 ........
 ........
 ........
 ........

 *.J.....
 .U.N....
 ....N...
 ........
 ........
 ........
 ........
 ........

 *.JN....
 .U......
 ....N...
 ........
 ........
 ........
 ........
 ........

 *.JN....
 .U..N...
 ........
 ........
 ........
 ........
 ........
 ........

 J..N....
 .U..N...
 ........
 ........
 ........
 ........
 ........
 ........

 ゲームクリア！
```

入力例2

```
up
up
left
right
left
up
left
down
up
```

```
up
up
up
up
up
up
up
up
up
right
```

出力例2

```
*.......
 .*......
 ..*.....
 ...N....
 ..J.N...
 .....U..
 ........
 ........

 *.......
  .*......
  ..*N....
  ........
  ..J.N...
  .....U..
  ........
  ........

 *.......
  .*......
  ..*N....
  ....N...
  ..J.....
  .....U..
  ........
  ........

 *.......
  .*......
  ..*N....
  ....N...
 J.......
  .....U..
  ........
  ........

 *.......
  .*......
  ..*N....
  ....N...
 J.......
  .U......
  ........
  ........

 *.......
  .*......
  ..N.....
  ....N...
 J.......
  .U......
  ........
  ........

 *.......
  .*......
```

```
..N.N...
........
J.......
.U......
........
........

*.......
.*......
..N.N...
........
......J.
.U......
........
........

*.......
.U......
..N.N...
........
......J.
........
........
........

*.......
.UN.....
....N...
........
......J.
........
........
........

*.......
.UN.N...
........
........
......J.
........
........
........

 *.......
 .UN.N...
 ......J.
 ........
 ........
 ........
 ........
 ........

 *.......
 ..N.N...
 ......J.
 ........
 ........
 .U......
 ........
 ........

 *.N.....
 ....N...
 ......J.
 ........
 ........
 .U......
 ........
 ........

 *.N.N...
 ........
```

```
......J.
........
........
.U......
........
........

*.N.N.J.
........
........
........
........
.U......
........
........

*.N.N.J.
.U......
........
........
........
........
........
........

*...N.J.
.U......
........
........
........
........
........
..N.....

*.....J.
.U......
........
........
........
........
........
..N.N...

J.......
.U......
........
........
........
........
........
..N.N...

ゲームクリア！
```

入力例3

```
end
```

出力例3

```
*.......
.*......
..*.....
...N....
..J.N...
.....U..
```

```
.....
.....

ゲームを強制終了
```

解答の雛形

```python
# Objectのクラス
class Object:
    def __init__(self, x, y):
        # 初期配置の座標
        self.x = x
        self.y = y

    # フィールド上での表示
    def __str__(self):
        # 具体的な処理は子クラスで定義
        pass


# Playerのクラス
class Player(Object):
    # プレイヤーの駒の移動を行うメソッド
    def move(self, direction):
        if direction == "up":
            self.y -= self.get_move_unit()
        elif direction == "down":
            self.y += self.get_move_unit()
        elif direction == "left":
            self.x -= self.get_move_unit()
        elif direction == "right":
            self.x += self.get_move_unit()
        else:
            raise ValueError("無効な方向です")

    # プレイヤーの駒の移動距離を取得するメソッド
    def get_move_unit(self):
        # プレイヤーの駒のデフォルトの移動距離を1と設定
        return 1


# Normal Playerのクラス
class NormalPlayer(Player):
    # フィールド上での表示
    def __str__(self):
        return "N"


# Jump Playerのクラス
class JumpPlayer(Player):
    # フィールド上での表示
    def __str__(self):
        return "J"

    # プレイヤーの駒の移動距離を取得するメソッド
    def get_move_unit(self):
        return 2


# 敵のクラス
class Enemy(Object):
    def __init__(self, x, y):
        super().__init__(x, y)
        # 倒されたかどうかのフラグ
        self.defeated = False

    # フィールド上での表示
    def __str__(self):
        return "*" if not self.defeated else "."
```

```
# フィールドのクラス
class Field:
    def __init__(self, players, enemies):
        # フィールド上に配置するプレイヤーの駒と敵
        self.players = players
        self.enemies = enemies
        # フィールドの大きさ
        self.size = 5

    # 指定した座標のマスの文字を返すメソッド
    def get_cell(self, x, y):
        for obj in self.players + self.enemies:
            if x == obj.x and y == obj.y:
                return str(obj)
        return "."

    # フィールドの状態を出力するメソッド
    def show(self):
        field = []
        for y in range(self.size):
            for x in range(self.size):
                cell = self.get_cell(x, y)
                field.append(cell)
            field.append("\n")
        print("".join(field))


# ゲームに必要なインスタンスの作成
players = [NormalPlayer(3, 3), NormalPlayer(4, 4), JumpPlayer(2, 4)]
enemies = [Enemy(0, 0), Enemy(1, 1), Enemy(2, 2)]
field = Field(players, enemies)

# フィールドの初期状態の表示
field.show()

# 動かす駒を表す変数
turn = 0

while True:
    # 駒を移動する方向を入力する
    direction = input()
    # 入力された文字列が'end'の場合は、無限ループを終了する
    if direction == "end":
        print("ゲームを強制終了")
        break
    # 入力された方向に駒を移動する
    field.players[turn].move(direction)

    for enemy in field.enemies:
        # 駒が倒されていない敵と同一座標に到達したら、その敵を倒したことにする
        if field.players[turn].x == enemy.x and field.players[turn].y == enemy.y:
            enemy.defeated = True

    # フィールドの状態を表示
    field.show()

    # 全部の敵を倒したらクリア
    if all([enemy.defeated for enemy in field.enemies]):
        print("ゲームクリア！")
        break

    # 駒を交互に移動させる処理
    turn = (turn + 1) % len(players)
```