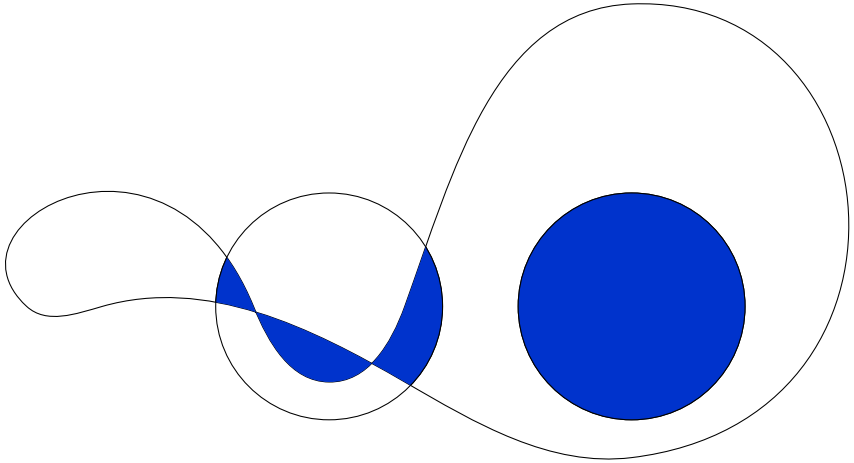


計算機プログラミング

学際科学科　松島慎 2024 年 5 月 23 日

この講義（前半）の進め方

この講義の目標は複雑な問題を解くためのアルゴリズムを実装する力を身につけることである。初回はオンライン講義なので、進め方の説明が主な目標である。

与えられた問題の解決手段としてアルゴリズムを理解する力も重要であるが、理解しているアルゴリズムを実装し（＝動作するプログラムに書き下すこと）実行する力も重要である。プログラミングは自分で試行錯誤をすることで覚えるものなので、この講義では DOMjudge を使って十分な試行回数を確保することを試みる。

DOMjudge は設定した問題の解法となるプログラムを提出（submit）すると自動で採点することができるシステムである。この講義を受講する学生は`http://163.220.176.180:9000/` にアクセスし User の登録を済ませること。（Register now? をクリック）

- Username：学生証番号。半角ハイフンあり（例：J3-220465）
- Full name (optional)：氏名。学生証の記述の通り（例：東大太郎）
- Email address (optional)：空欄（連絡は UTOL を通じて行う）
- Team name：任意だが他の人とかぶらないようにする（公開情報となるので注意。例：J3-220465）
- Affiliation：**Use existing affiliation を選択**し、自分の学年科類を選択
- Password：任意（教員に見られても問題ないものを使うこと。**忘れないように!**）

第 0 章

Python の実行環境

0.1 python2 と python3

この講義で扱う言語は Python であるが、Python は Python 3.0 系と Python 2.0 系では大きく仕様が異なるので注意が必要である。この講義では Python 3.0 系を用いたプログラミングを習得する。

0.2 インタラクティブモードと標準モード

Python のプログラムを実行する方法は二通りある。一つは**インタラクティブモード**と呼び、インタプリタがセルと呼ばれる部分ごとにプログラムを処理し、その都度出力を受け取ることを繰り返す方法である。もう一つは**標準モード**と呼びインタプリタがファイルに書かれた文を上から一つずつ処理して最後の文を処理したら終了する方法である。

python3 のプログラムの仕様を理解するためにはインタラクティブモードで実行するのが手軽で便利

である。少し長いプログラムの場合は標準モードで実行すると良い。

0.3　インタラクティブモードの実行環境

インタラクティブモードの実行環境としては Python Notebook があげられる。情報教育棟の iMac を使う場合はすでにインストールされている Anaconda Navigator を使う方法がある。Finder からアプリケーションの下にある Anaconda Navigator を起動して、そこから Jupyter Notebook を選択する。またはターミナル[*1]から以下のコマンドを実行する。ここではプロンプト文字列は$としてあり、$以降が実際に入力する文字列である（今後も実行例については同様に黒字は入力、灰色字は計算機の出力を表す。）

```
$ jupyter-notebook
```

より単純にインタラクティブモードを使う方法として、ターミナルから以下を実行することもできる：

```
$ python3
```

すると以下のような python3 のプロンプト文字列「>>>」が表示されるので、その後に入力される文は python3 のプログラムと解釈されて実行される。例えば 1+2 を入力して Enter キーを押すとこの式の評価結果が出力される

[*1] ターミナルがわからない時はhttps://hwb.ecc.u-tokyo.ac.jp/wp/information-2/cui/をみよ。

インタラクティブモードでの実行例

```
>>> 1+2
3
```

自分の PC が使いたい場合はhttps://utokyo-ipp.github.io/index.html を参考にして Google Colaboratory を使うのが簡単である。Google Colaboratory で Jupyter Notebook が使えて、Python を動かすことができる。または自分の PC に Anaconda をインストールする。次に Anaconda Navigator を起動して、そこから Jupyter Notebook が使用することができる。

0.4　標準モードの実行環境

標準モードではプログラムはテキストファイルとして編集・保存し、この段階ではプログラムは実行されない。単純なテキストエディタとしては Vim, Emacs[*2]の使用を勧めるが統合開発環境として Visual StudioCCode (VSCode) や pyCharm などを使うのも便利である。例えば、`hello.py`[*3] という名前でテキストファイルを作成し、内容を以下のように「`print('Hello World!')`」という一行だけにして保存すれば、これは`Hello World!`と出力するだけのプログラムである。

`hello.py` の内容

```
print('Hello World!')
```

標準モードでは**標準入力**を受けとり、**標準出力**を返すことでプログラムを実行する。おおざっぱには、標準入力とはキーボードから入力された文字列であり標準出力とはスクリーンに現れる文字列である。

[*2] Emacs の使い方は HWB の説明http://hwb.ecc.u-tokyo.ac.jp/current/literacy/editor/emacs/参照

[*3] Python のプログラムは拡張子を`.py` とするのが慣例である。

ターミナル上でプログラム hello.py を実行するには以下のようにコマンドを入力する。

標準モード (ターミナル) での実行例

```
$ python3 hello.py
Hello World!
```

この例では標準入力をうけとらず、標準出力に`Hello World!`という文字列を出力する。カレントフォルダに hello.py がないとこのコマンドはエラーとなるので注意すること。

0.4.1　標準入力と標準出力 (input 関数と print 関数)

標準入力を受け取るためには**input 関数**を用いることができる。`a=input()` をターミナル上から実行するとプログラムはキーボードから入力された文字列を `a` という変数に格納する。

標準出力を返すには**print 関数**を用いることができる。`print("hello")` を実行すると引数の値である hello を標準出力に出力し、最後に改行文字を出力する。最後に出力する文字を改行文字以外に指定したい場合は`print("hello",end="XX")`とすると、最後に XX を出力する。特に、`end=""`とすれば改行せずに標準出力のみを出力することができる。

例えば、以下のような関数は標準入力として得られた文字列を 2 回繰り返して出力するプログラムである。

repeat.py の内容

```
a = input() # aに標準入力の内容を文字列として格納する
print(a) # aに格納された内容を標準出力に出力する
print(a) # 再びaに格納された内容を標準出力に出力する
```

これは文字列を入力として受け取り、同じ文字列を二回改行して出力するプログラムである[*4]。この内容を repeat.py という名前のファイルに保存してターミナル上で以下のように実行することができる。

標準モードでの `repeat.py` の実行例

```
$ python repeat.py
aaaa
aaaa
aaaa
```

1 行目がプログラムを実行する命令であり、2 行目がプログラムが入力として受け取る文字列 `aaaa` である。どちらもキーボードから入力して最後に Enter を押す。3 行目と 4 行目がプログラムの標準出力として出力されたものである。すなわちこれは

入力例

```
aaaa
```

出力例

```
aaaa
aaaa
```

に対応するプログラムの例である。

[*4] 行の中で # 以降は**コメント**であり、インタプリタはこれを処理しない。

目次

第 1 章

基本の文法

変数は値の入れ物である。「変数名 = 式」を実行すると、式の値が計算され、指定された変数に**代入**される。代入の効果は、式のなかで再び変数名が出てきたときに代入された値として使われることである。

インタラクティブモードでの実行例

```
>>>a=3
>>>a*a
9
```

/は実数の割り算、//は整数の商、%は余り、**は巾乗となる。以下の様に行うと複数の変数を同時に代入することができる。

インタラクティブモードでの実行例

```
>>>a,b,c=1,10,100
>>>a+b+c
111
```

変数名としては一般的に英字の後に英数字か「_」を並べたものを用いるが、変数名に使える文字はアルファベット以外にもある。例えばひらがなやギリシャ文字なども使用可能である。詳しくはリファレンスを参照のこと。一方、構文で使う for などのキーワード（予約語）は変数名としては使えない。定義されているキーワードは keyword.kwlist に格納されており以下のようにして確認できる。

インタラクティブモードでの動作例

```
>>> import keyword
>>> keyword.kwlist
['False', 'None', 'True', 'and', 'as', 'assert', 'break', 'class', 'continue', 'def', 'del', 'elif',
 'else', 'except', 'finally', 'for', 'from', 'global', 'if', 'import', 'in', 'is', 'lambda', '
nonlocal', 'not', 'or', 'pass', 'raise', 'return', 'try', 'while', 'with', 'yield']
```

変数には**型**があり、型によって内部で行われる演算も変わってくるので、意識しておくことが必要である。型により整数と実数（浮動小数）は区別される。変数を初期化する場合は小数点のあるなしで整数と実数が区別される。例えば、`a=1`と書くとこれは整数型の 1 となり、`a=1.0`と書くと浮動小数型の 1 となる。変数の型は type 関数で確かめることができる。

インタラクティブモードでの実行例

```
>>>a=1
>>>type(a)
<class 'int'>
>>>b=1.0
>>>type(b)
<class 'float'>
>>> c="1"
>>> type(c)
<class 'str'>
```

`<class 'int'>`はその変数が整数型であることを意味しており、`<class 'float'>`はその変数が浮動小数型であることを意味している。`<class 'str'>`はその変数が文字列型であることを意味している。

実数型を整数型に変換するには「int(」「)」で囲む。数値を文字列に変換するには「str(」「)」で囲む。一般にデータを、ある「型」に変換するには「型 (」「)」で囲む。`"Hello"`のように「`"`」で囲まれたものは文字列である。「`"`」の代わりに「`'`」を使うこともできる。

1.1　分岐処理（if 文）

条件分岐による分岐処理は以下の形式で書くことができる。

```
1 if 条件:
2     文1
3     ...
4 else:
5     文2
6     ...
```

上の例ではまず 1 行目の条件が評価され、True（真）が返ってきた場合は文 1 からはじまるブロック、False（偽）が返ってきた場合文 2 からはじまるブロックが実行される。ブロックの範囲は**インデント**により指定されるので、2 行目以降 else までを全てインデントして書く必要がある。インデントとは行頭に配置されているスペースあるいはタブのことである。ファイル中で一貫していればスペースの数はいくつでもよい。数値の比較などに便利な条件の文法として以下の文法がある。

Python の文法	数学記号	意味
`x > y`	$>$	x が y より大きい
`x >= y`	$\geq$	x が y 以上
`x == y`	$=$	x と y が等しい[*1]
`x < y`	$<$	x が y より小さい
`x <= y`	$\leq$	x が y 以下
`x != y`	$\neq$	x と y が異なる
条件 1 `and` 条件 2	$\wedge$	条件 1 かつ条件 2
条件 1 `or` 条件 2	$\vee$	条件 1 または条件 2
`not` 条件 1	$\neg$	条件 1 ではない

実行例を以下に示す：

インタラクティブモードでの実行例

```
>>> print((1 >= 0))
True
>>> print((0 >= 1))
False
>>> print((3 != 3))
False
>>> print(False or True)
True
>>> print(True and False)
False
>>> print(False and True)
False
>>> print(not True)
False
>>> print(not False)
True
```

例えば 2 つの変数の値の大小による分岐処理は以下のように書ける。

インタラクティブモードでの実行例

```
>>>x,y = 1,2
>>>if x > y:
...     print("x is larger than y")
...else:
...     print("x is smaller than or equal to y")
...
x is smaller than or equal to y
```

3 つ以上の分岐は if ... elif .. else ... を組合せて実現する。例えば、符号を判定するプログラムは以下のように書ける。

sign.py

```
input_string = input()
x = float(input_string)
if x < 0 :
    print("x is negative")
elif x > 0:
    print("x is positive")
else:
    print("x is zero")
```

標準モード（ターミナル）での実行例

```
$python3 sign.py
23
s is positive
$python3 sign.py
-23
s is negative
```

`input()`は0.4.1節で説明したように標準入力を文字列として返すので、1 行目で`input_string`には標準入力の内容が一つの文字列として格納される。

例えば $-1 \leq x \leq 1$ の時は x、それ以外の場合は 0 とするには次のように書くことができる。不等式は連結して評価することができる。リファレンスにあるように「`x < y <= z`」は「`x < y and y <= z`」

と等価になる。

1.2　反復処理（for 文・while 文）

Python での反復処理は for 文または while 文を使ってかける。for 文は繰り返しの指定を以下のように行う。繰り返す部分は直後でインデントしたブロックである。

```
for 変数 in range(開始,終了+1):
    文
    ...
```

「range(開始, 終了 +1)」の部分に関して「開始」の値が 0 である場合は 0 を省略でき、「range(終了 +1)」とすることができる。

以下は for 文を用いて 1 から 10 までの和を計算する例である。

インタラクティブモードでの実行例

```
>>>s = 0
>>>for i in range(1,11):
...     s = s+i
...
>>>print(s)
55
```

for 文のなかに for 文を入れることもできる。この場合インデントの深さにより for の繰り返しの範囲を判断する。

インタラクティブモードでの実行例

```
>>>s = 0
>>>for i in range(1,3):
...     for j in range(1,3):
...         s = s+i*j
...         print(s)
...
1
3
5
9
```

何回くりかえすかはわからないが、ある条件が真となるまである処理を繰り返したい場合もある。このような反復処理は、While 文を用いて書くことができる。

```
while  条件:
    文
    ...
```

While を用いる反復処理では、反復する文において条件の評価が変わるような処理を行わないと無限ループあるいは何も実行しないプログラムになるので注意が必要である。

1.3　モジュールのインポートと利用

モジュールは意味がまとまったいくつかの関数や変数の定義などの文からなるプログラムであり、必要に応じて自分の実行環境に取り込む（インポートする）ことができる。Python には豊富な標準ライブラリが用意されており、これらのモジュールを適宜インポートすることができる。標準ライブラリの一覧はリファレンスを参照のこと。

```
    import math # math モジュールのインポート
    import copy # copy モジュールのインポート
```

$\sqrt{x}, \sin(x), \cos(x), \log x, \log_2 x, e^x, \pi$ を計算するには、`import math`を実行してから、`math.sqrt(x)`、`math.sin(x)`、`math.cos(x)`、`math.log(x)`、`math.log2(x)`、`math.exp(x)`、`math.pi`を使う。

インタラクティブモードでの実行例

```
>>> import math
>>> math.pi
3.141592653589793
>>> math.exp(10.0)
2.6881171418161356e+43
```

`2.6881171418161356e+43`は $2.6881171418161356 \times 10^{43}$ の意味である。

1.4　リスト

リストは要素となるデータの並びであり不特定数の要素を一つの変数で制御する**抽象データ型**の一つである。リストはプログラミング言語によらない概念であるがここでは Python のリストを学ぶ。

Python ではリストは「[値, 値,...]」のように可変個の値を指定することで定義することができる。このようにして作成したリストを変数に代入することで、リストを値にもつ変数を作ることができる。リストのに格納している各値は要素とよばれ、0 番目の値、1 番目の値などと呼ぶ。リストは次のように操作できる。

インタラクティブモードでの実行例

```
>>> a = [0,1] # 2 個の整数値で初期化。
>>> print(a) # リスト全体を表示
[0, 1]
>>> print(a[0]) # リストの指定した要素を表示
0
>>> print(a[1]) # リストの指定した要素を表示
1
>>> print(len(a)) # リストの長さ。要素数。
2
```

`len` はリストを引数とする関数で、リストに含まれる要素の数を返す。

以下の形で変数`i`が`0`から`n-1`まで変化していく反復処理により式を評価し、それぞれの結果を要素とするリストが a に格納される。このような表記をリストの**内包表記**という。

```
a = [ 式 for i in range(n)]
```

例えば以下のようにすると a にはリスト`[0,0,0,0,0,0,0,0,0,0]`が格納される。

インタラクティブモードでの実行例

```
>>> a = [0 for i in range(10)]
>>> print(a)
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

あるいは偶数のリストなどは以下のように作成することができる。

インタラクティブモードでの実行例

```
>>> b = [ 2*i for i in range(5)]
>>> print(b)
[0, 2, 4, 6, 8]
```

2 次元リストを作るには、リストの中身をリストにすればよい[*2]。(for の後の変数 (i や j) は異なるものを使うこと)

インタラクティブモードでの実行例

```
>>> b = [[0 for i in range(3)] for j in range(3)]
>>> print(b)
[[0, 0, 0], [0, 0, 0], [0, 0, 0]]
```

リストの連続する一部分を **部分リスト**という。このような部分リストの指定は、Python では**スライス**によって実現できる。b がリストである時

```
b[i:j]
```

は b のスライスとよばれ、i 番目から j-1 番目までの部分リストを指す。先頭から j-1 番目までの部分リストは `b[0:j]` で表現できる。`i==0`の時はこれを省略でき、`b[:j]` とも表現できる。一方、i 番目から末尾までの部分リストは `b[i:len(b)]` であるが `j==len(b)` の時はこれを省略でき、`b[i:]` とも表現できる。上の 2 つの省略形を組み合わせると、b の先頭から末尾までの部分リストを b[:] で指定できる。

インタラクティブモードでの実行例

```
>>> c = [ i for i in range(7) ]
>>> print(c[3:4])
[3]
>>> print(c[:4])
[0, 1, 2, 3]
>>> print(c[4:])
[4, 5, 6]
```

[*2] 以下の例では [0 for i in range(10)] は 10 回実行される。b = [a for j in range(10)] ではうまくいかない (理由は後で)。

1.4.1　リストの複製

リストの複製は Python では copy モジュールの copy 関数により行う。使用例を以下に示す。

インタラクティブモードでの実行例

```
>>>import copy
>>>a = [1, 2, 3, 4, 5]
>>>a2 = copy.copy(a) # aの内容を複製する
>>>a2[0] = 100
>>>print(a)
[1, 2, 3, 4, 5]
>>>print(a2)
[100, 2, 3, 4, 5]
```

a1 は変更されていない一方で a2 だけが変更されていることに注意する。copy 関数を使う代わりに = を使うと以下のようになる

インタラクティブモードでの実行例（つづき）

```
>>>b = [1, 2, 3, 4, 5]
>>>b2 = b # bの内容を複製するわけではない
>>>b2[0] = 100
>>>print(b)
[100, 2, 3, 4, 5]
>>>print(b2)
[100, 2, 3, 4, 5]
```

b2 だけが変更されているわけではなく b1 も値が変更されていることに注意する。各自結果を確認してみること。

リスト同士を「==」で比較すると各要素が等しいかどうかを評価する。「is」を使うと、リストが同じ実態を指しているかどうかを評価する。例えば、上の二つの例の続きで以下のように動作する。

インタラクティブモードでの実行例（つづき）

```
>>>print(a == a2)
False
>>>print(b == b2)
True
>>>a2[0] = 1
>>>print(a)
[1, 2, 3, 4, 5]
>>>print(a2)
[1, 2, 3, 4, 5]
>>>print(a2 is a)
False
>>>print(b2 is b)
True
```

b[:] は要素の並びは元と同じリストであるが、元のリストのコピーが作られるので、copy 関数の代わりに使われることがある。

インタラクティブモードでの実行例

```
>>>c = [1, 2, 3, 4, 5]
>>>c2 = c[:] # copy関数と同じ動作
>>>c2[0] = 100
>>>print(c)
[1, 2, 3, 4, 5]
>>>print(c2)
[100, 2, 3, 4, 5]
```

リストを要素とするリストなどの多次元リストを複製したい場合は deepcopy 関数を使う。

インタラクティブモードでの実行例

```
>>>import copy
>>>d = [ [i+j for i in range(3) ] for j in range(3) ]
>>>d2 = copy.deepcopy(d)
[[0, 1, 2], [1, 2, 3], [2, 3, 4]]
```

1.5　鉄道路線の駅名リスト

鉄道路線の駅名のリストから、指定した駅名を探したり、新しい駅を追加するような操作を考えてみよう。この例題を扱いながらリストの操作を慣れつつ、必要な Python の知識を得ることがねらいである。まず、渋谷、新泉、駒場東大前、という鉄道路線の駅名は順番に並んでいるのでリストで表現することにする。

1.5.1　リストの入力と出力 (split メソッド)

まずはいくつかの鉄道路線を定義しておこう。

```
1 inokashira = ["渋谷駅","神泉駅","東大前駅","駒場駅","池ノ上駅","下北沢駅","新代田駅","東松原駅","明大
      前駅","永福町駅","西永福駅","浜田山駅","高井戸駅","富士見ヶ丘駅","久我山駅","三鷹台駅","井の頭公
      園駅","吉祥寺駅"]
2 yamanotesen = ["品川駅","大崎駅","五反田駅","目黒駅","恵比寿駅","渋谷駅","原宿駅","代々木駅","新宿駅"
      ,"新大久保駅","高田馬場駅","目白駅","池袋駅","大塚駅","巣鴨駅","駒込駅","田端駅","西日暮里駅","
      日暮里駅","鶯谷駅","上野駅","御徒町駅","秋葉原駅","神田駅","東京駅","有楽町駅","新橋駅","浜松町
      駅","田町駅","高輪ゲートウェイ駅"]
3 odakyuusen = ["新宿駅","南新宿駅","参宮橋駅","代々木八幡駅","代々木上原駅","東北沢駅","下北沢駅","世
      田谷代田駅","梅ヶ丘駅","豪徳寺駅","経堂駅","千歳船橋駅","祖師ヶ谷大蔵駅","成城学園前駅","喜多見
      駅","狛江駅","和泉多摩川駅","登戸駅","向ヶ丘遊園駅","生田駅","読売ランド前駅","百合ヶ丘駅","新百
      合ヶ丘駅","柿生駅","鶴川駅","玉川学園前駅","町田駅","相模大野駅","小田急相模原駅","相武台前駅","
      座間駅","海老名駅","厚木駅","本厚木駅","愛甲石田駅","伊勢原駅","鶴巻温泉駅","東海大学前駅","秦野
      駅","渋沢駅","新松田駅","開成駅","栢山駅","富水駅","螢田駅","足柄駅","小田原駅"]
4 isesakisen = ["浅草駅","業平橋駅","押上駅","曳舟駅","東向島駅","鐘ヶ淵駅","堀切駅","牛田駅","北千住駅
      ","小菅駅","五反野駅","梅島駅","西新井駅","竹ノ塚駅","谷塚駅","草加駅","松原団地駅","新田駅","蒲
      生駅","新越谷駅","越谷駅","北越谷駅","大袋駅","せんげん台駅","武里駅","一ノ割駅","春日部駅","北
      春日部駅","姫宮駅","宮代町","東武動物公園駅"]
5 touyoko = ["渋谷駅","代官山駅","中目黒駅","祐天寺駅","学芸大学駅","都立大学駅","自由が丘駅","田園調布
      駅","多摩川駅","新丸子駅","武蔵小杉駅","元住吉駅","日吉駅","綱島駅","大倉山駅","菊名駅","妙蓮寺
      駅","白楽駅","東白楽駅","反町駅","横浜駅","新高島駅","みなとみらい駅","馬車道駅","日本大通り駅",
      "元町・中華街駅"]
6 fukutoshin = ["渋谷駅","明治神宮前駅","北参道駅","新宿三丁目駅","東新宿駅","西早稲田駅","雑司が谷駅",
      "池袋駅","要町駅","千川駅","小竹向原駅","氷水台駅","平和台駅","地下鉄赤塚駅","地下鉄成増駅","和
```

```
      光市駅"]
7  denentoshisen = ["渋谷駅","池尻大橋駅","三軒茶屋駅","駒沢大学駅","桜新町駅","用賀駅","二子玉川駅","二
      子新地駅","高津駅","溝の口駅","梶が谷駅","宮崎台駅","宮前平駅","鷺沼駅","たまプラーザ駅","あざみ
      野駅","江田駅","市が尾駅","藤が丘駅","青葉台駅","田奈駅","長津田駅","つくし野駅","すずかけ台駅",
      "南町田グランベリーパーク駅","つきみ野駅","中央林間駅"]
8  ooimachi = ["大井町駅","下神明駅","戸越公園駅","中延駅","荏原町駅","旗の台駅","北千束駅","大岡山駅","
      緑が丘駅","自由が丘駅","九品仏駅","尾山台駅","等々力駅","上野毛駅","二子玉川駅","溝の口駅"]
9  nikkosen = ["東武動物公園駅","杉戸高野台駅","幸手駅","南栗橋駅"]
10 hanzomonsen = ["渋谷駅","表参道駅","青山一丁目駅","永田町駅","半蔵門駅","九段下駅","神保町駅","大手町
      駅","三越前駅","水天宮前駅","清澄白河駅","住吉駅","錦糸町駅","押上駅"]
```

井の頭線の渋谷駅〜下北沢駅間には、渋谷駅, 神泉駅, 東大前駅, 駒場駅, 池ノ上駅, 下北沢駅の駅があったが、駒場駅と東大前駅が統合して、駒場東大前駅となった。この変化を駅名のリストの操作で表してみよう。まず、東大前駅の位置を探す。Python のリストでは、index メソッドを使うと、リスト中の何番目に指定した要素があるかを調べることができる。index メソッドは要素 e がリスト x の何番目にあるかを探すのを`index(x,e)`と書かずに、`x.index(e)`のように書く。単に引数の 1 つが前に来ただけであるが、それにより「x に対して定義された index を使う」という指定になる。このようなメソッドを**インスタンスメソッド**という。

index メソッドの実行例を以下に示す。以下を実行すると、`inokashira`中の「駒場東大前駅」の位置を変数`i`に格納することができる。index メソッドで検索した駅が存在してない場合はエラーとなってしまう。

インタラクティブモードでの実行例

```
>>>inokashira = ["渋谷駅","神泉駅","東大前駅","駒場駅","池ノ上駅","下北沢駅","新代田駅","東松原駅","明大前駅","永福町駅","西永福駅","浜田山駅","高井戸駅","富士見ヶ丘駅","久我山駅","三鷹台駅","井の頭公園駅","吉祥寺駅"]
>>>i = inokashira.index("東大前駅")
>>>print(i)
2
>>>i = inokashira.index("駒場東大前駅")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: '駒場東大前駅' is not in list
```

リスト`a` について **in 演算子**を使うと、「`i` in `a`」という構文により、`i`が`a`に入っているかを評価することができる。

インタラクティブモードでの実行例

```
>>> b = [0,2,4,6]
>>> i = 0
>>> print( i in b )
True
>>> print( (i+1) in b)
False
```

「” 駒場東大前駅” in inokashira」により、” 駒場東大前駅” が inokashira に入っているかを調べることができるので、調べた結果が True の時だけ操作するようにすればエラーが避けられる。

```
if "駒場東大前駅" in inokashira:
    i = inokashira.index("駒場東大駅")
```

次に、” 東大前駅” と” 駒場駅” を除き、そこに” 駒場東大前駅” を入れる。一つのやり方はリスト`inokashira`の” 東大前駅” の前までと、リスト`inokashira`の” 駒場駅” の次以降を求めておき、” 駒場東大前駅” を途中に挟んで繋げるというものである。

リスト`inokashira`の” 東大前駅” の前までは、” 東大前駅” が i 番目なので、`inokashira[0:i]`で

取り出すことができる。`[0:i]`は 0 番目 (先頭) から i 番目の前 (i 番目は含まない) という指定である。リスト`inokashira`の” 駒場駅” の次以降は、” 駒場駅” が`i+1`番目で、” 駒場駅” の次は i+2 番目となるので、`inokashira[i+2:]`で取り出すことができる。2 つのリストを繋げるのは「+」でできる。「+」は数値の場合は加算の意味になるが、リストの場合は連結の意味となる。

Python ではスライスに別のリストを代入することにより、スライスの要素の数も変えることができる。すなわち、以下のようにすれば`inokashira`が所望の通りに書き換わる

```
1 inokashira[i:i+2] = ["駒場東大前駅"]
```

このように元のデータを変更してしまう操作を**破壊的**という。それに対して元のリストを変更せずに計算結果を新しいリストとして返すような操作を**非破壊的**という。整数や実数の計算は非破壊的だった。例えば、`a+1` の計算をしても、a の値は変わらなかった。一方, `a+=6` などの操作は破壊的である。リストなどこれから扱うデータの操作には一見そうは見えないが破壊的なものも多いので注意する。破壊的操作をなるべく行わないでプログラムを書くのはバグを防ぐために有効な方法である。

例えば今回の場合、新しい変数 inokashirasen_new を以下のように導入することで破壊的な操作をさけることができる[*3]。

```
1 inokashira_new = inokashira[:i]+["駒場東大前駅"]+inokashira[i+2:]
```

+ でリストとリストの結合をさせるためには [” 駒場東大前駅”] のように要素が 1 つでもリストにしておかなければならない。これを inokashira[:i]+” 駒場東大前駅” とすると、リストに文字列を加算することになってうまくいかない。

キーボードから上記のようなリストを入力するのはとても煩雑であるため、バグが混入しやすい（=

[*3] 同じ a=6 という文でも初めて変数を導入する際の文は初期化と呼び破壊的とは呼ばない。二回目以降は代入と呼び、破壊的である。

間違えやすい）プログラムの典型例であるといえる。より一次ソースに近いような形として以下のようなファイル inokashira.txt が存在する場合を考える。

inokashira.txt の内容

```
1 渋谷駅,神泉駅,駒場東大前駅,池ノ上駅,下北沢駅,新代田駅,東松原駅,明大前駅,永福町駅,西永福駅,浜田山駅,
  高井戸駅,富士見ヶ丘駅,久我山駅,三鷹台駅,井の頭公園駅,吉祥寺駅
```

yamanote.txt の内容

```
1 品川駅,大崎駅,五反田駅,目黒駅,恵比寿駅,渋谷駅,原宿駅,代々木駅,新宿駅,新大久保駅,高田馬場駅,目白駅,池
  袋駅,大塚駅,巣鴨駅,駒込駅,田端駅,西日暮里駅,日暮里駅,鶯谷駅,上野駅,御徒町駅,秋葉原駅,神田駅,東京駅,
  有楽町駅,新橋駅,浜松町駅,田町駅,高輪ゲートウェイ駅
```

この時各駅が要素になっているようなリストをプログラム上で扱う方法を説明する。まず、`list_stdin_example.py`という名前のプログラムの標準入力にこのファイルの内容を与えるには以下のようにプログラムを実行すればよい。

list_stdin_example.py のターミナルでの実行例

```
1 $ python3 list_stdin_example.py < inokashira.txt
```

example_stdin_example.py の内容

```
1 input_string=input()
2 inokashira=input_string.split(',')
```

この時標準入力として与えられるのは「,」を含むひとつながりの文字列であり自然にリストになるわけではない。`inokashira` という名前の変数にリストとして格納するには以下のようにすればよい。文字列型の変数`X`に対して`X.split(',')`は文字列を「,」で区切ってそれぞれの部分文字列をそれぞれの要素とするリストを返す。そのため、実行例の通りに実行した場合、2 行目では各駅名が要素となるリストが変数 inokashira に格納される。ただし、このプログラムではその後何もしないので、標準出

力には何も出力されない。

1.5.2　直通運転 (reverse メソッド)

東急東横線は副都心線と直通運転を始めた。fukutoshin と touyoko を連結すれば良いのだが、fukutoshin が渋谷発となっているのでそのままでは連結できない。そこでリストを反転させる reverse メソッドを使う。reverse メソッドは破壊的なので適用する前にコピーしておく。あとは重複する渋谷駅を除いた touyoko[1:] を連結すれば良い。

reverse_example.py の内容

```
1 import copy
2 fukutoshin = ["渋谷駅","明治神宮前駅","北参道駅","新宿三丁目駅","東新宿駅","西早稲田駅","雑司が谷駅"
  ,"池袋駅","要町駅","千川駅","小竹向原駅","氷水台駅","平和台駅","地下鉄赤塚駅","地下鉄成増駅","和光市
  駅"]
3 touyoko = ["渋谷駅","代官山駅","中目黒駅","祐天寺駅","学芸大学駅","都立大学駅","自由が丘駅","田園調
  布駅","多摩川駅","新丸子駅","武蔵小杉駅","元住吉駅","日吉駅","綱島駅","大倉山駅","菊名駅","妙蓮寺駅"
  ,"白楽駅","東白楽駅","反町駅","横浜駅","新高島駅","みなとみらい駅","馬車道駅","日本大通り駅","元町・
  中華街駅"]
4 f = copy.copy(fukutoshin)
5 f.reverse()
6 direct = f + touyoko[1:]
7 print(fukutoshin)
8 print(touyoko)
9 print(direct)
```

reverse_example.py の実行例

```
$ python3 reverese_example.py
['渋谷駅', '明治神宮前駅', '北参道駅', '新宿三丁目駅', '東新宿駅', '西早稲田駅', '雑司が谷駅', '池袋駅', '要町駅', '千川駅', '小竹向原駅', '氷水台駅', '平和台駅', '地下鉄赤塚駅', '地下鉄成増駅', '和光市駅']
['渋谷駅', '代官山駅', '中目黒駅', '祐天寺駅', '学芸大学駅', '都立大学駅', '自由が丘駅', '田園調布駅', '多摩川駅', '新丸子駅', '武蔵小杉駅', '元住吉駅', '日吉駅', '綱島駅', '大倉山駅', '菊名駅', '妙蓮寺駅', '白楽駅', '東白楽駅', '反町駅', '横浜駅', '新高島駅', 'みなとみらい駅', '馬車道駅', '日本大通り駅', '元町・中華街駅']
['和光市駅', '地下鉄成増駅', '地下鉄赤塚駅', '平和台駅', '氷水台駅', '小竹向原駅', '千川駅', '要町駅', '池袋駅', '雑司が谷駅', '西早稲田駅', '東新宿駅', '新宿三丁目駅', '北参道駅', '明治神宮前駅', '渋谷駅', '代官山駅', '中目黒駅', '祐天寺駅', '学芸大学駅', '都立大学駅', '自由が丘駅', '田園調布駅', '多摩川駅', '新丸子駅', '武蔵小杉駅', '元住吉駅', '日吉駅', '綱島駅', '大倉山駅', '菊名駅', '妙蓮寺駅', '白楽駅', '東白楽駅', '反町駅', '横浜駅', '新高島駅', 'みなとみらい駅', '馬車道駅', '日本大通り駅', '元町・中華街駅']
```

1.5.3　リストを用いた反復処理

反復処理を行う for 文の別の文法として「for 変数名 in リスト」のように指定すると、変数名へリストの各要素を代入して反復処理を実行することができる。

インタラクティブモードでの実行例

```
>>>for x in inokashira:
...     print(x,end="")
...
...
渋谷駅神泉駅東大前駅駒場駅池ノ上駅下北沢駅新代田駅東松原駅明大前駅永福町駅西永福駅浜田山駅高井戸駅富士見ヶ丘駅久我山駅三鷹台駅井の頭公園駅吉祥寺駅>>>
>>>for x in inokashira:
...     print(x,end=",")
...
...
渋谷駅,神泉駅,東大前駅,駒場駅,池ノ上駅,下北沢駅,新代田駅,東松原駅,明大前駅,永福町駅,西永福駅,浜田山駅,高井戸駅,富士見ヶ　丘駅,久我山駅,三鷹台駅,井の頭公園駅,吉祥寺駅,>>>
```

実引数は並べられた順序に仮引数に渡されるが、end=","では、endと名付けられた引数に値","を

渡す。このような引数を Python では**キーワード引数**と呼ぶ。

末尾に「,」を表示させないようにするためには if 文を用いて末尾であるかを条件にして分岐処理を行うこともできるが、join メソッドを使っても行うことができる。join メソッドは引数のリストに対して途中に指定された文字列 (この例では「,」) を挟んだ文字列を作って、これを返り値とする。挟む文字列に対して、join メソッドを適用することに注意する。

インタラクティブモードでの実行例

```
1 >>> x=",".join(inokashira)
2 >>> print(x)
3 渋谷駅,神泉駅,東大前駅,駒場駅,池ノ上駅,下北沢駅,新代田駅,東松原駅,明大前駅,永福町駅,西永福駅,浜田山
  駅,高井戸駅,富士見ヶ　丘駅,久我山駅,三鷹台駅,井の頭公園駅,吉祥寺駅
4 >>>
```

1.5.4　リストの延伸 (append メソッド)

路線が伸びる場合を考えてみよう。横浜市営地下鉄ブルーラインはあざみ野から新百合ヶ丘まで延伸することが決定している。まず、現在のブルーラインをリスト blueline に保存しよう。

```
1 blueline = ["湘南台駅","下飯田駅","立場駅","中田駅","踊場駅","戸塚駅","舞岡駅","下永谷駅","上永谷駅",
  "港南中央駅","上大岡駅","弘明寺駅","蒔田駅","吉野町駅","阪東橋駅","伊勢佐木長者町駅","関内駅","桜木町
  駅","高島町駅","横浜駅","三ツ沢下町駅","三ツ沢上町駅","片倉町駅","岸根公園駅","新横浜駅","北新横浜駅"
  ,"新羽駅","仲町台駅","センター南駅","センター北駅","中川駅","あざみ野駅"]
```

リストの末尾への要素の追加は append メソッドでできる。

```
1 blueline.append("あざみ野")
2 blueline.append("鷲山付近")
3 blueline.append("すすき野付近")
4 blueline.append("ヨネッティ王禅寺付近")
5 blueline.append("新百合ヶ丘")
```

演習

DOMjudge の Contest「prog_1」にある問題へプログラムを提出せよ[*4]。問題 1-1,1-2,1-3,1-4,... はそれぞれ以下の問 1, 問 2, 問 3, 問 4,... に対応している。

問題で指定された入力を標準入力として受け付けて、問題で指定された出力を標準出力とするようなプログラムを作成できれば、今回はどのような形でも構わない。テストケースがすべて通るようになったら、より一般的な入力に対しても正しく出力できるような書き方を考えてみると良い。pdf からコピー・貼り付けを行うと予期せぬエラーが発生することがあるので注意すること。

問 1　Hello World! を出力するプログラムを提出せよ。

入力例（入力はなし）

出力例

```
Hello World!
```

問 2　与えられた文字列をそのまま返すプログラムを提出せよ。

入力例 1

```
Hello, World!
```

出力例 1

```
Hello, World!
```

入力例 2

```
Hello, World?
```

[*4] DOMjudge にログインすると現在いる Contest 名が右上に表示されているので、表示されている Contest 名をクリックした後、提出したい問題がある Contest 名をクリックして Contest を移動する。Contest 名の 2 つ左に Submit ボタンがあるのでこれをクリックして表示されるインターフェースから hello.py を submit する。

出力例 2

```
Hello, World?
```

問 3　与えられた文字列の末尾に「?」を加えて返すプログラムを提出せよ。

入力例 1

```
Hello, World!
```

出力例 1

```
Hello, World!?
```

入力例 2

```
Hello, World?
```

出力例 2

```
Hello, World??
```

問 4　与えられた文字列を 9 回くり返して末尾に「!」を加えて返すプログラムを提出せよ。

入力例 1

```
OK
```

出力例 1

```
OKOKOKOKOKOKOKOKOK!
```

入力例 2

```
?
```

出力例 2

```
?????????!
```

問 5　inokashira.txt の内容が標準入力として与えられたとき、駒場東大前駅が駒場駅と東大前駅に分裂したリストを構成し、これを表示する（すなわちこのリストを引数として print() を実行する）プログラムを作成せよ。入力：　駅を列挙した文字列。要素は「,」で区切られ、二つのリストの間には「boundary」という文字列があるとする。

出力：　駅のリスト

入力

```
渋谷駅,神泉駅,駒場東大前駅,池ノ上駅,下北沢駅,新代田駅,東松原駅,明大前駅,永福町駅,西永福駅,浜田山駅,高井戸駅,富士見ヶ丘駅,久我山駅,三鷹台駅,井の頭公園駅,吉祥寺駅
```

出力

```
['渋谷駅', '神泉駅', '駒場駅', '東大前駅', '池ノ上駅', '下北沢駅', '新代田駅', '東松原駅', '明大前駅', '永福町駅', '西永福駅', '浜田山駅', '高井戸駅', '富士見ヶ丘駅', '久我山駅', '三鷹台駅', '井の頭公園駅', '吉祥寺駅']
```

問 6　ある路線の全駅の文字列と急行が止まる駅の文字列を入力とし、急行に止まらない駅のリストの内容を返すプログラムを作成せよ。

入力：　ある路線の駅列挙した文字列とその路線の急行が止まる駅を列挙した文字列をつなげた文字列。要素は「,」で区切られ、二つのリストの間には「boundary」という文字列があるとする。

出力：　急行が止まらない駅のリスト

入力

```
渋谷駅,神泉駅,駒場東大前駅,池ノ上駅,下北沢駅,新代田駅,東松原駅,明大前駅,永福町駅,西永福駅,浜田山駅,高井戸駅,富士見ヶ丘駅,久我山駅,三鷹台駅,井の頭公園駅,吉祥寺駅,boundary,渋谷駅,下北沢駅,明大前駅,永福町駅,久我山駅,吉祥寺駅
```

出力

```
['神泉駅', '駒場東大前駅', '池ノ上駅', '新代田駅', '東松原駅', '西永福駅', '浜田山駅', '高井戸駅', '富士見ヶ丘駅', '三鷹台駅', '井の頭公園駅']
```

問 7　2 つの路線の路線を乗り継いで最初の路線の始発駅から 2 番目の路線の終着駅までの駅リストを表示するプログラムを作成せよ。乗り継ぎ駅が複数ある場合、始発駅から最も近い駅で乗り継ぐとする。

入力：　1 つ目の路線の駅と 2 つ目の路線の駅をつなげた文字列。要素は「,」で区切られ、二つの路線の間には「boundary」という文字列があるとする。

出力：　最初の路線の始発駅から 2 番目の路線の終着駅までの駅のリスト

入力例 1

```
品川駅,大崎駅,五反田駅,目黒駅,恵比寿駅,渋谷駅,原宿駅,代々木駅,新宿駅,新大久保駅,高田馬場駅,目白駅,池袋駅,大塚駅,巣鴨駅,駒込駅,田端駅,西日暮里駅,日暮里駅,鶯谷駅,上野駅,御徒町駅, 秋葉原駅, 神田駅,東京駅,有楽町駅,新橋駅,浜松町駅,田町駅,高輪ゲートウェイ,boundary,渋谷駅,神泉駅,駒場東大前駅,池ノ上駅,下北沢駅,新代田駅,東松原駅,明大前駅,永福町駅,西永福駅,浜田山駅,高井戸駅,富士見ヶ丘駅,久我山駅,三鷹台駅,井の頭公園駅,吉祥寺駅
```

出力例 1

```
['品川駅', '大崎駅', '五反田駅', '目黒駅', '恵比寿駅', '渋谷駅', '神泉駅', '駒場東大前駅', '池ノ上駅', '下北沢駅', '新代田駅', '東松原駅', '明大前駅', '永福町駅', '西永福駅', '浜田山駅', '高井戸駅', '富士見ヶ丘駅', '久我山駅', '三鷹台駅', '井の頭公園駅', '吉祥寺駅']
```

問 8　ある路線の全駅の文字列と急行が止まる駅の文字列を入力とし、各区間の通過する駅の数を求めるプログラムを作成せよ。

入力：　全駅と急行の止まる駅のリストをつなげた文字列。要素はカンマ「,」で区切られ、二つのリストの間には「boundary」という文字列があるとする。

出力：　間の駅の数のリスト

入力例 1

```
渋谷駅,池尻大橋駅,三軒茶屋駅,駒沢大学駅,桜新町駅,用賀駅,二子玉川駅,二子新地駅,高津駅,溝の口駅,梶が谷駅,宮崎台駅,宮前平駅,鷺沼駅,たまプラーザ駅,あざみ野駅,江田駅,市が尾駅,藤が丘駅,青葉台駅,田奈駅,長津田駅,つくし野駅,すずかけ台駅,南町田グランベリーパーク駅,つきみ野駅,中央林間駅,
boundary,渋谷駅,三軒茶屋駅,二子玉川駅,溝の口駅,鷺沼駅,たまプラーザ駅,あざみ野駅,青葉台駅,長津田駅,南町田グランベリーパーク駅,中央林間駅
```

出力例 1

```
[1, 3, 2, 3, 0, 0, 3, 1, 2, 1]
```

入力例 2

```
渋谷駅,神泉駅,駒場東大前駅,池ノ上駅,下北沢駅,新代田駅,東松原駅,明大前駅,永福町駅,西永福駅,浜田山駅,高井戸駅,富士見ヶ丘駅,久我山駅,三鷹台駅,井の頭公園駅,吉祥寺駅,boundary,渋谷駅,下北沢駅,明大前駅,永福町駅,久我山駅,吉祥寺駅
```

出力例 2

```
[3, 2, 0, 4, 2]
```

問 9　リストの i 番目から j 番目だけを反転させるプログラムを作成せよ。

入力：`i,j,x` へ代入する 3 つの値をカンマでつなげた文字列。$0 \leq$ `i` $\leq$ `j` $<$ `len`(x) であると仮定する。`i` と `j` には整数値、`x` には整数（一桁とは限らない）を要素とするリストが与えられる。

出力：リスト x の i 番目から j 番目の前までを反転させたリストの内容。

入力例 1

```
0,2,[1,2,3,4,5,6]
```

出力例 1

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

第 2 章

関数とクラス

2.1 関数の定義と呼び出し

関数は以下の文法で定義することができる。

```
def 関数名(引数1,引数2,...):
  ...
  ...
  return 返り値
```

def に続けて関数名と「(」を書き、その中に**引数** (ひきすう) を「,」で区切り必要な回数繰り返す。その後に「):」を入れる。関数の定義中では引数を**仮引数**とよび、関数を呼び出す際の引数と区別することがある。この文脈で、関数を呼び出す際の引数を**実引数**という。仮引数は関数の定義中で変数として抽象的に扱われる。関数の内容はその次の行からインデントして書き、ブロックとする。関数を呼び出して、評価した結果を**返り値**という。返り値は return 文で定義される。return 文がない場合は、ブロックが終了した時点で None という特別な値が返り値となる。

定義された関数は以下を含む文が評価されると呼び出される。

```
関数名(引数1,引数2,...)
```

関数を呼び出す時の引数と関数の定義の引数は原則として左から順に対応する。関数を呼び出す時の引数は実引数は仮引数とは異なる名前の変数でもよいし、式でも良い。関数が呼び出される際に実引数の値が計算され、その値が仮引数に代入されて、関数の定義の中のブロックが実行される。

半径 r の円の面積を計算する area 関数を定義してみよう。

インタラクティブモードでの実行例

```
>>>import math
>>>def area(r):
...     return math.pi*(r**2)
...
>>>
```

area 関数を呼び出して半径 10.0 の円の面積を計算した結果を変数 a に格納する文は以下のようになる。

インタラクティブモードでの実行例

```
>>>a=area(10.0)
>>>print(a)
314.1592653589793
```

ヘロンの公式は三角形の三辺 a, b, c に対しその三角形の面積が $S = \sqrt{s(s-a)(s-b)(s-c)}$ で表されるというものである。ここで $s = \frac{a+b+c}{2}$ である。これを用いて三辺の情報から面積を計算する関数を以下のように書くことができる。

インタラクティブモードでの実行例

```
>>>import math
>>>def heron(a,b,c):
...     s=0.5*(a+b+c)
...     return math.sqrt(s*(s-a)*(s-b)*(s-c))
...
>>>t=1
>>>u=1
>>>v=1
>>>heron(t,u,v)
0.4330127018922193
```

上の二つの例で実引数はそれぞれ`10.0`と`t,u,v`である。仮引数は`r`と`a,b,c`である。ここからわかるように実引数と仮引数は左から順に対応する。

関数を定義する際、仮引数の後に「= 値」を書くと対応する実引数が指定されなかった場合の値を指定することができる。この値をデフォルト引数値という。

インタラクティブモードでの実行例

```
>>>def f(a=0,b=1,c=2):
...     print(a,b,c)
...
>>>f()
0 1 2
```

関数の呼び出しの際は一部の引数はデフォルト引数値を利用しながら一部の引数だけ実引数により指定したいときは「仮引数名 = 実引数」により仮引数を指定することでどの仮引数に対応する実引数かを指定できる（実引数でなく式を書いてもよい）。このような引数を**キーワード引数**という[*1]。一方左からの順番で対応させる引数のことを**位置引数**という。

[*1] キーワード引数は順不同である。より詳細はリファレンスを参照

```
>>>f(c=-2)
0 1 -2
```

2.2　局所変数と大域変数

関数の中で定義された変数は関数の外からはアクセスできない。このような変数は **局所変数** (ローカル変数) と呼ばれる。また、局所変数がアクセスできる範囲をその変数の**スコープ**という。

global_local_example1.py の内容

```
# ここでは変数 a は未定義
def f():
    a = 1 # 局所変数aを定義した
    print(a) # 局所変数aの値を表示する
    #ここまでがaのスコープであり、ここで変数aが使えなくなる。
    #（fの定義おわり）＝ 局所変数aのスコープはここまで
f() # fが呼び出される。1が表示される
print(a) # 関数の外では定義されていないので未定義のエラーとなる。
```

global_local_example1.py の実行例

```
$ python3 global_local_example1.py
1
Traceback (most recent call last):
  File "global_local_example.py", line 8, in <module>
    print(a)
NameError: name 'a' is not defined
```

一番外の（＝どの関数の中でもない）文で定義した変数は **大域変数** (グローバル変数) と呼ばれる。グローバル変数はどこからでも参照できる。

global_local_example2.py の内容

```
a=2 # 大域変数aを定義した。
def f():
    print(a) # 大域変数aの値を表示する
    #（fの定義おわり）
f() # fが呼び出される。 2が表示される。
print(a) # 大域変数aの値が表示される。
```

global_local_example2.py の実行例

```
$ python3 global_local_example2.py
2
2
```

関数の中で大域変数に代入 (しようと) すると同じ変数名の局所変数ができ、代入は局所変数に対して行われる。

global_local_example3.py の内容

```
a=3 # 大域変数aを定義した。
def f():
    a=1 # 局所変数aが新しく定義され、1が格納される。
    print(a) # 局所変数aの値が表示される。
    #（fの定義おわり）＝ 局所変数aのスコープはここまで
f() # 1が表示される
print(a) # 大域変数aの値が表示される。
```

global_local_example3.py の実行例

```
$ python3 global_local_example3.py
1
3
```

関数の中で「global 変数名」を実行すると大域変数に代入できるようになる。同じ変数名の大域変数と局所変数がある場合は、その変数名では局所変数の方（よりスコープの短い方）が参照される。大域変数はいつどこで値が変更されたかがわかりにくい。インタラクティブモードで実行する場合は実行さ

れた順番が判然としないため、特に注意が必要である。

2.3　関数による副作用

関数の中で仮引数の値を変更しても、実引数の値は変わらない。

インタラクティブモードでの実行例

```
>>>def f(a): # 仮引数a
...     a=1 # 仮引数aに代入する
...     print(a) # 関数fのなかではaの値は1
...
>>>a=0
>>>f(a) # aを実引数として呼び出し
1
>>>print(a) # 実引数の値は0のままである。
0
```

値渡しの場合、仮引数への代入が実引数に影響しないのは、引数がリストでも同じである。

インタラクティブモードでの実行例

```
>>>def f(a): # 仮引数a
...     a=[1,1] # 仮引数aに代入する
...     print(a) # 関数fのなかではaの値は[1,1]
...
>>>a=[0,0]
>>>f(a)
[1, 1]
>>>print(a) # 実引数の値は[0,0]のままである。
[0, 0]
```

ただし、実引数としてリストを渡し、関数の中でリストの要素に代入すると、実引数のリストの要素も変わる。

インタラクティブモードでの実行例

```
>>>def f(a): # 仮引数a
...     a[0]=100 # 仮引数aの要素を変更する。
...     a[1]=200 # 仮引数aの要素に変更する。
...     print(a) # 関数fのなかではaの値は[1,1]
...
>>>a=[0,0]
>>>f(a) # aの値は[100,200]に変更された。
[100, 200]
>>>print(a)
[100, 200]
```

関数にリストを渡して要素を書き換えてもらい、返り値以外の方法で、呼び出し側が書き換えられた値を利用することも多いので、副作用の利用は意外に使う。ただし、呼び出す側からすると値がどこで変更されたのか分かりにくいので注意が必要である。

関数の副作用を使うと典型的な例として、リストの i 番目と j 番目の要素を入れ替える swap 関数の例がある。副作用を使わずにリスト a から入れ替えたリストを計算するようにすると以下のようになる。

```
def swap(a,i,j):
    if i == j:
        return a[:] # 他の場合と合わせてコピーを作る。
    elif i < j:
        return a[:i]+[a[j]]+a[i+1:j]+[a[i]]+a[j+1:]
    else:
        return a[:j]+[a[i]]+a[j+1:i]+[a[j]]+a[i+1:]
```

この方法は非破壊的であるという点では良いのだが、入力とほとんど同じリストを新たに作るので効率が悪い。元のリストを関数の中で変更し副作用を利用する場合は以下のように書ける。

```
def swap(a,i,j):
    a[i], a[j] = a[j], a[i]
```

2.4　関数の再帰呼び出し

関数の定義の中で自分自身を呼び出すことが、無限ループに陥らないように設計されている限り、可能である。そのような関数の呼び出し方を**再帰呼び出し**という。もっとも簡単な例の一つは以下のようなものである。

recursion_example.py の内容

```
1 def recursion_example(i):
2     if i == 0:
3         print("recursion_finished.")
4     else:
5         print("This is", i, "th recursion.")
6         recursion_example(i-1)
7
8 recursion_example(4)
```

recursion_example.py の実行結果

```
1 $python3 recursion_example.py
2 This is 4 th recursion.
3 This is 3 th recursion.
4 This is 2 th recursion.
5 This is 1 th recursion.
6 recursion_finished.
```

n 番目のフィボナッチ数 F_n は次のような関係を満たす。

$$F_n = \begin{cases} 1 & n = 1 \\ 1 & n = 2 \\ F_{n-1} + F_{n-2} & n \geq 3 \end{cases}$$

これを使って F_n を求める関数`Fibonacci(n)`は次のように再帰的に定義することができる。以下

はこれを定義およびテストするプログラムである。

Fibonacci_example.py の内容

```
def Fibonacci(n):
    """n番目のフィボナッチ数を再帰計算により行う関数
    Args:
        n(int): 1以上の整数値
    Returns:
        int: n番目のフィボナッチ数
    Examples:
        >>>Fibonacci(1)
        1
        >>>Fibonacci(6)
        10
    """
    if n == 1:
        return 1
    elif n == 2:
        return 1
    else:
        return Fibonacci(n-1)+Fibonacci(n-2)
X=int(input())
print(Fibonacci(X))
```

Fibonacci_example.py の実行例

```
$python3 Fibonacci_example.py
10
55
```

「"""」で囲まれた部分は複数行にまたがってコメントを書くことができる。2 行目から 12 行目は要件を定義を行うためのコメントであり、人がこれを読んで関数の仕様を理解するためのものである。

もう少し複雑な例としてハノイの塔の問題を解くプログラムを考えよう。スタートの棒 a、ゴールの棒 c、作業用の棒 b があり、問題はスタートの棒に積まれた n 枚の板をからゴールの棒に移すのだが、積まれた板は下に向かって大きくなり、小さな板の上に大きな板を乗せてはいけないというものである。

入力 n に対し、手順の長さは指数的に長くなるが、この手順を書き出す関数は再帰呼び出しを利用す

ることで簡単に書ける。すなわち、入力 n に対するハノイの塔を解く手順は以下のように書き下せる：

1. 入力 n-1 に対するハノイの塔の手順を用いて n-1 枚の積まれた板を棒 a から棒 b まで移す
2. n 番目の板を棒 a から棒 c へ移す
3. 入力 n-1 に対するハノイの塔の手順を用いて n-1 枚の積まれた板を棒 b から棒 c まで移す

これを利用した解法は以下のような形である。

```
def hanoi(n,a,b,c):
    """任意の枚数の板に対してハノイの塔の問題を解く（手順を出力する）関数
    Args:
        n (int) : 板の数
        a (str) : スタートの棒の名前
        b (str) : 作業用の棒の名前
        c (str) : ゴールの棒の名前
    Returns:
        None (標準出力に手順を出力する)
    Examples:
        >>> hanoi(3,"a","b","c")
        1 番目の板を a から c に移動する。
        2 番目の板を a から b に移動する。
        1 番目の板を c から b に移動する。
        3 番目の板を a から c に移動する。
        1 番目の板を b から a に移動する。
        2 番目の板を b から c に移動する。
        1 番目の板を a から c に移動する。
    """
    if n == 1:
        print(n,"番目の板を",a,"から",c,"に移動する。")
    else:
        hanoi(n-1,a,c,b)
        print(n,"番目の板を",a,"から",c,"に移動する。")
        hanoi(n-1,b,a,c)
```

2.5　クラス

クラスは関数と変数の集まりを一つの変数で扱う仕組みであり、クラスを定義することによって変数や配列よりも複雑なデータ構造を表現し、操作することができる。クラスを用いて定義した変数はオブジェクトと呼ばれ様々な機能を持つとみなすことができる。クラスはそのようなオブジェクトを初期化・複製をする機能を持っており、さらにオブジェクトを操作する関数として**インスタンスメソッド**が定義され、オブジェクトが管理する変数として**インスタンス変数**が定義される。

クラスは次のように定義することができる。クラスの名前は先頭を大文字にするのが普通である。

```
class クラスの名前:# クラスの定義
    def __init__(self, 引数1, 引数2,...): # 初期化メソッドの定義
        self.インスタンス変数名1 = インスタンス変数1の初期値
        self.インスタンス変数名2 = インスタンス変数2の初期値
        ...
        # （初期化メソッドの定義おわり）
    def インスタンスメソッド名1(self, 引数1, 引数2,...):# 1つ目のインスタンスメソッドの定義
        インスタンスメソッド1の定義
        ...
        # （1つ目のインスタンスメソッドの定義おわり）
    def インスタンスメソッド名2(self, 引数1, 引数2,...):# 2つ目のインスタンスメソッドの定義
        インスタンスメソッド2の定義
        ...
        # （2つ目のインスタンスメソッドの定義おわり）
    ...
    # （クラスの定義おわり）
```

例えば、年月日を扱う Date クラスを以下の様に定義することができる。

Date.py の内容

```
1  class Date:
2      def __init__(self,y,m,d):
3          self.year = y
4          self.month = m
5          self.day = d
6      def __str__(self):
7          return "今日は"+str(self.year)+"年"+str(self.month)+"月"+str(self.day)+"日です。"
8      def is_new_year_day(self):
9          return (self.month == 1) and (self.day == 1)
```

この Date クラスはインスタンス変数として year と month と day を持つ。

2.5.1　インスタンスの初期化 (__init__ メソッド)

ここまでのプログラムを実行しただけではクラスが定義されているだけで実際のオブジェクトは生成されない。初期化メソッドは以下のような文法で呼び出して、クラスのインスタンスを返り値とする関数のように実行することができる。

```
インスタンス名=クラス名(引数1, 引数2,...)
```

この文を実行することにより定義されたクラスのインスタンスを初期化することができる。初期化する際には初期化メソッド（コンストラクタ）と呼ばれる特殊なメソッド`__init__(self, 引数1, 引数2,...)`が呼び出される。

初期化メソッドの定義では第一引数として存在した「`self`」は呼び出しの際には現れないことに注意。クラスで扱うインスタンス変数は初期化メソッド内で定義するのが典型的である。

以下の文を実行すると year,month,day を一つのスコープで制御している today というオブジェクトが初期化される。

Date.py の内容 (つづき)

```
11 today = Date(2022,5,2)
```

仮引数は self を加えて 4 つであるが、実引数は 3 つであることに注意。today というオブジェクトの中で year,month,day のインスタンス変数はそれぞれ 2022,5,2 で初期化されている。これらインスタンス変数の値は以下の文法でアクセスすることができ、値を評価および変更することができる。

```
インスタンス名.インスタンス変数
```

Date.py の内容 (つづき)

```
12 print("今年は"+str(today.year)+"年です。")
13 print("今月は"+str(today.month)+"月です。")
14 print("今日は"+str(today.day)+"日です。")
15 today.day = 31
16 print("今日は"+str(today.day)+"日です。")
```

Date.py の実行例

```
1 $python3 Date.py
2 今年は2022年です。
3 今月は5月です。
4 今日は2日です。
5 今日は31日です。
```

インスタンスメソッドはインスタンスに対して以下のような文法により呼び出す。

```
1 インスタンス名.インスタンスメソッド(引数1,引数2,...)
```

selfは書かないことに再度注意する。元日であるかを判定するインスタンスメソッド (is_new_year_day) を持つ。インスタンス today に対し**is_new_year_day**メソッドを使う際は以下のようになる。以下の様に使用する。

```
1 today.is_new_year_day()
```

__str__メソッドは返り値を文字列に持つインスタンスメソッドであり、上記の文法によって呼び出す方法のほか、そのクラスのインスタンスを print 関数の引数にしたときに出力する文字列（すなわち1章で説明した「型 ()」の形の関数の一つである str 関数の引数にしたときに返す文字列）を定義する関数である。以下の様に使用する。

```
1 print(today)
```

2.6　クラスの継承

サブクラスというのはあるクラス (**スーパークラス**と呼ばれる) の変数やメソッドを受け継いで定義するクラスのことである。あるクラスの変数やメソッドを受け継ぐことをクラスの**継承**と呼ぶ。サブクラスではスーパークラスの変数とメソッドを受け継ぐだけでなく、新しいメソッドを定義することやスーパークラスで定義されているメソッドをサブクラスで新たに定義することができる。これを**オーバーライド**と呼ぶ。

サブクラスを定義する際は、以下のようにクラス名の後の () 中にスーパークラスとして利用したいクラス名を入れてそのサブクラスを定義する。

```
class クラス名(スーパークラス名):
```

スーパークラスとして使いたいクラスはこの文よりも上で Date が定義されているか、Date が定義されているモジュールがインポートされている必要がある。

例として、Date のサブクラスとして、和暦のクラス JDate を作ってみる。サブクラスを定義する場合は class のところで以下のように定義する。

```
#既に上でDateクラスが定義されているとする
class JDate(Date):
    def __init__(self,g,y,m,d):
        if g == "令和":
            self.year = y + 2018
        elif g == "平成":
            self.year = y + 1988
        elif g == "昭和":
            self.year = y + 1925
        elif g == "大正":
            self.year = y + 1911
        elif g == "明治":
            self.year = y + 1867
        else: # "西暦"を想定
            self.year = y
        self.month = m
        self.day = d
```

JDate クラスで__init__メソッドをオーバーライドしたので、Date クラスで定義した__init__メソッドは JDate クラスでは使えない。以下のようにして和暦の表記で年月日を指定してインスタンスを初期化することになる。（簡単のため元号の開始月日は考えず、年単位となっている。）

```
someday = JDate("昭和",43,12,21)
```

この JDate クラスには__str__メソッドが定義されていないが`print(someday)`という文を実行すると Date クラスに定義されている__str__メソッドを someday が self に代入された形で実行される。したがって、`print(someday)`を実行した場合の出力は「このインスタンスは 2001 年 1 月 1 日です。」という形になる。

もぢ、JDate クラスに対するインスタンスメソッド__print__が以下のように定義されている場合はオーバーライドが起きる。すなわち、`print(someday)`を実行した場合、JDate クラスに定義されている__str__メソッドが優先的に実行される。したがって出力は「このインスタンスは平成 13 年 1 月 1 日です。」という形になる。

```
#class JDate(Date):のスコープで以下のようなインスタンスメソッドが定義されているとする
    def __str__(self):
        if (self.year > 2019) or (self.year == 1989 and self.month > 5 ) or (self.year == 1989 and self.month == 5 and self.day >= 1):
            gengo = "令和"
            jyear = self.year - 2018
        elif (self.year > 1989) or (self.year == 1989 and self.month > 1) or (self.year == 1989 and self.month == 1 and self.day >= 8):
            gengo = "平成"
            jyear = self.year - 1988
        elif (self.year > 1926) or (self.year == 1926 and self.month == 12 and self.day >= 25):
            gengo = "昭和"
            jyear = self.year - 1925
        elif (self.year > 1912) or (self.year == 1912 and self.month > 7) or (self.year == 1912 and self.month == 7 and self.day >= 30):
            gengo = "大正"
            jyear = self.year - 1911
        elif (self.year > 1868) or (self.year == 1868 and self.month > 1) or (self.year == 1868 and self.month == 1 and self.day >= 25):
            gengo = "明治"
            jyear = self.year - 1867
        else:
            gengo = "out-of-range"
            jyear = self.year
        s = "このインスタンスは"+str(gengo)+str(jyear)+"年"+str(self.month)+"月"+str(self.day)+"日です。"
        return s
```

以下の用例を見てみよう。

```
new_century_day=JDate(2001,1,1)
print(new_century_day) # このインスタンスは平成13年1月1日です。
```

変数 new_century_day には JDate クラスのインスタンスが格納されているので、`print(new_century_day)`と実行すると JDate クラスの`__str__`が呼び出される。一方変数 today には Date クラスのインスタンスが格納されているので、`print(today)`と実行すると Date クラスの`__str__`が呼び出される。このようにオブジェクトに応じて異なるメソッドが呼びだされる。これが可能であるのは Python 言語が

多相性（ポリモーフィズム）をもつためであり、詳述はしないがその実現は**動的結合**により行われている。

2.6.1　そのほかの特殊メソッド（__eq__ メソッド・__gt__ メソッド）

Date クラスも JDate クラスも年月日を表すので、時間の前後関係という意味でインスタンス間の大小関係が比較可能である。そこで、年月日が等しいかを判定するメソッド==を定義してみよう。__eq__という名前の特殊なインスタンスメソッドを定義することによりd==d2が評価できるようになる[*2]。同様に__gt__という名前のインスタンスメソッドを定義することによりd>=d2が評価できるようになる。1 行目と 2 行目のfrom functools import total_orderingと@total_orderingは、ここで定義した__eq__と__gt__から残りの 4 つ、__ne__、__ge__、__lt__、__le__を自動的に生成するための指定である。

[*2] 実はクラスのデフォルトの == は is で定義されるので == は定義しなくても実行できる。しかし、これはオブジェクトの実体としての比較をしてしまうので、同じ年月日でも違うものと判定されてしまう。

eq_ge_example.py の内容

```
from functools import total_ordering
@total_ordering
class Date:
    def __init__(self,year,month,day):
        self.year = year
        self.month = month
        self.day = day
    def __eq__(self,other):
        return (self.year == other.year) and (self.month == other.month) and (self.day == other.day)
    def __gt__(self,other):
        if self.year > other.year:
            return True
        elif self.year == other.year:
            if self.month > other.month:
                return True
            elif self.month == other.month:
                if self.day > other.day:
                    return True
        return False

class JDate(Date):
    def __init__(self,g,y,m,d):
        if g == "昭和":
            self.year = y + 1925
        self.month = m
        self.day = d

d1 = JDate("昭和",3,3,3)
d2 = JDate("昭和",3,3,3)
print(d1 == d2)

p1 = Date(1981,2,8)
p2 = JDate("昭和",29,6,7)
print(p1 > p2)
```

標準モード (ターミナル) での実行例

```
$ python3 eq_ge_example.py
True
True
```

このように Date クラスに定義しておけば、JDate クラスのオブジェクト同士でも JDate オブジェクトと Date オブジェクトの間でも使えるようになる。その他にも様々な特殊メソッドがある。リファレンスを参照。

2.7　exec 関数

exec 関数は文字列を受け取るとその文字列を Python のプログラムの式または文として評価を行う。すなわち、`exec(X)`がある場所にあたかも X がプログラム中に書かれているかのように振る舞う。今回の演習の入力と出力は全てその入力がプログラム中で実行可能な文字列とそれがプログラム中に書かれているときに想定される出力の組になっている。そのため、適切に関数やクラスを定義するプログラムを書いた後、最終行に exec(input()) と書けば課題に正解することができる。

演習

DOMjudge 中の Contest 名を prog_2 へ変更すれば以下の問題が現れる。各問題に対して受け付けた入力に応じて結果が変わるようにプログラムを作成し、提出せよ。template を用意してあるので、これをダウンロードして加筆するのがよい。pdf からコピー・貼り付けを行うと予期せぬエラーがでることがあるので自分で書いていくことを推奨する。
UTOL 上の課題は改善点を自分で考えて実装してみた人のための報告場所です。
UTOL 上に提出しても課題に正解したかどうかは評価されないので注意してください。
UTOL 上に提出する際は、改善点をプログラムのコメントまたは提出時のコメントに明記すること。

問 1　Fibonacci 関数を定義し、入力として関数を呼び出すプログラムの文字列が与えられたときに、出力として想定される結果を出力するプログラムを提出せよ。

Fibonacci 関数の要件

```
def Fibonacci(n):
    """n番目のフィボナッチ数を再帰計算により行う関数
    Args:
        n(int): 1以上の整数値
    Returns:
        int: n番目のフィボナッチ数
    Examples:
        >>>Fibonacci(1)
        1
        >>>Fibonacci(6)
        8
    """
```

実行例 1

```
print(Fibonacci(1))
```

出力例 1

```
1
```

実行例 2

```
print(Fibonacci(6))
```

出力例 2

```
8
```

実行例 3

```
print(Fibonacci(30))
```

出力例 3

```
832040
```

問 2　Date クラスを作成して上の説明で扱ったメソッドを以下の要件を満たすように定義し、入力に各メソッドを呼び出すプログラムの文字列を与えられたときに、出力に想定される結果を出力す

るプログラムを提出せよ。

Date クラスの要件

```
class Date:
    def __init__(self,y,m,d):
        """初期化メソッド。year,month,dayをインスタンス変数として初期化する
            Args:
                y(int): インスタンス変数yearをyで初期化する
                m(int): インスタンス変数monthをmで初期化する
                d(int): インスタンス変数dayをdで初期化する
        """
    def is_new_year_day(self):
        """元日かどうかを判定するメソッド
            Args:
                None
            Returns:
                bool: インスタンスが元日かどうか
        """
    def __str__(self):
        """インスタンスの示す年月日を表示する文字列
            Args:
                None
            Returns:
                str: 「このインスタンスはX年Y月Z日です。」という形式の文字列。
        """
    def __gt__(self):
        """インスタンスが示す年月日が与えられた入力の年月日より後かどうか判定する。
        Args:
            rhs(Date): 比較対象のDateインスタンス
        Returns:
            bool: rhsとselfが示す年月日が等しい場合True,そうでない場合False
        """
    def __eq__(self, rhs):
        """インスタンスが示す年月日が与えられた入力の年月日と等しいかどうか判定する。
        Args:
            rhs(Date): 比較対象のDateインスタンス
        Returns:
            bool: rhsとselfが示す年月日が等しい場合True,そうでない場合False
        """
```

すなわち、以下のような実行例に対し標準出力として出力例が出てくるようにせよ。

実行例 1

```
print(Date(1996,2,3))
```

出力例 1

```
このインスタンスは1996年2月3日です。
```

実行例 2

```
print(Date(1996,2,3).is_new_year_day())
```

出力例 2

```
False
```

問 3　上の説明で登場した Date クラスを継承した JDate クラスを作成し、__init__メソッドと__str__メソッドをオーバーライドして、以下のような入力を受け付けて出力を返すようにせよ。

JDate クラスの要件

```
class JDate(Date):
    def __init__(self,g,y,m,d):
        """
            Args:
                g(str):
                y(int): インスタンス変数yearをgとyから西暦を計算し、その値で初期化する
                m(int): インスタンス変数monthをmで初期化する
                d(int): インスタンス変数dayをdで初期化する
            Returns:
                self
        """
    def __str__(self):
        """インスタンスの示す年月日を表示する文字列
            Args:
                None
            Returns:
                str: 「このインスタンスは元号X年Y月Z日です。」という形式の文字列
        """
```

入力例 1

```
print(JDate("昭和",62,1,1).is_new_year_day())
```

出力例 1

```
True
```

入力例 2

```
print(JDate("昭和",62,2,3) == Date(1987,2,3) )
```

出力例 2

```
True
```

入力例 3

```
print(JDate("平成",3,4,30))
```

出力例 3

```
このインスタンスは平成3年4月30日です。
```

問 4　鉄道路線クラス (Line) と各メソッドを以下の要件を満たすように定義し、次の駅 (next)、前の駅 (prev)、次の急行駅 (next_exp)、前の急行駅 (prev_exp) が計算できるようにせよ。初期化には全駅のリストと急行が止まる駅のリストを受けとる。

Line クラスの要件

```
class Line:
    def __init__(self, arr1, arr2):
        """初期化メソッド
            Args:
               arr1(list): 全駅のリスト
               arr2(list): 急行が止まる駅のリスト
        """
    def next(self, s):
        """入力された駅の次の駅を出力する
            Args:
                s(str) : 駅の名前
            Returns:
                str: 次の駅の文字列
        """
    def prev(self, s):
        """入力された駅の前の駅を出力する
            Args:
                s(str) : 駅の名前
            Returns:
                str: 前の駅の文字列
        """
    def next_exp(self, s):
        """入力された駅の次の駅を出力する
            Args:
                s(str) : 駅の名前
            Returns:
                str: 次の駅の文字列
        """
    def prev_exp(self, s):
        """入力された駅の前の駅を出力する
            Args:
                s(str) : 駅の名前
            Returns:
                str: 前の駅の文字列
        """
```

入力例 1（二つのリストの入力は長いので省略してある）

```
Line(["渋谷駅","神泉駅",..,"吉祥寺駅"],["渋谷駅","下北沢駅",..,"吉祥寺駅"]).next("渋谷駅")
```

出力例 1

```
神泉駅
```

入力例 2：（二つのリストの入力は長いので省略してある）

```
Line(["渋谷駅","神泉駅",..,"吉祥寺駅"],["渋谷駅","下北沢駅",..,"吉祥寺駅"]).next_exp("渋谷駅")
```

出力例 2

```
下北沢駅
```

入力例 3：（二つのリストの入力は長いので省略してある）

```
Line(["渋谷駅","神泉駅",..,"吉祥寺駅"],["渋谷駅","下北沢駅",..,"吉祥寺駅"]).prev_exp("明大前駅")
```

出力例 3

```
下北沢駅
```

問 5　鉄道路線クラス (Line) を継承した Line2 クラスを作成し、全駅リストにない駅についてこれらの関数を使った場合エラーメッセージを以下のように出力するようにせよ。また、全駅リストにはあるが急行駅リストにはない駅について next_exp または prev_exp を呼び出した場合、別のエラーメッセージを出力するようにせよ。さらに、始発駅に前の駅を求めたり、終着駅に次の駅を求めた場合、別のエラーメッセージを出力するようにせよ（例を参照すること）

入力例 1（二つのリストの入力は長いので省略してある）

```
Line2(["渋谷駅","神泉駅",..,"吉祥寺駅"],["渋谷駅","下北沢駅",..,"吉祥寺駅"]).prev_exp("明大駅")
```

出力例 1

```
no such station exists.
```

入力例 2

```
Line2(["渋谷駅","神泉駅",..,"吉祥寺駅"],["渋谷駅","下北沢駅",..,"吉祥寺駅"]).prev_exp("神泉駅")
```

出力例 2

```
express does not stop at this station.
```

入力例 3

```
Line2(["渋谷駅","神泉駅",..,"吉祥寺駅"],["渋谷駅","下北沢駅",..,"吉祥寺駅"]).prev_exp("渋谷駅")
```

出力例 3

```
input is a terminal station.
```

問 6　複素数クラス (Comp) を定義せよ。初期化には 2 つの整数型を受け取り、print 関数で”X + Yi” などと出力されるようにする

Comp クラスの要件

```
class Comp:
    def __init__(self, a, b):
        """初期化メソッド
            Args:
                a(int): 実部を表すインスタンス変数reをaで初期化する
                b(int): 虚部を表すインスタンス変数imをbで初期化する
        """
    def __str__(self):
        """
            Returns:
                str: "X + Yi"あるいは"X - Yi"の形式の文字列。Xは実数部でYは虚数部。純虚数の場合 "0 + 3i"などの形式、実数の場合はそのまま。
        """
```

入力例 1

```
print(Comp(2,2))
```

出力例 1

```
2 + 2i
```

入力例 2

```
print(Comp(-3,-2))
```

出力例 2

```
-3 - 2i
```

入力例 3

```
print(Comp(-3,0))
```

出力例 3

```
-3
```

問 7　複素数クラス (Comp) に対し、絶対値に基づく大小比較が計算できるようにせよ。

Comp クラスの要件（つづき）

```
class Comp:
    def __eq__(self, rhs):
        """
            Args:
                rhs(Comp): 比較対象のインスタンス
            Returns:
                bool: 絶対値が等しいかどうか
        """
    def __ge__(self,rhs):
        """
            Args:
                rhs(Comp): 比較対象のインスタンス
            Returns:
                bool: 絶対値が比較対象のインスタンス以上かどうか
        """
```

入力例 1

```
print(Comp(3,4) == Comp(4,3) == Comp(5,0))
```

出力例 1

```
True
```

入力例 2

```
print(Comp(3,3) < Comp(4,3) < Comp(5,0))
```

出力例 2

```
False
```

問 8　複素数クラス (Comp) に対し、加算と乗算 (+,*) が計算できるようにせよ。これらの演算子は__add__メソッドおよび__mul__メソッドを定義すれば使えるようになる。

Comp クラスの要件（つづき）

```
class Comp:
    def __add__(self, rhs):
        """
            Args:
               rhs(Comp): 足す対象のインスタンス
            Returns:
               Comp: 和の結果を表すCompインスタンス
        """
    def __mul__(self,rhs):
        """
            Args:
               rhs(Comp): 掛ける対象のインスタンス
            Returns:
               Comp: 積の結果を表すCompインスタンス
        """
```

入力例 1

```
print(Comp(1,1) + Comp(0,3))
```

出力例 1

```
1 + 4i
```

入力例 2

```
print(Comp(1,2) * Comp(0,3))
```

出力例 2

```
-6 + 3i
```

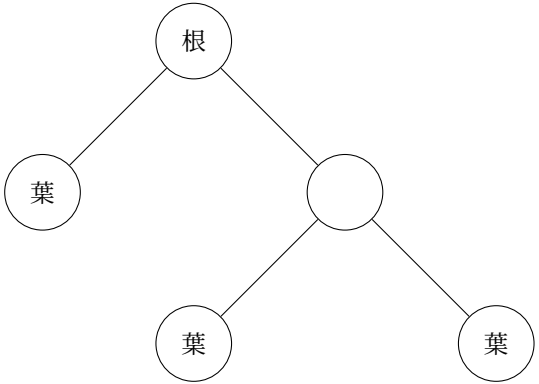
第 3 章

木

木 (tree) は **節点** (node) と呼ばれる有限個の要素の集合が階層構造を持つ場合にその構造を表現する抽象データ型である[*1]。本章では節点に数値か演算子を値として持たせることで一つの木が一つの数式 (の項) を表す**式木** (expression tree) を題材に、木に関する操作のアルゴリズムを学ぶ。節点と節点の関係の構造を指すときは**木構造** (tree structure)、親、子、孫の層構造を指すときは**階層構造** (hierarchical structure) という。

節点の集合が空集合である場合も特別に木とみなし**空木**と呼ぶことにする。木が空木でない場合、**根** (root) と呼ばれる特別な節点が存在する。根でない節点には必ずその節点の**親** (parent) と呼ばれる節点がひとつ存在し、親と呼ばれる節点は対象の節点よりも一つ階層が上であることを意味する。節点 n が節点 n' の親であるとき、節点 n' は節点 n の**子** (child) であるという。子を持たない節点を**葉** (leaf) という。木に含まれる任意の節点に対し、根からたどるために必要な親子関係の反復回数をその節点の**深**

[*1] 木の応用としては、検索木、ヒープ木、ハフマン木、trie、splay tree などがある。



さ (depth) という。根の子は深さが 1 であり、根自身は深さが 0 である。与えられた木の全ての葉の深さの最大値をその木の**高さ** (height) という。根だけの木の高さは 0、根の下に葉がある木の高さは 1 となる。空木の高さは-1 と定義される。

木に関する用語の一部には家系図由来のものがある。例えば、共通の親を持つ子同士を**兄弟姉妹** (sibling)、子、孫、... をまとめて**子孫** (descendant)、親、祖親、... をまとめて**祖先** (ancestor) という。

節点に対して、その節点の子の数を**出次数** (out degree) という。出次数は節点によって違う。葉は出次数が 0 の節点である。葉以外の出次数が 2 の木を**二分木** (binary tree) と呼ぶ。

木に含まれる任意の節点に対し、その節点と子孫全体は、その節点を根とする木とみなすことができる。この木を元の木の**部分木** (subtree) という。これはファイルシステムを木構造と見た時のサブフォルダに該当する。

子の間に順序が定められている木を**順序木** (ordered tree)、子の間に順序が定められていない木を**非順序木** (unordered tree) という。

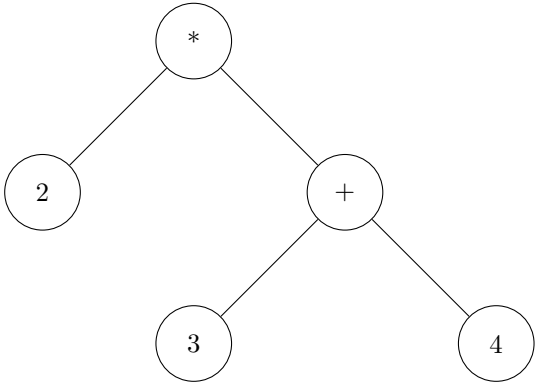


図3.1　$2 * (3 + 4)$ を表す式木

3.1　式木の構成

数値と演算 (加減と乗算) と括弧のみからなる数式を二分木で表すことを考える。+ や − などの演算子を要素とする節点をつくり、被演算子をその子とする。− は左の子から右の子を引き算するものであり逆ではない。つまり、子の間に順序があるので、順序木で表す。式 $2 * (3 + 4)$ を式木で表すと図 7.1のような木となる。

式木を Python で実装する場合以下のような方法が考えられる。まず、節点を表すオブジェクトはクラス Node のインスタンスとして表現し、節点がもつ値とともに、親と左の子と右の子をそれぞれ left と right という名前のインスタンス変数に持つようにする。そして、一つの木を表現するオブジェクトはクラス Tree のインスタンスとみなす。木はインスタンス変数として根に対応する節点のオブジェク

トのみを持つ。すなわち、以下のような Tree クラスと Node クラスを用意する。

インタラクティブモードでの実行例

```
>>>class Node:
...     def __init__(self, v):
...         self.value = v
...         self.left = None
...         self.right = None
...
>>>class Tree:
...     def __init__(self, root_node):
...         self.root = root_node
...
```

Noneは「何もない」ことを示す特別な（Nontype 型の）値である。

図 7.1に対応する式木は以下のようにして構成できる。

インタラクティブモードでの実行例（つづき）

```
>>>n1 = Node('2')
>>>n2 = Node('*')
>>>n3 = Node('3')
>>>n4 = Node('+')
>>>n5 = Node('4')
>>>
>>>n2.left, n2.right = n1, n4
>>>n4.left, n4.right = n3, n5
>>>t = Tree(n2)
```

3.2　式木の走査

木のインスタンスの根から全ての節点を辿りその情報を表示させたいことがある。上の動作例で最後に構成した木 t が正しく構成されているかを確認する場合、あるいは式木の節点を全て辿って全体が示

す数式を計算するプログラムを書きたい場合などがわかりやすい例だろう。木に対して全ての節点を辿ることを木を**走査** (traverse) する、という。二分木の走査の方法は複数考えられるが、再帰により実装が簡単な木の走査方法の一つに **深さ優先探索** (depth first search) がある。深さ優先探索はある節点の子孫を葉に到達するまで辿って、子孫を辿りきって初めて節点の兄弟姉妹へ進む。深さ優先探索は以下のように再帰的に呼び出される dfs 関数によって実現される。

```
1 def dfs(n):
2     if n == None:
3         return
4     else:
5         dfs(n.left)
6         dfs(n.right)
```

ただし、この関数はただ節点を辿ることだけを行い、結果としても (dfs 関数を呼び出す以外のことは) 何もしない。上述した Tree クラスのインスタンスの根に関してこの dfs 関数を実行する (`dfs(t.root)` を実行する) と、最初に`n2`に対する dfs 関数が呼ばれ、その中で実引数が`n1`である dfs 関数が呼ばれる。この dfs 関数の中でさらに`n1`の左の子である`None`に対して dfs 関数が呼ばれる。この dfs 関数の実行では`n==None`は真であり、`None`を返す。`n1`が実引数である dfs 関数の最後の処理は`n1`の右の子である`None`に対する dfs 関数の評価である。これも同様に`None`を返す。ここまでの処理で`n2`に対する dfs 関数の 5 行目の実行が終了し、6 行目が実行される。図3.2に以降の処理も含め全体の流れを図示している。

深さ優先探索をしながら節点の値を出力する方法は、dfs が自分自身を引数とした呼び出しを 1) 行った直後に出力する場合、2) 終えて返り値を返す直前に出力する場合の二通りが考えられる。これらをそれぞれ **preorder**（行きがけ順）、**postorder**（帰りがけ順）という。

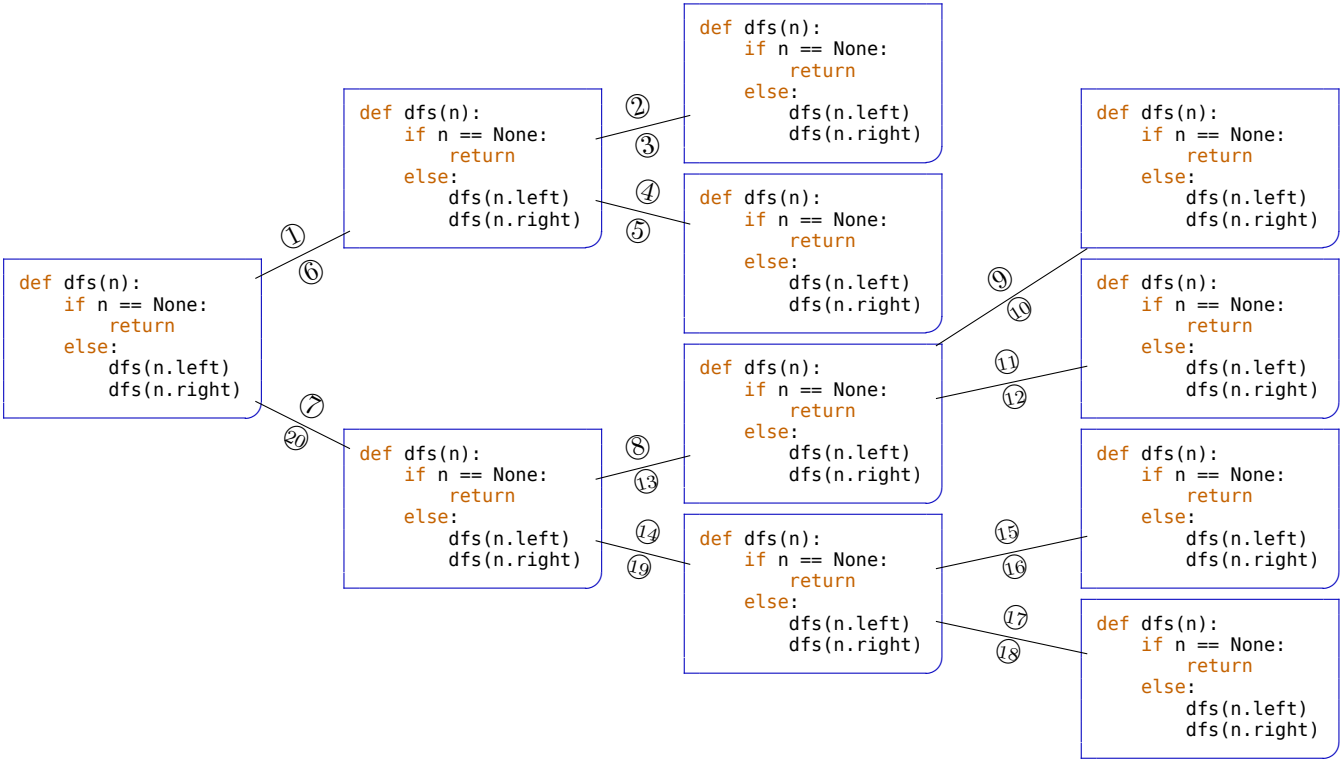


図3.2　式木 2*(3+4) に対する深さ優先探索の流れ。図中の番号は式木 2*(3+4) に対応する Tree オブジェクト t に対して dfs(t.root) とした場合の関数呼び出しと関数終了の順番を示している。postorder で節点の情報を出力する場合は、例えば根の情報は①の直前に出力する。preorder の場合は⑳の直後に出力する。inorder の場合は⑥と⑦の間に出力する

インタラクティブモードでの実行例（つづき）

```
>>>def preorder(n):
...     if n == None:
...         return
...     else:
...         print(n.value, end=" ")
...         preorder(n.left)
...         preorder(n.right)
...
```

インタラクティブモードでの実行例（つづき）

```
>>>def postorder(n):
...     if n == None:
...         return
...     else:
...         postorder(n.left)
...         postorder(n.right)
...         print(n.value, end=" ")
...
```

さらに、dfs 関数が自分自身を引数とした呼び出しの中で、左の子に関する呼び出しが実行された後、右の子に関する呼び出しが実行される前に出力などの処理をする方法も考えられ、これは**inorder**（通りがけ順）と呼ばれる。

インタラクティブモードでの実行例（つづき）

```
>>>def inorder(n):
...     if n == None:
...         return
...     else:
...         inorder(n.left)
...         print(n.value, end=" ")
...         inorder(n.right)
...
```

式木の場合を考えると葉は値となるので、preorder や postorder の出力は二つの異なる式木が同一に

なることはない。すなわち、preorder や postorder の表示は括弧が不要となる。一方、inorder では二つの異なる式木が同じ表示となることがある。

インタラクティブモードでの動作例（つづき）

```
>>> preorder(t.root)
* 2 + 3 4
>>> postorder(t.root)
2 3 4 + *
>>> inorder(t.root)
2 * 3 + 4
```

上記の例のような単純な実装では最後にも空白文字が表示されることに注意。

3.3　式木の計算

Tree に関してその式木の計算結果を求めるメソッド evaluate() を定義してみよう。計算結果を再帰的に辿ることで簡単に実装することができる。

```
#class Node:以降のブロック内に以下のようなメソッドを追加する
    def evaluate(self):   # 式の値を再帰的に計算する
        if self.left == self.right == None:   # 葉なら、
            return float(self.value)    # その値を返す（文字列を浮動小数点数へ変換）
        else:
            l = self.left.evaluate()  # 左の子の値
            r = self.right.evaluate() # 右の子の値
            if self.value == "+": #演算子は「+」「-」「*」のみを許すとする。
                return l+r
            elif self.value == "-":
                return l-r
            elif self.value == "*":
                return l*r
            else:
                print("値が不正です。：", self.value)
#class Tree:以降のブロック内に以下のようなメソッドを追加する
    def evaluate(self):
        return self.root.evaluate()
```

演習

UTOL 上の課題は改善点を自分で考えて実装してみた人のための報告場所です。UTOL 上に提出しても課題に正解したかどうかは評価されないので注意してください。UTOL 上に提出する際は、改善点をプログラムのコメント中または提出時のコメントに明記すること。

DOMjudge 中の Contest 名を prog_3 へ変更すれば以下の問題が現れる。各問題に対し、要件をよく読んで受け付けた入力に対応する出力がされるプログラムを作成し、提出せよ。template を用意してあるので、これをダウンロードして加筆するのがよい。pdf からコピー・貼り付けを行うと予期せぬエラーがでることがあるので自分で書いていくことを推奨する。

問 1　上で説明した Node クラスの初期化メソッド`__init__`および`__str__`メソッドを実装せよ。

要件

```
class Node:
    def __init__(self, x):
        """初期化メソッド。引数xをvalueへ代入する。
        Args:
            x(str) : 節点に割り当てる文字列。値は数値とは限らない。
        """
    def __str__(self):
        """表示のための特殊メソッド。valueの値を表示する。
        Args:
            None
        Returns:
            str: x.valueの値が文字列であればその文字列、Noneであれば空の文字列""を返す。
        """
```

入力例 1

```
print(Node("*"))
```

出力例 1

```
*
```

入力例 2

```
print(Node(None))
```

出力例 2（空文字「""」が出力される）

問 2　Node クラスに二つの Node インスタンスを引数として、それぞれを自分自身の右の子・左の子とするような副作用をもち、自分自身を返り値とする `set_nodes` メソッドを追加せよ。

要件

```
class Node:
    def set_nodes(self, x, y):
        """引数のNodeインスタンスを自分自身の左の子・右の子とする
        Args:
            x : 左の子となるNodeインスタンス
            y : 右の子となるNodeインスタンス
        Returns:
            Node: 子を持つように変更された自分自身
        Examples:
            >>> n = Node("*").set_nodes(Node("3"),Node("4"))
            >>> n.left.value
            3
            >>> n.right.value
            4
        """
```

入力例 1

```
print(Node("*").set_nodes(Node("3"),Node("4")).left)
```

出力例 1

```
3
```

入力例 2

```
print(Node("*").set_nodes(Node("3"),Node("4")).right)
```

出力例 2

```
4
```

問 3　3.2節で説明されている 3 つの関数 `preorder, postorder, inorder` を実装せよ。

要件

```
def preorder(n):
    """節点nを根とする（部分）木の内容を行きがけ順で標準出力に表示する。
    Args:
        n(Node): Node インスタンス
    Returns:
        None
    Examples:
        >>> n = Node("*").set_nodes(Node("3"),Node("4"))
        >>> preorder(n)
        * 3 4
    """
def postorder(n):
    """節点nを根とする（部分）木の内容を帰りがけ順で標準出力に表示する。
    Args:
        n(Node): Node インスタンス
    Returns:
        None
    Examples:
        >>> n = Node("*").set_nodes(Node("3"),Node("4"))
        >>> postorder(n)
        3 4 *
    """
def inorder(n):
    """節点nを根とする（部分）木の内容を通りがけ順で標準出力に表示する。
    Args:
        n(Node): Node インスタンス
    Returns:
        None
    Examples:
        >>> n = Node("*").set_nodes(Node("3"),Node("4"))
        >>> inorder(n)
        3 * 4
    """
```

入力例 1

```
preorder( Node("*").set_nodes(Node("3"),Node("+").set_nodes(Node("4"),Node("5"))) )
```

出力例 1

```
* 3 + 4 5
```

入力例 2

```
postorder( Node("*").set_nodes(Node("3"),Node("+").set_nodes(Node("4"),Node("5"))) )
```

出力例 2

```
3 4 5 + *
```

入力例 3

```
inorder( Node("*").set_nodes(Node("3"),Node("+").set_nodes(Node("4"),Node("5"))) )
```

出力例 3

```
3 * 4 + 5
```

問 4　Tree クラスにインスタンスに含まれる節点数を出力する`size`メソッドを実装せよ

要件

```
class Tree:
    def size(self):
        """節点数を出力するメソッド。
        Args:
            None
        Returns:
            int: 木がもつ節点の総数
        Examples:
            >>> n = Node("*").set_nodes(Node("3"),Node("4"))
            >>> Tree(n).size()
            3
            >>> n2 = Node("+").set_nodes(Node("9"),n)
            >>> Tree(n2).size()
            5
        """
```

入力例 1

```
print(Tree(Node("*").set_nodes(Node("3"),Node("4"))).size())
```

出力例 1

```
3
```

入力例 2

```
print(Tree(None).size())
```

出力例 2

```
0
```

問 5　Tree クラスに式木がインスタンスとして表現されている場合にその表式を出力する__str__メソッドを実装せよ。木は必ず二分木になっていることを仮定してよい。ここでは演算子や数値の間に空白文字を入れないこと。すべてを含む「()」も必ずつけるものとする。空木には空の文字列を返すこと。

要件

```
class Tree:
    def __str__(self):
        """表示のためのメソッド。木の内容を表示する。
        Args:
            None
        Returns:
            str: 自分自身の式木としての内容。空木には空の文字列を返す。
        Examples:
            >>> print(Tree(None))

            >>> n = Node("*").set_nodes(Node("3"),Node("4"))
            >>> print(Tree(n)))
            (3*4)
            >>> n2 = Node("+").set_nodes(Node("9"),n)
            >>> print(Tree(n2))
            (9+(3*4))
        """
```

入力例 1

```
print(Tree(Node("*").set_nodes(Node("3"),Node("4"))))
```

出力例 1

```
(3*4)
```

入力例 2

```
print(Tree(Node("*").set_nodes(Node("3"),Node("*").set_nodes(Node("7"),Node("9")))))
```

出力例 2

```
(3*(7*9))
```

問 6　上で説明した高さの定義に従って、Tree クラスに自分自身の高さを返す `height` メソッドを定義せよ。

```
def height(self):
        """木の高さを返すメソッド
        Args:
            None
        Returns:
            int: 高さ
        Examples:
            >>> print(Tree(None).height())
            -1
            >>> n = Node("*").set_nodes(Node("3"),Node("4"))
            >>> t = Tree(n)
            >>> print(t.height())
            1
            >>> n2 = Node("+").set_nodes(Node("9"),n)
            >>> t2 = Tree(n2)
            >>> print(t2.height())
            2
        """
```

入力例 1

```
print(Tree(Node("*").set_nodes(Node("3"),Node("4"))).height())
```

出力例 1

```
1
```

入力例 2

```
print(Tree(None).height())
```

出力例 2

```
-1
```

第 4 章

パーサー

パーサーとは一般に文字列を構造へと変換する機構のことである。例えば前章で説明したようにノードをつなげて式木を構築するのではなく、数式を表す文字列から式木を構築することを考える。入力文字列については簡単のため、数値は全て一文字とし、すべての数字と演算子の間は空白文字が一つだけはいっているとする。

4.1 スタック

最初に**スタック** (stack) という抽象データ型 (Abstract data type) を説明する。抽象データ型とは言語や実装に依存することなく、その型のインスタンスに対してできる操作とその計算量を記述したものである。

スタックはデータの要素を追加する push 操作と取り出しを行う pop 操作ができる抽象データ型である。取り出しは最後に追加された要素が取り出される。追加された要素が一つもないスタックからは取

り出しはできない。Python のリストはスタックの具体的な実装といえ、append メソッドは push 操作を実現しており、pop メソッドは pop 操作を実現する。他にもスタックには取り出さずに要素の情報のみを取得する peek 操作やスタックが空であるかを判定する is_empty 操作が実装されることが多い。

stack_example.py の内容

```
1 s = [] # リストをスタックとしてみる。
2 s.append(1) # appendがpush操作に対応している
3 s.append(2)
4 s.append(3)
5 v = s.pop() #popがpop操作に対応している。最後に追加した要素が取り出される。
6 print(v,s)  #vは3, sは[1,2]になっている
```

stack_example.py の動作例

```
1 $ python3 stack_example.py
2 3 [1, 2]
```

4.2　postorder のパーサー

数式が与えられたときに、それが構成する式木に対して postorder で節点の値を出力する場合に出てくる順番で文字列（数値あるいは演算子）が与えられたとき、式木を構成する`parse_postorder_string`関数を考える。すなわち、この関数は入力として「`1 2 + 3 *`」のように postorder で書かれた式を文字列を受け取り、図4.1のような式木を返す。処理がどのように進むかを入力`1 2 + 3 *`に対して説明すると以下のようになる。まず、要素がそれぞれの値を持つ節点となっている配列 [1, 2, +, 3, *] を構成する。その後この配列に対する反復処理を以下のように行えば、最後にスタックに残ったのが式木 (の根) となる。

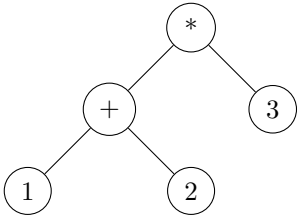


図4.1　postorder で `1 2 + 3 *` が表す式木

1. `1`を見たら、`1`をスタックに追加する。
2. `2`を見たら、`2`をスタックに追加する。
3. `+`を見たら、演算の対象 (`1`と`2`) をスタックから取り出し、それらを`+`の左右の子としてスタックに追加する。
4. `3`を見たら、`3`をスタックに追加する。
5. `*`を見たら、演算の対象 (`+`と`3`) をスタックから取り出し、それらを`*`の左右の子としてスタックに追加する。

これを図示すると図4.2のようになる。

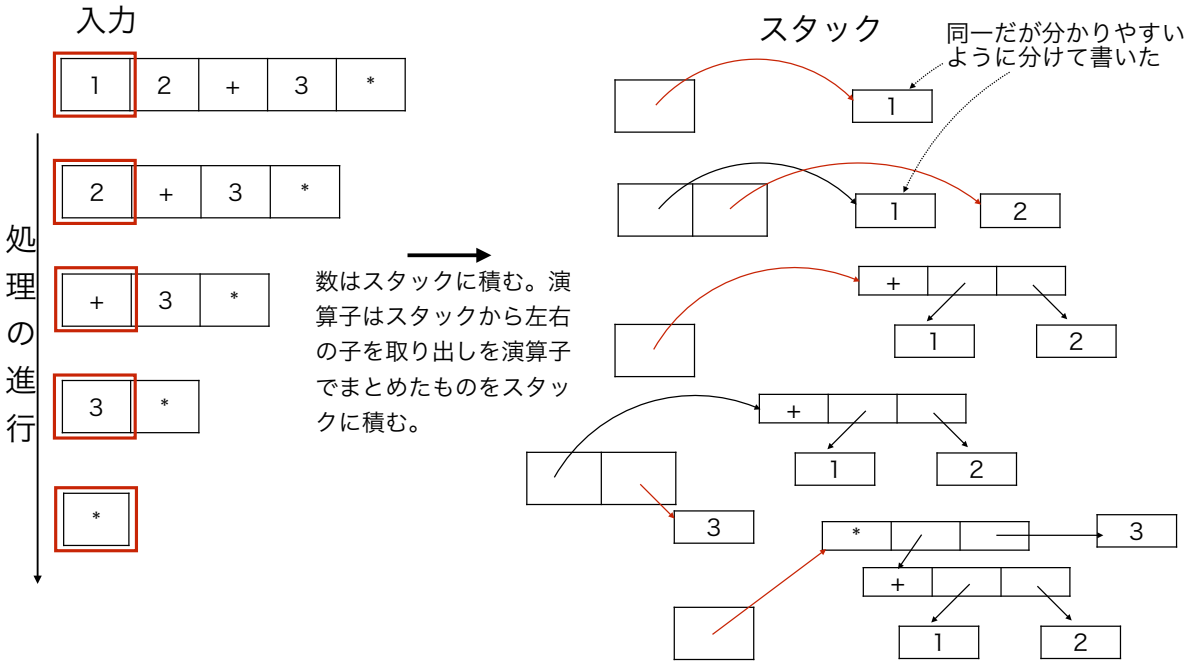


図4.2　postorder のパーサーのふるまいの例

これを疑似アルゴリズムとして書くと以下のようになる。

```
1: procedure PARSE_POSTORDER_STRING(input)
2:     for input の先頭から順に、要素 e を取り出して以下を行う。 do
3:         if e が演算子 then
4:             スタックから要素を取り出し (pop)、それを e の右の子とする。
5:             スタックから要素を取り出し (pop)、それを e の左の子とする。
6:             e をスタックに追加する (push)。
7:         else  # e は数値
8:             e をスタックに追加する。
9:     return スタックの先頭
```

postorder で書かれた文字列からその式木を表現する Tree クラスのオブジェクトを自分自身に構成するインスタンスメソッドは Tree クラス内で以下のようにすれば達成できる。

```
1  # Treeクラスに以下のインスタンスメソッドが実装されているとする
2  def parse_postorder_string(self,line):
3      node_array = [ Node(x) for x in line.split(" ")]
4      stack = []
5      for n in node_array:
6          if n.value in "+-*":
7              r,l = stack.pop(),stack.pop()
8              stack.append(n.set_nodes(l,r))
9          else:
10             stack.append(n)
11     self.root=stack.pop()
12     return self
```

文字列には引数の値で区切ったリストを返す組み込みメソッドsplit()が用意されている。

インタラクティブモードでの動作例

```
1  >>> t = Tree(None) # 空の木を構成しておく
2  >>> t.parse_postorder_string("1 2 + 3 *") #引数に対応する木を自分自身に構成する
3  >>> print(t) # 前章の問５でやったもの
4  ((1+2)*3)
```

4.3　inorder のパーサー

例えば inorder の記法が `3 * 4 + 4` となる木は * が根になる 3 * (4 + 4) という木と + が根になる (3 * 4) + 4 という木の二通りあり、一意に定めることができない (図4.3参照)。ここでは`3 * 4 * 3`や`3 - 4 - 3`などの表現については 3 * (4 * 3) や 3 - (4 - 3) となるように、さらに`3 * 4 + 3`や`3 + 4 * 3` は加減乗除の優先順位にあわせて (3 * 4) + 3 や 3 + (4 * 3) となるように木を構成するようなパーサーを考える。この場合、文字列に対しては以下のような再帰的に定められた処理をすればよい。

1. 文字列が数値である場合はその数値を値とする節点を構成する。
2. 文字列が +, − を含まない文字列であれば 1. の処理をするか、あるいは数値を左の子とするような * を値とする節点を構成する（右の子は +− を含まない文字列である）
3. 一般の文字列は 2. の処理をするか、あるいは +− を含まない文字列を左の子とするような + か-を値とする節点を構成する（右の子は一般の文字列である）

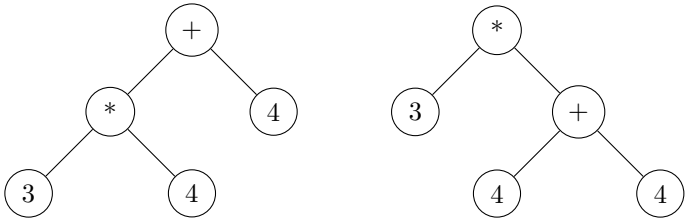


図4.3　inorder で `3 * 4 + 4` となる 2 つの式木

1.2.3. に対応するそれぞれの処理をここでは F,T,E と呼ぶことにする。それぞれが対象とする文字列の例は以下のようなものである。

F: 0,1,2,3,4,5,6,7,8,9
T: 2*3 , 3*3*3 , 2*3*4*3,...
E: 2*3+4, 4-1+2*3 , 1-2*3 , ...

E の処理を再帰的に用いることで内部で T の処理が再帰的に呼び出される。それらの処理の中で適宜 F の処理が呼び出され、最終的にはどんな正しい文も解析（木にもどす）ことができる。

これを**BNF**(Backus Naur Form) と呼ばれる文法の記述法に則って書き下すと以下のようになる。

- F ::= 数値
- T ::= $F \mid F * T$
- E ::= $T \mid T + E \mid T - E$

左辺は右辺のどれかからなるので、どれかに置き換えていくと展開できる。例えば`3 * 3 * 3`という文を解析する時、$E \to T \to F*T \to F*F*T \to F*F*F$ という形で、E から始まり最終的には全て F からなる式になる。`3 + 3 * 3` であれば、$E \to T+E \to T+T \to F+T \to F+F*T \to F+F*F$ という形で、E から始まり最終的には全て F からなる式になる。

F, T, E それぞれの文法規則を適用して左辺を右辺に変換する関数をそれぞれ f,t,e とすると、e,t,f はそれぞれ以下のような関数になる。e が最初に適用を試みるべき規則であり、最終的には根となるべき節点がスタックに残っている。前節と同様に式木を表現する Tree クラスのオブジェクトを自分自身に構成するインスタンスメソッドは以下のように実装できる。

```
1 # Treeクラスに以下のインスタンスメソッドが実装されているとする
2     def parse_inorder_string(self, line):
```

```
        node_array = [Node(x) for x in line.split(" ")]
        self.root = self.e(node_array)
        return self

    def f(self, node_array):
        if len(node_array) == 0:
            return None
        elif node_array[0].value in "1234567890":
            return node_array.pop(0)
        else:
            print("syntax error")
            return

    def t(self, node_array):
        n = self.f(node_array)
        if len(node_array) != 0 and node_array[0].value == "*":
            return node_array.pop(0).set_nodes(n,self.t(node_array))
        else:
            return n

    def e(self, node_array):
        n = self.t(node_array)
        if len(node_array) != 0 and node_array[0].value in "+-":
            return node_array.pop(0).set_nodes(n, self.e(node_array))
        else:
            return n
```

演習

UTOL 上の課題は改善点を自分で考えて実装してみた人のための報告場所です。UTOL 上に提出しても課題に正解したかどうかは評価されないので注意すること。UTOL 上に提出する際は、改善点をプログラムのコメント中または提出時のコメントに明記すること。

DOMjudge 中の Contest 名を prog_3 へ変更すれば以下の問題が現れる。各問題に対して受け付けた入力に応じて結果が変わるようにプログラムを作成し、提出せよ。UTOL から template をダウンロードして加筆するのを推奨する。pdf からコピー・貼り付けを行うと予期せぬエラーがでることがあるの

で自分で書いていくことを推奨する。

問 1　4.2節で述べたように preorder で書かれた文字列を受け取り、表される式木を自分自身に構築して返すインスタンスメソッドを`parse_postorder_string`メソッドとして Tree クラスに実装せよ。

要件

```
class Tree:
    def parse_postorder_string(self,line):
    """postorderで書かれた文字列をパースする
        Args:
            line(str): postorderで書かれた文字列
        Returns:
            Tree: 自分自身
        Examples:
            >>> t = Tree(None)
            >>> print(t.parse_postorder_string("1 2 + 4 * 3 +")) #前回の問5で__str__を実装した場合
            (((1+2)*4)+3)
    """
```

入力例 1

```
print(Tree(None).parse_postorder_string("4 5 * 4 -"))
```

出力例 1

```
((4*5)-4)
```

問 2　4.3節で述べたようにように inorder で書かれた文字列を受け取り、表される式木を自分自身に構築して返すインスタンスメソッドを`parse_inorder_string`メソッドとして Tree クラスに実装せよ。

要件

```
class Tree:
    def parse_inorder_string(self,line):
    """ inorderの記法で書かれた文字列をパースする。乗除算は加減算より優先度が高く、同じ優先度の場合、先の演算子が優先する。優先された演算子が根に近くなる。
        Args:
            line(str): 文字列
        Returns:
            Tree: 自分自身
        Examples:
            >>> t = Tree(Node(None))
            >>> t.parse_inorder_string("2 - 2 + 4 * 3 + 2")
            >>> print(t) #前回の問５で実装した場合
            (2-(2+((4*3)+2))
    """
```

入力例１

```
print(Tree(None).parse_string("2 - 2 + 4 * 3 + 2"))
```

出力例１

```
(2-(2+((4*3)+2))
```

問 3　3.3節で述べた evaluate で冪乗演算子「^」を追加したものを実装せよ。式木上の葉の数値は絶対値が０以上の９以下の整数のみ、演算子は「+」「-」「*」「^」のみを許すとする。空木には None を返し、計算の過程で`3^(0-2)`などの負の冪乗が現れた時も None を返す。

※Python の文法では冪乗の計算は**を使う。

要件

```
class Tree:
    def evaluate(self):
        """式木の計算結果を出力する。式木上の数値は絶対値が 0 以上の 9 以下の整数のみ、演算子は「+」「-」「*」「^」のみを許すとする。負の冪乗は扱わない。
        Args:
            None
        Returns:
            int: 自分自身の式木としての式の計算結果。空木にはNoneを返す。
        Examples:
            >>> n = Node("*").set_nodes(Node("3"),Node("2"))
            >>> Tree(n).evaluate()
            6
            >>> n2 = Node("-").set_nodes(Node("9"),n)
            >>> Tree(n2).evaluate()
            -3
            >>> Tree( Node("^").set_nodes(Node("2"),n) ).evaluate()
            64
            >>> Tree( Node("^").set_nodes(Node("9"),n2) ).evaluate()
            None
        """
```

入力例 1

```
print(Tree(Node("*").set_nodes(Node("3"),Node("4"))).evaluate())
```

出力例 1

```
12
```

入力例 2

```
print(Tree(Node("^").set_nodes(Node("2"),Node("*").set_nodes(Node("3"),Node("4")))).
evaluate())
```

出力例 2

```
4096
```

入力例 2

```
print(Tree(Node("^").set_nodes(Node("2"),Node("-").set_nodes(Node("3"),Node("4")))).
evaluate())
```

出力例 2（print(None) の結果）

```
None
```

問 4　式木 x の葉のなかに k が現れる回数を返すインスタンスメソッド count を実装せよ。

要件

```
class Tree:
    def count(self,k):
        """式木の葉のなかにkが現れる回数を返す
        Args:
            str: 検索する文字列
        Returns:
            int: 結果。空木にはNoneを返す。
        Examples:
            >>> Tree(None).count("1")
            None
            >>> n = Node("*").set_nodes(Node("3"),Node("4"))
            >>> Tree(n).count("6")
            0
            >>> n2 = Node("+").set_nodes(Node("4"),n)
            >>> Tree(n2).count("4")
            2
        """
```

入力例 1

```
print(Tree(Node("*").set_nodes(Node("3"),Node("4"))).count("3"))
```

出力例 1

```
1
```

入力例 2

```
print(Tree(Node("^").set_nodes(Node("2"),Node("*").set_nodes(Node("3"),Node("4")))).
count("3"))
```

出力例 2

```
1
```

問 5　式木 x と y が同じ式を表しているかどうかを判定するインスタンスメソッド__eq__を実装せ

よ。木構造が一致するかを判定すればよく、$a+b=b+a$ のような代数的な等価性は考えなくて良い。

要件

```
class Tree:
    def __eq__(self,t):
        """自分自身と式木tが同じ式を表しているかどうかを判定する
        Args:
            Tree: 式木
        Returns:
            bool: 要素と構造が等しければTrue、さもなくばFalse
        Examples:
            >>> n = Node("*").set_nodes(Node("3"),Node("4"))
            >>> n2 = Node("*").set_nodes(Node("3"),Node("4"))
            >>> print(Tree(n) == Tree(n2))
            True
            >>> n.value = "-"
            >>> print(Tree(n) == Tree(n2))
            False
        """
```

入力例 1

```
print(Tree(Node("*")) == Tree(Node("*")) )
```

出力例 1

```
True
```

入力例 2

```
print(Tree(None) == Tree(None) )
```

出力例 2

```
True
```

問 6　式木 x を複製するインスタンスメソッド deepcopy() を作れ。ここで複製するということは、t2=t.deepcopy() としたとき、t は t2 と同じ数式を表しているが、t と t2 は節点を共有しないようにするということである。

要件

```
class Tree:
    def deepcopy(self):
        """式木を複製する
        Args:
            str: 検索する文字列
        Returns:
            Tree: 複製結果
        Examples:
            >>> n = Node("*").set_nodes(Node("3"),Node("4"))
            >>> t = Tree(n)
            >>> t2 = t.deepcopy()
            >>> n.set_nodes(Node("30"),Node("40"))
            >>> t2.evaluate()
            12
        """
```

入力例 1

```
print(Tree(Node("*")) == Tree(Node("*")).deepcopy())
```

出力例 1

```
True
```

入力例 2

```
print(Tree(None) == Tree(None).deepcopy() )
```

出力例 2

```
True
```

問 7　c1 と c2 を引数とし、式木の葉に c1 が現れたら c2 に破壊的に置き換えて自分自身を返す`replace`メソッドを実装せよ。例えば、(2+3)*(3+4) の式木 t に t.replace(”3”,”5”) と適用すると、t は (2+5)*(5+4) となり返される。

要件

```
class Tree:
    def replace(self,a,b):
        """式木の特定の値を破壊的に置き換える
        Args:
            a: 検索する文字列
            b: 置換する文字列
        Returns:
            Tree: 複製結果となる自分自身
        Examples:
            >>> n = Node("*").set_nodes(Node("3"),Node("4"))
            >>> t = Tree(n)
            >>> t2 = t.replace("3","0")
            >>> print(t2)
            0+4
            >>> t is t2
            True
        """
```

入力例 1

```
print(Tree(Node("4")).replace("4","5"))
```

出力例 1

```
5
```

入力例 2

```
print(Tree(Node("-").set_nodes(Node("2"), Node("+").set_nodes(Node("2"), Node("4")))).
replace("2","5"))
```

出力例 2

```
(5-(5+4))
```

問 8　c1 と c2 を引数とし、式木の葉に c1 が現れたら c2 に非破壊的に置き換えて新しい木を返す`replace_and_copy`メソッドを実装せよ。例えば、(2+3)*(3+4) の式木 t に対し t.replace_and_copy(”3”,”5”) と適用すると、(2+5)*(5+4) に対応する式木が返される。

要件

```
class Tree:
    def replace_and_copy(self,a,b):
        """式木の特定の値を破壊的に置き換える
        Args:
            a: 検索する文字列
            b: 置換する文字列
        Returns:
            Tree: 置換結果となる新しい木
        Examples:
            >>> n = Node("*").set_nodes(Node("3"),Node("4"))
            >>> t = Tree(n)
            >>> t2 = t.replace_and_copy("3","3")
            >>> print(t2)
            3+4
            >>> t is t2
            False
        """
```

入力例 1

```
print(Tree(Node("4")).replace(_and_copy"4","5"))
```

出力例 1

```
5
```

入力例 2

```
print(Tree(Node("-").set_nodes(Node("2"), Node("+").set_nodes(Node("2"), Node("4")))).
replace_and_copy("2","5")))
```

出力例 2

```
(5-(5+4))
```

第 5 章

探索木

今まで扱ってきた式木は一つの木が数式という一つの情報を表現していた。一方で多くの実用的な場面で表れる木構造は、節点に独立した情報を格納したデータベースの様に扱うことが多い。それは木構造が効率的に特定の節点を探索する様々なアルゴリズムがあるからである。

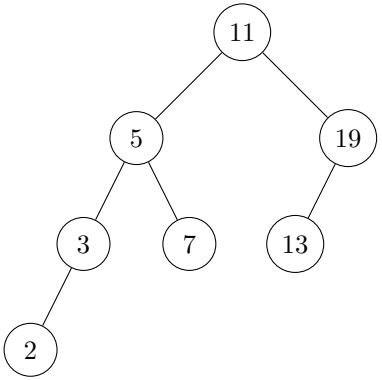
例えばある文章に出現した単語一つ一つが木の節点に対応するような場合などでは、任意の単語がこの木＝文章に含まれるかどうか効率的に判定することができる。このような文脈で情報の探索のために使われる木を**探索木** (search tree) と呼ぶ。ここでは二分木（3章参照）である探索木を扱うが、探索木が二分木に限定されているわけではない。

節点には多くの場合 key と value という 2 種類の変数が格納される。key は探索する際に手がかりとする見出しのような値であり、上の例であれば単語のようなものである。value はデータの内容であり、検索の結果返すべき値を格納する。上の例であればその単語が何回あらわれたか、どこで現れたかなどの情報を持たせることが可能である。あるいは DOMjudge に提出されたプログラムの提出時間が key

でプログラム自身や提出者の情報が value であれば、特定の時間に提出されたプログラムを効率的に検索することなどが可能である。key は文字列である場合が多いが、本章では簡単のため整数値が格納されていると考える。

5.1　二分探索木

木が二分探索木条件を満たすとは **全ての節点において左の子孫の値 $\leq$ 自分自身の値 $\leq$ 右の子孫の値を満たすこと**である。以下は二分探索木条件を満たす例である。



5.1.1　二分探索木の探索

木が二分探索木条件を満たしている間は指定されたデータの探索が以下のように簡単にできる。第3章の深さ優先探索の例では関数とインスタンスメソッドを両方使った実装を紹介した。ここではインターフェースとなるsearchメソッドと内部で Node インスタンスを再帰的な処理を行う_search_by_recursionメソッド[*1]を両方インスタンスメソッドとして実装する例を紹介する。

```
class Node:
    def __init__(self, key, value):
        self.key = key
        self.value = value
        self.left = None
        self.right = None

class BinaryTree:
    def __init__(self):
        self.root = None
    def search(self, key):
        return self._search_by_recursion(self.root, key)
    def _search_by_recursion(self, node, key):
        if node == None:
            return None #ないことが分かった
        if key == node.key:
            return node #データが見つかった
        elif key < node.key: # 左の子を探索
            return self._search_by_recursion(node.left, key)
        else:# 右の子を探索
            return self._search_by_recursion(node.right, key)
```

_search_by_recursionメソッドは、指定されたキーを持つ節点を二分探索木から再帰的に探索する。再帰的な動作に関しては以下のようにベースケースと再帰ステップの 2 つの側面を整理すると理

[*1] 別のインスタンスメソッド内からしか呼び出さないインスタンスメソッドのインスタンスメソッド名は慣例として「_」から始める

解しやすい。

ベースケース： ベースケースは 2 つある。一つ目は、探索が終了した節点が None である（つまり、探索が葉まで達した）場合である。この場合探索は失敗し、メソッドは None を返す。二つ目は、探索が終了した節点のキーが探索対象のキーと一致する場合である。この場合、探索は成功し、メソッドは該当の節点を返す。

再帰ステップ： 探索対象のキーが現在の節点のキーより小さい場合、メソッドは現在の節点の左の節点に対して再帰的に探索を行う。逆に、探索対象のキーが現在の節点のキー以上の場合、メソッドは現在の節点の右の節点に対して再帰的に探索を行う。

`_search_by_recursion`メソッドは各ステップで探索対象のキーと現在の節点のキーを比較することで、探索範囲を半分に絞り込むことができ、大規模な木でも高速に探索を行うことが可能となる。

5.1.2　二分探索木への節点の挿入

節点の挿入を二分探索木条件を満たすように行うことも同様に再帰的に行うことができる。以下のように挿入する節点の親となるべき節点を根から子を再帰的に辿ることで見つける。

```
# BinaryTreeクラスに以下のインスタンスメソッドが定義されているとする。
    def insert(self, key, value):
        self.root = self._insert_by_recursion(self.root, key, value)
    def _insert_by_recursion(self, n, key, value):
        if n == None:
            return Node(key, value) # 子がなければ、そこに節点を作って登録

        if key < n.key:
            n.left = self._insert_by_recursion(n.left, key, value)
        else: # key >= n.key の場合
            n.right = self._insert_by_recursion(n.right, key, value)
        return n
```

以下のように使う。

インタラクティブモードでの実行例

```
>>> bt =  BinaryTree()
>>> bt.insert(5,"five")
>>> bt.insert(3,"three")
>>> bt.insert(6,"six")
>>> for i in [2,3,4,5,6,7]:
...     s = bt.search(i)
...     if s != None:
...         print(s.value)
...
...
three
five
six
```

5.1.3　二分探索木の削除

二分探索木からノードを削除する方法を考える。二分探索木からノードを削除するには、3 つのケースを考慮する必要があり、例えば以下のような方法で二分探索木の特性を維持しながら再帰的に節点を削除することができる：

削除する節点が葉である場合：このケースは最も簡単で、単に節点を削除する。

削除する節点が 1 つの子を持つ場合：削除する節点をその子で置き換える。

削除する節点が 2 つの子を持つ場合：削除する節点を右部分木の中で最小値をもつ節点で置き換え、その最小値を持つ節点を右部分木から削除する。最後の場合は部分木の中で最小値を求める必要があるがこれも再帰的に求めることができる。プログラムの例は以下のようなものが考えられる

```
#class BinaryTree: のブロック内で以下のようなメソッドを作成する。
    def delete(self, key):
        self.root = self._delete(self.root, key)

    def _delete_by_recursion(self, node, key):
        if node == None:
```

```
            return node
        if key < node.key:
            node.left = self._delete_by_recursion(node.left, key)
        elif key > node.key:
            node.right = self._delete_by_recursion(node.right, key)
        else: # key == node.key なので、この節点を削除する
            if node.left == None:
                return node.right
            elif node.right == None:
                return node.left
            # 左右の子がともに存在する場合
            # 右側の部分木の最小値を探す
            right_min_node = self._find_min_by_recursion(node.right) #適切に定義されているとする
            node.overwrite(right_min_node) #適切に定義されているとする
            node.right = self._delete_by_recursion(node.right, node.key)
        return node
```

5.1.4　二分木の探索の計算量

この方法での探索の計算量（complexity）を見積もる。ある h に対して、葉から根に至る節点列の長さが全て h か $h-1$ になっている 2 分木の場合を考えると、高々 h 個節点を見れば探索ができる。この木のデータ数 n は h と $n \geq 1+2+\cdots+2^{h-2}+1$ という関係にある。逆にいうと $h \leq \log_2(n)+1$ となる。従って、データ数 n に対して、$\log_2(n)+1$ 個の節点を見るだけで探索ができることになる。具体的な数値でいうと $n=10000$ の時、$\log_2(n)+1=14.29$、$n=1000000$ の時、$\log_2(n)+1=20.93$ である。これは線形探索の場合の平均 $n/2$ 回に比べて非常に良くなっている。

このように木がまんべんなく繁るようにデータが入力されれば良いが、入力の順序によっては非常に木が高くなる場合がある。例えば 1,2,3,4, というように昇順にデータが入力されると、右の子だけが追加されるので線形リストと同じ木ができてしまい探索効率が悪くなる。

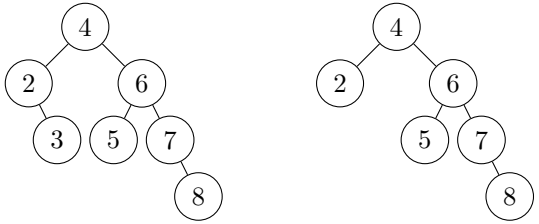


図5.1　AVL の条件を (ギリギリ) 満たす木の例（左）と AVL の条件を (ギリギリ) 満たさない木の例（右）

5.2　AVL 木 (Adelson-Velskii-Landis Tree)

木がアンバランスに茂らないようにした探索木を**平衡木** (balanced tree) と呼ぶ。平衡木には、AVL 木、赤黒木、2-3 木、k 平衡木など数多く開発されている。以降では G.M.**A**delson-**V**elskii と E.M.**L**andis によって開発された AVL 木について説明する。

全ての葉について深さが h か $h-1$ と揃った状態に保てれば理想的だが、節点を追加するときの仕事が多くなり過ぎるという問題がある。そこで、この条件を若干緩和したものとして以下のような条件を満たす木を**AVL 木**と呼ぶ:

「木の任意の節点に対し、左の子を根とする部分木の高さと右の子を根とする部分木の高さの差が高々 1 である」

便宜上、空木は AVL 木であるとする。以降では各節点に対しその節点のすべての子が「左の子を根とする部分木の高さと右の子を根とする部分木の高さの差が高々 1 である」ことをその節点が AVL 条件をみたす。ということにする。すなわち、木が AVL 木であることは木を構成するすべての節点が AVL 条件を満たすことと同値の条件である。

以下の実装例では Node に height というインスタンス変数を導入し、自分を根とする木の高さの情報を格納する。None で表される空木の高さにも対応するために、高さを得るためには必ず`get_height`メソッドを経由することにする。自分自身が AVL 木であるかどうかを判定するインスタンスメソッドである`is_AVL`メソッドは以下のように再帰を用いて簡単に書ける。

AVLTree クラスの冒頭部分の実装例

```
class AVLNode:
    def __init__(self, key, value, height):
        self.key = key
        self.value = value
        self.height = height
        self.left = None
        self.right = None
class AVLTree(BinaryTree):
    def get_height(self, n):
        """
            指定された節点を根とする部分木の高さを返す
        """
        if n == None:
            return -1
        else:
            return n.height
    def is_AVL(self):
        """
            自分自身がAVL木であるかどうかを再帰的に判定する。空木はAVL木であるとする。
        Returns:
            bool: AVL木ならTrue、そうでなければFalse
        """
        if self.root == None:
            return True
        else:
            if -1 <= self.get_height(self.root.left) - self.get_height(self.root.right) <=1:
                root_node_is_AVL = True
            else:
                root_node_is_AVL = False

            left_subtree, right_subtree = BinaryTree(self.root.left), BinaryTree(self.root.right)
            return root_node_is_AVL and left_subtree.is_AVL() and right_subtree.is_AVL()
```

5.2.1　AVL 木への節点の探索・削除

AVL 木はあくまで二分探索木であるので前節の探索が同様に可能である。削除に関しても同様の議論により前節の削除が利用可能である。

5.2.2　AVL 木への節点の挿入

上述のように単純な二分探索木と同様の挿入方法を採用すると挿入前は AVL 木にであったのに挿入後に AVL 木でなくなる場合がある。その場合は AVL 木の条件を満たすように木を組みなおす必要があり、その操作を平衡化という。平衡化の方法については、以下で詳しく説明する。

5.2.3　AVL 木への節点の平衡化

AVL 木への節点の挿入を二分探索木条件を満たように行った結果、AVL 木であった木が AVL 木でなくなった場合、AVL 条件を（ギリギリ）満たさない節点が存在する。すなわち、右の部分木の高さと左の部分木の高さの差が 2 である節点が存在する。そのような節点は節点を追加した際に辿った節点（= 追加した節点の祖先）のうちの一つか複数の節点である。AVL 条件を満たさない節点を組み替えて、AVL 条件をみたし、かつ、二分探索木条件を保つようにすることを節点の平衡化とよび、AVL 木でない木を AVL 木であるように組み替えることを木の平衡化と呼ぶことにすると、挿入の帰りがけに再帰的に節点の平衡化を行うことで木の平衡化を行うことができる。節点の平衡化は木の単回転 (single rotation) あるいは複回転 (double rotation) を行うことで達成できる。

ここでは、ある節点について左の子を根とする部分木の高さが右の子を根とする部分木より高い時、その節点は左に寄るということにする。逆に、右の子を根とする部分木の高さが左の子を根とする部分

木より高い時、その節点は右に寄るということにする。さらに、右の子を根とする部分木の高さが左の子を根とする部分木の高さと等しいとき、その節点は均衡するということにする。

　すると、AVL 条件を満たさない節点が左に寄り、左の子も左に寄る場合、単回転を行いさらに、AVL 条件を満たさない節点が左に寄り、左の子は右に寄る場合、複回転を行う。ということができる。以下ではそれぞれを左の子の高さが右の子の高さより 2 高いような節点があるとして単回転と複回転を説明する。以下の説明ではこの節点を A と呼び、A の左の子を B と呼ぶとする。

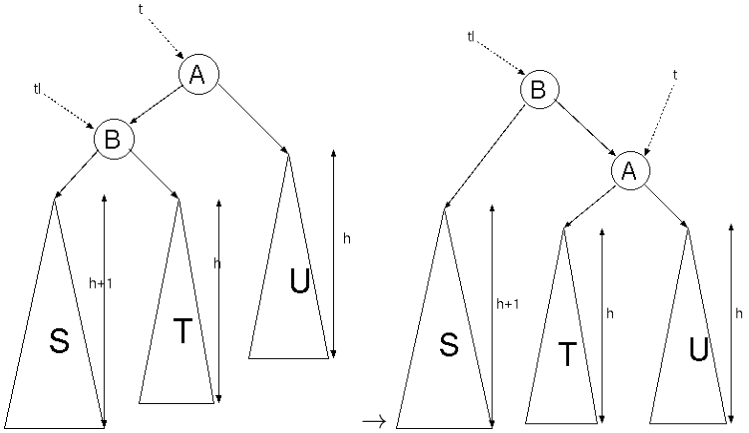


図5.2　single right rotation の前後の（部分）木の様子。左の木が single right rotation の後右の木となることで、A および B の左右の部分木の高さが等しくなる。

単回転

挿入の操作前は AVL 条件を満たしていたとすると、B を根とする部分木に節点が追加されたことにより高さが 1 増えたことになる。この場合、挿入の操作により B が左に寄ったか、B が右に寄ったかのいずれかである。左に寄った場合は、節点 B と節点 A を図5.2(右) のように組みかえることにより、AVL 木の条件を満たすようにできる。この操作を右単回転 (single right rotation) と呼ぶ。右単回転の前の状態で、二分探索木条件を満たすのであれば、右単回転の後でも二分探索木条件を満たす。理由を一言でいうと、どちらの木も以下の関係を満たすことが二分探索木条件であるからである：

$$S\text{の任意の値} < B\text{の値} < T\text{の任意の値} < A\text{の値} < U\text{の任意の値}$$

右単回転を行う実装は以下のように書ける。

```
#class AVLTree:以下のブロックに以下のようなインスタンスメソッドがあるとする
    def _single_right_rotation(self,A):
        """
        AVL木の節点に対しsingle right rotationを行う。
        Args:
            A(AVLNode): single right rotationを行う節点（図中のA）
        Returns:
            AVLNode: single right rotation後の節点（図中のB）
        Examples:
            >>> n = AVLNode("3", "A", 3)
            >>> n.left = AVLNode("2", "B", 2)
            >>> n.left.left = AVLNode("1", "S", 1)
            >>> n = single_right_rotation(n)
            >>> print(n.left.value, n.value, n.right.value)
            S B A
        """
        B = A.left
        A.left = B.right # TをB.rightからA.leftへ移動
        B.right = A      # AをB.rightへ移動
        A.height = 1 + max(self.get_height(A.left),self.get_height(A.right)) #高さを更新
        B.height = 1 + max(self.get_height(B.left),self.get_height(B.right)) #高さを更新
        return B            # BがAの代わりになる
```

左単回転（single left rotation）はこの左右を反転した場合である。すなわち、右の部分木の高さが左の部分木の高さより 2 高い節点 A に対し、右の節点の右の部分木が左の部分木よりも 1 高い場合に、木を平衡化する操作である。左単回転は以下のように書ける。

```
#class AVLTree:以下のブロックに以下のようなインスタンスメソッドがあるとする
    def _single_left_rotation(self,A):
        """
        AVL木の節点に対しsingle left rotationを行う。
        Args:
            A(AVLNode): single left rotationを行う節点
        Returns:
            AVLNode: single left rotation後の最も根に近い節点
        Examples:
            >>> n = AVLNode("1", "A", 3)
            >>> n.right = AVLNode("2", "B", 2)
            >>> n.right.right = AVLNode("3", "S", 1)
            >>> n = single_left_rotation(n)
            >>> print(n.left.value, n.value, n.right.value)
            A B S
        """
        B = A.right
        A.right = B.left
        B.left = A
        A.height = 1 + max(self.get_height(A.left),self.get_height(A.right)) #高さを更新
        B.height = 1 + max(self.get_height(B.left),self.get_height(B.right)) #高さを更新
        return B
```

複回転

図5.3のように左の子を根とする部分木の高さが 2 高く、挿入の操作により B が右に寄った場合は、AVL 木が節点を挿入した後に、上述の単回転は行えない。この場合は節点 B を左回転した後、節点 C を右回転する。これを**複回転** (double rotation) と呼ぶ。特にこの場合 left right rotation とよび、左右反転した状況の場合を right left rotation と呼ぶ。

double rotation の後も二分探索木条件は満たすようになっている。これは、一言でいうと、double

rotation の前後どちらの木も以下の関係を満たすことが二分探索木条件であるためである。

$$S\text{の任意の値} < B\text{の値} < T\text{の任意の値} < C\text{の値} < U\text{の任意の値} < A\text{の値} < V\text{の任意の値}$$

これらの４通りの回転操作により、AVL 木が節点を挿入した後に左右の高さが 2 違う場合は必ず単回転か複回転を行うことができる。図5.3にある複回転（`left_right_rotation`）と左右反転した`right_left_rotation`は以下のようになる。

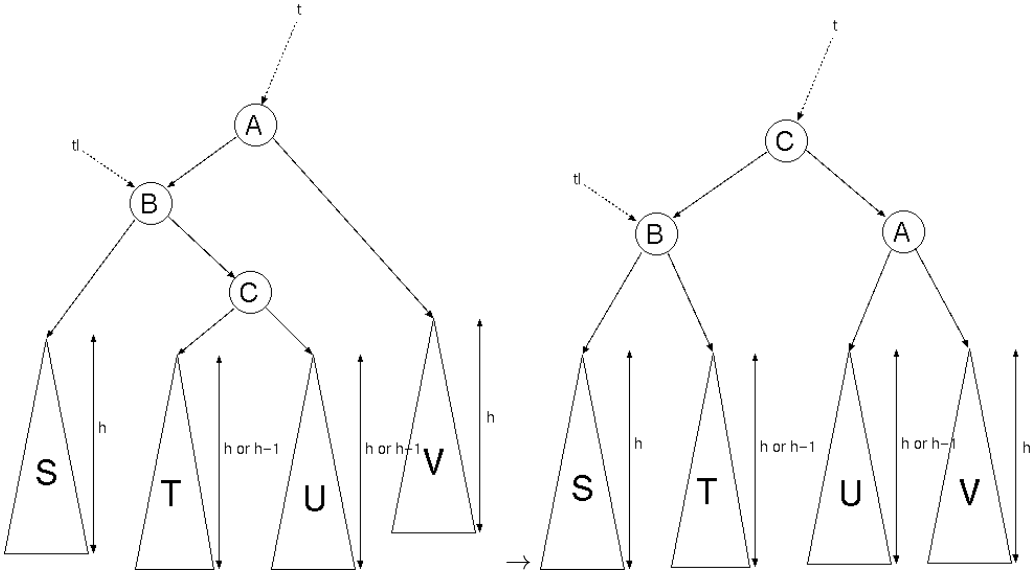


図5.3　left right rotation の前後の（部分）木の様子。左の木が複回転の後右の木となることで、A および B の左右の部分木の高さの差が 1 以下となる。

```
#class AVLTree:以下のブロックに以下のようなインスタンスメソッドがあるとする
    def _left_right_rotation(self,A):
        """
        AVL木の節点に対しleft right rotationを行う。
        Args:
            A(AVLNode): left right rotationを行う節点（図中のA）
        Returns:
            AVLNode: left right rotation後の節点（図中のC）
        Examples:
            >>> n = AVLNode("3", "A", 3)
            >>> n.left = AVLNode("1", "B", 2)
            >>> n.left.right = AVLNode("2", "C", 1)
            >>> n = left_right_rotation(n)
            >>> print(n.left.value, n.value, n.right.value)
            B C A
        """
        B = A.left
        C = A.left.right
        #Bを左回転
        B.right = C.left    # TをB.rightからA.leftへ移動
        C.left = B          # AをB.rightへ移動
        #Cを右回転
        A.left = C.right  # UをC.rightからA.leftへ移動
        C.right = A       # AをC.rightへ移動

        A.height = 1 + max(self.get_height(A.left),self.get_height(A.right)) #高さを更新
        B.height = 1 + max(self.get_height(B.left),self.get_height(B.right)) #高さを更新
        C.height = 1 + max(self.get_height(C.left),self.get_height(C.right)) #高さを更新
        return C            # CがAの代わりになる
    def _right_left_rotation(self,A):
        """
        AVL木の節点に対しright left rotationを行う。
        Args:
            A(AVLNode): right left rotationを行う節点
        Returns:
            AVLNode: right left rotation後の最も根に近い節点
        Examples:
            >>> n = AVLNode("1", "A", 3)
            >>> n.right = AVLNode("3", "B", 2)
            >>> n.right.left = AVLNode("2", "C", 1)
            >>> n = right_left_rotation(n)
            >>> print(n.left.value, n.value, n.right.value)
            A C B
```

```
            >>> print(n.left.height, n.height, n.right.height)
        """
        B = A.right
        C = A.right.left
        #Bを右回転
        B.left = C.right
        C.right = B
        #Cを左回転
        A.right = C.left
        C.left = A
        A.height = 1 + max(self.get_height(A.left),self.get_height(A.right)) #高さを更新
        B.height = 1 + max(self.get_height(B.left),self.get_height(B.right)) #高さを更新
        C.height = 1 + max(self.get_height(C.left),self.get_height(C.right)) #高さを更新
        return C
```

AVL 木の挿入の実装

平衡化を行う節点は挿入の際に通った節点のうちのいずれかである。上述した木の回転により AVL 木であることを保つためには、AVL 木の挿入の帰りがけに平衡化を行う必要がある。

```
#class AVLTree:以下のブロックに以下のようなインスタンスメソッドがあるとする
    def insert(self, key, value):
        """
        指定されたkeyとvalueを持つ節点を自分自身にノードを挿入し、AVL条件を保つ。
        """
        self.root = self._insert_by_recursion(self.root, key, value)
        return self
    def _insert_by_recursion(self, n, key, value):
        """
        指定されたkeyとvalueを持つ節点を指定した節点以下の部分木内に挿入し、AVL条件を保つ。
        Args:
            n (Node): 挿入する部分木の根
            key (int): 挿入する節点のkeyの値
            value (int): 挿入する節点のvalueの値

        Returns:
            Node: 挿入後の部分木の根
        """
        if n == None:
            return AVLNode(key, value, 0) # 空木ならば、そこに節点を作って登録
```

```
        else:
            if key < n.key:
                n.left = self._insert_by_recursion(n.left, key, value)
            else:
                n.right = self._insert_by_recursion(n.right, key, value)

        #挿入の帰りがけに木の深さを更新する。
        n.height = 1 + max(self.get_height(n.left), self.get_height(n.right))

        #必要であれば木の平衡化を行う
        h_left,h_right = self.get_height(n.left),self.get_height(n.right)
        if h_left - h_right == 2:
            h_left_left,h_left_right = self.get_height(n.left.left),self.get_height(n.left.right)
            if h_left_left - h_left_right == 1:
                n = self._single_right_rotation(n)
            elif h_left_right - h_left_left == 1:
                n = self._left_right_rotation(n)
        elif h_right - h_left == 2:
            h_right_left,h_right_right = self.get_height(n.right.left),self.get_height(n.right.right)
            if h_right_right - h_right_left == 1:
                n = self._single_left_rotation(n)
            elif h_right_left - h_right_right == 1:
                n = self._right_left_rotation(n)
        return n
```

以下のように使う。

インタラクティブモードでの実行例

```
avltree = AVLTree()
>>> for i in [1,2,3,4,5,6,7,8,9]:
>>>     avltree.insert(10-i,str(10-i))
>>>
>>> for i in [-1,0,1,2,3,4,5,6,7,8,9,10]:
>>>     s = avltree.search(i)
>>>     if s != None:
>>>         print(s.value)
1
2
3
4
5
6
7
8
9
```

5.2.4　AVL 木の計算量（発展）

AVL 木での探索で節点を辿る回数を考える。高さ h の AVL 木の節点の数を $S(h)$ で表す。すると、AVL 木の条件から $S(h) \geq 1+S(h-1)+S(h-2)$ となる。最悪の場合である $S(h) = 1+S(h-1)+S(h-2)$ を考える。$F(h) = S(h)+1$ とおくと $F(h) = F(h-1)+F(h-2)$ となる。この漸化式はフィボナッチ数列なので、適当な係数 a と b に対して、$F(h) = a((\sqrt{5}+1)/2)^h + b((\sqrt{5}-1)/2)^h$ と書ける。$0 < ((\sqrt{5}-1)/2)^h < 1$ なので、h の増加につれて $((\sqrt{5}-1)/2)^h$ は小さくなり、$((\sqrt{5}+1)/2)^h$ の項が支配的となる。そこで、データ数 n に対して、辿る節点数はおよそ $\log_{(\sqrt{5}+1)/2}(n)$ に比例する。例えば、$n = 10000$ の時 $\log_{(\sqrt{5}+1)/2}(n) = 19.14$、$n = 1000000$ の時 $\log_{(\sqrt{5}+1)/2}(n) = 28.71$ となる。

5.3　複数行の入力

今回以降の演習では複数行の入力を受け取り、複数行の出力を返すプログラムを提出する。（問 3 以降）例えば key が 0,1,2,3,4,5,6 である要素を value をそれらを表す英単語として探索木に格納させたい時、それらの情報を以下のように表現する。すなわち、入力の最初の行は挿入するデータの数を表し、入力の二行目以降はそれぞれの要素の key と value の情報を各行が key の値、空白文字、value の値の連続で表す。

7 つの要素を表現する複数行の入力例

```
7
0 zero
1 one
2 two
3 three
4 four
5 five
6 six
```

このような入力に対し、探索木を構築するためには以下のようなプログラムを用いればよい。

```
#これより前で insertメソッドが定義されているクラスのインスタンスmytreeが初期化されたとする。
n=int(input())
for i in range(n):
    line = input().split(" ")
    key, value = int(line[0]), line[1]
    mytree.insert(key,value)
```

以下の演習問題ではこれらの入力の後、さらに木の操作に対する情報が追加されるので、それに応じて提出するプログラムを変える必要がある。

演習

UTOL 上の課題は改善点を自分で考えて実装してみた人のための報告場所です。UTOL 上に提出しても課題に正解したかどうかは評価されないので注意すること。UTOL 上に提出する際は、改善点をプログラムのコメント中または提出時のコメントに明記すること。
DOMjudge 中の Contest 名を prog_5 へ変更すれば以下の問題が現れる。各問題に対して受け付けた入力に応じて結果が変わるようにプログラムを作成し、提出せよ。UTOL から template をダウンロードして加筆するのがよい。pdf からコピー・貼り付けを行うと予期せぬエラーがでることがあるので自分で書いていくことを推奨する。

問 1　5.1節で述べた BinaryTree クラスの insert メソッドを実装し、入力として（BinaryTree インスタンスを作成して insert メソッドを実行する）プログラムの文字列を受け取り、想定される結果を出力するプログラムを提出せよ。

要件

```
class BinaryTree:
    def __init__(self):
        """
        初期化メソッド
        """
    def insert(self, key, value):
        """
        指定されたkeyとvalueを持つ節点を自分自身内に挿入する。
        Args:
            key (int): 挿入する節点のkeyの値
            value (any): 挿入する節点のvalueの値
        Returns:
            BinaryTree: 挿入後の自分自身

        Examples:
            >>> bt = BinaryTree()
            >>> bt.insert(10, "v10").insert(5, "v5")
            >>> print(bt.root.key)
            10
            >>> print(bt.root.left.value)
            v5
        """
    def search(self, key):
        """
        指定されたkeyを持つ節点を自分自身内で検索する。
        Args:
            key (int): 探索するkeyの値
        Returns:
            Node: 見つかった節点。存在しない場合はNoneを返す。

        Examples:
            >>> bt = BinaryTree()
            >>> bt.root = Node(10, "ten")
            >>> bt.root.left = Node(5, "five")
            >>> bt.root.right = Node(15, "fifteen")
            >>> print(bt.search(5))
            five
            >>> bt.search(20) == None
            True
        """
```

入力例 1

```
print(BinaryTree().insert(1,"a").root.value)
```

出力例 1

```
a
```

入力例 2

```
print(BinaryTree().insert(1,"a").insert(2,"b").insert(3,"c").root.right.right.value)
```

出力例 2

```
c
```

問 2　同様に search メソッドを実装し、入力として（BinaryTree インスタンスを作成して search メソッドを実行する）プログラムの文字列を受け取り、想定される結果を出力するプログラムを提出せよ。

要件

前問を参照

入力例 1

```
print(BinaryTree().insert(1,"a").insert(2,"b").insert(3,"c").search(3))
```

出力例 1

```
c
```

入力例 2

```
print(BinaryTree().insert(1,"a").insert(2,"b").insert(3,"c").search(0) == None)
```

出力例 2

```
True
```

問 3　5.3節で説明した形式で与えられる複数行の標準入力から BinaryTree クラスのインスタンスへデータを挿入した後に、データを key の値の小さい順にデータの value を出力するプログラムを作成せよ。

入力例 1

```
3
1 one
5 five
2 two
```

出力例 1

```
one
two
five
```

入力例 2

```
5
3 three
5 five
2 two
4 four
1 one
```

出力例 2

```
one
two
three
four
five
```

ヒント：深さ優先探索で木を走査するだけでよい

問 4　指定された整数 s, t に対し、key が s 以上 t 以下の節点に格納されている value のリストを返す`search_between`メソッドを、BinaryTree に定義したプログラムを提出せよ。ただし不要な部分木をたどらないようにすること。入力の形式は最終行を除いて前問と同様であり、最終行は s の値、「,」、t の値が与えられる。出力は返すべき value が key の小さい順に格納されたリストの内容を表示する。いずれかの key が存在しない場合空のリスト出力する。

要件

```
class BinaryTree:
    def search_between(self, s, t):
    """
        keyがs以上t以下の節点に格納されているvalueのリストを返す
        Args:
            s(int):探索するkeyの値
            t(int):探索するkeyの値
        Returns:
            List: s以上t以下のkeyの値であるデータのvalueが入っているリスト
    """
```

入力例 1

```
6
1 one
4 four
5 five
2 two
3 three
6 six
3,5
```

出力例 1

```
["three", "four", "five"]
```

入力例 2

```
7
1 one
4 four
5 five
2 two
7 seven
3 three
6 six
10,20
```

出力例 2

```
[]
```

問 5　5.2節で述べた AVLTree クラスを実装し、効率よくデータを検索するプログラムを作成せよ。入力は5.3節で説明したような複数行の入力の形式で挿入する節点を指定した後、同様に検索する key の数を表す行と、検索する key 自身を表す行がその数だけつづく。

要件

```
class AVLTree(BinaryTree):
    def __init__(self):
        """
        初期化メソッド
        """
    def insert(self, key, value):
        """
        指定されたkeyとvalueを持つ節点を自分自身に節点を挿入し、AVL条件を保つ。
        Args:
            key (int): 挿入する節点のkeyの値
            value (any): 挿入する節点のvalueの値
        Returns:
            AVLTree: 自分自身
        Examples:
            >>> avl = AVLTree()
            >>> avl.insert(10, "v10").insert(5, "v5").insert(15, "v15")
            >>> print(avl.root.key)
            v10
            >>> print(avl.root.left.key)
            v5
            >>> print(avl.root.right.key)
            v15
        """
```

入力例 1

```
3
10 v10
5 v5
15 v15
1
10
```

出力例 1

```
v10
```

入力例 2

```
3
10 v10
5 v5
15 v15
2
10
5
```

出力例 2

```
v10
v5
```

入力例 3

```
3
10 v10
5 v5
15 v15
4
7
8
9
10
```

出力例 3

```
No record
No record
No record
v10
```

第 6 章

無向グラフ

ここでは、「もの」の「つながり方」を表す「グラフ」について説明する。「つながり方」に向きの無い「グラフ」を無向グラフ（undirected graph）という。ある節点 A とつながっている節点 B を節点 A に**隣接する**という。グラフは節点と、節点と節点を結ぶ辺（edge,arc）からなる。例えば、図の左の迷路は場所を節点、場所から場所に行ける事を場所の間の辺で表すことで図の右のグラフで表現することができる。数学的には有限集合である節点の集合 V に対し、一つのグラフ G は集合 V と辺の集合 $E \subset V \times V$ の組で表され、$G = (V, E)$ と表すことができる。

6.1 キュー

キュー (queue) とはデータの要素に関して追加と取り出しができる抽象データ型である。取り出しは先に追加されたデータから優先的に要素が取り出される。要素が一つもないキューからは取り出しはできない。キューへの要素の追加は enqueue と呼ばれ、取り出しは dequeue と呼ばれる。取り出しの優

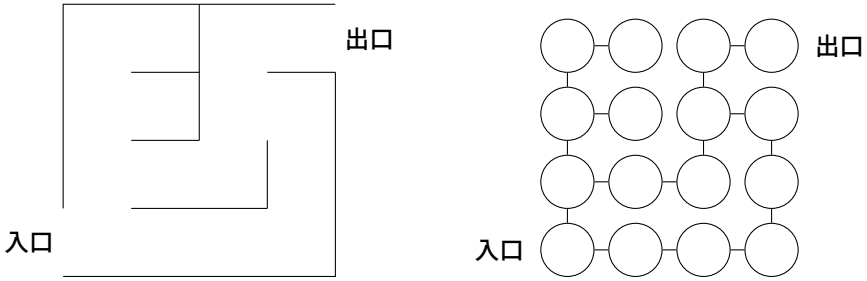


図6.1　迷路と対応する無向グラフ

先順位がスタックとは反対であるという意味でスタックと対照的な抽象データ型である。キューの方式を**先入れ先出し** (First in First out, FIFO)、スタックの方式を**後入れ先出し** (Last in First out, LIFO) と呼ぶこともある。Python のリストをキューとみなすこともできるし、queue ライブラリをインポートすることもできる。以下では queue ライブラリを用いる。

test_queue.py の内容

```
1 import queue
2 q = queue.Queue() #queueライブラリにQueueクラスが定義されている
3 q.put(1) # putメソッドでenqueue
4 q.put(2)
5 q.put(3)
6 v = q.get() #getメソッドでdequeue
7 print(v)  #vは1
```

標準モード (ターミナル) での実行例

```
1 $ test_queue.py
2 1
```

6.2　グラフの表現

与えられたグラフを計算機上で表現することを考える。ここでは隣接リスト表現と隣接行列表現の二つの方法を説明する。与えられたグラフの節点には適当に 1 から n までの番号が付与されているとする（n は節点の総数）。すると辺は二つの整数の組で表せることになる。

6.2.1　グラフの隣接リスト表現

各節点に対して、隣接する節点の番号のリストを作ることでグラフを表現することができる。これを**隣接リスト表現** (adjacency list representation) という。縦のリストの添字が 1 のところに要素が (2,5) のリストがあることで、節点 1 と節点 2、節点 1 と節点 5 の間に辺があることを表す。グラフを表すクラスにはこの（二重）リストをインスタンス変数に持つような以下のクラスを用いる。

```
class Graph:
    def __init__(self,n):
        # 全節点数を記録
        self.num_nodes = n
        # 節点数nの枝のないグラフで初期化する
        self.adjacency_list = [ [] for _ in range(n) ]
    def connect(self, x, y):
        """
            自分自身が表すグラフに枝を加える
            Args:
            x,y (int): 枝を加える2節点の番号
        """
        if not y in self.adjacency_list[x]:
            self.adjacency_list[x].append(y)
        if not x in self.adjacency_list[y]:
            # xとyをつないだら、yとxもつないでおく。
            self.adjacency_list[y].append(x)
```

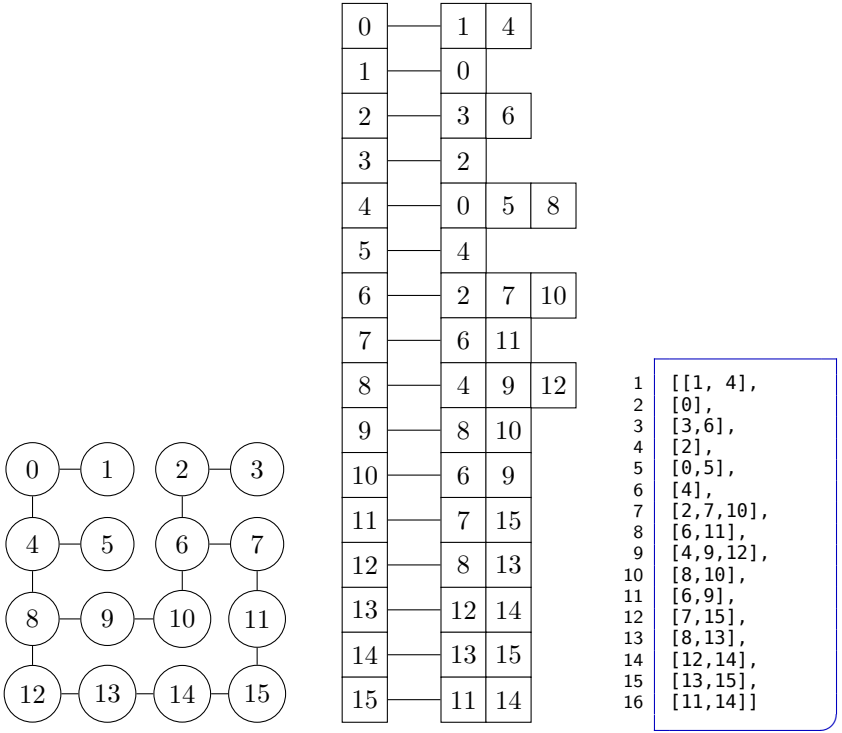


図6.2　無向グラフとその隣接リスト表現

6 行目の `_` は `i` などの変数と同じだが暗に変数を反復操作の中で使わないことを意味する。

6.2.2　グラフの隣接行列表現

隣接リストとは異なるグラフの表現である隣接行列表現を見てみよう。次のような行列を考える。

- 頂点 i と頂点 j が辺で結ばれている $\rightarrow$ 行列の $i+1$ 行 $j+1$ 列要素が 1 である
- 頂点 i と頂点 j が辺で結ばれていない $\rightarrow$ 行列の $i+1$ 行 $j+1$ 列要素が 0 である

この規則により、グラフを行列で表現したものを**隣接行列**（adjacency matrix）という。無向グラフの隣接行列は常に対称行列である。例えば、上述した迷路を表現する無向グラフの隣接行列表現は以下のようになる。

$$
\begin{pmatrix}
0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0
\end{pmatrix}
$$

6.3　データの入力

ここでは6.2.1節のグラフの情報が以下のようなテキストファイルに格納されているとする。ここでは最初の行は節点数、その後にそれぞれの枝の情報が行ごとに記述されている。

graph_data.txt の内容

```
16
0 1
0 4
2 3
2 6
4 5
4 8
6 7
6 10
7 11
8 9
8 12
9 10
11 15
12 13
13 14
14 15

```

最後に改行のみの行が入っていることに注意する。以下は上の文字列ファイルを入力としてグラフを構築するプログラムの一例である

graph_constract.py の内容

```
1  N = input()
2  g = Graph(int(N))
3  while True:
4      string_list = input().split(" ") #string_list は['0', '1']のようなものになる
5      if len(string_list) == 0:
6          break
7      g.connect(int(string_list[0]),int(string_list[1]))
```

このプログラムを標準モードで以下の様に実行すると内部でグラフが作られる。

標準モード (ターミナル) での実行例

```
1  $ python3 graph_construct.py < graph_data.txt
```

6.4　経路探索問題

始点 (source) と呼ばれる節点から終点 (target) と呼ばれる節点まで辺を辿っていく経路を見つける問題を**経路探索問題**という。迷路をグラフで表した場合、このグラフで経路探索問題を解くことは迷路を解くことに相当する。まずは各節点が始点から到達可能であるか（＝経路が存在するか）を判定することを考え、その後経路そのものを探索する方法を考える。

6.4.1　深さ優先探索による到達判定

各節点が到達可能か判定するためには、木の場合の深さ優先探索と同様に始点から隣接している節点を再帰的に走査していけばよい。ただし、グラフでは同じ節点を何度も辿ってしまう可能性があるので、まだ訪れてない節点だけ辿る工夫を施す必要がある。

```
1: procedure VISIT_DFS(節点 i = 始点)
2:     if 節点 i が既に訪れた節点 then
3:         終了
4:     else まだ訪れていない節点
5:         節点 i に訪れたことを記録する
6:         for i' in 節点 i に隣接している節点 do
7:             節点 i' に対して、VISIT_DFS(i') を実行する
```

プログラムは例えば以下のようにすればよい。ここでは各節点に到達したかどうかを記録するリストを用意し、すでに到達した場合は visit を呼ばないようにしている。

```
18 #class Graphのスコープ（17行目までは上を参照）
19     def visit_dfs(self, source):
20         """
21             深さ優先探索で与えられた節点から到達可能な節点を調べる
22             Args:
23                 source(int): 始点とする節点
24             Returns:
25                 list: sourceから到達できるかどうかが真偽値で与えれれた節点数の長さのリスト
26         """
27         visited = [ False for i in range(self.num_nodes)] # 全てFalseで初期化する
28         self._visit_by_recursion(source, visited)
29         return visited
30     def _visit_by_recursion(self, i, visited):
31         """
32             対象とする節点を訪れて終点であれば終了する。隣接する節点に再帰的にvisitを呼びだす。
33             Args:
34                 i: 対象とする節点
35                 visited(list): 各節点を訪れたかどうかを記録するリスト
36             Returns:
37                 None
38         """
39         if visited[i] == True:
40             return
41         else:
42             visited[i] = True
43             for j in self.adjacency_list[i]:
44                 self._visit_by_recursion(j,visited)
```

6.4.2　幅優先探索による到達判定

深さ優先探索では最短経路が見つかるとは限らない。これは隣接する節点を見つけることとその節点を処理することが LIFO になっているからである。これを FIFO にすることで最短経路を見つけることができる。すなわち、節点を処理するたびに隣接する節点をキューに追加し、新しく節点を処理すると

きはキューから取り出すようにすればよい。このような探索方法を**幅優先探索**（breadth first search）と呼ぶ。幅優先探索でグラフを辿るアルゴリズムは以下のようになる。

1: **procedure** VISIT_BFS(始点)
2: 　キューに始点を追加
3: 　**while** キューが空でない **do**
4: 　　キューから節点 i を取り出す
5: 　　節点 i に訪れたことを記録する
6: 　　**for** 節点 i' **in** i に隣接する節点 **do**
7: 　　　節点 i' に訪れたことがなければキューに i' を追加

これはキューを使って以下のように実装することができる。

```
45  #class Graphのスコープ（44行目までは上を参照）
46      def visit_bfs(self, source):
47          """
48              幅優先探索で与えられた節点から到達可能な節点を調べる
49              Args:
50                  source(int): 始点とする節点
51              Returns:
52                  list: sourceから到達できるかどうかが真偽値で与えれれた節点数の長さのリスト
53          """
54          q = queue.Queue()
55          q.put(source)
56          visited = [ (i==source) for i in range(self.num_nodes)] # sourceだけがTrue, あとはFalseで初期化する
57          while not q.empty():
58              u = q.get()
59              for v in self.adjacency_list[u]:
60                  if visited[v] == False:
61                      q.put(v)
62                      visited[v] = True
63          return visited
```

6.4.3　経路探索

到達可能な節点への経路を知るためには visited に真偽値ではなく、その節点にたどりつく直前に調べていた節点の番号を記録しておくだけで十分である。節点 i にたどり着く直前に調べていた節点が i' だとすると、始点から i にたどり着くために i' から i の枝を辿る経路があることを意味する。

```
62  #class Graphのスコープ（61行目までは上を参照）
63      def find_path_bfs(self,source,target):
64          """
65              幅優先探索で始点から終点までの経路を出力する
66              Args:
67                  source(int): 始点とする節点
68                  target(int): 終点とする節点
69              Returns:
70                  list: sourceからtargetまでの経由する節点を要素とするリスト
71          """
72          visited = [-1 for _ in range(self.num_nodes)]
73          # 真偽値ではなく -1が入っている(-1はたどり着かないことを意味する)
74          q = queue.Queue()
75          q.put(source)
76          visited[source] = source #始点はたどり着ける判定にしておきたいので適当な数値を入れておく
77          while not q.empty():
78              u = q.get()
79              for v in self.adjacency_list[u]:
80                  if visited[v] == -1:
81                      visited[v] = u
82                      q.put(v)
83
84          path = [] #pathには出力用の番号を格納する
85          u = target
86          while not u == source:
87              path.append(u)
88              u = visited[u]
89          path.append(source)
90          path.reverse()
91          return path
```

演習

UTOL 上の課題は改善点を自分で考えて実装してみた人のための報告場所です。UTOL 上に提出しても課題に正解したかどうかは評価されないので注意すること。UTOL 上に提出する際は、改善点をプログラムのコメント中または提出時のコメントに明記すること。
DOMjudge 中の Contest 名を prog_6 へ変更すれば以下の問題が現れる。各問題に対して受け付けた入力に応じて結果が変わるようにプログラムを作成し、提出せよ。

問 1　幅優先探索で始点から終点までの最短経路を出力するプログラムを実装せよ。最初の行は全節点数、始点の節点番号、終点の節点番号が与えらえる。次の行からは辺の情報が行ごとに記述されており、最後に空行が挿入されている。始点から終点までの最短経路は必ず一つだけ存在するとしてよい。(出力は最後に空白文字を入れないようにする。)

要件

```
class Graph:
    def find_path_bfs(self,source,target):
        """
            幅優先探索で始点から終点までの経路を出力する
            Args:
                source(int): 始点とする節点
                target(int): 終点とする節点
            Returns:
                list(int): sourceからtargetまでの経由する節点を要素とするリスト
        """
```

入力例 1

```
4 2 4
0 1
0 2
2 3
```

出力例 1

```
1 0 2 3
```

入力例 2

```
16 13 4
0 1
0 4
2 3
2 6
4 5
4 8
6 7
6 10
7 11
8 9
8 12
9 10
11 15
12 13
13 14
14 15
```

出力例 2

```
12 8 9 10 6 2 3
```

問 2　入力と出力は問 1 と同じであるが、経路が存在しない場合も考える（経路が存在する場合はただ一つとする）。経路が存在しない場合は”No path” と出力する。

要件

問 1 と同じ。経路が存在しない場合は"No path"と出力する。

入力例 1

```
16 13 4
0 1
0 4
2 3
2 6
4 5
4 8
6 7
6 10
7 11
8 9
8 12
9 10
11 15
12 13
13 14
14 15
```

出力例 1

```
12 8 9 10 6 2 3
```

入力例 2

```
16 13 4
0 1
0 4
2 6
4 5
4 8
6 7
6 10
7 11
8 9
8 12
9 10
11 15
12 13
13 14
14 15
```

出力例 2

```
No path
```

問 3　幅優先探索で始点から終点までの経路長が奇数であるような最短経路を出力するプログラムを実装せよ。最初の行は全節点数、始点の節点番号、終点の節点番号が与えらえる。経路が存在しない場合は”No path” と出力する。

要件

```
class Graph:
    def find_odd_path(self,source,target):
        """
            幅優先探索で始点から終点までの長さが奇数の経路を出力する
            Args:
                source(int): 始点とする節点
                target(int): 終点とする節点
            Returns:
                list(int): sourceからtargetまでの経由する節点を要素とするリスト
        """
```

入力例 1

```
16 13 4
0 1
0 4
2 3
2 6
4 5
4 8
6 7
6 10
7 11
8 9
8 12
9 10
11 15
12 13
13 14
14 15
```

出力例 1

```
No path
```

入力例 2

```
4 2 4
0 1
0 2
2 3
```

出力例 2

```
1 0 2 3
```

問 4　与えられた無向グラフが二部グラフであるかどうかを判定するインスタンスメソッド（`is_biparate` メソッド）を実装し、二部グラフならば節点 1 を含む節点集合を番号が小さい順に出力する。二部グラフとは全て節点がどちらかに入るような 2 つの節点集合 V_1 と V_2 が、V_1 の節点同士あるいは V_2 の節点同士の間に枝が存在しないようにとれるようなグラフのことで

ある。

要件

```
class Graph:
    def is_bipartite(self):
        """
        自分自身が二部グラフかどうかを判定し、0を含む節点集合（独立集合）を計算する。
        Returns:
            list(int): グラフが二部グラフであれば0を含む節点集合、二部グラフでなければ空のリストを返す。
        """
```

入力例 1

```
4
0 1
0 2
2 3
```

出力例 1

```
0 3
```

入力例 2

```
5
0 1
0 2
1 3
2 4
```

出力例 2

```
0 3 4
```

入力例 3

```
16 15
0 1
0 4
2 4
2 6
3 10
4 5
4 7
6 7
7 10
7 12
8 9
8 13
9 10
11 14
12 13
13 14
14 15
```

出力例 3

```
0 2 3 5 7 9 11 13 15
```

入力例 4

```
7
0 1
1 2
2 3
3 4
4 5
5 6
6 0
```

出力例 4

```
Graph is not bipartite
```

第 7 章

有向グラフ

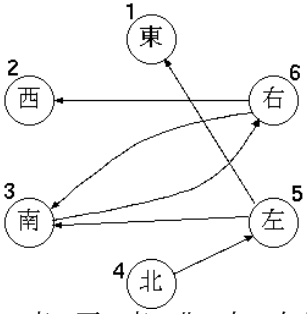
「つながり方」に向きのある「グラフ」は有向グラフ (directed graph) という。無向グラフと同様に節点と枝があるが、枝はただ二つの節点をつないでいるだけではなく向きを持ち、ある節点から出てある節点へ入る。

例として語の定義を考える。辞書では易しい語によって難しい語を定義していく。A を定義するために B を使い、B を定義するために A を使っている場合、それを循環定義（circular definition）という。広辞苑 (第四版) では、東、西、南、北、左、右が次のように定義されている[*1]。

東　四方の一。太陽の出る方：東方 ↔ 西
西　四方の一。日の入る方角 ↔ 東
南　四方の一。日の出る方に向かって右の方向 ↔ 北

[*1] 循環定義の例に広辞苑の東、西、南、北、左、右があることは、人文的数学のすすめ、銀林浩、日本評論社の示唆による。

北　方角の一。日の出る方に向かって左の方向
左　南を向いたとき、東にあたる方
右　南を向いたとき、西にあたる方



東、西、南、北、左、右を節点としてグラフを書くと次のようになる。このグラフは矢印の元の語が矢印の先の語を使って定義されていることを示している。このように枝に方向性があるグラフを有向グラフ (directed graph) という。木データモデルは有向グラフの特殊なケースということができる。

有向グラフと語の世界は以下のように対応している。辿っていくと元の節点に戻るような枝の列をサイクルという。サイクルのない有向グラフを DAG(Directed Acyclic graph) とよぶ。節点 x に入る枝の数のことを x の**入次数**という。節点 x から出る枝の数のことを x の**出次数**という。

7.1　numpy モジュール

numpy モジュールは行列の表現や操作などの高度な数値計算を行うためのモジュールである。numpy モジュールをインポートし、np というエイリアスを設定するには以下のようにする。

```
1 import numpy as np
```

これにより numpy.zeros() などの関数は np.zeros() などと呼び出すことになる。

インタラクティブモードでの実行例

```
1  >>> import numpy as np
2  >>> A = np.zeros([3,3]) # ゼロ行列に対応
3  >>> B = np.eye(3)       # 単位行列に対応
4  >>> C = np.ones([3,3])  # 全ての要素が1であるような行列に対応
5  >>> print(A)
6  [[0. 0. 0.]
7   [0. 0. 0.]
8   [0. 0. 0.]]
9  >>> print(B)
10 [[1. 0. 0.]
11  [0. 1. 0.]
12  [0. 0. 1.]]
13 >>> print(C)
14 [[1. 1. 1.]
15  [1. 1. 1.]
16  [1. 1. 1.]]
```

各要素にアクセスするためには`C[0,0]`などとすればよい。二重配列のように`C[0][0]`とすることも可能である。行列の乗算は`np.dot()`を用いる。

インタラクティブモードでの実行例

```
1  >>> print( A[0,0] )
2  0.0
3  >>> print( np.dot(A,B) )
4  [[0. 0. 0.]
5   [0. 0. 0.]
6   [0. 0. 0.]]
7  >>> print( np.dot(C,C))
8  [[3. 3. 3.]
9   [3. 3. 3.]
10  [3. 3. 3.]]
```

```
16
0 1
4 0
4 5
8 4
8 9
9 10
10 6
6 2
2 3
12 8
12 13
13 14
14 15
15 11
11 7
```

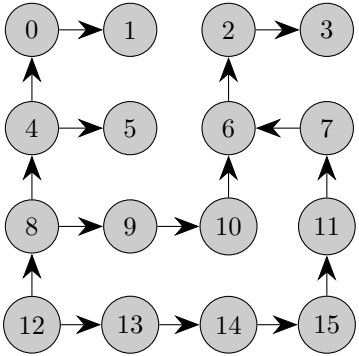


図7.1　有向グラフの場合の入力となる文字列と構築される DAG。最初の 16 は全節点数を示し、その後は各行が始点と終点の番号のペアで表される。最後に空行がある。

7.2　有向グラフの表現

有向グラフも無向グラフと同様に隣接リストで表現することができる。すなわち、それぞれの配列の要素はその節点から出る枝が入る節点の集合を表現するようにする。有向グラフの場合は枝を相互につなげるのではなく、一方的につなげるようにプログラムを変更するだけでよい。ただし、本節では節点番号は 0 から始まるものとする。

```
1  class DirectedGraphAdjacencyList:
2      #  隣接リストによる有向グラフのクラス
3      def __init__(self,n):
4          self.num_nodes = n
5          self.list = [ [] for _ in range(n) ]
```

```
    def connect(self, x, y):
        if not y in self.list[x]:
            self.list[x].append(y)
```

前述したようにグラフは隣接行列の表現も可能である。無向グラフの場合は隣接行列が対称行列であったが、有向グラフでは一般に非対称になる。隣接行列による実装は以下のようになる。隣接行列で表現したグラフでの枝 (x, y) の作成は $A_{x,y}$ に 1 を代入することに対応し、隣接行列で表現したグラフでの枝 (x, y) の存在の判定は $A_{x,y}$ の要素が 1 であるかを判定することに対応する。

```
class DirectedGraphAdjencyMatrix:
    #  隣接行列による有向グラフのクラス
    def __init__(self,n):
        self.num_nodes = n
        self.matrix = np.zeros([n,n]) #n行n列の零行列を作る関数
    def connect(self, x, y):
        self.matrix[x,y] = 1 # 行列のx,y 成分を1とする
```

7.3　有向グラフの経路探索（抽象クラスと具象クラス）

前節の無向グラフの経路探索アルゴリズムは有向グラフの場合も同様に動作する。経路探索のアルゴリズムは有向グラフの具体的な実装に関わらず、与えられた節点から枝が出ている節点のリストを得るインスタンスメソッド`get_adjacent_nodes`が適切に実装されていれば、実装可能である。

そこで、ここでは一般的なグラフの**抽象クラス**（Abstract Class）である AbstractGraph クラスを定義し、経路探索のアルゴリズムをこの抽象クラスの**抽象メソッド**として実装する例を紹介する。抽象クラスは必ず継承して使うクラスであり、具体的なグラフの表現方法を実装する下位のクラスがこれを継承して使う。Python では抽象基底クラスを扱う abc モジュールを利用して以下のような形式で使う[*2]：

[*2] @ から始まる部分は Python ではデコレータとよばれ、その後の関数やメソッド等に機能を加える。

```
from abc import ABC, abstractmethod
class 抽象クラス名:
    @abstractmethod
    def 抽象メソッド名 1(self, 引数 1, 引数 2):
        pass
    @abstractmethod
    def 抽象メソッド名 2(self, 引数 1, 引数 2):
        pass
class 具象クラス名 1(抽象クラス名):
    def 抽象メソッド名 1(self, 引数 1, 引数 2):
        具体的なメソッド 1 の内容
        return メソッド 1 の返り値
    def 抽象メソッド名 2(self, 引数 1, 引数 2):
        具体的なメソッド 1 の内容
        return メソッド 1 の返り値
```

具体的には以下のような AbstractGraph クラスを抽象基底クラスとして定義し、AbstractGraph クラスの上で経路探索のためのメソッド`find_path_bfs`を定義することができる。

```
1  from abc import ABC, abstractmethod
2
3  class AbstractGraph:
4      @abstractmethod
5      def connect(self, x, y):
6      """
7      xからyへの枝を張る抽象メソッド
8      """
9          pass
10     @abstractmethod
11     def get_adjacent_nodes(self,x):
12     """
13     xから張られている枝の集合を取得する抽象メソッド
14     """
15         pass
16     def find_path_bfs(self,source,target):
17     """
18         幅優先探索で始点から終点までの経路を出力する抽象メソッド
19         Args:
20             source(int): 始点とする節点
21             target(int): 終点とする節点
22         Returns:
```

```
23              list: sourceからtargetまでの経由する節点を要素とするリスト
24      """
25          visited = [-1 for _ in range(self.num_nodes)]
26          q = queue.Queue()
27          q.put(source)
28          visited[source] = source
29          while not q.empty():
30              u = q.get()
31              for v in self.get_adjacent_nodes(u):
32                  if visited[v] == -1:
33                      visited[v] = u
34                      q.put(v)
35
36          if visited[target] == -1:
37              return
38          path = [] #pathには出力用の番号を格納する
39          u = target
40          while not u == source:
41              path.append(u)
42              u = visited[u]
43          path.append(source)
44          path.reverse()
45          return path
```

具象クラスは抽象クラスを継承して実装するクラスであり、抽象クラスでは定義されていない具体的な実装を行う。実際にプログラム中でインスタンス生成を行うときは、必ず具象クラスに対して用いることになる。隣接リストによる実装と隣接行列による実装をそれぞれ抽象クラスを継承するように別のクラスで定義して、抽象メソッドであった`connect`と`get_adjacent_nodes`の具体的な実装をすると以下のようになる。

```
5000  class DirectedGraphAdjacencyList(AbstractGraph):
5001      def __init__(self,n):
5002          # 全節点数を記録
5003          self.num_nodes = n
5004          # 節点数nの枝のないグラフで初期化する
5005          self.list = [ [] for _ in range(n) ]
5006      def connect(self, x, y):
5007          if not y in self.list[x]:
5008              self.list[x].append(y)
```

```
5009     def get_adjacent_nodes(self, x):
5010         return self.list[x]
5011 class DirectedGraphAdjencyMatrix(AbstractGraph):
5012     def __init__(self,n):
5013         self.num_nodes = n
5014         self.matrix = np.zeros([n,n]) #n行n列行列を作る関数
5015     def connect(self, x, y):
5016         self.matrix[x,y] = 1 # 行列のx,y成分を1とする
5017     def get_adjacent_nodes(self, x):
5018         r = []
5019         for i in range(self.num_nodes):
5020             if self.matrix[x,i] == 1:
5021                 r.append(i)
5022         return r
```

具体的なプログラムでは以下のように具象クラスのインスタンスを作成して用いる。同じテキストファイルから構築されるグラフの違いに注意。

具体的なプログラムの例 (DAG_find_path.py の内容)

```
1 string_list = input().split() #string_list は['16']のようなものになる
2 g = DirectedGraphAdjencyList(int(string_list[0])) #DirectedGraphAdjencyMatrixに変えても動作する
3 s,t = int(string_list[1]), int(string_list[2])
4 while True:
5     string_list = input().split() #string_list は['1', '2']のようなものになる
6     if len(string_list) == 0:
7         break
8     g.connect(int(string_list[0]),int(string_list[1]))
9 print(g.find_path_bfs(s,t))
```

実行例

```
1 $ python3 DAG_find_path.py < graph.txt
2 [ 12, 8, 9, 10, 6, 2, 3]
```

7.4　深さ優先によるサイクルの検出

節点 n から辿った先にサイクルがあるかどうかは、次のように定義された n を引数にしてvisit_and_detect_cycleを呼び出せばよい。

```
1: procedure visit_and_detect_cycle(節点 i)
2:     ここから先で節点 i に来た時に節点 i を既に訪れたことが分かるように記録する。
3:     for 節点 i から出る枝が入る節点 j do
4:         if 節点 n から節点 i までの経路の途中で既に j に訪れた then
5:             サイクルを検出。
6:         else
7:             visit_and_detect_cycle(節点 j) を実行する。
```

visit_and_detect_cycleは深さ優先探索の過程で訪れる節点から n までの経路に True が格納されているようなリスト visited を維持していく。抽象クラスにおける実装は以下のようになる。

```
#class AbstractGraph の中で定義する
    def find_cycle(self):
        """
        サイクルがあるか判定する
            Args:
                None
            Returns:
                Bool : サイクルがあるかどうか
        """
        for s in range(self.num_nodes):
            visited = [ False for _ in range(self.num_nodes)]
            if self.visit_and_detect_cycle(s,visited):
                return True
        return False
    def visit_and_detect_cycle(self, u, visited):
        """
        与えられた節点uから辿る先にサイクルがあるか判定する
            Args:
                u(int):
                visited(list(bool)): 今走査している経路に各節点が含まれているか否かを要素としたリスト
            Returns:
                Bool : uからたどった先にサイクルがあるかどうか
        """
        has_cycle = False
        for v in self.get_adjacent_nodes(u):
            if visited[v]:
                return True
            else:
                visited[v] = True
                if self.visit_and_detect_cycle(v,visited):
                    has_cycle = True
                visited[v] = False
        return has_cycle
```

7.5　トポロジカルソート

枝 (x, y) がある時に、x を y より前に置くような “整列” を**トポロジカルソート**という。サイクルがあるとトポロジカルソートは不可能であり、トポロジカルソートが可能ならサイクルはない。一つの DAG に対してトポロジカルソートの出力となり得るリストは複数通りあることがある。サイクルがないことが分かっている DAG に対するトポロジカルソートは以下のように求めることができる。

function topological_sort(G)
    各節点の入次数を計算し、x の入次数を $d_{\mathrm{in}}(x)$ としておく。
    $d_{\mathrm{in}}(x) = 0$ の節点 x に対して、x をキューに入れる。
    while キューが空でない **do**
        キューから x を取り出す
        x をリスト（スタック）に追加
        for y **in** 枝 (x, y) がある節点 **do**
            $d_{\mathrm{in}}(y)$ を 1 減らす。
            新たに $d_{\mathrm{in}}(y) = 0$ となった節点 y をキューに入れる。
    return リスト

次のようなメソッドを実装すればよい。

```
#class AbstractGraph の中で定義する
    def compute_indegrees(self):
        """
        各節点の入次数を要素とするリストを計算する
            Args:
                None
            Returns:
                list : 各節点の入次数を要素とするリスト
        """
        indegree = [0 for _ in range(self.num_nodes)]
        for i in range(self.num_nodes):
            for j in self.get_adjacency_nodes(i):
                indegree[j] += 1
```

```
        return indegree

    def topological_sort(self):
        """
        トポロジカルソートの計算を行う
            Args:
                None
            Returns:
                list : トポロジカルソートの計算結果
        """
        sorted_index = []
        indegree = self.compute_indegrees()
        q = queue.Queue()
        #各節点の入次数を計算し0になっている節点をキューに入れる
        for i in range(self.num_nodes):
            if indegree[i] == 0:
                q.put(i)

        while not q.empty():
            i = q.get()
            sorted_index.append(i)
            for j in self.get_adjacent_nodes(i):
                indegree[j] -= 1
                if indegree[j] == 0:
                    q.put(j)

        return sorted_index
```

7.6　隣接行列の積と経（発展）

隣接行列の巾乗を作り、その対角成分に 0 以外の要素がないかどうかでサイクルを検出することもできる。これは次の原理による。

- 節点 i から節点 j への枝がある。$\rightarrow A_{i,j} > 0$
- 節点 j から節点 k への枝がある。$\rightarrow A_{j,k} > 0$

- 節点 i から節点 j への枝があり、節点 j から節点 k に枝がある。$\rightarrow A_{i,j}A_{j,k} > 0 \rightarrow A_{i,1}A_{1,k} + \cdots + A_{i,j}A_{j,k} + \cdots > 0 \rightarrow (A^2)_{i,k} > 0$

従って、

- 節点 i から節点 k へ枝を 2 つ辿ればいける。$\rightarrow (A^2)_{i,k} > 0$

となる。逆に、

- $(A^2)_{i,k} > 0 \rightarrow$ ある j が存在して $A_{i,j} > 0$ かつ $A_{j,k} > 0 \rightarrow$ ある節点 j を経由しても節点 i から節点 k にいける

となる。以上 2 つをまとめると、

- 節点 i から節点 k へ、枝を 2 つ辿ればいける。$\leftrightarrow (A^2)_{i,k} > 0$

となる。一般化すると、

- 節点 i から節点 k へ、枝を n 個辿ればいける。$\leftrightarrow (A^n)_{i,k} > 0$

となる。これを使うと、

- 節点 i から節点 i へ、枝を n 個辿ればいける。$\leftrightarrow (A^n)_{i,i} > 0$

いろいろな n に対して A^n を計算して対角成分を調べることでサイクルを検出することができる。n は最大で全節点数だけ調べればよい。