

# プログラミングA2

## 第7回

練習問題解答例

# 復習問題：stack\_queue2.py

インスタンス変数としてリストを持つのではなく、`list`クラスを継承して`Stack`と`Queue`を実装せよ。

stack\_queue2.py

```
class Stack(list):  
    def push(self, item):  
        self.append(item)  
  
    def pop(self):  
        if len(self) == 0:  
            return None  
        return super().pop()
```

stack\_queue2.py

```
class Queue(list):  
    def enqueue(self, item):  
        self.append(item)  
  
    def dequeue(self):  
        if len(self) == 0:  
            return None  
        return self.pop(0)
```

※`super()`により親クラス(`list`クラス)の`pop()`を呼び出さないと、自クラス(`Stack`クラス)内の`pop()`の中で自分自身を呼び出す再帰呼び出し(`recursive call`)になってしまう。

※ちなみに、`stack_queue1.py`のように、  
**継承**「is-a」ではなく他クラスのインスタンスを属性としてもつことを  
**合成**、または、**包含**「has-a」と呼び、継承と対照的な概念である。

# 練習問題：rev\_pol\_not.py

## ●逆ポーランド記法の数式評価

- オペレータ(演算子)をオペランドの後ろに記述する事で計算の優先順位を表す記法である。(括弧を使わない)

中置記法：100 - (2 + 3) x (4 + 5)

逆ポーランド記法：100 2 3 + 4 5 + x -

中置記法：5 x (4 + 3)

逆ポーランド記法：5 4 3 + x

- ①スタックを空にする
- ②対象文字列を順に走査する
  - ③数値が現れたらスタックにpushする
  - ④演算子が現れたら
    - ⑤スタックから2つの数値をpopし、取り出した順にrightとleftとする
    - ⑥数式文字列を評価し、結果を文字列としてスタックにpushする

# 解答例：rev\_pol\_not.py

rev\_pol\_not.py

```
class RevPolishNotation:
```

```
    operators = {"+", "-", "*", "/"}
```

```
    def __init__(self, expression: str):  
        self.items: list = expression.split()
```

← 逆ポーランド記法の数式文字列を  
スペース区切りで分割

```
    def calculate(self):
```

① `stk = deque()`

② `for s in self.items:`

← 逆ポーランド記法のリスト

`try:`

`float(s)`

③ `stk.append(s)`

`except Exception as e:`

④ `if s in __class__.operators:`

⑤ `right= stk.pop()` # 後に入れた方が演算子の右側

`left = stk.pop()` # 先に入れた方が演算子の左側

`expression = left+s+right`

`result = eval(expression)` # 数式文字列を評価

⑥ `stk.append(str(result))`

# 実行例：rev\_pol\_not.py

rev\_pol\_not.py

```
rpn = RevPolishNotation("1 2 + 3 4 + *")
rpn.calculate()

print("-"*30)
rpn = RevPolishNotation("5 4 3 + *")
rpn.calculate()

print("-"*30)
rpn = RevPolishNotation("3 4 + 1 2 - *")
rpn.calculate()
```

実行例

```
1+2  3
3+4  7
3*7  21
-----
4+3  7
5*7  35
-----
3+4  7
1-2  -1
7*-1 -7
```

# 練習問題：shortest\_path.py

## ● 幅優先探索 (Breadth First Search) による 迷路上の距離計算

shortest\_path.py

```
class ShortestPath:
    def __init__(self, map: Maze):
        self.map = map

    def bfs(self):
        ここを実装する

if __name__ == "__main__":
    tate, yoko = 9, 15
    maze = Maze(tate, yoko)
    maze.show_maze()
    sp = ShortestPath(maze)
    sp.bfs()
    maze.show_maze()
```

イメージ図

-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	12	11	10	-1	12
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	11	-1	9	-1	11
-1	-1	-1	-1	-1	-1	-1	-1	12	-1	10	-1	8	-1	10
-1	-1	-1	-1	-1	-1	-1	-1	11	-1	9	-1	7	-1	9
-1	-1	-1	-1	-1	-1	-1	-1	10	9	8	7	6	7	8
-1	-1	-1	-1	-1	-1	-1	-1	11	-1	9	-1	5	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1	12	-1	10	-1	4	3	2
-1	-1	-1	-1	-1	-1	13	-1	11	-1	9	-1	5	-1	1
-1	-1	-1	-1	-1	13	12	11	10	9	8	7	6	-1	0

← 縦9マス，横15マスの迷路生成

← BFSでスタートマスから全マスの距離を計算

# 迷路生成：maze\_maker.py

## ● 1 マス

maze\_maker.py

```
@dataclass
class Cell:
    """
    迷路の1マスを表わすクラス
    state: " "が床, "#"が壁, "S"がスタート, "G"がゴール
    """
    y: int
    x: int
    state:str = " " # デフォルトで床
    dist: int = -1 # スタートからの距離
    adj: list = field(default_factory=list)
    parent: "Cell" = None
```

← 空のlistをデフォルト値とする場合

# 迷路生成：maze\_maker.py

## ● 迷路全体

maze\_maker.py

```
class Maze:
```

```
    def __init__(self, tate, yoko):  
        self.tate, self.yoko = tate, yoko  
        self.map = self.generate()
```

← Cellインスタンスが並ぶ二次元リスト

```
    def generate(self):
```

```
        """
```

```
        self.tate x self.yokoの迷路を生成する  
        Cellインスタンスが並ぶ二次元リストmaze_lstを返す
```

```
        """
```

```
        省略
```

```
    def get_adj(self, current: Cell) -> list[Cell]:
```

```
        """
```

```
        隣接するマスのうち、壁でないマスのリストを返す
```

```
        """
```

```
        省略
```

```
    def show_maze(self):
```

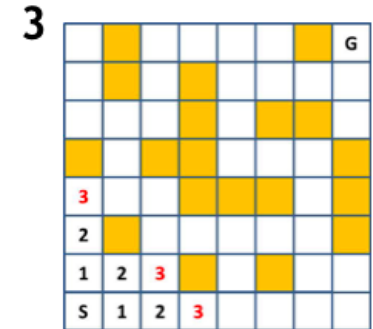
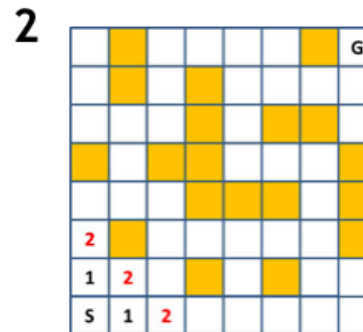
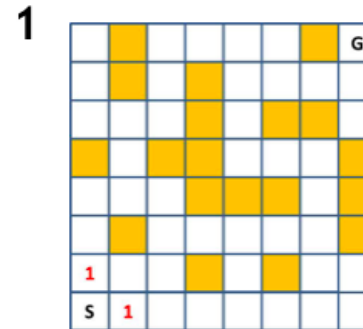
```
        省略
```



# キューと幅優先探索

## ●幅優先探索

- スタート(距離:0)から近い順に距離を求めていく
  - スタートから1歩で行けるマスをクリックに入れる(距離:1)
  - キューのマスを取り出し、それらから1歩で行けるマスをクリックに入れる(距離:2)  
※既出マスは除く
  - キューのマスを取り出し、それらから1歩で行けるマスをクリックに入れる(距離:3)  
※既出マスは除く



# アルゴリズム

## ●幅優先探索による迷路上の距離計算

- ①空のキューにスタートマスを追加する
  - スタートマスの距離は0に設定する
- ②キューが空になるまで、以下を繰り返す
  - ③キューからマスを1つdequeueする → `current`とする
  - ④`current`から1歩で行けるマスに対して、 → `cell`とする
    - ⑤既に訪れていたら(=距離の値が設定されていたら)何もしない
    - ⑥`cell`の距離に、`current`の距離+1を設定する
    - ⑦`cell`をキューにenqueueする

# 解答例：shortest\_path.py

shortest\_path.py

```
def bfs(self):
    que = deque()
    start = self.map.start
    start.dist = 0
    que.append(start)
    while True:
        try:
            current = que.popleft()
        except:
            break
        print(current)
        if current.state == "G":
            break
        for cell in self.map.get_adj(current):
            if cell.dist != -1: # ⑤既に訪れていたなら
                continue
            cell.dist = current.dist+1
            que.append(cell)
```

← スタートマスの  
距離を0に設定

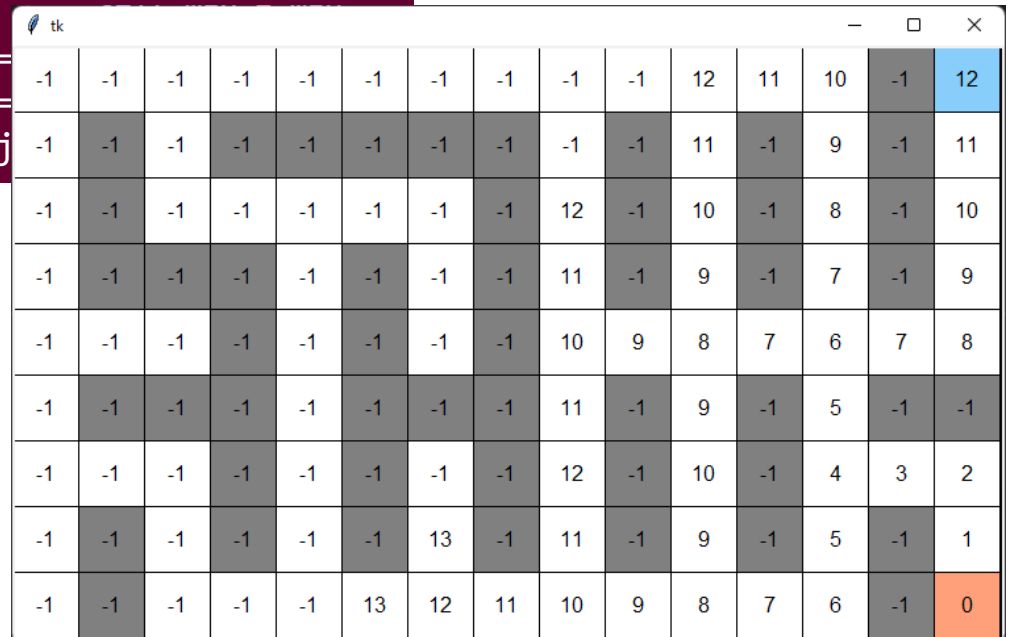
← ④currentの隣接マス  
を取得(壁を除く)

# 実行例：shortest\_path.py

## 実行例

```
Cell(y=8, x=14, state='S', dist=0, adj=[], parent=None)
Cell(y=7, x=14, state=' ', dist=1, adj=[], parent=None)
Cell(y=6, x=14, state=' ', dist=2, adj=[], parent=None)
Cell(y=6, x=13, state=' ', dist=3, adj=[], parent=None)
Cell(y=6, x=12, state=' ', dist=4, adj=[], parent=None)
Cell(y=7, x=12, state=' ', dist=5, adj=[], parent=None)
Cell(y=5, x=12, state=' ', dist=5, adj=[], parent=None)
```

```
:
Cell(y=6, x=8, state=' ', dist=12, adj=
Cell(y=8, x=6, state=' ', dist=12, adj=
Cell(y=0, x=14, state='G', dist=12, adj=
```



-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	12	11	10	-1	12
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	11	-1	9	-1	11
-1	-1	-1	-1	-1	-1	-1	-1	12	-1	10	-1	8	-1	10
-1	-1	-1	-1	-1	-1	-1	-1	11	-1	9	-1	7	-1	9
-1	-1	-1	-1	-1	-1	-1	-1	10	9	8	7	6	7	8
-1	-1	-1	-1	-1	-1	-1	-1	11	-1	9	-1	5	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1	12	-1	10	-1	4	3	2
-1	-1	-1	-1	-1	-1	13	-1	11	-1	9	-1	5	-1	1
-1	-1	-1	-1	-1	13	12	11	10	9	8	7	6	-1	0