

2022年11月14日(月) 4限
@研究棟A302

プログラミングA2 第7回

担当：伏見卓恭

連絡先：fushimity@edu.teu.ac.jp

居室：研A1201

プログラミングA2の流れ

- 第 1回：＜復習編＞関数，ファイル入出力，コンテナデータ型
- 第 2回：＜復習編＞クラスとオブジェクト
- 第 3回：＜文法編＞関数の高度な利用法 1
- 第 4回：＜文法編＞関数の高度な利用法 2
- 第 5回：＜文法編＞オブジェクト指向プログラミング
- 第 6回：＜応用編＞データ構造とアルゴリズム 1
- 第 7回：＜応用編＞データ構造とアルゴリズム 2
- 第 8回：＜実践編＞HTTPクライアント
- 第 9回：＜実践編＞スクレイピング
- 第10回：＜実践編＞データベース
- 第11回：＜実践編＞並行処理
- 第12回：＜総合編＞総合演習(複合問題)
- 第13回：＜総合編＞まとめ
- 第14回：＜総合編＞Python力チェック ← 確認テストのこと

本日のお品書き

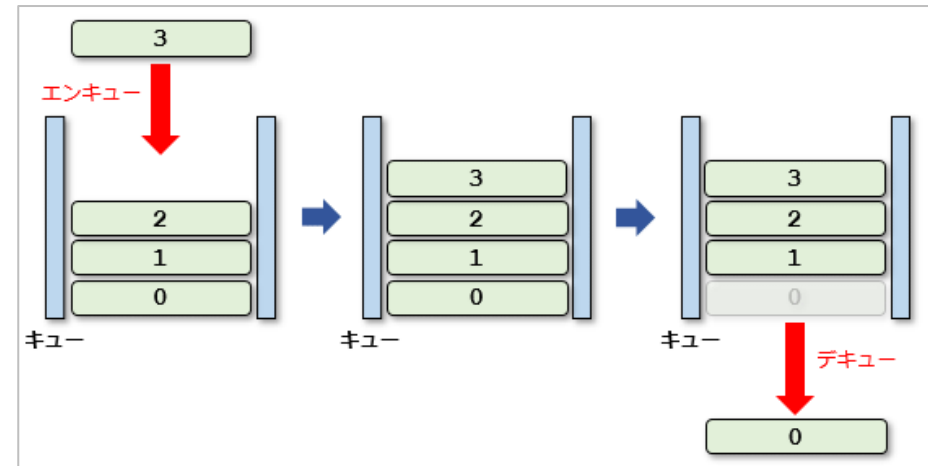
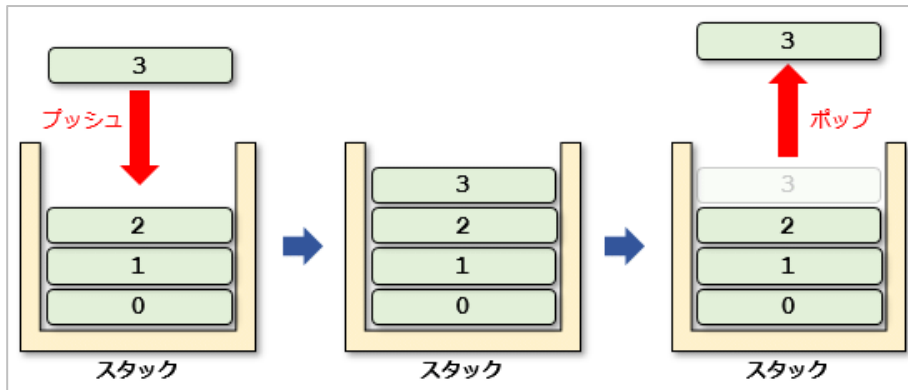
- データ構造と**アルゴリズム 2**
 - スタックとキューとデック
 - スタックを用いたアルゴリズム
 - キューを用いたアルゴリズム
 - 優先度付きキュー~~を用いたアルゴリズム~~
 - 時間計測
 - ハッシュ値の役割

スタックとキューとデック

スタックとキュー

スタック(stack)とは, LIFO(Last In, First Out)のデータ構造である.
= 最後に追加(push)した要素から順に抽出(pop)される

キュー(queue)とは, FIFO(First In, First Out)のデータ構造である.
= 最初に追加(enqueue)した要素から順に抽出(dequeue)される



図の出典: <https://atmarkit.itmedia.co.jp/ait/articles/1908/06/news015.html>

デック(deque)とは, スタックとキューの機能を備えたデータ構造である.
= 先頭にも末尾にも要素を追加, 先頭からも末尾からも要素を抽出できる.

コード例：stack_queue1.py

インスタンス変数としてリストを持つような
StackクラスとQueueクラスを実装せよ。

stack_queue1.py

```
class Stack:
```

```
    def __init__(self):  
        self._lst = []
```

← インスタンス変数
としてリストを持つ

```
    def push(self, item):  
        self._lst.append(item)
```

↑
内部リストにappend

```
    def pop(self):  
        if len(self._lst) == 0:  
            return None  
        return self._lst.pop()
```

↑
内部リストの末尾をポップ

stack_queue1.py

```
class Queue:
```

```
    def __init__(self):  
        self._lst = []
```

```
    def enqueue(self, item):  
        self._lst.append(item)
```

```
    def dequeue(self):  
        if len(self._lst) == 0:  
            return None  
        return self._lst.pop(0)
```

↑
内部リストの先頭をポップ

実行例：stack_queue1.py

stack_queue1.py

```
if __name__ == "__main__":
    stk = Stack()
    stk.push(Monster("ピカチュウ"))
    stk.push(Monster("カイリュウ"))
    stk.push(Monster("ヤドラン"))
    stk.push(Monster("ピジョン"))
    print(stk)

    print("-"*10)
    for i in range(5):
        print(stk.pop())
    print("#####")
    que = Queue()
    que.enqueue(Monster("ピカチュウ"))
    que.enqueue(Monster("カイリュウ"))
    que.enqueue(Monster("ヤドラン"))
    que.enqueue(Monster("ピジョン"))
    print(que)

    print("-"*10)
    for i in range(5):
        print(que.dequeue())
```

実行例

```
4: ピジョン
3: ヤドラン
2: カイリュウ
1: ピカチュウ
```

```
-----
ピジョン
ヤドラン
カイリュウ
ピカチュウ
None
```

```
#####
1: ピカチュウ 2: カイリュウ 3: ヤ
ドラン 4: ピジョン
-----
ピカチュウ
カイリュウ
ヤドラン
ピジョン
None
```

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(継承):  
    def push(self, item):  
        自身がリスト(item)  
  
    def pop(self):  
        if len(self) == 0:  
            return None  
        return ※注意が必要
```

stack_queue2.py

```
class Queue(継承):  
    def enqueue(self, item):  
        自身がリスト(item)  
  
    def dequeue(self):  
        if len(self) == 0:  
            return None  
        return 自身がリスト
```

※`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。

標準ライブラリのスタック

deque : <https://docs.python.org/ja/3.9/library/collections.html#collections.deque>

queueモジュール : <https://docs.python.org/ja/3.9/library/queue.html>

●collections.deque

dequeの使い方

```
from collections import deque
```

```
stk = deque()  
stk.append(要素)  
stk.pop() # 要素の抽出
```

※要素がない場合, IndexErrorになる

stack_queue4.py

```
stk = deque()  
stk.append(Monster("ピカチュウ"))  
stk.append(Monster("カイリュウ"))  
stk.append(Monster("ヤドラン"))  
stk.append(Monster("ピジョン"))  
for i in range(5):  
    print(stk.pop())
```

●queue.LifoQueue

LifoQueueの使い方

```
from queue import LifoQueue
```

```
stk = LifoQueue()  
stk.put(要素)  
stk.get() # 要素の抽出  
stk.empty() # 空チェック
```

※要素がない場合, ブロック状態になる

stack_queue3.py

```
stk = LifoQueue()  
stk.put(Monster("ピカチュウ"))  
stk.put(Monster("カイリュウ"))  
stk.put(Monster("ヤドラン"))  
stk.put(Monster("ピジョン"))  
for i in range(5):  
    print(stk.get())
```

標準ライブラリのキュー

deque : <https://docs.python.org/ja/3.9/library/collections.html#collections.deque>

queueモジュール : <https://docs.python.org/ja/3.9/library/queue.html>

●collections.deque

dequeの使い方

```
from collections import deque
```

```
que = deque()  
que.append(要素)  
que.popleft() # 要素の抽出
```

※要素がない場合, IndexErrorになる

stack_queue4.py

```
que = deque()  
que.append(Monster("ピカチュウ"))  
que.append(Monster("カイリュウ"))  
que.append(Monster("ヤドラン"))  
que.append(Monster("ピジョン"))  
for i in range(5):  
    print(que.popleft())
```

●queue.Queue

Queueの使い方

```
from queue import Queue
```

```
que = Queue()  
que.put(要素)  
que.get() # 要素の抽出  
que.empty() # 空チェック
```

※要素がない場合, ブロック状態になる

stack_queue3.py

```
que = Queue()  
que.put(Monster("ピカチュウ"))  
que.put(Monster("カイリュウ"))  
que.put(Monster("ヤドラン"))  
que.put(Monster("ピジョン"))  
for i in range(5):  
    print(que.get())
```

スタックを用いた アルゴリズム

スタックを用いたアルゴリズム例

●括弧の対応関係をチェックする

- ①スタックを空にする
- ②対象文字列を順に走査する
 - ③左括弧が現れたらスタックにpushする
 - ④右括弧が現れたら
 - ⑤スタックが空でなかったら, popする
 - ⑥スタックが空ならエラーを返す (=対応する左括弧がない)

check_brackets.py

```
def check(self):
```

```
①  stk = deque()
```

```
②  for s in self.sentence:
```

```
③      if s in __class__.left:  
          stk.append(s)
```

```
④      if s in __class__.right:  
          try:
```

```
⑤          item = stk.pop()  
          if __class__.left[item] == __class__.right[s]:  
              print(f"一致しました:{item} {s}")
```

```
⑥      except Exception as e:  
          print(f"一致する左括弧がありません{e}")
```

実行例：check_brackets.py

check_brackets.py

```
bc = BracketChecker("(())(())((()()))(())(())")
bc.check()
print("-"*30)
bc = BracketChecker("if ((key in self and self[key] != value) or (key not in self)):")
bc.check()
print("-"*30)
bc = BracketChecker("\delta(d) \&\& \frac{1}{\gamma(d)-1} \sum_{\substack{j = 1 \\ \forall c_j, c_{j+1} \in \Gamma(d)}}^{\gamma(d)-1} \{c_{j+1}.time - c_j.time\}")
bc.check()
```

実行例

一致しました：()
一致しました：()
一致しました：()
一致しました：()
一致しました：()
一致しました：()
一致しました：()
一致しました：()
一致しました：()
一致する左括弧がありません
一致しました：()
一致する左括弧がありません
一致する左括弧がありません
一致しました：()

実行例

一致しました：[]
一致しました：()
一致しました：()
一致しました：()

実行例

一致しました：()
一致しました：{ }
一致しました：()
一致しました：{ }
一致しました：{ }
一致しました：()
一致しました：{ }
一致しました：{ }
一致しました：()
一致しました：{ }
一致しました：{ }
一致しました：{ }
一致しました：{ }

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:`
 `try:`

← 逆ポーランド記法のリスト

数値に変換

③ `スタックに数字を追加`

```
    except Exception as e:
```

④ `if s in __class__.operators:`

⑤ `right =` `スタックからpop` `# 後に入れた方が演算子の右側`
`left =` `# 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 `スタックに計算結果の数字を追加`

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
```


キューを用いた アルゴリズム

キューを用いたアルゴリズム例

● ラウンドロビン・スケジューリング

- キューに到着したプロセスを順に処理するが、各プロセスはタイムクオンタムだけ処理され、終わらなかった場合はキューの一番後ろに回される。
- ①空のキューにプロセスを追加する
- ②キューが空になるまで、以下を繰り返す
 - ③キューから**dequeue**したプロセスに対して、所要時間からクオンタム分を引き、
 - ④残り時間が0より大きかったらキューに**enqueue**する
 - ⑤残り時間が0以下ならプロセス終了を**print**する

round_robin.py

```
def schedule(self, processes):  
    ① que = deque(processes)  
    ② while True:  
        try:  
            ③ p = que.popleft()  
            p.duration -= self.quantum  
            ④ if p.duration > 0:  
                que.append(p)  
                print(f"未完: {p}")  
            ⑤ else:  
                print(f"完了: {p}")  
        except Exception as e:  
            break
```

実行例：round_robin.py

round_robin.py

```
@dataclass
class Process:
    title: str = None
    duration: int = 0

if __name__ == "__main__":
    rr = RoundRobin(5) # 1回の処理は5秒のみ
    rr.schedule([Process(f"プロセス{i+1}", randint(1, 15))
                 for i in range(10)])
```

実行例

```
deque([Process(title='プロセス1', duration=13), Process(title='プロセス2',
duration=7), Process(title='プロセス3', duration=3), ..., Process(title='プロ
セス10', duration=11)])
未完：Process(title='プロセス1', duration=8)
未完：Process(title='プロセス2', duration=2)
完了：Process(title='プロセス3', duration=-2)
：
全所要時間：108
```

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

← 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
```

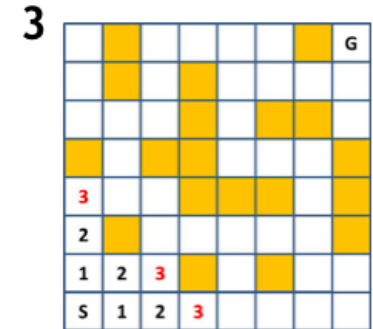
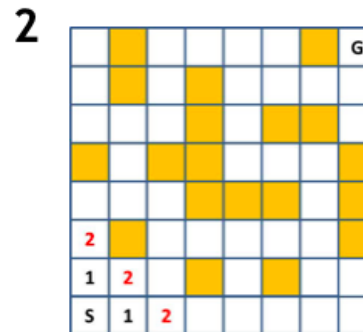
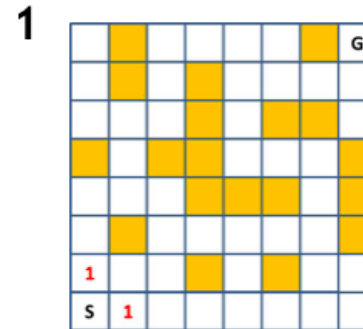
```
    def show_maze(self):
```

```
        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
    ①スタートマス enqueue
    while True:
        try:
            current = ③キューからマスを1つdequeue
        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 = ⑥currentの距離+1を設定する
            ⑦cellをキューにenqueueする
```

← スタートマスの
距離を0に設定

③キューからマスを1つdequeue

← ④currentの隣接マス
を取得(壁を除く)

⑥currentの距離+1を設定する

⑦cellをキューにenqueueする

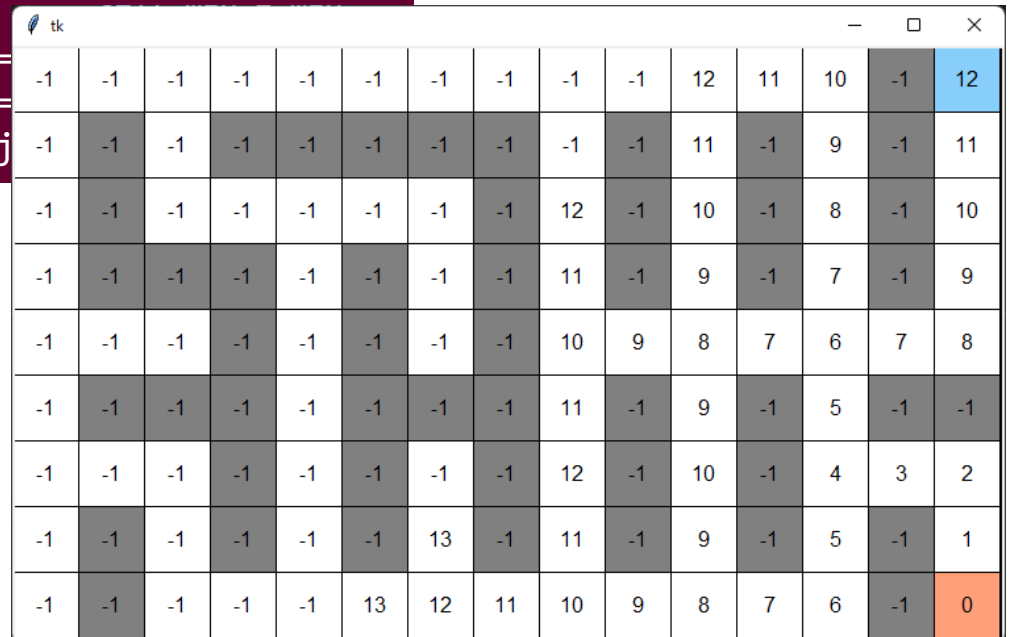
実行例：shortest_path.py

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=
```

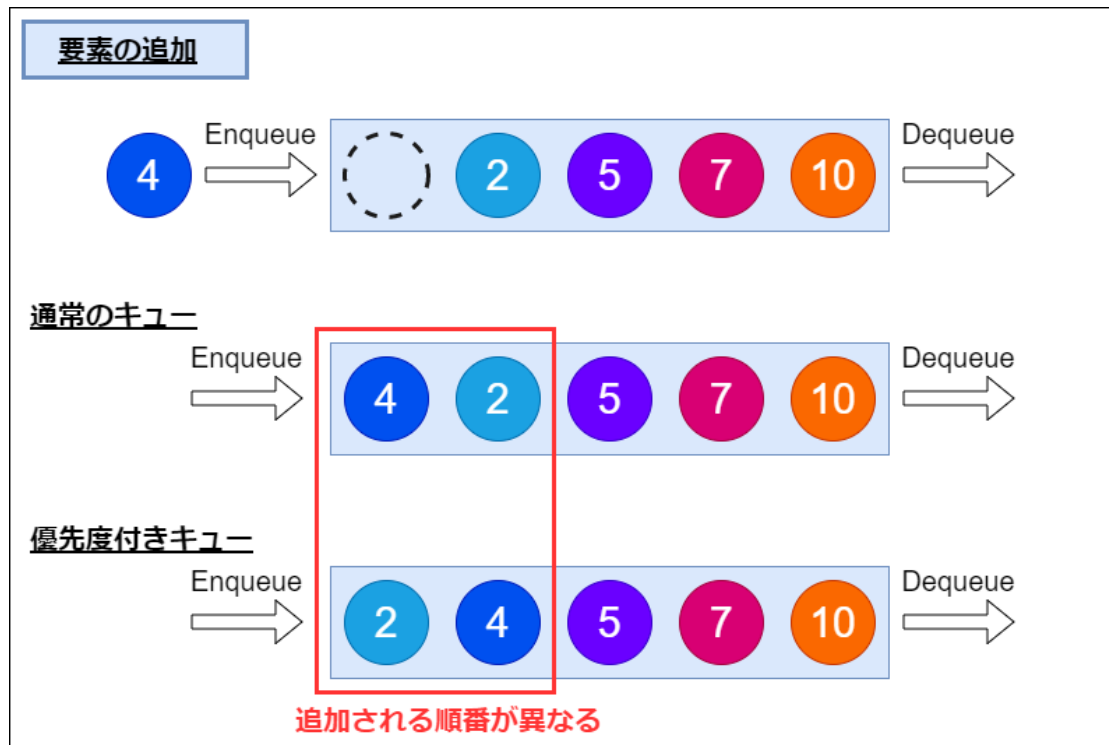


-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	12	11	10	-1	12
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	11	-1	9	-1	11
-1	-1	-1	-1	-1	-1	-1	-1	12	-1	10	-1	8	-1	10
-1	-1	-1	-1	-1	-1	-1	-1	11	-1	9	-1	7	-1	9
-1	-1	-1	-1	-1	-1	-1	-1	10	9	8	7	6	7	8
-1	-1	-1	-1	-1	-1	-1	-1	11	-1	9	-1	5	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1	12	-1	10	-1	4	3	2
-1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

優先度付きキュー

優先度付きキューとは

優先度付きキュー(priority queue)とは、基本的にはFIFO(First In, First Out)のデータ構造であるが、各要素が優先度を持っており、優先度が高い要素から順に抽出(dequeue)されるように要素を追加(enqueue)する。



図の出典：<https://novnote.com/priority-queue-heapsort-impl-cpp/565/>

例題：prior_queue1.py

インスタンス変数としてリストを持つような
PriorityQueueクラスを実装せよ。

prior_queue1.py

青字追加箇所

```
class PriorityQueue:
    def __init__(self):
        self._lst = []

    def enqueue(self, item):
        self._lst.append(item)
        self._lst.sort(key=lambda item: item.level, reverse=True)

    def dequeue(self):
        if len(self._lst) == 0:
            return None
        return self._lst.pop(0)
```

↑ enqueueするたびに
内部リストを優先度で降順ソート

実行例：prior_queue1.py

prior_queue1.py

```
que = PriorityQueue()
que.enqueue(Monster(25, "ピカチュウ"))
que.enqueue(Monster(60, "カイリュウ"))
que.enqueue(Monster(40, "ヤドラン"))
que.enqueue(Monster(20, "ピジョン"))
print(que)

for i in range(2): print(que.dequeue())

que.enqueue(Monster(15, "コダック"))
que.enqueue(Monster(10, "コラッタ"))
que.enqueue(Monster(30, "ズバット"))
que.enqueue(Monster(45, "ギャロップ"))
print(que)
```

キューに入っている要素のうち
優先度（レベル）が高い要素が
前に来るようにenqueueされる
↓

実行例

1: カイリュウ(Lv.60) 2: ヤドラン(Lv.40) 3: ピカチュウ(Lv.25) 4: ピジョン(Lv.20)

カイリュウ(Lv.60)
ヤドラン(Lv.40)

1: ギャロップ(Lv.45) 2: ズバット(Lv.30) 3: ピカチュウ(Lv.25) 4: ピジョン(Lv.20)
5: コダック(Lv.15) 6: コラッタ(Lv.10)

標準ライブラリの優先度付きキュー

heapq : <https://docs.python.org/ja/3/library/heapq.html>

queueモジュール : <https://docs.python.org/ja/3.9/library/queue.html>

●heapq

heapqの使い方

```
import heapq

que = []
heapq.heappush(que, (優先度, 要素))
heapq.heappop(que) # 要素の抽出
```

※要素がない場合, IndexErrorになる

prior_queue2.py

```
que = []
heapq.heappush(que, (25, "ピカチュウ"))
heapq.heappush(que, (60, "カイリュウ"))
heapq.heappush(que, (40, "ヤドラン"))
heapq.heappush(que, (20, "ピジョン"))

for i in range(5):
    print(heapq.heappop(que))
```

●queue.PriorityQueue

PriorityQueueの使い方

```
from queue import PriorityQueue

que = PriorityQueue()
que.put((優先度, 要素))
que.get() # 要素の抽出
que.empty() # 空チェック
```

※要素がない場合, ブロック状態になる

prior_queue3.py

```
que = PriorityQueue()
que.put((25, "ピカチュウ"))
que.put((60, "カイリュウ"))
que.put((40, "ヤドラン"))
que.put((20, "ピジョン"))
for i in range(5):
    print(que.get())
```

時間計測

timeモジュール

<https://docs.python.org/ja/3/library/time.html>

●time() :

- 現在時刻(1970年1月1日0時0分0秒からの経過秒)を返す
- システム依存の誤差が生じる

●perf_counter() :

- パフォーマンスカウンターの値を返す
- **sleep**も含めてカウントする

●process_time() :

- 現在のプロセスのシステムおよびユーザーCPU時間の合計値を返す
- **sleep**はカウントしない

●sleep(秒) :

- 引数で指定した秒だけ、処理を一時停止する

時間計測：time_measure.py

time_measure.py

```
from time import time, perf_counter, process_time, sleep

begin = [time(), perf_counter(), process_time()]
sleep(3)
end = [time(), perf_counter(), process_time()]
print("sleep(3):")
for b, e in zip(begin, end):
    print((e-b))

begin = [time(), perf_counter(), process_time()]
s = ""
for i in range(1000000): s+=str(i)
end = [time(), perf_counter(), process_time()]
print("str concat1")
for b, e in zip(begin, end):
    print((e-b))

begin = [time(), perf_counter(), process_time()]
s = "".join([str(i) for i in range(1000000)])
end = [time(), perf_counter(), process_time()]
print("str concat2")
for b, e in zip(begin, end):
    print((e-b))
```

実行例

```
sleep(3):
3.0051469802856445
2.9898312999999996
0.0
#####
str concat1
1.145174503326416
1.1455904999999995
1.140625
#####
str concat2
0.1543118953704834
0.16059710000000038
0.15625
```

- `process_time()`は`sleep`の時間を計測しない
- 「+」演算子による文字列結合のたびに新たな`str`インスタンスを生成する→遅い

ハッシュ値の役割

集合の要素や辞書のキーは、一意性の制約がある。
すなわち、等価なオブジェクトを複数登録できない。
新たなオブジェクトを集合や辞書に登録するたびに、多数の要素やキーとの等価性をチェックするのは大変である。
ハッシュ値により、オブジェクトの等価性チェックの事前チェックをする。

- 命題： $a == b \Rightarrow \text{hash}(a) == \text{hash}(b)$
- 対偶： $\text{hash}(a) \neq \text{hash}(b) \Rightarrow a \neq b$

事前チェック：

ハッシュ値が等しくないオブジェクト同士は、等価でない。 【確定】

等価でないオブジェクトが同じハッシュ値を持つことはあり得るので、事前チェックにより等価と判断された【暫定】オブジェクトは、
`__eq__`による本番チェックにより等価性をチェックする。 【確定】

不適切なハッシュ値(その1)

hash_value.py

```
class Monster:
    def __init__(self, title):
        self.title = title

    def __repr__(self):
        return self.title

    def __eq__(self, other:"Monster"):
        return self.title == other.title

    def __hash__(self):
        """
        不適切なハッシュ値を返す
        等価なオブジェクトでも異なるハッシュ値となる可能性がある
        """
        return randint(1, 1000000000)
```

確かめよう：hash_value.py

hash_value.py

```
pika1 = Monster("ピカチュウ")  
pika2 = Monster("ピカチュウ")  
fushi = Monster("フシギダネ")
```

```
print(pika1 == pika2) →  
print(hash(pika1) == hash(pika2)) →  
print(pika1 == fushi) →  
print(hash(pika1) == hash(fushi)) →
```

```
mon_set = set([pika1, pika2, fushi])  
# 一応ハッシュ値を返す__hash__を持つので、集合の要素となれる  
print(mon_set)
```

__eq__()では等価なオブジェクトだが、→
ハッシュ値が異なるので事前チェックで
等価でないという結論に至り、
複数登録できてしまった例

実行例

```
True  
False  
False  
False
```

```
{ピカチュウ,  
ピカチュウ,  
フシギダネ}
```

不適切なハッシュ値(その2)

hash_value.py

```
class Monster:
    def __init__(self, title):
        self.title = title

    def __repr__(self):
        return self.title

    def __eq__(self, other: "Monster"):
        return self.title == other.title

    def __hash__(self):
        """
        不適切なハッシュ値を返す
        等価でないオブジェクトでも同じハッシュ値となる
        """
        return 1
```

確かめよう : hash_value.py

hash_value.py

```
pika1 = Monster("ピカチュウ")  
pika2 = Monster("ピカチュウ")  
fushi = Monster("フシギダネ")
```

```
print(pika1 == pika2) →  
print(hash(pika1) == hash(pika2)) →  
print(pika1 == fushi) →  
print(hash(pika1) == hash(fushi)) →
```

```
mon_set = set([pika1, pika2, fushi])  
# 一応ハッシュ値を返す__hash__を持つので、集合の要素となれる  
print(mon_set)
```

実行例

```
True  
True  
False  
True
```

```
{ピカチュウ,  
フシギダネ}
```

__eq__()では等価でないオブジェクトだが、→
ハッシュ値が等しいので事前チェックで
等価の可能性ありという結論に至った。
その後の__eq__()による本番チェックで
等価でないことが確定し、いずれも登録できた例

※時間のかかる本番チェックが必要になり非常に効率が悪い

適切なハッシュ値

hash_value.py

```
class Monster:
    def __init__(self, title):
        self.title = title

    def __repr__(self):
        return self.title

    def __eq__(self, other: "Monster"):
        return self.title == other.title

    def __hash__(self):
        """
        適切なハッシュ値を返す
        等価なオブジェクトは同じハッシュ値となる
        """
        return hash(self.title)
```


確かめよう：hash_value.py

hash_value.py

```
pika1 = Monster("ピカチュウ")  
pika2 = Monster("ピカチュウ")  
fushi = Monster("フシギダネ")
```

```
print(pika1 == pika2) →  
print(hash(pika1) == hash(pika2)) →  
print(pika1 == fushi) →  
print(hash(pika1) == hash(fushi)) →
```

```
mon_set = set([pika1, pika2, fushi])  
# 一応ハッシュ値を返す__hash__を持つので、集合の要素となれる  
print(mon_set)
```

実行例

```
True  
True  
False  
False
```

↑
オブジェクトの等価性と
ハッシュ値の等価性が一致

```
{ピカチュウ,  
フシギダネ}
```

ハッシュ値による事前チェック

●ハッシュ値による事前チェックは効率がいいの？

- 類似のハッシュ値をバケット(バケツ)に入れて、そのバケツ内で等価なオブジェクトを探索する
= 探索空間が小さいから効率がいい

●ハッシュ値の取りうる値の範囲が狭いと【衝突】が頻繁に発生し効率悪い

- バケツが少ない
= 多くのオブジェクトがバケツ内に入る
= 探索空間が広いから効率が悪い

オブジェクト	ハッシュ値	バケット番号
Monster("ピカチュウ")	432194871	43219
Monster("ピカチュウ")	432194871	43219
Monster("フシギダネ")	902318911	90231
Monster("フシギソウ")	902312478	90231

※ハッシュ値そのものをバケット番号にする場合が多い

確かめよう : hash_value.py

hash_value.py

```
titles = read_names("lec06/data/poke_names.txt")
mons = set([Monster(title) for title in titles])
begin = process_time()
for _ in range(999):
    mons |= set([Monster(title) for title in titles])
end = process_time()
print(end-begin)
print(len(mons))
```

↑

251匹のMonsterインスタンスの集合
の和集合を1000回計算

● 不適切なハッシュ値(その1 : 乱数)

※確かめよう : 乱数の範囲を狭めてみると遅くなるよ

実行例

0.546875
251000

● 不適切なハッシュ値(その2 : 全て1)

実行例

7.09375
251

● 適切なハッシュ値

実行例

0.125
251