Computer Science

# COMPSCI 130 S1

**Assignment**

| | |
|---|---|
| Due: | **11.59 pm Saturday 22nd May 2021** |
| Worth: | **6% of the final mark** |
| Marked out of: | **120** |

## *Introduction*

Solitaire (also known as “Klondike” and “Patience”) is a game which is played with a standard 52-card deck, without Jokers. An example is shown in Figure 1.

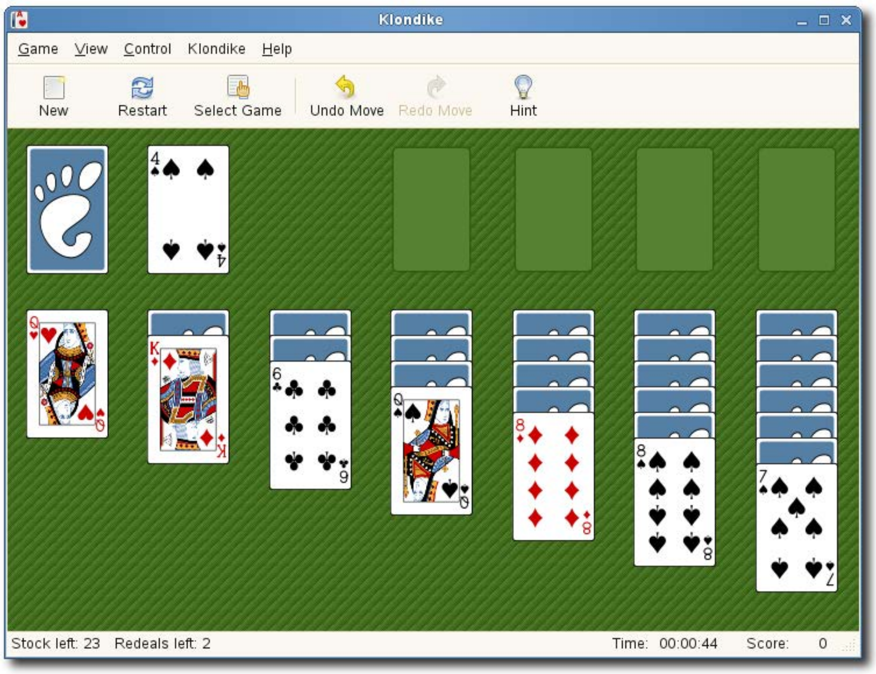

Figure 1 A version of Klondike [1]

After shuffling, seven piles of cards are laid from left to right. Each pile begins with one upturned card. From left to right, each pile contains one more card than the last. The first and left-most pile contains a single upturned card, the second pile contains two cards (one downturned, one upturned), the third contains three (two downturned, one upturned), and so on, until the seventh pile which contains seven cards (six downturned, one upturned). Figure 1 shows an example.

The four foundations (light rectangles in the upper right of Figure 1) are built up by suit from Ace (low in this game) to King, and the tableau piles can be built down by alternate colours, and partial or complete piles can be moved if they are built down by alternate colours also. Any empty piles can be filled with a King or a pile of cards with a King. The aim of the game is to build up a stack of cards starting with two and ending with King, all of the same suit. Once this is accomplished, the goal is to move this to a foundation, where the player has previously placed the Ace of that suit. Once the player has done this, they will have "finished" that suit, the goal being to finish all suits, at which time the player would have won (see https://en.wikipedia.org/wiki/Klondike_%28 solitaire%29).

In this assignment, you are going to implement a simplified version of this game using Python.

## *Warning*

**DO NOT SUBMIT SOMEONE ELSE'S WORK:**

- The work done on this assignment must be your own work. Think carefully about any problems you come across, and try to solve them yourself before you ask anyone for help.
- Under no circumstances should you take or pay for a copy of someone else's work. This will be penalised heavily.
- Under no circumstances should you give a copy of your work to someone else.
- The Computer Science department uses copy detection tools on the files you submit. If you copy from someone else, or allow someone else to copy from you, this copying will be detected and disciplinary action will be taken.
- To ensure you are not identified as cheating you should follow these points:
    - Always do individual assignments by yourself.
    - Never give another person your code.
    - Never put your code in a public place (e.g., forum, your web site).
    - Never leave your computer unattended. You are responsible for the security of your account.
    - Ensure you always remove your USB flash drive from the computer before you log off.

## *Your name, UPI and other notes*

- All files should also include your name and UPI in a comment at the beginning of the file.
- All your files should be able to be compiled without requiring any editing.
- All your files should include good layout structure

## *The simplified game rule*

The input to the program is a shuffled deck of cards represented by a Python list. From this deck of cards *n* piles of cards are generated, where *n* is computed from the number of cards in the deck. The first pile (0) begins with one upturned card and *n*-1 downturned cards. The other piles (the tableau piles) are empty at the beginning. An example with *n*=10 is shown below (a '*' indicates a downturned card).

```
0: 6 * * * * * * * * *
1:
2:
3:
```

The first item of the list representing a pile is called the top of the pile, the last item of the list is the bottom of the pile. In the example above, 6 is the top of pile 0.

We can add a card at the bottom of a pile if the card is the direct successor of the currently bottom card – we call this process "building down". We can move partial or complete tableau piles if they are built down in order. In the example below, the pile 1 can be moved to pile 2 since the pile 1 is built down and since the top card of pile 1 is exactly one lower than the bottom card in pile 2, i.e. after moving, pile 2 is still built down in order.

```
0: 2 * * * * * *
1: 5 4 3
2: 6
3:
```

to

```
0: 2 * * * * * *
1:
2: 6 5 4 3
3:
```

We can fill an empty pile with any card. If a player does not want to (or cannot) move the top card of the first pile (0) to any other tableau pile, the player can get the next card of that pile (0) and put the original one to the back of the first pile (0). For example, assume we have the situation below:

```
0: 6 * * * * * * * * *
1:
2:
3:
```

The player can move 6 to the back of pile (0) and the next card after 6 is upturned – in the example below it is 7:

```
0: 7 * * * * * * * * *
1:
2:
3:
```

OR: the player can move the top card of pile (0) to another pile – in the example below the card 6 is moved to pile (1) and the next card of pile (0) is upturned:

```
0: 7 * * * * * * * *
1: 6
2:
3:
```

The aim of the game is to build up a stack of cards in descending order starting from the card with the highest value to the card with the lowest value. Your assignment is divided into several sections for ease of completion. Please complete the assignment in order of the sections.

You can use CodeRunner to test your solution to each section. Note that these tests are not exhaustive and it is recommended to do more testing in an IDE. A program template is provided to help with testing and to enable you to play the game after you have completed and inserted the solutions to each section.

## Section 1: The class CardPile (25 marks)

Please create a class **CardPile** to implement a pile of cards. You should use a python **List** data structure to implement the pile. The first item of the list will be the top card of the pile, and the last item of the list will be the bottom card. A summary of the methods of this class is shown below:

```
class CardPile:
    def __init__(self):
    def add_top(self, item):          # adds a card to the top of the pile
    def add_bottom(self, item):       # adds a card to the bottom of the pile
    def remove_top(self):             # removes a card from the top of the pile
    def remove_bottom(self):          # removes a card from the bottom of the pile
    def size(self):                   # number of cards in the pile
    def peek_top(self):               # returns top card of the pile (without removing it)
    def peek_bottom(self):            # returns bottom card of the pile (without removing it)
    def print_all(self, index):       # print pile <index> – for pile 0 only the top card is upturned
```

**def __init__(self):**
You should use a python **List** data structure to implement the pile. The list's name should be **__items**.

**def add_top(self, item):**
This method adds a new item to the top of the pile (at **__items**[0])

**def add_bottom(self, item):**
This method adds a new item to the bottom of the pile (at **__items**[**SIZE**-1])

**def remove_top(self):**
This method removes and returns an item from the top of the pile (at **__items**[0])

**def remove_bottom(self):**
This method removes and returns an item from the bottom of the pile (at **__items**[**SIZE**-1])

**def size(self):**

This method returns the size of the pile

**def peek_top(self):**
This method returns the top card of the pile (at **__items**[0])

**def peek_bottom(self):**
This method returns the bottom card of the pile (at **__items**[**SIZE**-1])

**def print_all(self, index):**
This method prints out all the cards of the pile from top (first element of **__items**) to bottom (last element of **__items**) The items should be separated by a single space. However, if **index** is **0** (i.e. the first pile of cards), all the items besides the top one (the first item of pile 0) should be hidden by an '*'.

For example: if pile.__items is [4, 5, 6, 7, 8], the expected output for "pile.print_all(1)" should be "4 5 6 7 8" and the expected output for "pile.print_all(0)" should be "4 * * * *"

In Section 1, you are going to complete the implementation of this CardPile class.

You can test your code using CodeRunner (Assignment Question 1: "Solitaire - CardPile").

## Section 2: display(self) of class Solitaire (10 marks)

You are going to implement the **display()** method for the game Solitaire. The class **Solitaire** should include the **__init__(self, cards)** and **display(self)** methods. You can write the method **__init__(self, cards)** as below.

```
class Solitaire:
    def __init__(self, cards):
        self.__piles = []                                   # list of card piles
        self.__num_cards = len(cards)                       # number of cards
        self.__num_piles = (self.__num_cards // 8) + 3      # number of piles
        self.__max_num_moves = self.__num_cards * 2         # maximum number of moves
        for i in range(self.__num_piles):
            self.__piles.append(CardPile())
        for i in range(self.__num_cards):
            self.__piles[0].add_bottom(cards[i])

    def get_pile(self, i):
        return self.__piles[i]          # returns the i-th pile (required for use in CodeRunner test cases)

    def display(self):
        ......
```

**display(self)** is going to display the game layout. The game should have a number (**self.__num_piles)** of card piles. If **self.__num_piles** is 4, 4 rows of numbers should be printed out. However, the first row will display the first item only, the others are hidden by '*'. Thus, you can implement this function by **self.__piles[i].print_all(i)**. Each line should start with the pile number followed by a ':' and the items of the list separated by spaces.

An example is shown below:

**piles[0]** is [6, 5, 4]
**piles[1]** is [3, 2, 1, 0]
**piles[2]** is [14, 13, 12, 11]
**piles[3]** is [10, 9, 8, 7]

The display should be:

```
0: 6 * *
1: 3 2 1 0
2: 14 13 12 11
3: 10 9 8 7
```

In Section 2, you should implement the full program including both the classes **Solitaire** and **CardPile**.

You can test your code using CodeRunner (Assignment Question 2: “Solitaire - display”).

## Section 3: move(self, p1, p2) of class Solitaire (25 marks)

You are going to implement the **move()** method for the game Solitaire. The class **Solitaire** should include the **__init__(self, cards), display(self)** and **move(self, p1, p2)** methods.

The function **move(self, p1, p2)** moves the card from pile **p1** to pile **p2**. There are three types of moves.

**Condition 1 (p1 = p2 = 0):** move a card from the top of the first pile (front of the list) to the bottom of the first pile (rear of the list). This move is always valid. If the pile is empty, the move does not change anything.

```
0: 6 2 1
1: 3 2 1 0

After move(0, 0),
0: 2 1 6
1: 3 2 1 0
```

**Condition 2 (p1 = 0, p2 > 0):** move a card from the top of the first pile (front of the list) to the bottom of any other pile p2 (rear of the list). This move is valid only if the first pile is not empty. If pile 2 is not empty then the value of the moving card (*N1*) must be one less than the value of the card (*N2*) at the bottom of the destination pile (i.e., *N1* = *N2* - 1). If the move is not valid, it does not change anything.

```
0: 4 8
1: 7 6 5

Move(0, 1),
0: 8
1: 7 6 5 4
```

**Condition 3 (p1 > 0, p2 > 0):** move the whole pile of cards from p1 to the bottom of any other pile p2 (rear of the list). This move is valid only if both piles are not empty and if the value *N1* of the card on the top of the moving pile is one less than the value *N2* of the card at the bottom of the destination pile by one (i.e., *N1* = *N2* - 1). If the move is not valid, it does not change anything.

```
0: 8
1: 7 6 5
2: 4 3 2

Move(2, 1) is valid since 4 is one less than 5 and we get:
0: 8
1: 7 6 5 4 3 2
2:
```

In Section 3, you should implement the full program including both the classes **Solitaire** and **CardPile**.

You can test your code using CodeRunner (Assignment Question 3: “Solitaire - move”).

## Section 4: class Solitaire – full program (30 marks)

You are going to implement the complete program of the game Solitaire. The class **Solitaire** should include the following methods:

```
class Solitaire:
    def __init__(self, cards):
        self.piles = []
        self.__num_cards = len(cards)
        self.__num_piles = (self.__num_cards // 8) + 3
        self.__max_num_moves = self.__num_cards * 2
        for i in range(self.__num_piles):
            self.piles.append(CardPile())
        for i in range(self.__num_cards):
            self.piles[0].add_bottom(cards[i])

    def display(self):

    def get_pile(self, pilenumber):

    def move(self, p1, p2):

    def is_complete(self):

    def play(self):
        print("********************** NEW GAME *****************************")
        move_number = 1
        while move_number <= self.__max_num_moves and not self.is_complete():
            self.display()
            print("Round", move_number, "out of", self.__max_num_moves, end = ": ")
            row1 = int(input("Move from row no.:"),10)
            print("Round", move_number, "out of", self.__max_num_moves, end = ": ")
            row2 = int(input("Move to row no.:"),10)
            if row1 >= 0 and row2 >= 0 and row1 < self.__num_piles and row2 < self.__num_piles:
                self.move(row1, row2)
            move_number += 1

        if self.is_complete():
            print("You Win in", move_number - 1, "steps!\n")
        else:
            print("You Lose!\n")
```

You can write the **play(self)** method according to above code. You are going to implement the **is_complete(self)** method to finish the whole program. This function checks whether the player can win the game. The player can win the game if all of the following conditions are satisfied:

1. No card on the first pile
2. All the cards are on one of the other piles
3. All the cards are in decreasing order (this should be always true if your program prevents illegal moves)

This function returns **True** if the player wins the game and **False** otherwise.

You can test your code using CodeRunner (Assignment Question 4: “Solitaire – is_complete()”).

## Section 5: Creative Extension (15 marks)

Extend your program in a "cool" way, e.g. using a more attractive graphical representation, more functionalities, some simple AI etc. Please explain your extension in the file AssCool.pdf and submit a separate source file titled AssCool.py. The markers will evaluate your submission using the following criteria:

- Explanation (how detailed, clear and insightful is your explanation?)
- Novelty and creativity (how creative / interesting / novel / "cool" is your solution?)
- Technical difficulty (what is the complexity of your solution in terms of software development and/or algorithm development skills reflected in your solution?)

## *Submission*

Submit your assignment online via the assignment dropbox (https://adb.auckland.ac.nz/) at any time from the first submission date up until the final date. You will receive an electronic receipt. Submit ONE Ass.zip file containing the following files:

1. **Ass.py** (Python source file with the solution for Sections 1-4)
2. **AssCool.py** (Python source file with the solution for Section 5)
3. **AssCool.pdf** (pdf-file with your explanations for the extension described in Section 5)

Remember to include your name, UPI and a comment at the beginning of each file you create or modify. You may make more than one submission, but note that every submission that you make replaces your previous submission. Submit ALL your files in every submission. Only your very latest submission will be marked. Please double check that you have included all the files required to run your program and AssCool.pdf in the zip file before you submit it.

**Note 1. We will only mark your submitted code**. Your testing results on CodeRunner will not be counted. It is created for you to test your program only, but the tests might not be exhaustive.
**Note 2: Your program must compile and run to gain any marks.** We recommend that you check this on the lab machines before you submit. Your solutions in Ass.py must run on CodeRunner.

## *Marking Details*

**TOTAL: 120 marks**

| **Section 1: The class CardPile** | **25 marks** |
|---|---|
| Display the hidden numbers | 5 |
| Display all numbers | 5 |
| Manage the movement within one card pile | 5 |
| Manage the movements among a few card piles | 10 |

| **Section 2: : display(self) of class Solitaire** | **10 marks** |
|---|---|
| All numbers of all piles can be displayed correctly (test 1) | 2.5 |
| All numbers of all piles can be displayed correctly (test 2) | 2.5 |
| All numbers of all piles can be displayed correctly (test 3) | 2.5 |
| All numbers of all piles can be displayed correctly (test 4) | 2.5 |

| **Section 3: move(self, c1, c2) of class Solitaire** | **25 marks** |
|---|---|
| Move from pile 0 to pile 0 | 5 |
| Move from pile 0 to any other valid pile | 5 |
| Move from any valid pile to any valid pile | 5 |
| Invalid moves (should not change card piles) | 10 |

| **Section 4: full program** | **30 marks** |
|---|---|
| Test 1 of is_complete() after multiple moves | 3 |
| Test 2 of is_complete() after multiple moves | 3 |
| Test 3 of is_complete() after multiple moves | 3 |
| Test with win in 6 steps | 3 |
| Test with loss in 6 steps | 3 |
| Test with win in 15 steps | 3 |
| Test with loss in 20 steps | 3 |

| | |
|---|---|
| Test with win in 16 steps | 3 |
| Test with loss in 30 steps | 3 |
| Test with win in 24 steps | 3 |

| **Section 5: creative extension** | **15 marks** |
|---|---|
| Explanation (how detailed, clear and insightful is your explanation in the file A2cool.pdf?) | 5 |
| Novelty and creativity (how creative / interesting / novel / "cool" is your solution?) | 5 |
| Technical difficulty (what is the complexity of your solution in terms of software development and/or algorithm development skills reflected in your solution?) | 5 |

| **Section 6: Coding style (applies to all submitted sections)** | **15 marks** |
|---|---|
| Good meaningful variable names | 5 |
| Easy to read | 5 |
| Adheres to Python code conventions | 5 |

**Total: 120 marks**