Representing Playing Cards

- in the previous example we used a list to store an array of numbers

- lists can also be used to store information about real world objects with multi component attributes:

- Example: playing cards

Representing Playing Cards

```
 0 Ace   Clubs    13 Ace   Diamonds    26 Ace   Hearts    39 Ace   Spades
 1 Two   Clubs    14 Two   Diamonds    27 Two   Hearts    40 Two   Spades
 2 Three Clubs    15 Three Diamonds    28 Three Hearts    41 Three Spades
 3 Four  Clubs    16 Four  Diamonds    29 Four  Hearts    42 Four  Spades
 4 Five  Clubs    17 Five  Diamonds    30 Five  Hearts    43 Five  Spades
 5 Six   Clubs    18 Six   Diamonds    31 Six   Hearts    44 Six   Spades
 6 Seven Clubs    19 Seven Diamonds    32 Seven Hearts    45 Seven Spades
 7 Eight Clubs    20 Eight Diamonds    33 Eight Hearts    46 Eight Spades
 8 Nine  Clubs    21 Nine  Diamonds    34 Nine  Hearts    47 Nine  Spades
 9 Ten   Clubs    22 Ten   Diamonds    35 Ten   Hearts    48 Ten   Spades
10 Jack  Clubs    23 Jack  Diamonds    36 Jack  Hearts    49 Jack  Spades
11 Queen Clubs    24 Queen Diamonds    37 Queen Hearts    50 Queen Spades
12 King  Clubs    25 King  Diamonds    38 King  Hearts    51 King  Spades
```

-52 unique cards

-13 face values

-four suits

- if we can agree on ordering,
  each card can be represented by a number between 0-51
- so use the above standard contract-bridge ordering

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ..., 47, 48, 49, 50, 51]
```

hence **deck = list(range(52))**

Representing Playing Cards: Suit from card number

Solution #1

```
if cardnum < 13 :
    suit = 'Clubs'
elif cardnum < 26 :
    suit = 'Diamonds'
elif cardnum < 39 :
    suit = 'Hearts'
else :
    suit = 'Spades'
```

Solution #2

```
suit = cardnum / 13
if suit == 0 :
    suit = 'Clubs'
elif suit == 2 :
    suit = 'Diamonds'
elif suit == 3 :
    suit = 'Hearts'
else :
    suit = 'Spades'
```

Representing Playing Cards: Face value from card number

Solution #1: straightforward conversion from card-id to face value

```
if cardnum == 0 or cardnum == 13 or cardnum == 26 or cardnum == 39 :
    face_value = 'Ace'
elif cardnum == 1 or cardnum == 14 or cardnum == 27 or cardnum == 40 :
    face_value = 'Two'
...
```

Can we improve this using lists?

Representing Playing Cards: Face value from card number

Solution #1: straightforward conversion from card-id to face value

```
if cardnum == 0 or cardnum == 13 or cardnum == 26 or cardnum == 39 :
    face_value = 'Ace'
elif cardnum == 1 or cardnum == 14 or cardnum == 27 or cardnum == 40 :
    face_value = 'Two'
...
```

Solution #2: improve sol#1 using list membership tests

```
if cardnum in [0, 13, 26, 39]:
    face_value = 'Ace'
elif cardnum in [1, 14, 27, 40]:
    face_value = 'Two'
elif cardnum in [2, 15, 28, 41]:
    face value = 'Three'
```

...

Representing Playing Cards: Face value from card number

```
face_value = cardnum % 13
if face_value == 0 :
    face_value = 'Ace'
elif face_value == 2 :
    face_value = 'Two'
elif face_value == 3 :
    face_value = 'Three'
elif face_value == 4 :
    face_value = 'Four'
elif face_value == 5 :
    face_value = 'Five'
elif face_value == 6 :
    face_value = 'Six'
elif face_value == 7 :
    face_value = 'Seven'
elif face_value == 8 :
    face_value = 'Eight'
elif face_value == 9 :
    face_value = 'Nine'
elif face_value == 10 :
    face_value = 'Ten'
elif face_value == 11 :
    face_value = 'Jack'
elif face_value == 12 :
    face_value = 'Queen'
else :
    face_value = 'King'
```

Solution #3: consider the remainder when card-index divided by 13

Representing Playing Cards: Face value from card number

```
face_value = cardnum % 13
if face_value == 0 :
    face_value = 'Ace'
elif face_value == 2 :
    face_value = 'Two'
elif face_value == 3 :
    face_value = 'Three'
elif face_value == 4 :
    face_value = 'Four'
elif face_value == 5 :
    face_value = 'Five'
elif face_value == 6 :
    face_value = 'Six'
elif face_value == 7 :
    face_value = 'Seven'
elif face_value == 8 :
    face_value = 'Eight'
elif face_value == 9 :
    face_value = 'Nine'
elif face_value == 10 :
    face_value = 'Ten'
elif face_value == 11 :
    face_value = 'Jack'
elif face_value == 12 :
    face_value = 'Queen'
else :
    face_value = 'King'
```

<-- Solution #3: consider the remainder when card-index divided by 13

Solution #4: improve sol#3 by using list look-up

```
FACE_VALUES = ['Ace', 'Two', 'Three', 'Four', 'Five', 'Six', 'Seven', \
               'Eight', 'Nine', 'Ten', 'Jack', 'Queen', 'King']

face_value = cardnum % 13
print('The face value of card number', cardnum, 'is', FACE_VALUES[face_value])
```

Conc:
- using lists can make code more readable and easy to understand
- possibly the ones with lists will perform better than the case where there is a cascade of if statements.

Representing Playing Cards: Dealing a Hand

Problem: How can we deal a hand of cards from our deck of cards?

```
import random

# Create the deck of cards.
deck = range(52)

# Shuffle the deck of cards
for swaps in range(104):
    posn1 = random.randint(0, 51)
    posn2 = random.randint(0, 51)
    # Swap the cards at posn1 and posn2
   (deck[posn1], deck[posn2]) = (deck[posn2], deck[posn1])

# Create the empty hand.
hand = []

# Deal 5 cards from the deck into the hand.
for card in range( 0, 5 ):
    hand.append( deck.pop() )
```

Solution1: shuffle the deck and then either:

(i) use .**pop()** to select one random item at a time or

(ii) use **list slicing** to get as many random cards as needed.

Representing Playing Cards: Dealing a Hand

Problem: How can we deal a hand of cards from our deck of cards?

Solution2: randomly select cards from deck and add to the **hand** list

```
import random
deck = range(52)
hand = []
for card in range(5) :
    # Choose the card to deal.
    posn = random.randint(0, len(deck) - 1)
    # Append the number at that position to the hand.
    hand.append(deck[posn])
    # Delete that card from the deck.
    del(deck[posn])
```

Question Imagine a list with 1 Billion items,
which solution method would you use?

Representing Playing Cards: Dealing a Hand: Putting it together

```
# deal_a_hand.py
# This program deals a hand of 5 cards at random.
# CPSC128 Example code
# S. Bulut 2019, T. Topper 2015

import random

# Define handy string constants.
FACE_VALUES = ['Ace', 'Two', 'Three', 'Four', 'Five', 'Six',
               'Seven', 'Eight', 'Nine', 'Ten', 'Jack',
               'Queen', 'King']
SUITS = ['Clubs', 'Diamonds', 'Hearts', 'Spades']

# Create deck of cards.
deck = list(range(52))

# Create empty hand.
hand = []

# Deal 5 cards into hand.
for deal in range(5):
    posn = random.randint(0, len(deck) - 1)
    hand.append(deck[posn])
    del(deck[posn])

# Display the cards in the hand.
for card in hand:
    print(FACE_VALUES[card % 13], 'of', SUITS[card // 13])
```

Programming Practice
Poker hands: is it a “flush”?

Task: write code to detect if our hand is a flush, i.e. that all the cards in the hand are from the same suit.

since face values are irrelevant,

lets create a second list (lets call it **suit_list**) corresponding to the suit type

For simplicity represent suit type in numbers:
0 -> club
1 -> diamond
2 -> heart
3 -> spade

a hand: [47, 36, 20, 40, 14]
Nine of Spades
Jack of Hearts
Eight of Diamonds
Two of Spades
Two of Diamonds

corresponding suit list : [3, 2, 1, 3, 2]

an example of flush woud be:

suit_list: [2, 2, 2, 2, 2]

Programming Practice
Poker hands: is it a “flush”?

Task: write code to detect if our hand is a flush, i.e. that all the cards in the hand are from the same suit.

since face values are irrelevant,

lets create a second list (lets call it **suit_list**) corresponding to the suit type

```
suit_list = []
for card in hand:
    suit_list.append(card//13)
```

Programming Practice
Poker hands: is it a “flush”? (Sol’n 1)

```
28 if suit_list[1] == suit_list[0] and suit_list[2] == suit_list[1] and \
29     suit_list[3] == suit_list[2] and suit_list[4] == suit_list[3]:
30     print("That's a flush!")
31 else:
32     print("Sorry no flush here.")
```

What if there is more than 5 cards in ‘hand’? Let’s find a more general solution

Programming Practice
Poker hands: is it a “flush”? (Sol’n 1)

```
34 ## is it a flush? Sol_1b
35 flush = True # Start by assuming it is a flush.
36
37 # The first test compares the second card to the first card,
38 # so the initial previous card is the first card in the hand.
39 prev_card = suit_list[0]
40 for card in suit_list[1:]:
41     # Note: To loop starting with the second card we use a slice of the list,
42     # that begins in the second position, i.e. position #1.
43     if card != prev_card: # If card's suit not the same as previous one's.
44         flush = False # It's not a flush.
45     prev_card = card # Update previous card: This card will be the previous
46                      # card next time around.
47 if flush:
48     print("That's a flush!")
49 else:
50     print("Sorry, no flush here.")
51
```

further improvement: can we get grid of ‘prev_card’?

Programming Practice
Poker hands: is it a “flush”? (Sol’n 1)

```
53 ## Is it a flush? sol_1c
54 flush = True
55 for posn in range(1,len(suit_list)-1):
56     if suit_list[posn] != suit_list[posn-1]:
57         flush = False
58         break
59
60 if flush:
61     print("That's a flush!")
62 else:
63     print("Sorry, no flush here.")
```

Programming Practice
Poker hands: is it a “flush”? (Sol’n 2)

Another approach would be to count how many cards there are of each suit. If the count is ever the same as the length of the hand then it's a flush.

```
66 ## Is it a flush? sol2
67 flush = False
68 for suit in [0,1,2,3]:
69     count = 0
70     for card_suit in suit_list:
71         if card_suit == suit:
72             count = count + 1
73     if count == len(suit_list):
74         flush = True
75         break
76
77 if flush:
78     print("That's a flush!")
79 else:
80     print("Sorry, no flush here.")
81
```

Programming Practice
Poker hands: is it a “flush”? (Sol’n 3)

Yet another approach takes advantage of some of the built-in list methods. For example the count method counts how many times a value occurs in a list.

```
83 if suit_list.count(suit_list[0]) == len(suit_list):
84     print("That's a flush!")
85 else:
86     print("Sorry, no flush here.")
```

Programming Practice
Poker hands: is it a “flush”? (Sol’n 4)

Another approach that leverages Python's built-in list methods:

When sorted, the first and last item in a suit_list should be the same in a flush scenario:
[2, 1, 3, 0, 2] -> sorted -> [0, 1, 2, 2, 3]
above, first and last items aren’t the same
[2, 2, 2, 2, 2] -> sorted -> [2, 2, 2, 2 ,2]
first and last items are the same! its a flush!

```
89 suit_list.sort()
90 if suit_list[0] == suit_list[len(suit_list) - 1]:
91     print("That's a flush!")
92 else:
93     print("Sorry, no flush here.")
```

Poker hands: summary

With solutions 3 and 4 being so short why waste time on long solutions like 1 and 2?

- not all languages provide the rich set of list methods that Python does

- solutions 1 and 2 showed techniques that are useful in a wide variety of problems:

1. comparing elements of a list pairwise
2. looping from part way through a list to the end
3. looping through a list by index position
4. using flag variables
5. using counters

There's (almost) always more than one way to solve it!

Poker Hands: alternative representation

```
deck = [
['A','C'], ['2','C'], ['3','C'], ['4','C'], ['5','C'], ['6','C'], ['7','C'],
['8','C'], ['9','C'], ['10','C'], ['J','C'], ['Q','C'], ['K','C'],
['A','D'], ['2','D'], ['3','D'], ['4','D'], ['5','D'], ['6','D'], ['7','D'],
['8','D'], ['9','D'], ['10','D'], ['J','D'], ['Q','D'], ['K','D'],
['A','H'], ['2','H'], ['3','H'], ['4','H'], ['5','H'], ['6','H'], ['7','H'],
['8','H'], ['9','H'], ['10','H'], ['J','H'], ['Q','H'], ['K','H'],
['A','S'], ['2','S'], ['3','S'], ['4','S'], ['5','S'], ['6','S'], ['7','S'],
['8','S'], ['9','S'], ['10','S'], ['J','S'], ['Q','S'], ['K','S']]
```

Poker Hands: alternative representation

```
deck = [
['A','C'], ['2','C'], ['3','C'], ['4','C'], ['5','C'], ['6','C'], ['7','C'],
['8','C'], ['9','C'], ['10','C'], ['J','C'], ['Q','C'], ['K','C'],
['A','D'], ['2','D'], ['3','D'], ['4','D'], ['5','D'], ['6','D'], ['7','D'],
['8','D'], ['9','D'], ['10','D'], ['J','D'], ['Q','D'], ['K','D'],
['A','H'], ['2','H'], ['3','H'], ['4','H'], ['5','H'], ['6','H'], ['7','H'],
['8','H'], ['9','H'], ['10','H'], ['J','H'], ['Q','H'], ['K','H'],
['A','S'], ['2','S'], ['3','S'], ['4','S'], ['5','S'], ['6','S'], ['7','S'],
['8','S'], ['9','S'], ['10','S'], ['J','S'], ['Q','S'], ['K','S']]
```

```
FACE_VALUES = ['A', '2', '3', '4', '5', '6', '7', '8', '9', '10',
               'J', 'Q', 'K']
SUITS = ['C', 'D', 'H', 'S']

deck = []
for suit in SUITS :
    for face_value in FACE_VALUES :
        deck.append([face_value, suit])
```

Multidimensional lists:

What is this? [['X', 'O', ''], ['O', 'X', 'O'], ['', '', 'X']]

Multidimensional lists:

What is this? [['X', 'O', ''], ['O', 'X', 'O'], ['', '', 'X']]

When the sizes of **all** the lists are the same, this is called a
- multi-dimentional list
- in this case it is a 2D-list or array

Note: both outer and inner lists have 3-item in them.

This 2D lists represents the state of a Tic-Tac-Toe game

```
[['X', 'O',  ''],
 ['O', 'X', 'O'],
 [ '',  '', 'X']]
```

```
 X | O |
---+---+---
 O | X | O
---+---+---
   |   | X
```

Multidimensional lists:

define g = [['X', 'O', ''], ['O', 'X', 'O'], ['', '', 'X']]

g[0][0]	g[0][1]	g[0][2]
g[1][0]	g[1][1]	g[1][2]
g[2][0]	g[2][1]	g[2][2]

```
 X | O |
---+---+---
 O | X | O
---+---+---
   |   | X
```

```
if g[0][0]==g[0][1] and g[0][1]==g[0][2]:
    print(g[0][0], 'has won!')
else:
    print('No one has won in the top row.')
```

```
if g[0][0]==g[1][1] and g[1][1]==g[2][2]:
    print(g[0][0], 'has won!')
else:
    print('No one has won on the main diagonal.')
```

String processing: Pallindromes

Problem

Write a program that inputs a string and determines if it is a palindrome. A palindrome is a string that reads the same forwards or backwards, e.g. "bob" and "madam".
An excellent program would be able to deal with entire phrases by ignoring capitalization, spaces and punctuation in the input, e.g. "*A man, a plan, a canal, Panama!*" is a palindromic phrase and should be identified as such.

For the moment, ignore phrases and focus on words only:

How can we determine that a word is a pallindrome?

String processing: Pallindromes (Solution 1)

A pallindrome reads the same backwards and forwards:
1st letter forwards then should be the 1st letter backwards, i.e the last letter etc

```
if first_letter == last_letter and second_letter == second_last_lett
    Then it is a palindrome.
else
    It is NOT a palindrome.
```

Python'ize this pseudocode:

```
if s[0]==s[len(s)-1] and s[1]==s[len(s)-2] and s[2]==s[len(s)-3] a
    Then it is a palindrome.
else
    It is NOT a palindrome.
```

any problems here?

String processing: Pallindromes (Solution 1)

We don't know ahead of time how many test expressions we need (because we don't know ahead of time how long the string is).

So use a **loop** instead of **if**

```
 97 palindrome = True
 98 for offset in range(0, len(s)//2):
 99     if s[offset] != s[len(s)-1-offset]:
100         palindrome = False
101         break
102
103 if palindrome:
104     print("It is a palindrome!")
105 else:
106     print("It is NOT a palindrome.")
```

String processing: Pallindromes (Solution 2)

Python must have a built-in library that we can leverage to solve this problem!

Another approach: build the reverse of the string, and compare:

Lists have a built-in reverse method, but strings don't.
so, lets convert the string to a list first.

```
>>> s = "madam"
>>> slist = list(s)
>>> slist
['m', 'a', 'd', 'a', 'm']
>>>
```

Now we can reverse

```
>>> slist.reverse()
>>> slist
['m', 'a', 'd', 'a', 'm']
>>>
```

Note we can't do: slist == slist.reverse()
since **.reverse()** changes slist in place,
that is instead of creating a new string in reverse
it modified slist!

String processing: Pallindromes (Solution 2)

we can’t do slist == slist.revers(), instead convert back to string and compare!
For that purpose use the **.join()** method provided by string module

```
>>> iplist = ['199', '147', '23', '5']
>>> ip = '.'.join(iplist)
>>> ip
'199.147.23.5'
>>>
```

join items iplist with ‘.’ in b/w

In order to join char’s nothing in between use ‘’:

```
>>> s_reversed = ''.join(slist)
>>> s_reversed
'madam'
>>>
```

String processing: Pallindromes (Solution 2)

we can’t do slist == slist.revers(), instead convert back to string and compare!
For that purpose use the **.join()** method provided by string module

```
109 s = "madam"
110 slist = list(s)
111 slist.reverse()
112 s_reversed = ''.join(slist)
113 if s == s_reversed:
114     print("It is a palindrome!")
115 else:
116     print("It is NOT a palindrome.")
117
```

String processing: Pallindromes (Testing)

So far we have just used "madam" and been happy when our code correctly identified it as a palindrome, but that's not sufficient testing. What would be good additional tests?

String processing: Pallindromes (Testing)

So far we have just used "madam" and been pleased when our code correctly identified it as a palindrome, but that's not sufficient testing. What would be good additional tests?

- pallindromes with even number of letters
- non-palindromes with even & odd # of letters
- edge cases: zero and one-letter strings
- two letter strings: both palindrome & non-pallindrome
- non-alphabetic characters

A good set for testing ->

```
madam
maam
motor
moor
a
oo
at
A man, a plan, a canal, Panama!
```

String processing: Pallindromes (Testing)

Poor man's testing:

```
130 ## palindrome: poor man's testing
131 TESTS = ['madam', 'maam', 'motor', 'moor', 'a', 'oo', 'at',
132           'A man, a plan, a canal, Panama!']
133 print 'Testing Solution 1:'
134 for s in TESTS:
135     print s,
136     # Preprocess s to lower case and remove non-alphas.
137     ...
138
139     # Test to see if s is a palindrome
140     ...
```

Although this code does the job, it is an awkward way of doing testing our code because we had to modify our code substantially to test it.

Python allows us to make our palindrome testing code a stand alone function, and then to embed it into a module that does automatic testing when run on its own, but from which we can import the palindrome function if we need to use it.

Creating such modules and functions will be covered in the next class.

Palindromes: Phrases

```
## pallindrome: preprocessing for phrases
s = "A man, a plan, a canal, Panama."
print(s)
s_new = ''
for c in s:
    if c.isalpha():
        if c.isupper():
            s_new = s_new + c.lower()
        else:
            s_new = s_new + c
print("becomes")
print(s_new)
```

- convert all characters to lowercase
- get rid of punctuation & space etc

```
A man, a plan, a canal, Panama.
becomes
amanaplanacanalpanama
```