```python
"""
Analyzes a card

:reflection: Defined functions called get_suit and get_num_char for information hiding.
"""

def get_suit(hands, card_order):
    """
    Gets the card suit with the card order from hands

    :param hands: a list for hands
    :param card_order: an integer for the index of specific card
    :return: a string for the card suit
    """
    return hands[card_order][len(hands[card_order]) - 1]

def get_num_char(hands, card_order):
    """
    Gets the card number or character with the card order from hands

    :param hands: a list for hands
    :param card_order: an integer for the index of specific card
    :return: a string for the card number or character
    """
    return hands[card_order][0: len(hands[card_order]) - 1]

if __name__ == "__main__":
    print("expected suit: C, actual suit: ", get_suit(["AC", "2D", "3H", "4S", "5S"], 0))
    print("expected number: A, actual number: ", get_num_char(["AC", "2D", "3H", "4S", "5S"], 0))
```

```python
"""
Models a deck

:reflection: Used named constants like MAX_NUMBER because it is convenient to change their values throughout the code.
Used string for cards like "2C" (two clover) because it is easy to compare their numbers, characters, or suits
if their lengths and index are similar.
Defined functions called shuffle, draw, and card_remaining for the modularity.
"""

import random

MAX_NUMBER = 10
SUITS = ["C", "D", "H", "S"]
CHARACTERS = ["A", "J", "Q", "K"]

def create_number():
    """
    Creates a deck of number cards

    :return: a list for a deck of number cards
    """
    number_deck = []

    for i in range (2, MAX_NUMBER + 1):
        for j in SUITS:
            number_deck.append(str(i) + j)

    return number_deck

def create_standard():
    """
    Creates a standard deck of 52 cards

    :return: a list for a deck of cards
    """
    standard_deck = create_number()

    for i in CHARACTERS:
        for j in SUITS:
            standard_deck.append(i + j)

    return standard_deck

def shuffle(deck):
    """
    Shuffles the deck

    :param deck: a list for a deck of cards
    :return: a list for a shuffled deck of cards
    """
    return random.shuffle(deck)

def draw(deck):
    """
    Draw a card from the standard deck

    :param deck: a list for a deck of cards
    :return: a string drawn randomly
    """
    return deck.pop(random.randrange(0, len(deck)))

def cards_remaining(deck):
    """
    Returns a number of cards left in the deck

    :param deck: a list for a deck of cards
    :return: an integer for cards left
    """
    return len(deck)

if __name__ == "__main__":
    deck = create_standard()

    print("deck: ", deck)
    print("card: ", draw(deck))
    print("cards remaining: ", cards_remaining(deck))
```

```python
"""
Models hands and analyzes cards

:reflection: Imported files called deck and card for the modularity.
Used named constants like MAX_HANDS because it is convenient to change their values throughout the code.
Used helper functions like d.draw(deck) for the modularity.
Defined functions called hands_extend and card_remove for the modularity and information hiding.
Defined functions starts with "is_..." for the modularity.
Used helper functions like get_suit(hands, i) for the modularity and information hiding.
"""

import deck as d
import card as c

MAX_HANDS = 5
ATTEMPTS = 10000

def create_hands(deck):
    """
    Creates a list for pocker hands with 5 cards
    :return: a list for poker hands
    """
    hands = []

    for hand in range (MAX_HANDS):
        hands.append(d.draw(deck))

    return hands

def hands_extend(current_hands, given_hands):
    """
    Extends current hands with given hands

    :param current_hands: a list for current hands
    :param given_hands: a list for given hands
    :return: a list for extended hands
    """
    return current_hands.extend(given_hands)

def card_remove(hands, card_order):
    """
    Removes the card with the card order from hands

    :param hands: a list for hands
    :param card_order: an integer for card number
    :return: a list for modified hands
    """
    return hands.remove(hands[card_order])

def is_flush(hands):
    """
    Checks whether hands are flush

    :param hands: a list for hands
    :return: True if all cards have the same shape
    """
    for i in range(1, MAX_HANDS):
        if c.get_suit(hands, i) != c.get_suit(hands, i - 1):
            return False

    return True

def is_two_pair(hands):
    """
    Checks whether hands are two pair

    :param current_hands: a list for hands
    :return: True if hands have 2 pairs of the same number or character
    """
    current_hands = []
    hands_extend(current_hands, hands)

    total_pair = 0
    i = 1

    while len(current_hands) > i:
        if c.get_num_char(current_hands, 0) == c.get_num_char(
 current_hands, i):
             total_pair += 1

             card_remove(current_hands, i)
             card_remove(current_hands, 0)

             i = 1

         else:
             i += 1

         if i == len(current_hands):
             card_remove(current_hands, 0)

             i = 1

     if total_pair == 2:
         return True

     return False

 def is_pair(hands):
     """
     Checks whether hands are two pair

     :param current_hands: a list for hands
     :return: True if hands have a pairs of the same number or character
     """
     current_hands = []
     hands_extend(current_hands, hands)

     total_pair = 0
     i = 1

     while len(current_hands) > i:
         if c.get_num_char(current_hands, 0) == c.get_num_char(current_hands, i):
         total_pair += 1

         card_remove(current_hands, i)
             card_remove(current_hands, 0)

             i = 1

         else:
             i += 1

         if i == len(current_hands):
             card_remove(current_hands, 0)

             i = 1

     if total_pair == 1:
         return True

     return False

 def confirm_result(hands):
     """
     Confirms the actual result of hands

     :param hands: a list for hands
     :return:
     """
     if is_flush(hands):
         print(hands, " is flush")

     elif is_two_pair(hands):
         print(hands, " is two pair")

     elif is_pair(hands):
         print(hands, " is pair")

     else:
         print(hands, "is high card")

 if __name__ == "__main__":
     deck = d.create_standard()

     for i in range (10):
         hands = create_hands(deck)

         confirm_result(hands)
```

```python
"""
Computes a table

:reflection: Imported files called deck and hands for the modularity.
Used named constant like ATTEMPTS because it is convenient to change its value throughout the code.
Defined functions called first_row and get_result for the modularity.
Defined a function called get_percentage and get_content for the modularity and information hiding.
Used helper functions like d.create_standard() for the modularity.
"""

import deck as d
import hands as h

ATTEMPTS = 10000

def first_row():
    """
    Outputs the first row of the table

    :return:
    """
    print('{} {:>7} {:^7} {:>9} {:^7} {:>9} {:^7} {:>11} {:^7}'.format('# of hands', 'pairs', '%', '2 pairs', '%',
            'flushes', '%', 'high card', '%'))
def get_percentage(number, total_attempt):
    """
    Gets percentage with the given number and total attempt

    :param number: an integer for the division
    :param total_attempt: an integer for the division
    :return: an integer converted into a percentage
    """
    return number / total_attempt * 100

def get_result(total_attempt):
    """
    Gets each total number of flush, two pair, pair, and high card and computes each percentage

    :return: a list of total numbers and percentages
    """
    total_flush = 0
    total_two_pair = 0
    total_pair = 0
    total_high_card = 0

    deck = d.create_standard()
    d.random.shuffle(deck)

    for i in range(total_attempt):
        if d.cards_remaining(deck) < 5:
            deck = d.create_standard()
            d.shuffle(deck)

        hands = h.create_hands(deck)

        if h.is_flush(hands):
            total_flush += 1

        elif h.is_two_pair(hands):
            total_two_pair += 1

        elif h.is_pair(hands):
            total_pair += 1

        else:
            total_high_card += 1

    return [total_attempt, total_pair, get_percentage(total_pair, total_attempt), total_two_pair,
            get_percentage(total_two_pair, total_attempt), total_flush, get_percentage(total_flush, total_attempt),
            total_high_card, get_percentage(total_high_card, total_attempt)]

def get_content(total_result, content_order):
```

```python
    """
    Gets content from the given total result

    :param total_result: a list of total numbers and percentages
    :param content_order: an integer for the index of specific content
    :return: an integer or float for the specific content
    """
    return total_result[content_order]

def output_table(total_result):
    """
    Outputs the table with total numbers and percentages

    :param total_list: a list of total numbers
    :return:
    """
    print('{:>10,} {:>7} {: 06.2f} {:>10} {: 06.2f} {:>10} {: 06.2f} {:>12} {: 06.2f}'.
          format(get_content(total_result, 0), get_content(total_result, 1), get_content(total_result, 2),
                 get_content(total_result, 3), get_content(total_result, 4), get_content(total_result, 5),
                 get_content(total_result, 6), get_content(total_result, 7), get_content(total_result, 8)))

if __name__ == "__main__":
    first_row()
    output_table(get_result(ATTEMPTS))
```

```python
"""
A simple pocker game

:author: Chris Hegang Kim
:note: I affirm that I have carried out the attached academic endeavors with full academic honesty,
in accordance with the Union College Honor Code and the course syllabus.

:reflection: Imported a file called table for the modularity.
Used named constants like ATTEMPTS because it is convenient to change their values throughout the code.
Used helper functions like t.first_row() for the modularity.
"""

import table as t

ATTEMPTS = 10000
MAX_COLUMNS = 10

def play_rounds():
    """
    Starts the entire program running and prints the output table

    :return:
    """
    total_attempt = 0

    t.first_row()

    for i in range (MAX_COLUMNS):
        total_attempt += ATTEMPTS

        t.output_table(t.get_result(total_attempt))

if __name__ == "__main__":
    play_rounds()
```