

You can order print and ebook versions of *Think Python 3e* from Bookshop.org and Amazon.

Work done by Colin Howard starting in [Exercises](#) and indicated with boldface text

```
In [1]: from os.path import basename, exists

def download(url):
    filename = basename(url)
    if not exists(filename):
        from urllib.request import urlretrieve

        local, _ = urlretrieve(url, filename)
        print("Downloaded " + str(local))
    return filename

download('https://github.com/AllenDowney/ThinkPython/raw/v3/thinkpython.py');
download('https://github.com/AllenDowney/ThinkPython/raw/v3/diagram.py');

import thinkpython
```

Downloaded thinkpython.py

Downloaded diagram.py

(chapter_inheritance)=

Inheritance

The language feature most often associated with object-oriented programming is **inheritance**. Inheritance is the ability to define a new class that is a modified version of an existing class. In this chapter I demonstrate inheritance using classes that represent playing cards, decks of cards, and poker hands. If you don't play poker, don't worry -- I'll tell you what you need to know.

Representing cards

There are 52 playing cards in a standard deck -- each of them belongs to one of four suits and one of thirteen ranks. The suits are Spades, Hearts, Diamonds, and Clubs. The ranks are Ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, and King. Depending on which game you are playing, an Ace can be higher than King or lower than 2.

If we want to define a new object to represent a playing card, it is obvious what the attributes should be: `rank` and `suit`. It is less obvious what type the attributes should be. One possibility is to use strings like `'Spade'` for suits and `'Queen'` for ranks. A problem with this implementation is that it would not be easy to compare cards to see which had a higher rank or suit.

An alternative is to use integers to **encode** the ranks and suits. In this context, "encode" means that we are going to define a mapping between numbers and suits, or between numbers and ranks. This kind of encoding is not meant to be a secret (that would be "encryption").

For example, this table shows the suits and the corresponding integer codes:

Suit	Code
Spades	3
Hearts	2
Diamonds	1
Clubs	0

With this encoding, we can compare suits by comparing their codes.

To encode the ranks, we'll use the integer `2` to represent the rank `2`, `3` to represent `3`, and so on up to `10`. The following table shows the codes for the face cards.

Rank	Code
Jack	11
Queen	12
King	13

And we can use either `1` or `14` to represent an Ace, depending on whether we want it to be considered lower or higher than the other ranks.

To represent these encodings, we will use two lists of strings, one with the names of the suits and the other with the names of the ranks.

Here's a definition for a class that represents a playing card, with these lists of strings as **class variables**, which are variables defined inside a class definition, but not inside a method.

```
In [2]: class Card:
        """Represents a standard playing card."""

        suit_names = ['Clubs', 'Diamonds', 'Hearts', 'Spades']
        rank_names = [None, 'Ace', '2', '3', '4', '5', '6', '7',
                      '8', '9', '10', 'Jack', 'Queen', 'King', 'Ace']
```

The first element of `rank_names` is `None` because there is no card with rank zero. By including `None` as a place-keeper, we get a list with the nice property that the index `2` maps to the string `'2'`, and so on.

Class variables are associated with the class, rather than an instance of the class, so we can access them like this.

```
In [3]: Card.suit_names
```

```
Out[3]: ['Clubs', 'Diamonds', 'Hearts', 'Spades']
```

We can use `suit_names` to look up a suit and get the corresponding string.

```
In [4]: Card.suit_names[0]
```

```
Out[4]: 'Clubs'
```

And `rank_names` to look up a rank.

```
In [5]: Card.rank_names[11]
```

```
Out[5]: 'Jack'
```

Card attributes

Here's an `__init__` method for the `Card` class -- it takes `suit` and `rank` as parameters and assigns them to attributes with the same names.

```
In [6]: %%add_method_to Card

def __init__(self, suit, rank):
    self.suit = suit
    self.rank = rank
```

Now we can create a `Card` object like this.

```
In [7]: queen = Card(1, 12)
```

We can use the new instance to access the attributes.

```
In [8]: queen.suit, queen.rank
```

```
Out[8]: (1, 12)
```

It is also legal to use the instance to access the class variables.

```
In [9]: queen.suit_names
```

```
Out[9]: ['Clubs', 'Diamonds', 'Hearts', 'Spades']
```

But if you use the class, it is clearer that they are class variables, not attributes.

Printing cards

Here's a `__str__` method for `Card` objects.

```
In [10]: %%add_method_to Card

def __str__(self):
    rank_name = Card.rank_names[self.rank]
    suit_name = Card.suit_names[self.suit]
    return f'{rank_name} of {suit_name}'
```

When we print a `Card`, Python calls the `__str__` method to get a human-readable representation of the card.

```
In [11]: print(queen)
```

Queen of Diamonds

The following is a diagram of the `Card` class object and the `Card` instance. `Card` is a class object, so its type is `type`. `queen` is an instance of `Card`, so its type is `Card`. To save space, I didn't draw the contents of `suit_names` and `rank_names`.

```
In [12]: from diagram import Binding, Value, Frame, Stack

bindings = [Binding(Value(name), draw_value=False)
             for name in ['suit_names', 'rank_names']]

frame1 = Frame(bindings, name='type', dy=-0.5, offsetx=0.77)
binding1 = Binding(Value('Card'), frame1)

bindings = [Binding(Value(name), Value(value))
             for name, value in zip(['suit', 'rank'], [1, 11])]

frame2 = Frame(bindings, name='Card', dy=-0.3, offsetx=0.33)
binding2 = Binding(Value('queen'), frame2)

stack = Stack([binding1, binding2], dy=-1.2)
```

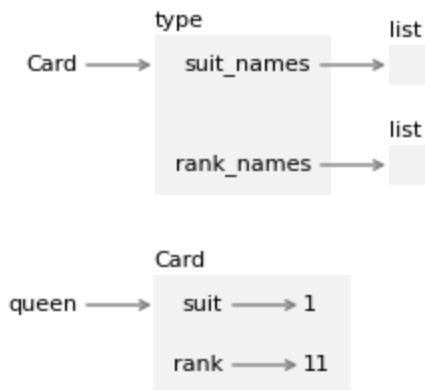
```
In [13]: from diagram import diagram, Bbox, make_list, adjust
```

```
width, height, x, y = [2.11, 2.14, 0.35, 1.76]
ax = diagram(width, height)
bbox = stack.draw(ax, x, y)

value = make_list([])
bbox2 = value.draw(ax, x+1.66, y)

value = make_list([])
bbox3 = value.draw(ax, x+1.66, y-0.5)

bbox = Bbox.union([bbox, bbox2, bbox3])
#adjust(x, y, bbox)
```



Every `Card` instance has its own `suit` and `rank` attributes, but there is only one `Card` class object, and only one copy of the class variables `suit_names` and `rank_names`.

Comparing cards

Suppose we create a second `Card` object with the same suit and rank.

```
In [14]: queen2 = Card(1, 12)
print(queen2)
```

Queen of Diamonds

If we use the `==` operator to compare them, it checks whether `queen` and `queen2` refer to the same object.

```
In [15]: queen == queen2
```

```
Out[15]: False
```

They don't, so it returns `False`. We can change this behavior by defining the special method `__eq__`.

```
In [16]: %%add_method_to Card
```

```
def __eq__(self, other):  
    return self.suit == other.suit and self.rank == other.rank
```

`__eq__` takes two `Card` objects as parameters and returns `True` if they have the same suit and rank, even if they are not the same object. In other words, it checks whether they are equivalent, even if they are not identical.

When we use the `==` operator with `Card` objects, Python calls the `__eq__` method.

```
In [17]: queen == queen2
```

```
Out[17]: True
```

As a second test, let's create a card with the same suit and a different rank.

```
In [18]: six = Card(1, 6)  
print(six)
```

```
6 of Diamonds
```

We can confirm that `queen` and `six` are not equivalent.

```
In [19]: queen == six
```

```
Out[19]: False
```

If we use the `!=` operator, Python invokes a special method called `__ne__`, if it exists. Otherwise it invokes `__eq__` and inverts the result -- so if `__eq__` returns `True`, the result of the `!=` operator is `False`.

```
In [20]: queen != queen2
```

```
Out[20]: False
```

```
In [21]: queen != six
```

```
Out[21]: True
```

Now suppose we want to compare two cards to see which is bigger. If we use one of the relational operators, we get a `TypeError`.

```
In [22]: queen < queen2
```

```
TypeError: '<' not supported between instances of 'Card' and 'Card'
```

To change the behavior of the `<` operator, we can define a special method called `__lt__`, which is short for "less than". For the sake of this example, let's assume that suit is more

important than rank -- so all Spades outrank all Hearts, which outrank all Diamonds, and so on. If two cards have the same suit, the one with the higher rank wins.

To implement this logic, we'll use the following method, which returns a tuple containing a card's suit and rank, in that order.

```
In [23]: %%add_method_to Card

def to_tuple(self):
    return (self.suit, self.rank)
```

We can use this method to write `__lt__`.

```
In [24]: %%add_method_to Card

def __lt__(self, other):
    return self.to_tuple() < other.to_tuple()
```

Tuple comparison compares the first elements from each tuple, which represent the suits. If they are the same, it compares the second elements, which represent the ranks.

Now if we use the `<` operator, it invokes the `__lt__` method.

```
In [25]: six < queen
```

```
Out[25]: True
```

If we use the `>` operator, it invokes a special method called `__gt__`, if it exists. Otherwise it invokes `__lt__` with the arguments in the opposite order.

```
In [26]: queen < queen2
```

```
Out[26]: False
```

```
In [27]: queen > queen2
```

```
Out[27]: False
```

Finally, if we use the `<=` operator, it invokes a special method called `__le__`.

```
In [28]: %%add_method_to Card

def __le__(self, other):
    return self.to_tuple() <= other.to_tuple()
```

So we can check whether one card is less than or equal to another.

```
In [29]: queen <= queen2
```

```
Out[29]: True
```

```
In [30]: queen <= six
```

```
Out[30]: False
```

If we use the `>=` operator, it uses `__ge__` if it exists. Otherwise, it invokes `__le__` with the arguments in the opposite order.

```
In [31]: queen >= six
```

```
Out[31]: True
```

As we have defined them, these methods are complete in the sense that we can compare any two `Card` objects, and consistent in the sense that results from different operators don't contradict each other. With these two properties, we can say that `Card` objects are **totally ordered**. And that means, as we'll see soon, that they can be sorted.

Decks

Now that we have objects that represent cards, let's define objects that represent decks. The following is a class definition for `Deck` with an `__init__` method takes a list of `Card` objects as a parameter and assigns it to an attribute called `cards`.

```
In [32]: class Deck:

        def __init__(self, cards):
            self.cards = cards
```

To create a list that contains the 52 cards in a standard deck, we'll use the following static method.

```
In [33]: %%add_method_to Deck

        def make_cards():
            cards = []
            for suit in range(4):
                for rank in range(2, 15):
                    card = Card(suit, rank)
                    cards.append(card)
            return cards
```

In `make_cards`, the outer loop enumerates the suits from `0` to `3`. The inner loop enumerates the ranks from `2` to `14` -- where `14` represents an Ace that outranks a King. Each iteration creates a new `Card` with the current suit and rank, and appends it to `cards`.

Here's how we make a list of cards and a `Deck` object that contains it.


```
In [34]: cards = Deck.make_cards()
deck = Deck(cards)
len(deck.cards)
```

Out[34]: 52

It contains 52 cards, as intended.

(section_print_deck)=

Printing the deck

Here is a `__str__` method for `Deck`.

```
In [35]: %%add_method_to Deck

def __str__(self):
    res = []
    for card in self.cards:
        res.append(str(card))
    return '\n'.join(res)
```

This method demonstrates an efficient way to accumulate a large string -- building a list of strings and then using the string method `join`.

We'll test this method with a deck that only contains two cards.

```
In [36]: small_deck = Deck([queen, six])
```

If we call `str`, it invokes `__str__`.

```
In [37]: str(small_deck)
```

Out[37]: 'Queen of Diamonds\n6 of Diamonds'

When Jupyter displays a string, it shows the "representational" form of the string, which represents a newline with the sequence `\n`.

However, if we print the result, Jupyter shows the "printable" form of the string, which prints the newline as whitespace.

```
In [38]: print(small_deck)
```

```
Queen of Diamonds
6 of Diamonds
```

So the cards appear on separate lines.

Add, remove, shuffle and sort

To deal cards, we would like a method that removes a card from the deck and returns it. The list method `pop` provides a convenient way to do that.

```
In [39]: %%add_method_to Deck

def take_card(self):
    return self.cards.pop()
```

Here's how we use it.

```
In [40]: card = deck.take_card()
print(card)
```

Ace of Spades

We can confirm that there are `51` cards left in the deck.

```
In [41]: len(deck.cards)
```

Out[41]: 51

To add a card, we can use the list method `append`.

```
In [42]: %%add_method_to Deck

def put_card(self, card):
    self.cards.append(card)
```

As an example, we can put back the card we just popped.

```
In [43]: deck.put_card(card)
len(deck.cards)
```

Out[43]: 52

To shuffle the deck, we can use the `shuffle` function from the `random` module:

```
In [44]: import random
```

```
In [45]: # This cell initializes the random number generator so we
# always get the same results.

random.seed(3)
```

```
In [46]: %%add_method_to Deck

def shuffle(self):
    random.shuffle(self.cards)
```

If we shuffle the deck and print the first few cards, we can see that they are in no apparent order.

```
In [47]: deck.shuffle()
         for card in deck.cards[:4]:
             print(card)
```

```
2 of Diamonds
4 of Hearts
5 of Clubs
8 of Diamonds
```

To sort the cards, we can use the list method `sort`, which sorts the elements "in place" -- that is, it modifies the list rather than creating a new list.

```
In [48]: %%add_method_to Deck

         def sort(self):
             self.cards.sort()
```

When we invoke `sort`, it uses the `__lt__` method to compare cards.

```
In [49]: deck.sort()
```

If we print the first few cards, we can confirm that they are in increasing order.

```
In [50]: for card in deck.cards[:4]:
         print(card)
```

```
2 of Clubs
3 of Clubs
4 of Clubs
5 of Clubs
```

In this example, `Deck.sort` doesn't do anything other than invoke `list.sort`. Passing along responsibility like this is called **delegation**.

Parents and children

Inheritance is the ability to define a new class that is a modified version of an existing class. As an example, let's say we want a class to represent a "hand", that is, the cards held by one player.

- A hand is similar to a deck -- both are made up of a collection of cards, and both require operations like adding and removing cards.
- A hand is also different from a deck -- there are operations we want for hands that don't make sense for a deck. For example, in poker we might compare two hands to see which one wins. In bridge, we might compute a score for a hand in order to make a bid.

This relationship between classes -- where one is a specialized version of another -- lends itself to inheritance.

To define a new class that is based on an existing class, we put the name of the existing class in parentheses.

```
In [51]: class Hand(Deck):  
         """Represents a hand of playing cards."""
```

This definition indicates that `Hand` inherits from `Deck`, which means that `Hand` objects can access methods defined in `Deck`, like `take_card` and `put_card`.

`Hand` also inherits `__init__` from `Deck`, but if we define `__init__` in the `Hand` class, it overrides the one in the `Deck` class.

```
In [52]: %%add_method_to Hand  
  
         def __init__(self, label=''):  
             self.label = label  
             self.cards = []
```

This version of `__init__` takes an optional string as a parameter, and always starts with an empty list of cards. When we create a `Hand`, Python invokes this method, not the one in `Deck` -- which we can confirm by checking that the result has a `label` attribute.

```
In [53]: hand = Hand('player 1')  
         hand.label
```

```
Out[53]: 'player 1'
```

To deal a card, we can use `take_card` to remove a card from a `Deck`, and `put_card` to add the card to a `Hand`.

```
In [54]: deck = Deck(cards)  
         card = deck.take_card()  
         hand.put_card(card)  
         print(hand)
```

Ace of Spades

Let's encapsulate this code in a `Deck` method called `move_cards`.

```
In [55]: %%add_method_to Deck  
  
         def move_cards(self, other, num):  
             for i in range(num):  
                 card = self.take_card()  
                 other.put_card(card)
```

This method is polymorphic -- that is, it works with more than one type: `self` and `other` can be either a `Hand` or a `Deck`. So we can use this method to deal a card from `Deck` to a `Hand`, from one `Hand` to another, or from a `Hand` back to a `Deck`.

When a new class inherits from an existing one, the existing one is called the **parent** and the new class is called the **child**. In general:

- Instances of the child class should have all of the attributes of the parent class, but they can have additional attributes.
- The child class should have all of the methods of the parent class, but it can have additional methods.
- If a child class overrides a method from the parent class, the new method should take the same parameters and return a compatible result.

This set of rules is called the "Liskov substitution principle" after computer scientist Barbara Liskov.

If you follow these rules, any function or method designed to work with an instance of a parent class, like a `Deck`, will also work with instances of a child class, like `Hand`. If you violate these rules, your code will collapse like a house of cards (sorry).

Specialization

Let's make a class called `BridgeHand` that represents a hand in bridge -- a widely played card game. We'll inherit from `Hand` and add a new method called `high_card_point_count` that evaluates a hand using a "high card point" method, which adds up points for the high cards in the hand.

Here's a class definition that contains as a class variable a dictionary that maps from card names to their point values.

```
In [56]: class BridgeHand(Hand):
        """Represents a bridge hand."""

        hcp_dict = {
            'Ace': 4,
            'King': 3,
            'Queen': 2,
            'Jack': 1,
        }
```

Given the rank of a card, like `12`, we can use `Card.rank_names` to get the string representation of the rank, and then use `hcp_dict` to get its score.

```
In [57]: rank = 12
rank_name = Card.rank_names[rank]
score = BridgeHand.hcp_dict.get(rank_name, 0)
rank_name, score
```

```
Out[57]: ('Queen', 2)
```

The following method loops through the cards in a `BridgeHand` and adds up their scores.

```
In [58]: %%add_method_to BridgeHand

def high_card_point_count(self):
    count = 0
    for card in self.cards:
        rank_name = Card.rank_names[card.rank]
        count += BridgeHand.hcp_dict.get(rank_name, 0)
    return count
```

```
In [59]: # This cell makes a fresh Deck and
# initializes the random number generator

cards = Deck.make_cards()
deck = Deck(cards)
random.seed(3)
```

To test it, we'll deal a hand with five cards -- a bridge hand usually has thirteen, but it's easier to test code with small examples.

```
In [60]: hand = BridgeHand('player 2')

deck.shuffle()
deck.move_cards(hand, 5)
print(hand)
```

```
4 of Diamonds
King of Hearts
10 of Hearts
10 of Clubs
Queen of Diamonds
```

And here is the total score for the King and Queen.

```
In [61]: hand.high_card_point_count()
```

```
Out[61]: 5
```

`BridgeHand` inherits the variables and methods of `Hand` and adds a class variable and a method that are specific to bridge. This way of using inheritance is called **specialization** because it defines a new class that is specialized for a particular use, like playing bridge.

Debugging

Inheritance is a useful feature. Some programs that would be repetitive without inheritance can be written more concisely with it. Also, inheritance can facilitate code reuse, since you can customize the behavior of a parent class without having to modify it. In some cases, the

inheritance structure reflects the natural structure of the problem, which makes the design easier to understand.

On the other hand, inheritance can make programs difficult to read. When a method is invoked, it is sometimes not clear where to find its definition -- the relevant code may be spread across several modules.

Any time you are unsure about the flow of execution through your program, the simplest solution is to add print statements at the beginning of the relevant methods. If

`Deck.shuffle` prints a message that says something like `Running Deck.shuffle`, then as the program runs it traces the flow of execution.

As an alternative, you could use the following function, which takes an object and a method name (as a string) and returns the class that provides the definition of the method.

```
In [62]: def find_defining_class(obj, method_name):
        """Find the class where the given method is defined."""
        for typ in type(obj).mro():
            if method_name in vars(typ):
                return typ
        return f'Method {method_name} not found.'
```

`find_defining_class` uses the `mro` method to get the list of class objects (types) that will be searched for methods. "MRO" stands for "method resolution order", which is the sequence of classes Python searches to "resolve" a method name -- that is, to find the function object the name refers to.

As an example, let's instantiate a `BridgeHand` and then find the defining class of `shuffle`.

```
In [63]: hand = BridgeHand('player 3')
        find_defining_class(hand, 'shuffle')
```

```
Out[63]: __main__.Deck
```

The `shuffle` method for the `BridgeHand` object is the one in `Deck`.

Glossary

inheritance: The ability to define a new class that is a modified version of a previously defined class.

encode: To represent one set of values using another set of values by constructing a mapping between them.

class variable: A variable defined inside a class definition, but not inside any method.

totally ordered: A set of objects is totally ordered if we can compare any two elements and the results are consistent.

delegation: When one method passes responsibility to another method to do most or all of the work.

parent class: A class that is inherited from.

child class: A class that inherits from another class.

specialization: A way of using inheritance to create a new class that is a specialized version of an existing class.

Exercises

```
In [64]: # This cell tells Jupyter to provide detailed debugging information
# when a runtime error occurs. Run it before working on the exercises.

%xmode Verbose
```

Exception reporting mode: Verbose

Ask a Virtual Assistant

When it goes well, object-oriented programming can make programs more readable, testable, and reusable. But it can also make programs complicated and hard to maintain. As a result, OOP is a topic of controversy -- some people love it, and some people don't.

To learn more about the topic, ask a virtual assistant:

- What are some pros and cons of object-oriented programming?
- What does it mean when people say "favor composition over inheritance"?
- What is the Liskov substitution principle?
- Is Python an object-oriented language?
- What are the requirements for a set to be totally ordered?

And as always, consider using a virtual assistant to help with the following exercises.

Begin work done by Colin Howard

I asked ChatGPT the middle question in the list above. It replied as follows:

The **Liskov Substitution Principle (Liskov Principle)** is one of the five SOLID principles of object-oriented design, and it states that:

"Objects of a superclass should be replaceable with objects of a subclass without affecting the correctness of the program."

In other words, if a class (S) is a subclass of class (T), you should be able to use an instance of (S) wherever an instance of (T) is expected, without altering the desirable properties or behavior of the program. Subtypes must be substitutable for their base types without introducing errors or unexpected behavior.

Key Points of Liskov Substitution Principle:

1. **Behavioral Consistency:** Subclasses should behave in a way that is consistent with the expectations set by the base class. If a method in the base class works with an instance of the base class, it should also work with an instance of a subclass.
2. **Inheritance should maintain invariants:** A subclass should not violate the invariants (rules or assumptions) that are present in the superclass. It should respect the same contract or expectations set by the base class.
3. **No weakening of preconditions:** The subclass should not introduce stricter conditions or constraints than the superclass. If a base class method expects a certain range of input, a subclass method should not impose more restrictive limits on that input.
4. **No strengthening of postconditions:** A subclass should not return results that are more restrictive or different than those promised by the superclass method. The results must fulfill the same contract.

Examples of Violating Liskov Substitution Principle:

Imagine you have a base class `Bird` with a method `fly()`, and a subclass `Penguin` that overrides this method:

- The `Bird` class may assume that all birds can fly. But if you have a `Penguin` subclass, you cannot expect it to fly. If you try to use a `Penguin` wherever a `Bird` is expected (and that `Bird` is assumed to be able to fly), the substitution would break the program.
- This violates Liskov's principle because you can't replace the `Bird` class with a `Penguin` without altering the expected behavior of the system.

Correct Approach:

To adhere to Liskov's principle in this case, you might:

- Use a more refined class hierarchy that reflects the different abilities of different birds, such as having `FlyingBird` and `NonFlyingBird` subclasses.
- Or, you could make `fly()` an interface or a trait that only applicable birds implement.

In this way, `Penguin` would not override `fly()` at all, and the `Bird` class would remain meaningful and its behavior would not be violated by the `Penguin` subclass.

Summary:

The Liskov Substitution Principle ensures that subclasses can be used interchangeably with their base classes, without introducing errors or unexpected behavior in a program. It emphasizes that the subclass must fulfill the promises made by the base class and must not weaken or distort the original behavior or expectations.

End work done by Colin Howard

Exercise

In contract bridge, a "trick" is a round of play in which each of four players plays one card. To represent those cards, we'll define a class that inherits from `Deck`.

```
In [65]: class Trick(Deck):
        """Represents a trick in contract bridge."""
```

As an example, consider this trick, where the first player leads with the 3 of Diamonds, which means that Diamonds are the "led suit". The second and third players "follow suit", which means they play a card with the led suit. The fourth player plays a card of a different suit, which means they cannot win the trick. So the winner of this trick is the third player, because they played the highest card in the led suit.

```
In [91]: cards = [Card(1, 3),
                  Card(1, 10),
                  Card(1, 12),
                  Card(2, 13)]
trick = Trick(cards)
print(trick)
```

```
3 of Diamonds
10 of Diamonds
Queen of Diamonds
King of Hearts
```

Write a `Trick` method called `find_winner` that loops through the cards in the `Trick` and returns the index of the card that wins. In the previous example, the index of the

winning card is 2 .

You can use the following outline to get started.

Begin work done by Colin Howard

```
In [ ]: %%add_method_to Trick

def find_winner(self):
    for i in range(3):
        if self.cards[i].suit != self.cards[i+1].suit:
            return i
    return 3
```

End work done by Colin Howard (testing below)

If you test your method with the previous example, the index of the winning card should be 2 .

```
In [95]: trick.find_winner()
```

```
Out[95]: 2
```

Exercise

The next few exercises ask to you write functions that classify poker hands. If you are not familiar with poker, I'll explain what you need to know. We'll use the following class to represent poker hands.

```
In [ ]: class PokerHand(Hand):
    """Represents a poker hand."""

    def get_suit_counts(self):
        counter = {}
        for card in self.cards:
            key = card.suit
            counter[key] = counter.get(key, 0) + 1
        return counter

    def get_rank_counts(self):
        counter = {}
        for card in self.cards:
            key = card.rank
            counter[key] = counter.get(key, 0) + 1
        return counter
```

`PokerHand` provides two methods that will help with the exercises.

- `get_suit_counts` loops through the cards in the `PokerHand` , counts the number of cards in each suit, and returns a dictionary that maps from each suit code to the number of times it appears.

- `get_rank_counts` does the same thing with the ranks of the cards, returning a dictionary that maps from each rank code to the number of times it appears.

All of the exercises that follow can be done using only the Python features we have learned so far, but some of them are more difficult than most of the previous exercises. I encourage you to ask an AI for help.

For problems like this, it often works well to ask for general advice about strategies and algorithms. Then you can either write the code yourself or ask for code. If you ask for code, you might want to provide the relevant class definitions as part of the prompt.

As a first exercise, we'll write a method called `has_flush` that checks whether a hand has a "flush" -- that is, whether it contains at least five cards of the same suit.

In most varieties of poker, a hand contains either five or seven cards, but there are some exotic variations where a hand contains other numbers of cards. But regardless of how many cards there are in a hand, the only ones that count are the five that make the best hand.

You can use the following outline to get started.

Begin work done by Colin Howard

```
In [102... %%add_method_to PokerHand

def has_flush(self):
    """Checks whether this hand has a flush."""
    suit_counts = self.get_suit_counts()
    for suit in suit_counts:
        if suit_counts[suit] >= 5:
            return True
    return False
```

End work done by Colin Howard *(testing below)*

To test this method, we'll construct a hand with five cards that are all Clubs, so it contains a flush.

```
In [103... good_hand = PokerHand('good_hand')

suit = 0
for rank in range(10, 15):
    card = Card(suit, rank)
    good_hand.put_card(card)

print(good_hand)
```

10 of Clubs
Jack of Clubs
Queen of Clubs
King of Clubs
Ace of Clubs

If we invoke `get_suit_counts`, we can confirm that the rank code `0` appears `5` times.

```
In [104... good_hand.get_suit_counts()
```

```
Out[104... {0: 5}
```

So `has_flush` should return `True`.

```
In [105... good_hand.has_flush()
```

```
Out[105... True
```

As a second test, we'll construct a hand with three Clubs and two other suits.

```
In [106... cards = [Card(0, 2),
               Card(0, 3),
               Card(2, 4),
               Card(3, 5),
               Card(0, 7),
               ]

bad_hand = PokerHand('bad hand')
for card in cards:
    bad_hand.put_card(card)

print(bad_hand)
```

2 of Clubs
3 of Clubs
4 of Hearts
5 of Spades
7 of Clubs

So `has_flush` should return `False`.

```
In [107... bad_hand.has_flush()
```

```
Out[107... False
```

Exercise

Write a method called `has_straight` that checks whether a hand contains a straight, which is a set of five cards with consecutive ranks. For example, if a hand contains ranks `5`, `6`, `7`, `8`, and `9`, it contains a straight.

An Ace can come before a two or after a King, so `Ace` , `2` , `3` , `4` , `5` is a straight and so it `10` , `Jack` , `Queen` , `King` , `Ace` . But a straight cannot "wrap around", so `King` , `Ace` , `2` , `3` , `4` is not a straight.

You can use the following outline to get started. It includes a few lines of code that count the number of Aces -- represented with the code `1` or `14` -- and store the total in both locations of the counter.

Begin work done by Colin Howard

```
In [111... %%add_method_to PokerHand

def has_straight(self, n=5):
    """Checks whether this hand has a straight with at least `n` cards."""
    counter = self.get_rank_counts()
    aces = counter.get(1, 0) + counter.get(14, 0)
    counter[1] = aces
    counter[14] = aces

    for i in range(1,11):
        if counter.get(i,0) > 0 and counter.get(i+1,0) > 0 and counter.get(i+2,
            return True

    return False
```

End work done by Colin Howard (testing below)

`good_hand` , which we created for the previous exercise, contains a straight. If we use `get_rank_counts` , we can confirm that it has at least one card with each of five consecutive ranks.

```
In [109... good_hand.get_rank_counts()
```

```
Out[109... {10: 1, 11: 1, 12: 1, 13: 1, 14: 1}
```

So `has_straight` should return `True` .

```
In [112... good_hand.has_straight()
```

```
Out[112... True
```

`bad_hand` does not contain a straight, so `has_straight` should return `False` .

```
In [113... bad_hand.has_straight()
```

```
Out[113... False
```

Exercise

A hand has a straight flush if it contains a set of five cards that are both a straight and a flush -- that is, five cards of the same suit with consecutive ranks. Write a `PokerHand` method that checks whether a hand has a straight flush.

You can use the following outline to get started.

Begin work done by Colin Howard

In [138...

```
%%add_method_to PokerHand

def get_suit_rank_counts(self):
    """Helper function to get a nested dictionary of card counts"""
    # initialize a nested dict with all zeros
    counter = {}
    for suit in range(4):
        counter[suit] = {}
        for rank in range(1,15):
            counter[suit][rank] = 0
    # count the actual number of cards of each suit and rank
    for card in self.cards:
        if (card.rank == 1 or card.rank == 14):
            counter[card.suit][1] += 1
            counter[card.suit][14] += 1
        else:
            counter[card.suit][card.rank] += 1
    return counter
```

In [123...

```
%%add_method_to PokerHand

def has_straightflush(self):
    """Check whether this hand has a straight flush."""
    # shortcut: if there are no straights and no flushes, there are no straight
    if not self.has_straight() or not self.has_flush():
        return False
    #
    counter = self.get_suit_rank_counts()
    for suit in range(4):
        suit_dict = counter[suit]
        for rank in range(1,11):
            if suit_dict[rank] > 0 and suit_dict[rank+1] > 0 and suit_dict[rank]
                return True
    return False
```

End work done by Colin Howard (testing below)

Use the following examples to test your method.

In [124...

```
good_hand.has_straightflush()    # should return True
```

Out[124...

```
True
```

In [125...

```
bad_hand.has_straightflush()    # should return False
```

Out[125... False

Note that it is not enough to check whether a hand has a straight and a flush. To see why, consider the following hand.

```
In [126... from copy import deepcopy

straight_and_flush = deepcopy(bad_hand)
straight_and_flush.put_card(Card(0, 6))
straight_and_flush.put_card(Card(0, 9))
print(straight_and_flush)
```

2 of Clubs
3 of Clubs
4 of Hearts
5 of Spades
7 of Clubs
6 of Clubs
9 of Clubs

This hand contains a straight and a flush, but they are not the same five cards.

```
In [127... straight_and_flush.has_straight(), straight_and_flush.has_flush()
```

Out[127... (True, True)

So it does not contain a straight flush.

```
In [128... straight_and_flush.has_straightflush()    # should return False
```

Out[128... False

Exercise

A poker hand has a pair if it contains two or more cards with the same rank. Write a `PokerHand` method that checks whether a hand contains a pair.

You can use the following outline to get started.

Begin work done by Colin Howard

```
In [ ]: %%add_method_to PokerHand

def has_pair(self):
    """Checks whether this hand has a flush."""
    rank_counts = self.get_rank_counts()
    for rank in rank_counts:
        if rank_counts[rank] >= 2:
            return True
    return False
```

End work done by Colin Howard (testing below)

To test your method, here's a hand that has a pair.

```
In [134... pair = deepcopy(bad_hand)
pair.put_card(Card(1, 2))
print(pair)
```

```
2 of Clubs
3 of Clubs
4 of Hearts
5 of Spades
7 of Clubs
2 of Diamonds
```

```
In [135... pair.has_pair()    # should return True
```

```
Out[135... True
```

```
In [136... bad_hand.has_pair()    # should return False
```

```
Out[136... False
```

```
In [137... good_hand.has_pair()    # should return False
```

```
Out[137... False
```

Exercise

A hand has a full house if it contains three cards of one rank and two cards of another rank. Write a `PokerHand` method that checks whether a hand has a full house.

You can use the following outline to get started.

```
In [ ]: %%add_method_to PokerHand

def has_full_house(self):
    return False
```

```
In [ ]:
```

You can use this hand to test your method.

```
In [ ]: boat = deepcopy(pair)
boat.put_card(Card(2, 2))
boat.put_card(Card(2, 3))
print(boat)
```

```
2 of Clubs
3 of Clubs
4 of Hearts
5 of Spades
7 of Clubs
2 of Diamonds
2 of Hearts
3 of Hearts
```

```
In [ ]: boat.has_full_house()    # should return True
```

```
Out[ ]: True
```

```
In [ ]: pair.has_full_house()    # should return False
```

```
Out[ ]: False
```

```
In [ ]: good_hand.has_full_house()    # should return False
```

```
Out[ ]: False
```

Exercise

This exercise is a cautionary tale about a common error that can be difficult to debug. Consider the following class definition.

```
In [ ]: class Kangaroo:
    """A Kangaroo is a marsupial."""

    def __init__(self, name, contents=[]):
        """Initialize the pouch contents.

        name: string
        contents: initial pouch contents.
        """
        self.name = name
        self.contents = contents

    def __str__(self):
        """Return a string representaion of this Kangaroo.
        """
        t = [ self.name + ' has pouch contents:' ]
        for obj in self.contents:
            s = '    ' + object.__str__(obj)
            t.append(s)
        return '\n'.join(t)

    def put_in_pouch(self, item):
        """Adds a new item to the pouch contents.

        item: object to be added
        """
        self.contents.append(item)
```

`__init__` takes two parameters: `name` is required, but `contents` is optional -- if it's not provided, the default value is an empty list.

`__str__` returns a string representation of the object that includes the name and the contents of the pouch.

`put_in_pouch` takes any object and appends it to `contents`.

Now let's see how this class works. We'll create two `Kangaroo` objects with the names `'Kanga'` and `'Roo'`.

```
In [ ]: kanga = Kangaroo('Kanga')
        roo = Kangaroo('Roo')
```

To Kanga's pouch we'll add two strings and `roo`.

```
In [ ]: kanga.put_in_pouch('wallet')
        kanga.put_in_pouch('car keys')
        kanga.put_in_pouch(roo)
```

If we print `kanga`, it seems like everything worked.

```
In [ ]: print(kanga)
```

```
Kanga has pouch contents:
  'wallet'
  'car keys'
  <__main__.Kangaroo object at 0x7f44f9b4e500>
```

But what happens if we print `roo`?

```
In [ ]: print(roo)
```

```
Roo has pouch contents:
  'wallet'
  'car keys'
  <__main__.Kangaroo object at 0x7f44f9b4e500>
```

Roo's pouch contains the same contents as Kanga's, including a reference to `roo`!

See if you can figure out what went wrong. Then ask a virtual assistant, "What's wrong with the following program?" and paste in the definition of `Kangaroo`.

```
In [ ]:
```

[Think Python: 3rd Edition](#)

Copyright 2024 [Allen B. Downey](#)

Code license: [MIT License](#)

