

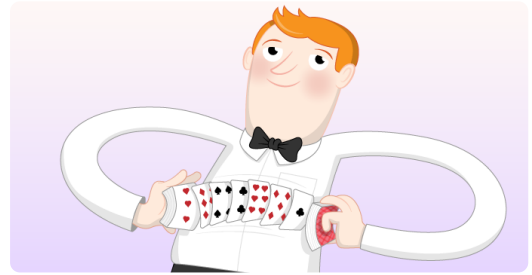


Projects

Deck of cards

Create a reusable object oriented model of a deck of cards

Python



Step 1 Introduction

Create a model of a deck of cards that can form the basis for building digital card game programs such as Poker or Gin Rummy.

What you will make

You will learn how to use the object-oriented programming paradigm in Python to create a reusable model of a deck of cards.

Object-oriented programming (OOP) is a way of organising your code so it is easier to understand, reuse, and change. OOP allows you to combine data (variables) and functionality and wrap them together inside **objects**.



Image by Rosapicci (Own work) **CC BY-SA 4.0** (<https://creativecommons.org/licenses/by-sa/4.0>), via Wikimedia Commons



What is object-oriented programming?

You have probably heard of object-oriented programming, but perhaps you are unsure about what it is. Maybe you have even attempted to read guides or books but got lost in the jargon.

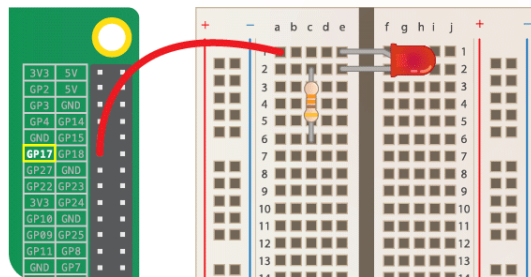
Object-oriented programming is integral to many programming languages and it is simply a different style of programming.

What is an object?

Objects are used to model things in code. An object can represent a physical item, such as an LED, or it can represent a digital unit, such as a bank account or a character in a computer game. An object is basically a group of data and functions. Because you can define your own objects, you can represent anything you like with an object!

Where would I have seen objects before?

Let's look at an example of an LED wired up to a Raspberry Pi computer. Don't worry if you have never wired up an LED or done other physical computing – the important thing here is the code!



On the left of the diagram are the Raspberry Pi's GPIO pins, which allow us to control components that are connected to them. The LED is connected to pin 17. To make the LED switch on, you would use the following Python code:

```
from gpiozero import LED
red = LED(17)
red.on()
```

To interact with the LED, we have created an `LED` object which represents the physical LED in code. It has the name `red` so that we can refer to that specific LED object.

```
red = LED(17)
```

We can wire up another LED to pin 21, and then create another object with a different name to represent it:

```
green = LED(21)
```

Why would I want to use objects?

In our example, we created an `LED` object to model a physical LED in code. We also included a command to control the LED, in this case to turn it `on`. Such commands are called methods – custom functions specifically designed to interact with an object.

One of the benefits of using object-oriented programming is that unnecessary details can be left out when we use a method. We do not need to know the specifics of exactly how a method works to be able to use it, we can just call it to achieve a desired outcome. In our example, we don't need to know anything about the `on()` method apart from the fact that using it on our `LED` object will make the physical LED light up.

What you will learn

This project covers elements from the following strands of the **Raspberry Pi Digital Making Curriculum** (<http://rpf.io/curriculum>):

- **Apply higher-order programming techniques to solve real-world problems** (<https://curriculum.raspberrypi.org/programming/maker/>)

Additional information for educators

If you need to print this project, please use the **printer-friendly version** (<https://projects.raspberrypi.org/en/projects/deck-of-cards/print>).

Use the link in the footer to access the GitHub repository for this project, which contains all resources (including an example finished project) in the 'en/resources' folder.

Step 2 What you will need

Hardware

- Any computer that can run Python 3

Software

- Python 3

If your computer doesn't have Python 3 installed, you can either install it, or create a new Python 3 project **on Trinket** (<https://rpf.io/python3new>).

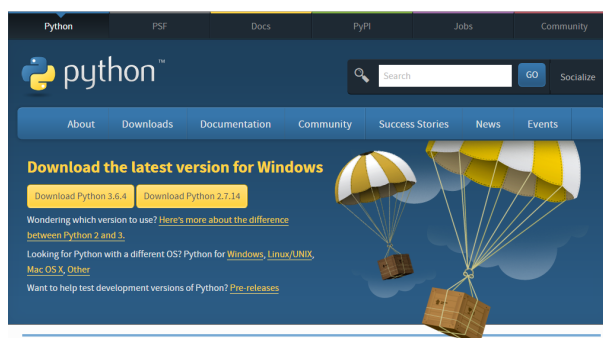
How to install Python 3

If Python 3 or IDLE isn't installed on your computer, follow the installation instructions below for your operating system.

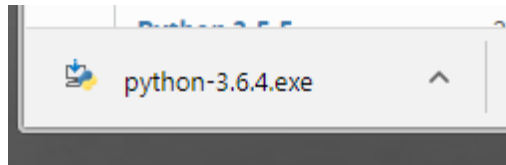
IDLE is a standard interactive development environment for writing and executing Python code that you will use in many of our Python projects.

Microsoft Windows

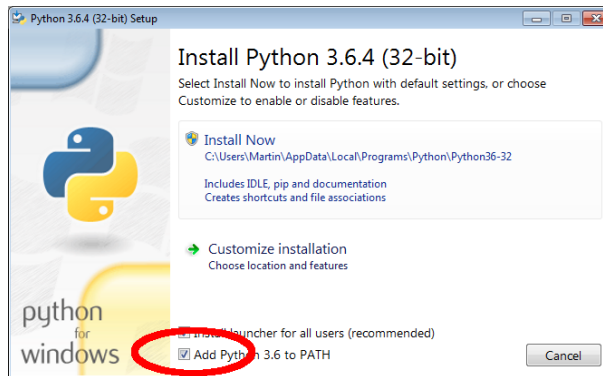
- Open your web browser and navigate to **www.python.org/downloads** (<https://www.python.org/downloads>).
- On this webpage, you will see a button to install the latest version of Python 3. Click it, and a download will start automatically.



- Click on the **.exe** file to run it. (It will have been saved in your **Downloads** folder, or wherever your computer saves downloaded files by default.)



- In the dialogue box that opens up, it is important to first tick the box next to **Add Python 3 to PATH**.



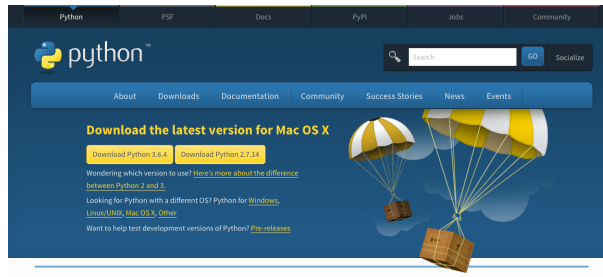
- Click **Install Now** and follow the install guide. The setup process will take a little time.



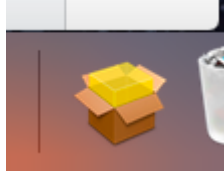
- Once the setup is complete, click **Done** and then close your web browser. Now you can go to the start menu to open IDLE.

macOS

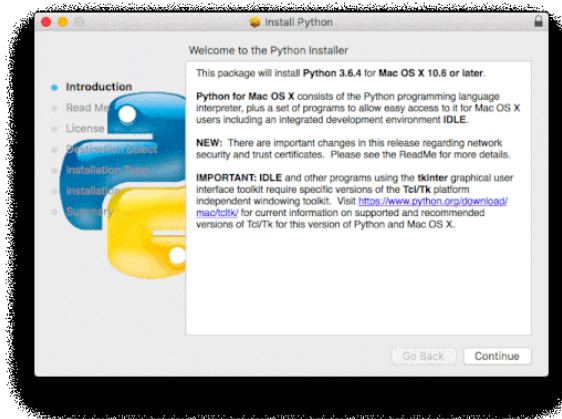
- Open your web browser and navigate to **www.python.org/downloads** (<https://www.python.org/downloads>).
- On this webpage, you will see a button to install the latest version of Python 3. Click it, and a download will start automatically.



- Click on the download in the dock to start the installation process.



- Click **Continue** and follow the installation guide. The installation may take a little time.



- When it's complete, click **Close**.
- Open IDLE from your Applications.

Linux (Debian-based distributions)

Most distributions of Linux come with Python 3 already installed, but they might not have IDLE installed. Use `apt` to check whether they are installed and install if they aren't.

- Open up a terminal window and type:

```
sudo apt update
sudo apt install python3 idle3
```

This will install IDLE (and Python 3), and you should then be able to find it in your Application menu.

Step 3 Create a class

A **class** is like a template for creating objects. Think of a class as being similar to a cookie cutter – it is a template for all the cookie objects you make. You can make as many instances of cookie objects as you want, and they will all start off from the same template.

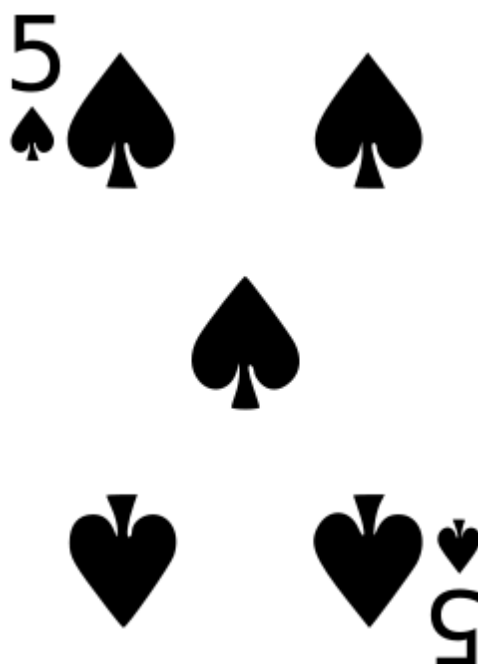
When you make real cookies, you make them with the same cookie cutter, but you can customise each individual cookie, for example by adding some icing or some sprinkles. It's the same with objects created using a class – you can customise each object by storing different data in it. Let's look at how this works in practice.

Create a class

You are going to begin by making a **Card** class that will act as a template for creating playing card objects.

Each card object is a separate **instance** of the **Card** class. For example, you might have a card object representing the 5 of Spades and another card object representing the 2 of Hearts.

Our playing cards will be represented as text rather than pictures like the one below.



You can choose whether to watch the video or to use the written instructions.

- Open a new Python file and save it as `card.py`.
- Begin by creating a `Card` class:

```
class Card:
```

Class names are usually written with a capital letter so that they are easily distinguishable from variable names.

Next, you are going to add a **method** to the class. Methods are very similar to functions, and we use them to interact with objects.

Methods

You may have already encountered functions when writing Python code. Functions allow us to give a name to a set of instructions. You can pass data to a function as parameters, and optionally you can have it return some data as a result.

The difference between a function and a method is that the method is called **on an object**. This means that a method can use all of the data stored inside the object, as well as any data that you pass to it as parameters.

Create an `__init__` method

In Python, every class has a special method called `__init__` that tells it how to create (or **initialise**) an object. This particular method name always has a double underscore on either side of `init`.

- Create an `__init__` method inside your `Card` class:

```
class Card:
    def __init__(self):
```



Why do I need the ``self`` in the brackets?

A method needs context in order to work. `self` is the reference to the object, and it needs to be the first parameter passed to any `Class` method. This is because the method needs to know what it is being

called on, so that it can use the data stored within the object.

Let's look at an example:

Outside of OOP, for two functions to share the same variable, it must be global:

```
name = "Laura"

def hi():
    print("Hi " + name)

def bye():
    print("Bye " + name)
```

Within a class, `self` can be used to share variables.

```
class Welcome():
    def __init__(self):
        self.name = "Laura"

    def hi(self):
        print("Hi " + self.name)

    def bye(self):
        print("Bye " + self.name)
```

Here, we defined the variable `self.name` and set its value to `"Laura"` within the `__init__` method that initialises object of this class. Thus all objects will contain a variable `self.name` set to `"Laura"`. The `hi()` and `bye()` methods we defined can now use the information stored in `self.name`.

Attributes

Attributes are pieces of information stored within an object, rather like a collection of variables associated with that object. The card object will begin with two attributes, `suit` and `number`, and we will prefix them with `self.` to show that they belong to the object instance.

- Add two attributes to your `__init__` method, and two parameters so that you can pass in their values as arguments when you create

the object:

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

Step 4 Instantiate an object

Let's test out our `Card` class by creating a card object. The object is an **instance** of the `Card` class, and creating it is also called **instantiating**.

- Below your class definition, instantiate a card object called `my_card` for the 6 of Hearts:

```
my_card = Card("hearts", "6")
```

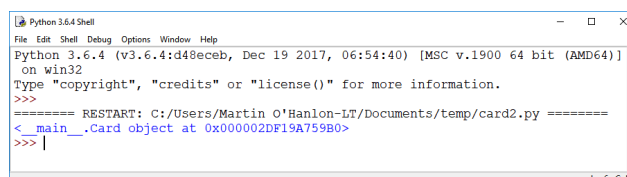
You may be wondering why the card number is `"6"` (a string) and not `6` (an integer). This is because some of the card "numbers" will be letters: `"J"`, `"Q"`, `"K"`, and `"A"`.

- Add a print statement to display the card object.

```
print(my_card)
```

- Run the program.

You're probably expecting to see an output containing `"hearts"` and `"6"`. What you will see instead is the text **representation** of your object – it is a `Card` object, and you are shown its address in your computer's memory:



```
Python 3.6.4 Shell  
File Edit Shell Debug Options Window Help  
Python 3.6.4 (v3.6.4:d48ecef, Dec 19 2017, 06:54:40) [MSC v.1900 64 bit (AMD64)]  
on win32  
Type "copyright", "credits" or "license()" for more information.  
>>>  
===== RESTART: C:/Users/Martin O'Hanlon-LT/Documents/temp/card2.py =====  
<__main__.Card object at 0x000002DF19A759B0>  
>>>
```

This output is created by a special method called `__repr__` (which is short for 'representation'). All objects in Python have this method by default, meaning you do not need to create `__repr__` yourself. It will

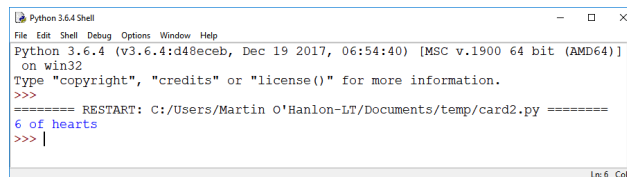
automatically be used whenever you tell your program to return a text representation of an object. However, you can **override** the default output of the `__repr__` method to change how your object is represented as text.

- Go back to your `Card` class definition and add in some code to override the `__repr__` method so that it describes the card in a more meaningful way:

```
def __repr__(self):  
    return self.number + " of " + self.suit
```

For example, if `self.number` is "5" and `self.suit` is "spades", this will print "5 of spades".

- Run the program again and check that your new way of representing the object as text works, e.g.



```
Python 3.6.4 Shell  
File Edit Shell Debug Options Window Help  
Python 3.6.4 (v3.6.4:d48ecef, Dec 19 2017, 06:54:40) [MSC v.1900 64 bit (AMD64)] ^  
on win32  
Type "copyright", "credits" or "license()" for more information.  
>>>  
===== RESTART: C:/Users/Martin O'Hanlon-LT/Documents/temp/card2.py =====  
6 of hearts  
>>> |
```

You can also customise this representation to your own liking.

Step 5 Attributes and properties

So you can change the attributes within an object, your class will need some **properties**.

Properties are the special methods that either **get** or **set** the value of an attribute, and are therefore referred to as the **getter** and the **setter**.

Getters and setters

For this section, you can either watch the video below, or follow the written instructions.

It is possible to access the attributes of an object directly. For example, you could add the following code at the bottom of your program to change the `suit` attribute of `my_card` and then display the object:

```
my_card.suit = "clubs"  
print(my_card)
```

However, accessing attributes directly is not a good idea, because people could do things like this:

```
my_card.suit = "dinosaurs"  
print(my_card)
```

Instead, let's create properties for your class: a **getter** and a **setter** for accessing the `suit` attribute.

- Before we start on this, go back to your `__init__` method and locate `self.suit`. Add an underscore `_` before `suit` to indicate that you do not want people to access this attribute directly.

```
def __init__(self, suit, number):  
    self._suit = suit
```



Does the underscore mean the attribute cannot be changed directly?

Adding an underscore is good practice and good programming style.

However, the underscore **will not prevent** people from changing the attribute directly – it is a convention which indicates that they **should not**. If you want to test this, add an underscore to your code for changing the attribute to `"dinosaurs"`:

```
my_card._suit = "dinosaurs"
```

You will see that you can still change the attribute just as before.

Creating a getter

- Go back to your `Card` class definition, add a new method called `suit`, and have it return the value of the `_suit` attribute.

```
def suit(self):  
    return self._suit
```

- Add a **decorator** to this method to say that it is a property.

```
@property  
def suit(self):  
    return self._suit
```

Now, whenever someone uses the value `my_card.suit` in their program, this getter will be called, and the user will receive the value of `self._suit` stored within the `my_card` object.



What's a decorator?

In object-oriented programming, decorators allow you to add additional behaviour (or functionality) to a class.

A decorator can be thought of as a wrapper to a method: it contains the method but it also extends its functionality.

The `@property` decorator in Python needs to be added to the getter method to define the method as a property.

Creating a setter

- Add another method. It is important that this method **is also called** `suit`. It should take a piece of data representing the new suit the user would like to set, and do a basic check to make sure that the piece of data is one of the usual suits available in a deck of cards.

```
def suit(self, suit):  
    if suit in ["hearts", "clubs", "diamonds",  
               "spades"]:  
        self._suit = suit  
    else:  
        print("That's not a suit!")
```

- Now add a decorator to this method to say that it is the setter for `suit`.

```
@suit.setter  
def suit(self, suit):
```

As with the getter method, this decorator defines the method as a property. Now, whenever someone tries to set the `suit` attribute (e.g. by typing `my_card.suit = "spades"`), the `@suit.setter` property will be called. In this example, the value `"spades"` will be passed to it as the `suit` parameter.

Notice that by using getter and setter properties and decorators, you can have two functions with the same name, one which is called when you get the value, and one which is called when you set the value.



Why do we use properties?

Why would we want to use the `@property` and `.setter` decorators to create properties instead of just creating a `get_suit()` method and a `set_suit()` method?

There are several reasons:

- It's shorter and nicer to be able to refer to `card.suit` rather than `card.get_suit()` and `card.set_suit()`. For example:

```
my_suit = card.suit  
card.suit = "spades"
```

is much neater than:

```
my_suit = card.get_suit()  
card.set_suit("spades")
```

- You can make complex functions look like simple operations. Think back to the LED example on step 1: you don't need to know the details of how to turn on an LED with your computer – `on()` does it all for you.
- If you use properties rather than allowing direct access to attributes and you need to change how the class works, you can do so without breaking any code that uses the class. For example,

the original `suit` setter you wrote simply stored the suit, but let's say you want to store it in capital letters. To do so, you can just change the code within the property:

```
@suit.setter
def suit(self, suit):
    if suit in ["hearts", "clubs", "diamonds",
               "spades"]:
        self._suit = suit.upper()
    else:
        print("That's not a suit!")
```

Now all suits will be stored as capitals, while any code that uses the `suit` property will still work.

If you let people access the `suit` attribute directly, you will not be able to change any aspects of its code later.

- Run the program. If you now try to change the card's suit to anything other than one of the suits in the list, you should see `"That's not a suit!"` appear, and the suit should not change.

Note that you currently don't have any validation in the `__init__` method, so you could still create the 2 of Dinosaurs like this:

```
another_card = Card("Dinosaurs", "2")
```

Step 6 Challenge: add a getter and setter

Now it's your turn. Follow the same principles as in the previous step to add a **getter** and a **setter** method for the attribute `number`.

Don't forget:

- In a deck of cards, each card may have a number between 2 and 10, or it may be a picture card or an ace (J, Q, K, A). Think about which data type you will need to use to handle this.

- Think about how you will check whether the input to your setter method is valid.

Challenge: aces high?

- Can you add another property which gives you the value of a card as an integer? This might be different for different games: some games count all “picture cards” (J, K, Q) as worth 10, and depending on the game, the ace may be high or low. How could you represent the value of a card in your `Card` model without being tied to any particular game?

Step 7 A deck of cards

Now that you have a basic model of a card, it's time to create a deck.

- Create a `Deck` class. You can either do this in the same file where you wrote your `Card` class, or in a separate one. If you do it in a different file (e.g. `deck.py`), you will need to import the `Card` class at the top of that file with this code:

```
from card import Card
```

In this line of code, `card` is the name of the Python file containing the class, minus the `.py` extension, and `Card` is the name of the class.

- Create a new `Deck` class and include an `__init__` method in it. This time we won't need any parameters other than `self`, which is compulsory.

```
class Deck:  
  
    def __init__(self):
```

- The `Deck` will need to store a list of cards, each of which will be a `Card` object. Add an attribute called `_cards` to the `__init__` method, and define it as an empty list for now.


```
class Deck:

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

- Now let's write a method to populate the deck with the 52 necessary cards. Begin by creating a method called `populate`:

```
def populate(self):
```

- Inside your method, define two lists. One should contain all the possible card suits, and the other all the possible card numbers, as strings:

```
def populate(self):
    suits = ["hearts", "clubs", "diamonds", "spades"]
    numbers = ["2", "3", "4", "5", "6", "7", "8", "9",
               "10", "J", "Q", "K", "A"]
```



Is there a more efficient way to make the list of numbers?

Yes! Instead of writing them all out, you could use **list comprehension**, which is a quick way of creating a new list based on an existing list.

So, to create a list containing the numbers 2 to 10 as strings, you could use the code:

```
numbers = [str(n) for n in range(2,11)]
```

This means:

- Give me `str(n)` (the string version of `n`)
- For every `n in range(2, 11)` (remember that the `range()` function will start at 2 and stop at (but not include) 11)

Then add a list containing the picture cards and the ace at the end:

```
numbers = [str(n) for n in range(2,11)] + ["J", "Q",
"K", "A"]
```

- So that `populate` generates the deck of cards, we just have to combine items from the two lists – for each suit, for each number, create a `Card` object. One way of doing this is with nested loops:

```
cards = []                                # Create an empty
list of cards
for suit in suits:                        # For each suit...
    for number in numbers:                # For each
number...
        # Create a new Card object and add it to the
list
        cards.append( Card(suit, number) )
self._cards = cards                       # Then point
self._cards at this list
```

However, using nested loops can make your code more complicated. To simplify the code, you can use **list comprehension**:

```
self._cards = [ Card(s, n) for s in suits for n in
numbers ]
```

This code means:

- Set `self._cards` to
- `[a list]`
- of `Card` objects
- containing every combination of `s, n`, looping through `for s in suits` and `for n in numbers`

If you would like to know more about list comprehensions, have a look at the information below.



Simple Python list comprehensions

- If you want to generate a list using Python, it is quite easy to do so using a **for loop**.

```
new_list = []
for i in range(10):
    new_list.append(i)

>>> new_list
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

- The same list can be created in a single line using a construct that exists in many programming languages: a **list comprehension**.

```
new_list = [i for i in range(10)]

>>> new_list
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

- You can use any **iterable** in a list comprehension, so making a list from another list is easy.

```
numbers = [1, 2, 3, 4, 5]
numbers_copy = [number for number in numbers]

>>> numbers_copy
[1, 2, 3, 4, 5]
```

- You can also do calculations within a list comprehension.

```
numbers = [1, 2, 3, 4, 5]
double = [number * 2 for number in numbers]

>>> double
[2, 4, 6, 8, 10]
```

- String operations can be done as well.

```
verbs = ['shout', 'walk', 'see']
present_participle = [word + 'ing' for word in verbs]

>>> present_participle
['shouting', 'walking', 'seeing']
```

- You can also extend lists quite easily.

```
animals = ['cat', 'dog', 'fish']
animals = animals + [animal.upper() for animal in animals]

>>> animals
['cat', 'dog', 'fish', 'CAT', 'DOG', 'FISH']
```

- Let's test whether your method properly constructs a deck. Go back to your `__init__` method, call the `populate()` method, then print out the list of cards:

```
def __init__(self):  
    self._cards = []  
    self.populate()  
    print(self._cards)
```

- Create an instance of the `Deck` class to check whether you are getting the deck you want.

Use this code:

```
my_deck = Deck()
```

You should see the following output:

```
[2 of hearts, 3 of hearts, 4 of hearts, 5 of hearts, 6  
of hearts, 7 of hearts, 8 of hearts, 9 of hearts, 10  
of hearts, J of hearts, Q of hearts, K of hearts, A of  
hearts, 2 of clubs, 3 of clubs, 4 of clubs, 5 of clubs,  
6 of clubs, 7 of clubs, 8 of clubs, 9 of clubs, 10  
of clubs, J of clubs, Q of clubs, K of clubs, A of clubs,  
2 of diamonds, 3 of diamonds, 4 of diamonds, 5 of  
diamonds, 6 of diamonds, 7 of diamonds, 8 of diamonds,  
9 of diamonds, 10 of diamonds, J of diamonds, Q of  
diamonds, K of diamonds, A of diamonds, 2 of spades, 3  
of spades, 4 of spades, 5 of spades, 6 of spades, 7  
of spades, 8 of spades, 9 of spades, 10 of spades, J  
of spades, Q of spades, K of spades, A of spades]
```

Step 8 Challenge: what else?

Now that you have created `Card` and `Deck` classes, what else could you add so that you could use them to program card games? Here are some ideas:

- Add a `shuffle()` method to randomise the order of the cards in the deck.
- Add a method to check whether a particular card is present in the deck or not.
- Add a `deal()` method to deal cards from the deck. How many cards will you deal to how many players?
- Could you create a `Hand` class to model the cards stored in a player's hand?

Published by **Raspberry Pi Foundation** (<https://www.raspberrypi.org>) under a **Creative Commons** license (<https://creativecommons.org/licenses/by-sa/4.0/>).

View project & license on GitHub (<https://github.com/RaspberryPiLearning/deck-of-cards>).