Unit 9

Learn something new

1. Python provides built-in data types like string and list. Each of these is used to store some kind of data and to manipulate it in appropriate ways. For the manipulations, we use methods—`lower`, `split`, `format`, `isdigit` and many others for strings, and `sort`, `append` and others for lists.

   One main feature of the `tkinter` module is that it introduces a host of additional data types like frames, labels and buttons, each equipped with its own methods—`get` for entry boxes, for example, and `pack` and `after` for all widgets. We'd like to be able to do the same, creating new data types relevant to any application we tackle.

   For example, if we're writing a banking program, we might like to work with account objects. The following hypothetical code would then set up an account for Emily Sanchez and deposit $100 into it:

   ```
   sampleAccount = Account('Emily Sanchez')
   sampleAccount.deposit(100)
   ```

   Or, for a poker program, we might want to work with deck objects that contain playing card objects. Then we could write the following to deal a hand of five cards and display their face values:

   ```
   d = Deck()
   d.shuffle()
   hand = [d.deal() for number in range(5)]
   for card in hand:
     print(card.faceValue())
   ```

   Here we're imagining that deck objects have a `shuffle` method that puts the cards in a random order and a `deal` method that returns the top card from the deck and removes it. We're also expecting card objects to have a `faceValue` method that returns a string containing the face value.

   Let's make all this a reality. We'll start by creating a card data type.

   ```
   class Card:
     def __init__(self, f, s):
       self.myFaceValue = f
       self.mySuit = s
     def faceValue(self):
       return self.myFaceValue
     def suit(self):
       return self.mySuit
   ```

```
card1 = Card('jack', 'spades')
card2 = Card('3', 'hearts')

print(card1.faceValue())
print(card2.suit())
```

In Python, we create a new data type with a `class` statement. The term **class** means a data type that's added rather than built in. We can use any name we like for a class, but it's conventional to use one starting with an uppercase letter.

To create an **instance** of the class—an individual object of this type—we use the class name as if it were a function. The line

```
card1 = Card('jack', 'spades')
```

makes a new card object and uses the name `card1` to refer to it. When an instance is created, the class function with the special name `__init__` is automatically called. The function name starts and ends with two underscore characters to distinguish it from names we might use ourselves for other purposes.

If you look at the definition of the `__init__` function above, you'll see that it takes three arguments. When we call it, though, we pass in only two. The first argument is always automatically set to refer to the newly created instance. We used the name `self` for this first argument, which is conventional. So when we write `Card('jack', 'spades')`, a call to `__init__` is generated with `f` set to `'jack'`, `s` set to `'spades'` and `self` set to the card object we're creating.

An instance will normally have **attributes**, which are just names associated with the instance and accessed using dot notation. In our example, the code

```
self.myFaceValue = f
```

creates an attribute called `myFaceValue` for the newly created card referred to by `self` and assigns it to the string `'jack'`.

To provide our new data type with methods, we just define functions within the `class` statement. Functions defined inside a class are no different than ordinary functions except in the way that they're called, attached with a dot to an object. For example, if `card2` refers to a card object, then

```
card2.suit()
```

calls the `suit` method of the card class. Moreover, and this is crucial, `card2` itself will automatically be passed to `suit` as the first argument. Looking back at the definition of `suit`, we can see that this means `self` will again refer to the current instance, i.e. `card2`.

Once the card class is defined, as in the code above, we can create as many cards as we like and each one will have its own `myFaceValue` and `mySuit` attributes. Calling a method like `suit` on a particular instance will return the value of `mySuit` for that instance.

Before we move on to create a data type for *decks* of cards, it will be useful to add one more method to the `card` class. As things stand, code like this

```
card1 = Card('jack', 'spades')
print(card1)
```

will produce output that looks something like this:

```
<__main__.Card object at 0x018CD830>
```

That's because, although Python knows how to display lists, strings and other built-in data types, we haven't yet told it what to do with cards. By default Python displays the technical information shown above. If we prefer, we can specify something else to be used in printing—and in `format` and all other contexts when a string is produced from an object—by adding a `__str__` function. Here's our final version of the card class:

```
class Card:
   def __init__(self, f, s):
      self.myFaceValue = f
      self.mySuit = s
   def __str__(self):
      return self.myFaceValue + ' of ' + self.mySuit
   def faceValue(self):
      return self.myFaceValue
   def suit(self):
      return self.mySuit
```

Now

```
card1 = Card('jack', 'spades')
print(card1)
```

will produce the output

```
jack of spades
```

2. The code for the deck class will start as follows:

```
import random

class Deck:
  faceValues = ['ace', '2', '3', '4', '5', '6', '7', '8',
               '9', '10', 'jack', 'queen', 'king']
  suits = ['clubs', 'diamonds', 'hearts', 'spades']
  def __init__(self):
    self.theCards = [Card(faceValue, suit)
                    for faceValue in Deck.faceValues
                    for suit in Deck.suits]
    self.shuffle()
  def shuffle(self):
    random.shuffle(self.theCards)
```

Each deck instance will contain a list called `theCards` that contains card objects, one for each card that has not yet been dealt. The `shuffle` method simply puts these cards in random order.

Notice that our code uses two names—`faceValues` and `suits`—that refer to attributes of the *class* rather than to particular *instances*. When we want to use these—inside the `__init__` method, for example—we write `Deck.faceValues` and `Deck.suits`. Actually, *all* names defined under the class header can be used in this way. Instead of writing

```
self.shuffle()
```

and passing the instance `self` to `shuffle` implicitly, we can write

```
Deck.shuffle(self)
```

passing `self` explicitly. We'll see soon why this may be necessary.

The `__init__` method for the deck class uses a list comprehension with two `for` parts. It's equivalent to the following less elegant code:

```
self.theCards = []
for faceValue in Deck.faceValues:
  for suit in Deck.suits:
    self.theCards.append(Card(faceValue, suit))
```

Note that, in either case, we're creating card instances using the code we wrote for the card class.

To complete the deck class, we'll just add two additional methods:

```
class Deck:
   .
   .
   .
  def deal(self):
    return self.theCards.pop()
  def cardsLeft(self):
    return len(self.theCards)
```

In the first, we employ the list method **pop**, which removes and returns the last item in a list.[1] We've also included a method to tell us how many cards remain in the deck.

Now here's an example of how we can use our new classes. It's a program that simulates dealing from two decks side by side and noting when there's a coincidence and the same card happens to be dealt from both decks.

```
class Card:
    .
    .
    .
class Deck:
    .
    .
    .
deck1 = Deck()
deck2 = Deck()
while deck1.cardsLeft() > 0:
  card1 = deck1.deal()
  card2 = deck2.deal()
     print('{0:18s} {1:18s}'.format(str(card1), str(card2)),
      end='')
  sameFaceValue = card1.faceValue() == card2.faceValue()
  sameSuit = card1.suit() == card2.suit()
  if sameFaceValue and sameSuit:
    print('  Coincidence!!!')
  else:
    print()
```

If you run this program a few times, you'll find that coincidences are surprisingly common.

[1] This means that we're dealing from the end of the deck rather than the beginning. So long as we're consistent, this can't make any difference. There are technical reasons why it's more efficient to delete items at the end of a list rather than the beginning, which is why **pop** is defined as it is.

3. One of the most useful and powerful features of classes is that we can easily define new classes as specialized versions of old ones. For example, if we've already designed a class for representing bank accounts in general, it will be easy to create a class for joint accounts—which belong to two people instead of just one. A joint account is just a regular account with a special feature—an additional attribute for the name of the second account holder.

   As a warm-up example, here's how we would specialize our deck class to create a new class called `personalDeck`. Each instance of this new class has an owner, whose name is displayed when we print the deck.

```
class Card:
    .
    .
    .
class Deck:
    .
    .
    .
class personalDeck(Deck):
  def __init__(self, name):
    Deck.__init__(self)
    self.owner = name
  def __str__(self):
    return "{0}'s deck".format(self.owner)
```

   The header line

```
class personalDeck(Deck):
```

   is best read "`personalDeck` is a kind of `Deck`." As a kind of deck, a personal deck **inherits** all the methods the deck class defines. That means these methods—`deal`, `shuffle`, etc.—are automatically defined for a personal deck without any additional effort on our part.

   Note that the first step in creating a personal deck is to carry out all the usual steps for initializing an ordinary deck. We arrange for that to be done with the line

```
Deck.__init__(self)
```

   In this case, we *must* use the technique of explicitly passing the instance. If we instead wrote

```
self.__init__()
```

we would be calling the **personalDeck.__init__** function, the very one we're trying to define.

Once the personalDeck class is defined, we can use it as follows:

```
pd = personalDeck('John')
print(pd)
print(pd.cardsLeft())
```

The output is

```
John's deck
52
```

4. Personalized decks of playing cards aren't of much use. For a more practical example, let's create a new kind of GUI widget. Frames, labels, buttons, entry boxes and all other **tkinter** widgets are defined as classes, so we can specialize them using the same technique we used to create **personalDeck**.

   When we write GUI programs we very often want to include a 'Quit' button that the user can click to exit. Here's a class that defines one by specializing the **Button** class provided by **tkinter**. As the example shows, once the class is written, we can add a 'Quit' button to any program with just one line.

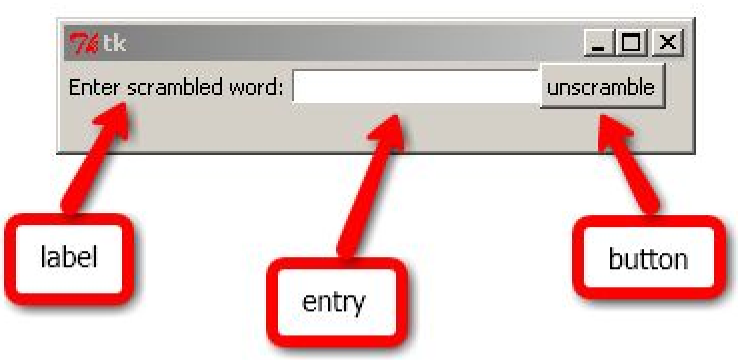
```
from tkinter import *

class quitButton(Button):
   def __init__(self, parent):
      Button.__init__(self, parent)
      self['text'] = 'Quit'
      self['command'] = parent.destroy
      self.pack(side=BOTTOM, anchor=E)

root = Tk()

quitButton(root)

mainloop()
```

   Here we've used the **destroy** method. When applied to the main window, it shuts down the program. If we apply it to a widget, it removes the widget and repacks all the remaining ones.[2]

[2] A more standard way to shut down would be to use the **quit** method, which stops the event loop. Since we're running under IDLE, and sharing its event loop, this would have the effect of stopping not only our program but IDLE too. We use **destroy** to avoid this and also because it will be useful to us a little later in the unit.

Notice that the `quitButton` is self packing. This makes it easier to use, but slightly non-standard—we have to remember that it isn't necessary to call `pack` for this particular widget. Notice also that, since we don't need to refer to the button created in the main part of the program, we haven't assigned a name to it.

5. Another handy widget we can build is one that combines an entry box, a label that tells the user what to type in it and a button to click when the typed input is ready for processing. This combination can be used to good effect in three of the four programs in the 'Apply what you know' section of Unit 8. For example, here's how it might be employed in a solution for Program 8.1:



What we want is a specialized kind of frame—one containing a label, entry box and button. Here's code for the new widget, which we'll call an enhanced entry box.

```
from tkinter import *

class enhancedEntry(Frame):
   def __init__(self, parent, prompt, actionText, action):
      Frame.__init__(self, parent)

      self.inputBoxLabel = Label(self)
      self.inputBoxLabel['text'] = prompt
      self.inputBoxLabel.pack(side=LEFT, fill=X)

      self.inputBox = Entry(self)
      self.inputBox.pack(side=LEFT, fill=X)

      self.button = Button(self)
      self.button['text'] = actionText
      self.button['command'] = action
      self.button.pack(side=LEFT, fill=X)

   def get(self):
      return self.inputBox.get()

   def setActionText(self, actionText):
      self.button['text'] = actionText

   def setPrompt(self, prompt):
      self.inputBoxLabel['text'] = prompt

   def setAction(self, cmd):
      self.button['command'] = cmd
```

When we want to create an instance of an enhanced entry box, we simply specify the text to be used for both the label and the button and the function to be called when the button is clicked. Our class provides methods to change any of those settings later, as well as one that returns the typed input stored in the ordinary entry box inside the enhanced one.

If we store this and the code for our ‘Quit’ button class in a file called `myWidgets.py`, we can write a complete program as follows. Notice how simple the GUI part of the code is.

```
from tkinter import *
from myWidgets import *

word = ''

def slide():
    global word
    word = userInput.get()
    result['text'] = ''
    doSlide()

def doSlide():
    charactersToShow = len(result['text']) + 1
    result['text'] = word[-charactersToShow:]
    if charactersToShow < len(word):
        root.after(100, doSlide)

root = Tk()

quitButton(root)

userInput = enhancedEntry(root, 'Enter text:', 'Go', slide)
userInput.pack(fill=X)

result = Label(root)
result.pack(side=LEFT, fill=X, anchor=W)

mainloop()
```

Run the program now to see what it does. Try typing ‘This is a slide test’ in the entry box and clicking ‘Go’.