Using objects in Python for CS2515

Class and objects as complex data types

An example card game object-oriented model

Note

We need to use classes and objects to represent the data structures and Abstract Data Types we will explore in CS2515, so this is a short description of some of the concepts and Python features that we will use.

CS2513 contains a much more detailed and principled treatment of Object-Oriented Programming in Python.

If there is any conflict in the description of the concepts or the use of the features, then when you are in CS2513, you should use the methodology presented in CS2513.

Python data types

Integers, Floats, Booleans – a variable of any of these types takes a single value at a time, and allows particular operations – e.g. division, comparison, ...

When we type (e.g.) x = 3 we are telling Python that x refers to a variable of type Integer, and it has value 3

Lists – a variable of this type maintains a sequence of other variables.

```
x = [3, “word”, 5.2]
x.reverse()
print(x)
```

gives output [5.2, “word”, 3]

A variable stores data.
The type of variable says:
- what type of data is stored
- what operations can be done on the data

Lists under the hood

Python specifies how a list behaves using an internal *class* definition. The class definition is a piece of code.
A list stores multiple items in numbered positions. You can add new items onto the end, read or change the value stored in a specified position, ...
You can ask a list how many items it has, reverse its order, ...

```
x = [3, “word”, 5.2]
x.reverse()
```

the square brackets tell python this is a list

x is a variable that is now referring to a list

tells the list stored in x to reverse the sequence of its elements – ‘reverse’ is a method specified in the class definition – when you create a list object, this method is available to you.

Example: patient records

Suppose we are asked to write software to manage a waiting list for operations in a hospital.

We need to keep a set of records for patients who need the operation. We want to order the patients by priority. We will need to compare two patients, to work out which one should come first. We will need to update patient information; e.g.

- as the patient moves through a treatment process, we update their record; maybe move them into a different list
- as a patient’s condition becomes more critical, their priority changes and we move them to the front of the list

We want to represent the ‘record’ as a *new type of variable*, change values assigned to it, compare against another, ...

Example: music files

Suppose we are want to write software to manage music files stored in a streaming service.

We need to keep a collection of music track descriptions. Each description contains info like artist, name, genre, bpm, number of times played, and so on. It should point to where the sound file is stored. We need to keep sub-collections of the descriptions in some order (i.e. playlists). At any time, we can ask the track to play (or to pause, or to stop)

We want to represent the ‘description’ as a new type of variable, change values assigned to it, read its info, invoke some behaviour from it (ie play), ...

Example: bank accounts

Suppose we are asked to write software to manage an online bank.

We need to keep a record for each customer. We want to maintain their bank balance, report it when asked, allow the deposit of cash into the account, and withdrawals from the account, allow interest to be applied, change the interest rate, lock the account when theft is suspected, and so on.

We want to represent the ‘account’ as a new type of variable, change values assigned to it, run some processes on it, ...

Classes

In order to allow Python to treat these complex things as variables, we need to describe them to Python, which we do by writing code.

Each type of entity we need will be described as a *class*

- *fields (*i.e. subsidiary variables*)* represent its properties
- *methods* (i.e. functions) specify its behaviour.

A class is a design for objects of a particular type.

class: PatientRecord

- pps
- name
- date_of_birth
- gender
- priority
- opDate
- before(otherpr)

class: MusicFile

- name
- artist
- genre
- bpm
- times_played
- location
- play()
- pause()

class: BankAccount

- number
- balance
- interest_rate
- get_balance()
- deposit()
- withdraw()
- apply_interest()

Objects

We can create individual objects from the class design. This is the same idea as creating individual variables corresponding to a data type.

x =

object: PatientRecord
- pps: 123456
- name: Jane McCarthy
- date_of_birth: 25-05-1953
- gender: female
- priority: 96
- opDate: 26-09-2023

z =

object: MusicFile
- name: In the midnight hour
- artist: Wilson Pickett
- genre: soul
- bpm: 112
- times_played: 58
- location: xrp34kls

v =

object: BankAccount
- number: 345678
- balance: 12,961
- interest_rate: 0.05

y =

object: PatientRecord
- pps: 738273
- name: Jimmy Walsh
- date_of_birth: 16-07-1968
- gender: male
- priority: 72
- opDate: None

w =

object: BankAccount
- number: 345678
- balance: 268
- interest_rate: 0.02

Using objects

We can update the values assigned to them.

Change x’s opDate to 28/09/2023

x =

object: PatientRecord
- pps: 123456
- name: Jane McCarthy
- date_of_birth: 25-05-1953
- gender: female
- priority: 96
- opDate: 26-09-2023

becomes

object: PatientRecord
- pps: 123456
- name: Jane McCarthy
- date_of_birth: 25-05-1953
- gender: female
- priority: 96
- opDate: 28-09-2023

We can run processes on them.
Ask x if it should be scheduled before y
It must compare its priority with y’s priority.

Yes (96 > 72)

object: PatientRecord
- pps: 123456
- name: Jane McCarthy
- date_of_birth: 25-05-1953
- gender: female
- priority: 96
- opDate: 28-09-2023

object: PatientRecord
- pps: 738273
- name: Jimmy Walsh
- date_of_birth: 16-07-1968
- gender: male
- priority: 72
- opDate: None

Using objects

Apply interest to w’s account.

w =

```
object: BankAccount
-    number: 345678
-    balance: 268
-    interest_rate: 0.02
```

w needs to multiply its balance by its interest rate

(268 * 0.02 == 5.36) and then add onto its balance.

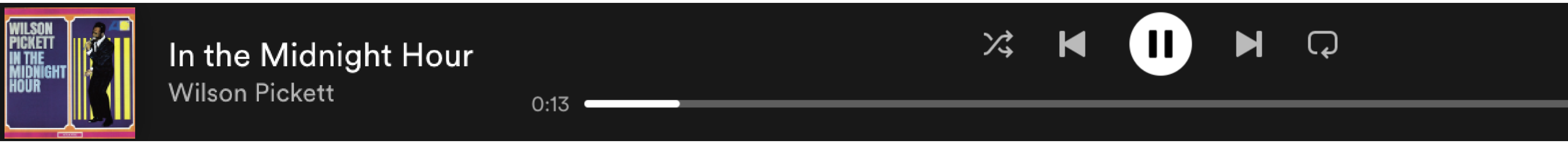
So w becomes

```
object: BankAccount
-    number: 345678
-    balance: 273.36
-    interest_rate: 0.02
```

Using objects

Playing the track linked from z

object: MusicFile
- name: In the midnight hour
- artist: Wilson Pickett
- genre: soul
- bpm: 112
- times_played: 58
- location: xrp34kls



After playing, it changes to

object: MusicFile
- name: In the midnight hour
- artist: Wilson Pickett
- genre: soul
- bpm: 112
- times_played: 59
- location: xrp34kls

Car

class

attributes: variables

fixed at start-up

weight
length
width
num gears
max acceleration
max braking
max lock

will change as the program runs

on?
location
speed
direction
wheel alignment
gear

behaviours:

start engine
accelerate
brake
turn wheel
change gear
stop engine

methods

code we write in the methods specify how the variables will change when the method is called

Objects

To use the classes in a program, we create *instances* or objects from the class.

Each instance gets its own copy of the variables.

Calling a method on the instance allows us to change the values of the variables for that instance.

myBMW

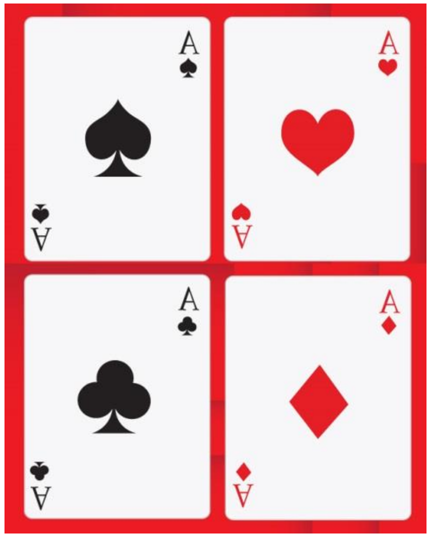
An object from the *car* class

Telling myBMW to accelerate by a certain amount will increase that specific object’s speed and in the game it will move forward more quickly

Example

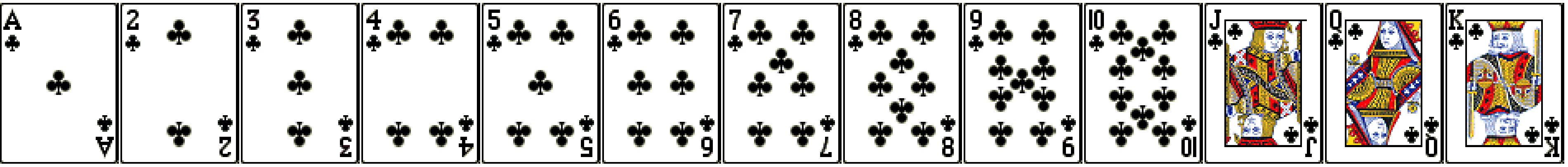
Design and implement the basic elements of card games.

Playing cards primer



Four suits: Spades, Hearts, Diamonds, Clubs

Each suit has 13 cards: Ace, 2, 3, ..., 9, 10, Jack, Queen, King



... which means 52 cards in a deck

```
c1 = Card(3,4)
c2 = Card(8,2)
c3 = Card(8,2)

print(c1.rank)
print(c2.suit)
print(c3)

if c2.is_equal(c3):
    print('Yes')
else:
    print('No')
```

creates a card

```
rank: 3
suit: 4
```

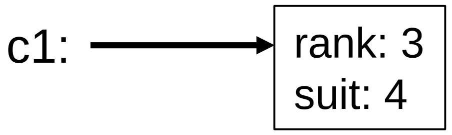
Suppose we have written code to design a Card class in Python. Each card must have a rank and suit. One card can compare itself to another, and say whether or not it represents the same info. We can now write other code that creates and uses Card objects.

and assign it to variable c1

```
c1 = Card(3,4)
c2 = Card(8,2)
c3 = Card(8,2)

print(c1.rank)
print(c2.suit)
print(c3)

if c2.is_equal(c3):
    print('Yes')
else:
    print('No')
```

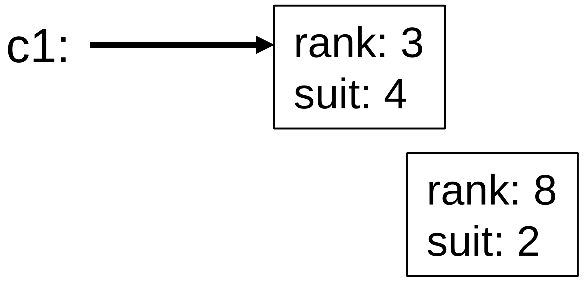


```
c1 = Card(3,4)
c2 = Card(8,2)
c3 = Card(8,2)

print(c1.rank)
print(c2.suit)
print(c3)

if c2.is_equal(c3):
    print('Yes')
else:
    print('No')
```

creates another card

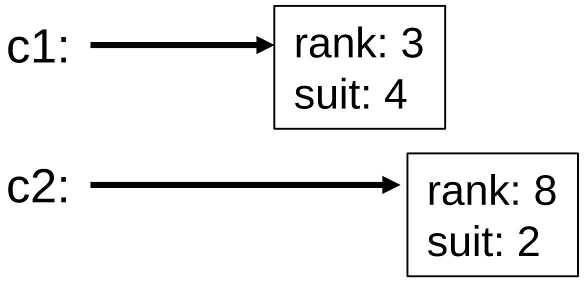


```
c1 = Card(3,4)
c2 = Card(8,2)
c3 = Card(8,2)

print(c1.rank)
print(c2.suit)
print(c3)

if c2.is_equal(c3):
    print('Yes')
else:
    print('No')
```

and assign it to variable c2

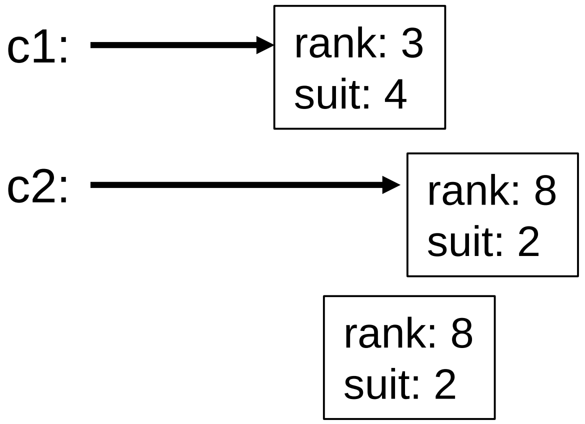


```
c1 = Card(3,4)
c2 = Card(8,2)
c3 = Card(8,2)

print(c1.rank)
print(c2.suit)
print(c3)

if c2.is_equal(c3):
    print('Yes')
else:
    print('No')
```

creates another card

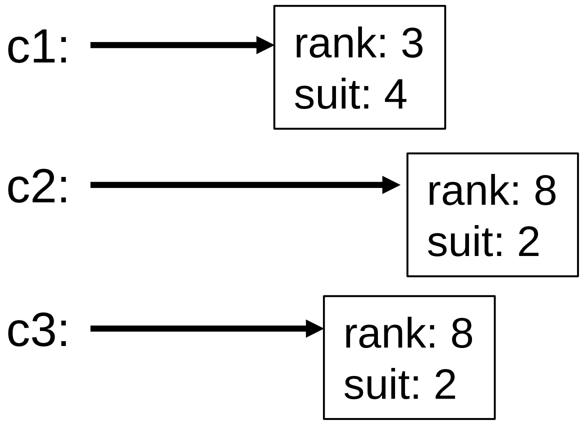


```
c1 = Card(3,4)
c2 = Card(8,2)
c3 = Card(8,2)

print(c1.rank)
print(c2.suit)
print(c3)

if c2.is_equal(c3):
    print('Yes')
else:
    print('No')
```

and assign it to variable c3

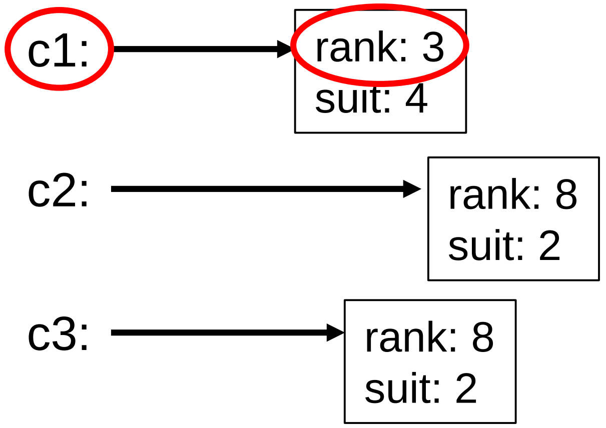


```
c1 = Card(3,4)
c2 = Card(8,2)
c3 = Card(8,2)

print(c1.rank)
print(c2.suit)
print(c3)

if c2.is_equal(c3):
    print('Yes')
else:
    print('No')
```

find the value of ‘rank’ for the object that was assigned to c1, and then print it



OUTPUT:

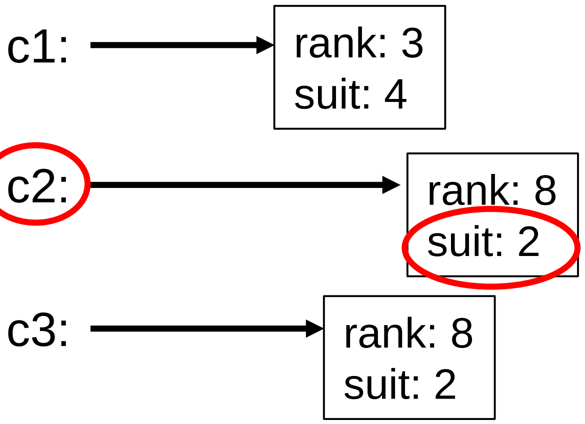
```
3
```

```
c1 = Card(3,4)
c2 = Card(8,2)
c3 = Card(8,2)

print(c1.rank)
print(c2.suit)
print(c3)

if c2.is_equal(c3):
    print('Yes')
else:
    print('No')
```

find the value of ‘suit’ for the object that was assigned to c2, and then print it



OUTPUT:

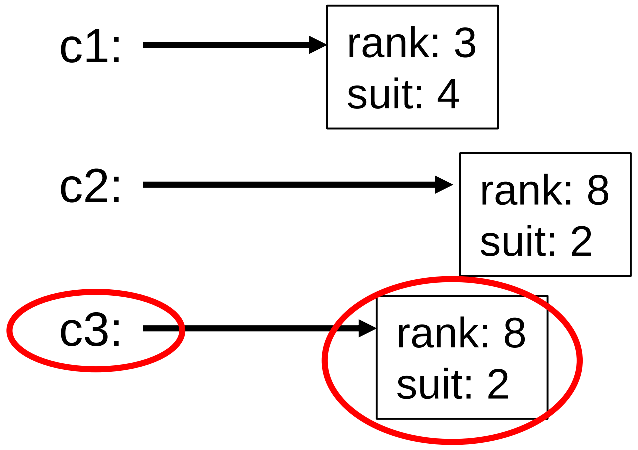
```
3
2
```

print the value assigned to c3

```
c1 = Card(3,4)
c2 = Card(8,2)
c3 = Card(8,2)

print(c1.rank)
print(c2.suit)
print(c3)

if c2.is_equal(c3):
    print('Yes')
else:
    print('No')
```



OUTPUT:

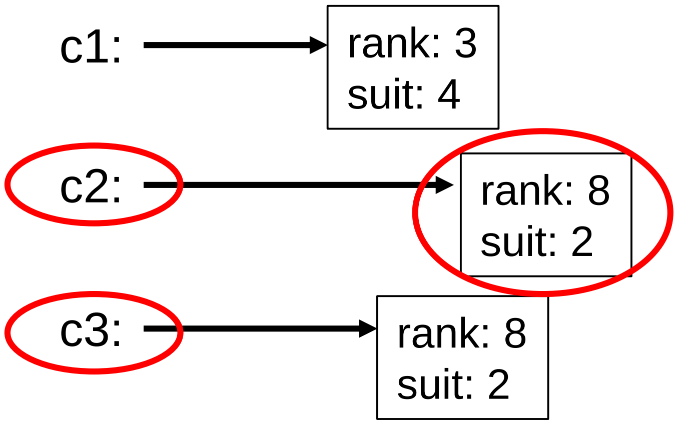
```
3
2
82
```

```
c1 = Card(3,4)
c2 = Card(8,2)
c3 = Card(8,2)

print(c1.rank)
print(c2.suit)
print(c3)

if c2.is_equal(c3):
    print('Yes')
else:
    print('No')
```

tell card c2 to report it is equal to c3; if the result is True, continue with the body of the if statement.



OUTPUT:

```
3
2
82
```

So, what code do we write that allows us to do these actions?

starts class definition – everything indented below this until the indentation stops is part of the definition of the class.

name of class

__init__(self, ...) describes how to initialise an object when it is created

```
class Card:

    def __init__(self, irank, isuit):
        self.rank = irank
        self.suit = isuit

    def __str__(self):
        return '' + str(self.rank) + str(self.suit)

    def is_equal(self, other):
        if self.rank == other.rank and self.suit == other.suit:
            return True
        return False
```

self.x in here creates a variable ‘x’ in the class

Note: double underscores before and after a method name (e.g. : __init__(…)) signify an internal method that Python will call at certain times – don’t use this pattern for other methods

must specify 'self' as input parameter in the method definition if you want to access any variables

__str__(self) specifies how to represent a card object as a string for a user to read

```
class Card:

    def __init__(self, irank, isuit):
        self.rank = irank
        self.suit = isuit

    def __str__(self):
        return '' + str(self.rank) + str(self.suit)

    def is_equal(self, other):
        if self.rank == other.rank and self.suit == other.suit:
            return True
        return False

    def is_higher(self, other):
        if self.rank > other.rank:
            return True
        return False
```

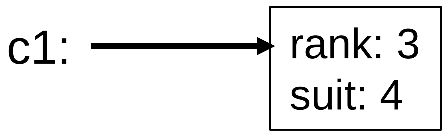
```
c1 = Card(3,4)
c2 = Card(8,2)
c3 = Card(8,2)

print(c1.rank)
print(c2.suit)
print(c3)

if c2.is_equal(c3):
    print('Yes')
else:
    print('No')
```

creates a card

Card(3,4) invokes the __init__(...) method, with input 3,4



```
class Card:

    def __init__(self, irank, isuit):
        self.rank = irank
        self.suit = isuit

    def __str__(self):
        return ’’ + str(self.rank) + str(self.suit)

    def is_equal(self, other):
        if self.rank == other.rank and self.suit == other.suit:
            return True
        return False

    def is_higher(self, other):
        if self.rank > other.rank:
            return True
        return False
```

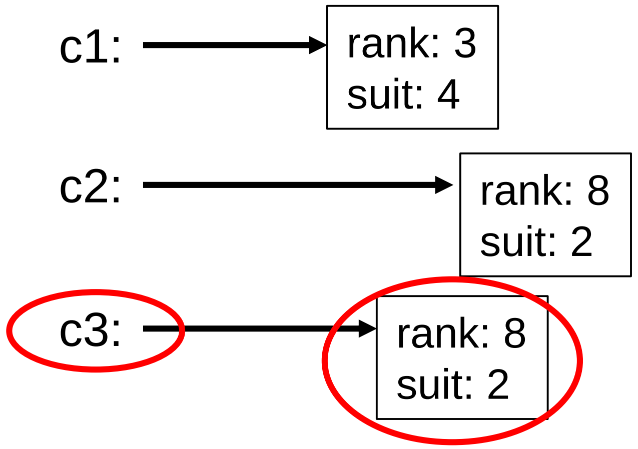
Note: 'self' was specified as an input parameter when we defined the methods, but we do not give it as input when we invoke the method

```
c1 = Card(3,4)
c2 = Card(8,2)
c3 = Card(8,2)

print(c1.rank)
print(c2.suit)
print(c3)

if c2.is_equal(c3):
    print('Yes')
else:
    print('No')
```

print the value assigned to c3
(the value is a Card object, which python has no built in method for, so calls our Card.__str__(...) method and gets a string in return, which it prints)



```
class Card:

    def __init__(self, irank, isuit):
        self.rank = irank
        self.suit = isuit

    def __str__(self):
        return ’’ + str(self.rank) + str(self.suit)

    def is_equal(self, other):
        if self.rank == other.rank and self.suit == other.suit:
            return True
        return False

    def is_higher(self, other):
        if self.rank > other.rank:
            return True
        return False
```

OUTPUT:

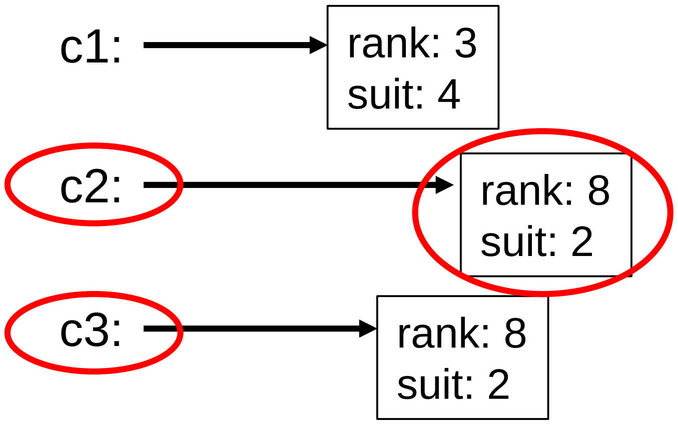
```
3
2
82
```

call the is_equal(...) method of the object assigned to c2, passing in the object assigned to c3; if the result is True, continue with the body of the if statement.

```
c1 = Card(3,4)
c2 = Card(8,2)
c3 = Card(8,2)

print(c1.rank)
print(c2.suit)
print(c3)

if c2.is_equal(c3):
    print('Yes')
else:
    print('No')
```



```
class Card:

    def __init__(self, irank, isuit):
        self.rank = irank
        self.suit = isuit

    def __str__(self):
        return '' + str(self.rank) + str(self.suit)

    def is_equal(self, other):
        if self.rank == other.rank and self.suit == other.suit:
            return True
        return False

    def is_higher(self, other):
        if self.rank > other.rank:
            return True
        return False
```

OUTPUT:

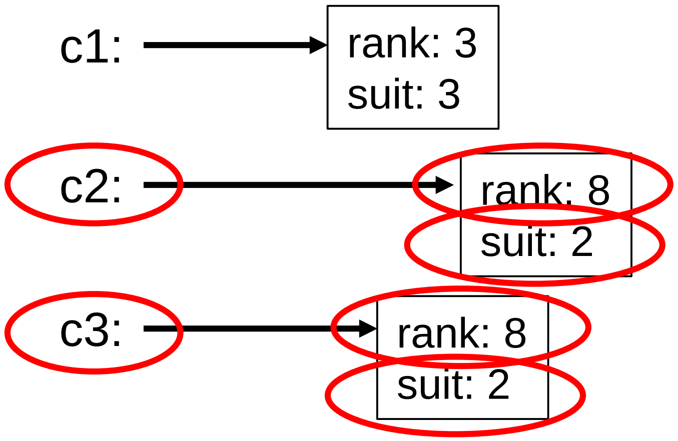
```
3
2
82
```

```
c1 = Card(3,4)
c2 = Card(8,2)
c3 = Card(8,2)

print(c1.rank)
print(c2.suit)
print(c3)

if c2.is_equal(c3):
    print('Yes')
else:
    print('No')
```

call the is_equal(...) method of the object assigned to c3, passing in the object assigned to c4; if the result is True, continue with the body of the if statement.



```
class Card:

    def __init__(self, irank, isuit):
        self.rank = irank
        self.suit = isuit

    def __str__(self):
        return ’’ + str(self.rank) + str(self.suit)

    def is_equal(self, other):
        if self.rank == other.rank and self.suit == other.suit:
            return True
        return False

    def is_higher(self, other):
        if self.rank > other.rank:
            return True
        return False
```

OUTPUT:

```
3
2
82
Yes
```

when running this method this time, self is c2; other is c3

```python
""" Class definition for a simple playing card. """
class Card:

    def __init__(self, rank, suit):
        """ Initialise the card.
            rank should be an integer in {1,2,...,13}
            suit should be an integer in {1,2,3,4}
        """
        self.rank = rank
        self.suit = suit

    def __str__(self):
        return "" + str(self.rank) + str(self.suit)

    def is_equal(self, other):
        """ Determine whether this instance is the same card as another. """
        if self.rank == other.rank and self.suit == other.suit:
            return True
        return False

    def is_higher(self, other):
        """ Determine whether this card is higher than another. """
        if self.rank > other.rank:
            return True
        return False

c1 = Card(3,4)
c2 = Card(8,2)
c3 = Card(8,2)

print(c1.rank)
print(c2.suit)
print(c3)
if c2.is_equal(c3):
    print('Yes')
else:
    print('No')
```

Exercise: Play your cards right

Exercise: Play your cards right

- A player sees a line of hidden cards, with one card face-up at the start. The player must predict whether the next card is higher or lower than the previous face-up card. If they get it wrong, they lose; if they get it right, they move on to the next card. If they predict all cards correctly, they win.
- Implement this using classes and objects in Python. You should have the following classes:
  - Card (as given in these notes) – test for higher or lower (and you decide if Ace is high or low)
  - Board – maintains a line of face-down cards, the current face-up card, plus a history of revealed cards. Methods:
    - Add card to the end of the hidden cards
    - Reveal the next card (and update accordingly)
    - Report number of cards remaining, the current card, and the history

Play your cards right (continued)

- Deck – all standard 52 playing cards. Methods:
  - Shuffle the cards
  - Deal the top card (and remove from the deck)

You will also need a method to play the game.

Overall, you will create 1 Deck object, which refers to 52 Card objects, and one Board object, which refers to some of the same 52 Card objects.

Once you have this working, adapt the game play so that a player has 2 passes – that is, can choose not make a prediction for at most 2 cards in the sequence.

Object Oriented Programming

There is much more to programming with objects than is described here.

Abstraction, encapsulation, inheritance, polymorphism, ...

All will be discussed and presented in CS2513.

For CS2515, we only really need the ideas presented here.

But we will design classes that specify other classes inside them (known as *object composition*)

- e.g. a Deck contains up to 52 Cards

Next lecture ...

Complexity review