Part III- Object Oriented Programming

Overview: Object Oriented Programming

- **Encapsulation**
- **Polymorhism**
- **Inheritence**

Object Oriented Programming (OOP)

- **so far we've been doing:**
  - procedural programming: programmer specifies a procedure, step-by-step instructions, to solve a problem
  - object-based programming: we began to operate on objects

```
a_list.append('Tom')
```

  - above we're invoking append method of a list object a_list
- **objects package data and code to operate on together into a convenient unit that's easy to use**

Object Oriented Programming (OOP)

- **go beyond pre-existing classes and**
- **create new classes of our own**
- **wish Python provided dice and playing_cards?**
- **just add them by defining new classes**
- **With this ability comes a new perspective:**
  - instead of focusing on what needs to be done (procedural)
  - focus on objects in a given problem and how they will interact
- **new jargon: encapsulation, inheritence, polymorhism**

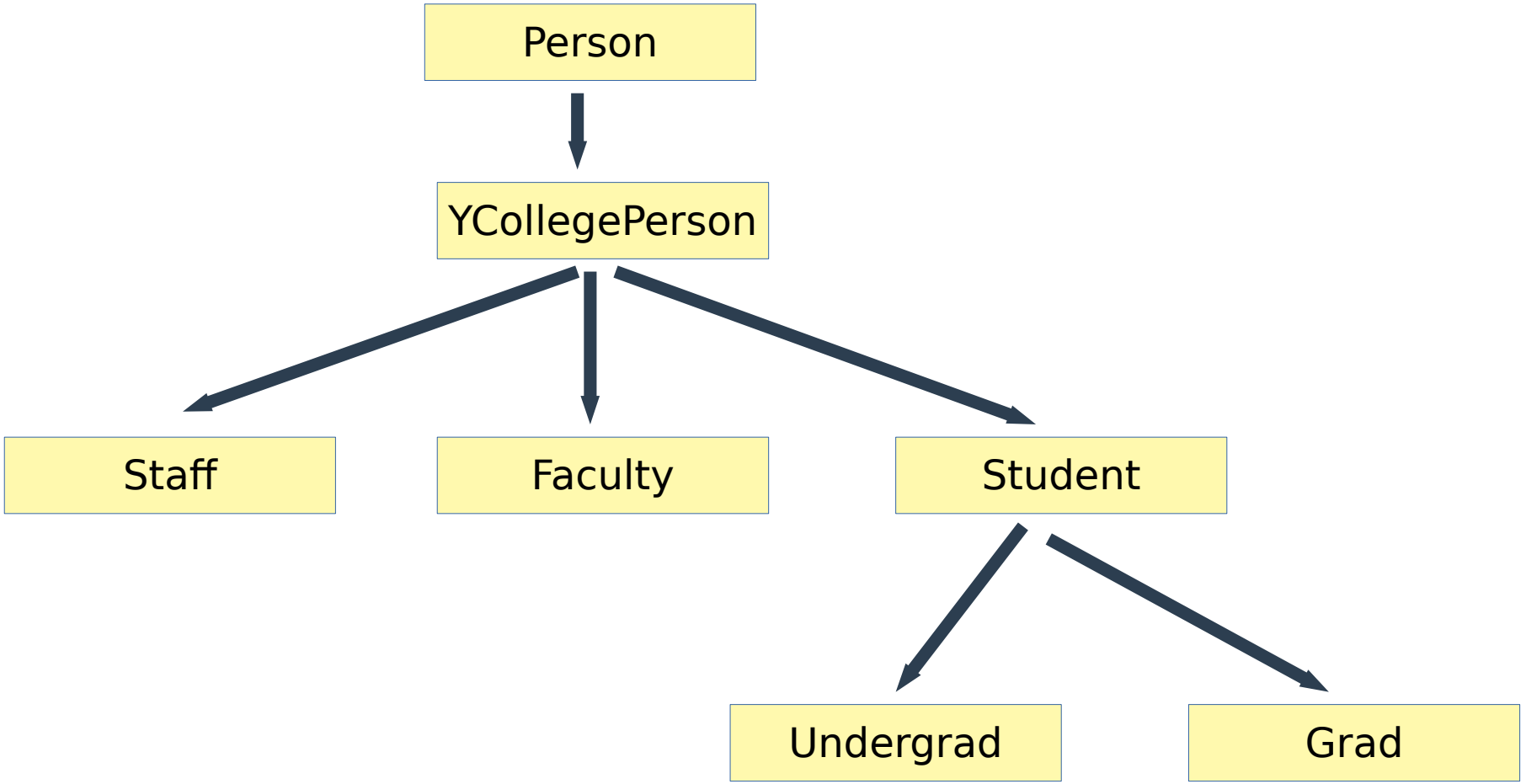
Encapsulation

- **The first benefit of OOP is encapsulation:**
  - bundling together data attributes and methods operating on them
- **A consequence: how an object is implemented under the hood is hidden from the code invoking it**
  - also called “information hiding’
  - we used lists/dictionaries without knowing how they’re implemented
  - a program using an object only needs to know the specification of a method (input parameter etc)
- **complete encapsulation: an object type is re-written and “dropped” into a working program, without other program components needing to be aware of the change**
- **Hence: encapsulation leads to modularity & reusability**

Inheritence

- **Experience of creating objects soon showed that**
  - many objects share similar features, hence
  - existing code can be re-used by one type object inheriting behaviour and properties from another
  - when ancestor object is updated, descendant object instantly inherits such improvements

Inheritence: example



Polymorhism

An operation/request triggers different behaviour in different context

```
x = y + 5
```

addition

```
salutation = 'Mr.' + last_name
```

append/concatenate

```
t = [3, 7] + [6, 2]
```

list expansion

Plan for the next three classes

All these OOP topics will be discussed today + in the following two lectures

Each lecture will be built around a main example

- **playing cards**
- **Hunt the Wumps**
- **Rational Numbers (Fractions)**

A simple example: Die class

```
1 # die_class_0.py
2 import random
3
4 class Die:
5     def __init__(self, n):
6         self.nsides = n
7
8     def roll(self):
9         spots = random.randint(1,self.nsides)
10        return spots
11
12 if __name__ == '__main__':
13    d1 = Die(6)
14    red = Die(20)
15
16    print('Rolling d1 ...', end='')
17    result = d1.roll()
18    print('result =', result)
19
20    print('Rolling red and d1 together gets you:', d1.roll() + red.roll() )
21    print('The die d1 has %d sides' % (d1.nsides) )
```

A simple example: Die class

```
1 # die_class_0.py
2 import random
3
4 class Die:
5     def __init__(self, n):
6         self.nsides = n
7
8     def roll(self):
9         spots = random.randint(1,self.nsides)
10        return spots
11
12 if __name__ == '__main__':
13    d1 = Die(6)
14    red = Die(20)
15
16    print('Rolling d1 ...', end='')
17    result = d1.roll()
18    print('result =', result)
19
20    print('Rolling red and d1 together gets you:', d1.roll() + red.roll() )
21    print('The die d1 has %d sides' % (d1.nsides) )
```

keyword **class** announces definition of a new object type

A simple example: Die class

```
1 # die_class_0.py
2 import random
3
4 class Die:
5     def __init__(self, n):
6         self.nsides = n
7
8     def roll(self):
9         spots = random.randint(1,self.nsides)
10        return spots
11
12 if __name__ == '__main__':
13    d1 = Die(6)
14    red = Die(20)
15
16    print('Rolling d1 ...', end='')
17    result = d1.roll()
18    print('result =', result)
19
20    print('Rolling red and d1 together gets you:', d1.roll() + red.roll() )
21    print('The die d1 has %d sides' % (d1.nsides) )
```

name of this new object type is **Die**
convention: class names are capitilized

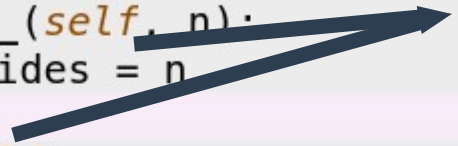
A simple example: Die class

```
1 # die_class_0.py
2 import random
3
4 class Die:
5     def __init__(self, n):
6         self.nsides = n
7
8     def roll(self):
9         spots = random.randint(1,self.nsides)
10        return spots
11
12 if __name__ == '__main__':
13    d1 = Die(6)
14    red = Die(20)
15
16    print('Rolling d1 ...', end='')
17    result = d1.roll()
18    print('result =', result)
19
20    print('Rolling red and d1 together gets you:', d1.roll() + red.roll() )
21    print('The die d1 has %d sides' % (d1.nsides) )
```

__init__ short for initialize
this function is executed whenever
an object of type **Die** is created

A simple example: Die class

```
1 # die_class_0.py
2 import random
3
4 class Die:
5     def __init__(self, n):
6         self.nsides = n
7
8     def roll(self):
9         spots = random.randint(1,self.nsides)
10        return spots
11
12 if __name__ == '__main__':
13    d1 = Die(6)
14    red = Die(20)
15
16    print('Rolling d1 ...', end='')
17    result = d1.roll()
18    print('result =', result)
19
20    print('Rolling red and d1 together gets you:', d1.roll() + red.roll() )
21    print('The die d1 has %d sides' % (d1.nsides) )
```



All class methods should use **self** as the first parameter. It provides access to the current object

A simple example: Die class

```
1 # die_class_0.py
2 import random
3
4 class Die:
5     def __init__(self, n):
6         self.nsides = n
7
8     def roll(self):
9         spots = random.randint(1,self.nsides)
10        return spots
11
12 if __name__ == '__main__':
13     d1 = Die(6)
14     red = Die(20)
15
16     print('Rolling d1 ...', end='')
17     result = d1.roll()
18     print('result =', result)
19
20     print('Rolling red and d1 together gets you:', d1.roll() + red.roll() )
21     print('The die d1 has %d sides' % (d1.nsides) )
```

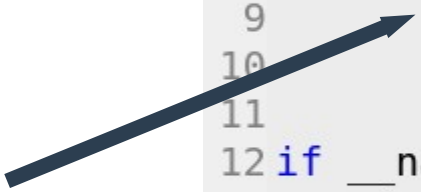
class **attribute**

Each **Die** object will have an attribute nsides!

A simple example: Die class

class **method**

Each **Die** object can be rolled!



```
1 # die_class_0.py
2 import random
3
4 class Die:
5     def __init__(self, n):
6         self.nsides = n
7
8     def roll(self):
9         spots = random.randint(1,self.nsides)
10        return spots
11
12 if __name__ == '__main__':
13     d1 = Die(6)
14     red = Die(20)
15
16     print('Rolling d1 ...', end='')
17     result = d1.roll()
18     print('result =', result)
19
20     print('Rolling red and d1 together gets y
21     print('The die d1 has %d sides' % (d1.nsi
```

A simple example: Die class

```
1 # die_class_0.py
2 import random
3
4 class Die:
5     def __init__(self, n):
6         self.nsides = n
7
8     def roll(self):
9         spots = random.randint(1,self.nsides)
10        return spots
11
12 if __name__ == '__main__':
13    d1 = Die(6)
14    red = Die(20)
15
16    print('Rolling d1 ...', end='')
17    result = d1.roll()
18    print('result =', result)
19
20    print('Rolling red and d1 together gets you:', d1.roll() + red.roll() )
21    print('The die d1 has %d sides' % (d1.nsides) )
```

Object constructors: each invokes **Die**'s __init__ method. Note: object constructor name is the same as the class name

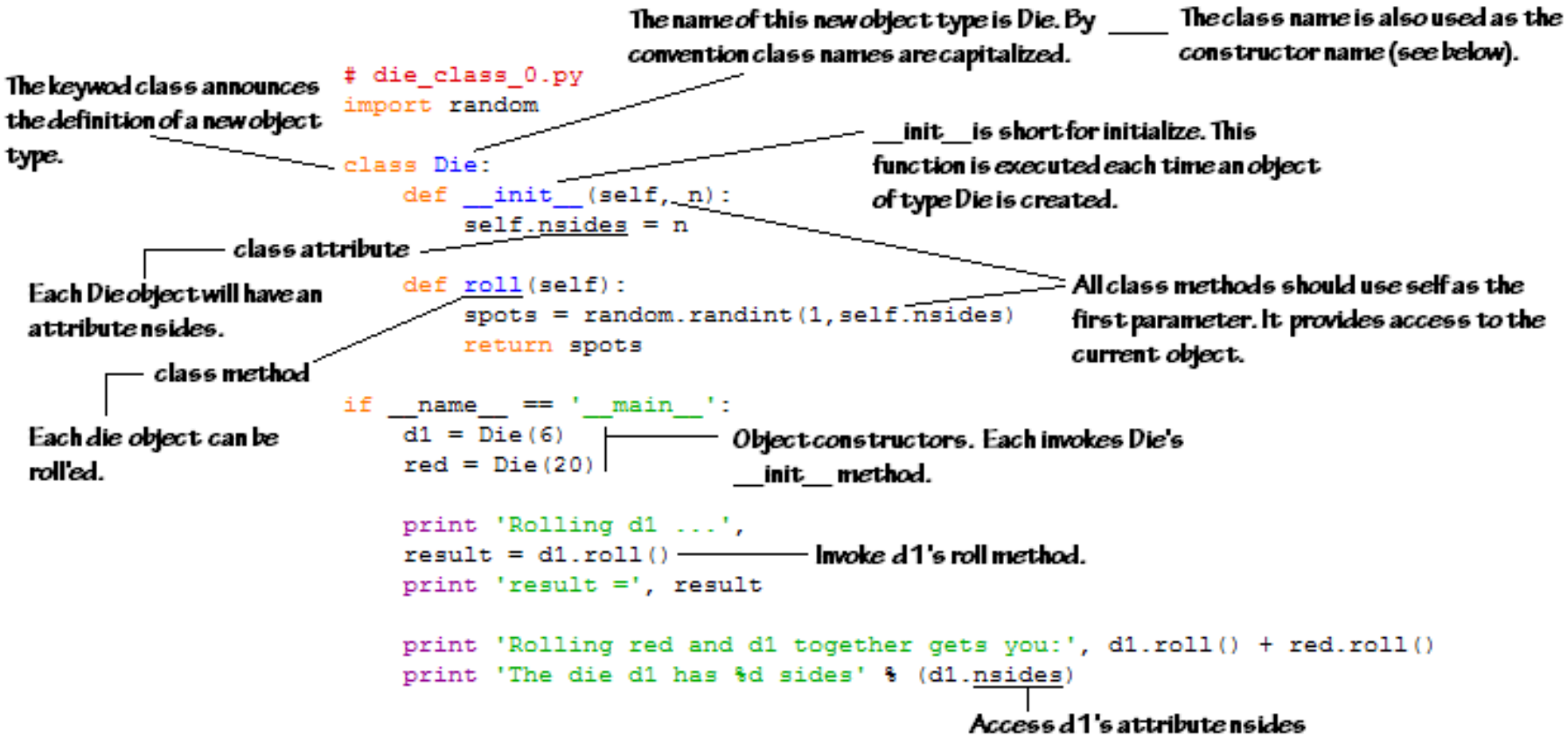
A simple example: Die class

```
 5     def __init__(self, n):
 6         self.nsides = n
 7
 8     def roll(self):
 9         spots = random.randint(1,self.nsides)
10         return spots
11
12 if __name__ == '__main__':
13     d1 = Die(6)
14     red = Die(20)
15
16     print('Rolling d1 ...', end='')
17     result = d1.roll()
18     print('result =', result)
19
20     print('Rolling red and d1 together gets you:', d1.roll() + red.roll() )
21     print('The die d1 has %d sides' % (d1.nsides) )
```

invoke d1's roll method

access d1's attribute nsides

Simple example: die class




```
# die_class_0.py
import random

class Die:
    def __init__(self, n):
        self.nsides = n

    def roll(self):
        spots = random.randint(1,self.nsides)
        return spots

if __name__ == '__main__':
    d1 = Die(6)
    red = Die(20)

    print 'Rolling d1 ...',
    result = d1.roll()
    print 'result =', result

    print 'Rolling red and d1 together gets you:', d1.roll() + red.roll()
    print 'The die d1 has %d sides' % (d1.nsides)
```

Python OOP syntax

```
class ClassName:
    def __init__(self, ...):
        ...

    def method1(self, ...):
        ...
    def method2(self, ...):
        ...
```

A simple programming practice: Person object

- Define person class
- with attributes name & birthdate
- with method “getage”
  - use: **datetime** module
- after **if __name__ = ‘__main__’:**
  - create two person objects:
  - p1 = Person(fullname, birthdate)
  - p2 = Person(fullname, birthdate)
  - see id(p1) not eq id(p2)
  - get their respective ages

```
1 # die_class_0.py
2 import random
3
4 class Die:
5     def __init__(self, n):
6         self.nsides = n
7
8     def roll(self):
9         spots = random.randint(1,self.nsides)
10        return spots
11
```

```
In [111]: import datetime as dt

In [112]: now = dt.datetime.now()

In [113]: year = now.year

In [114]: month = now.month

In [115]: day = now.day

In [116]: [year, month, day]
Out[116]: [2019, 6, 11]
```

```
In [165]: from person import *

In [166]: p1 = Person("Clark Kent",'1990/1/2')

In [167]: [p1.fullname, p1.birthdate, p1.getage()]
Out[167]: ['Clark Kent', '1990/1/2', '29 years 5 months 9days']

In [168]: p2 = Person("Albert Einstein",'1890/1/2')

In [169]: [id(p1), id(p2)]
Out[169]: [140518267866808, 140518267866752]
```

An extended example: Playing card classes

- Earlier we developed a series of functions to help write programs involving playing cards.

- Now we'll take the next step and make our code object-oriented.

- How shall we go about designing the class(es)?

There are many possible answers to this question, but for a small domain like this one (as opposed to creating an OS for example) a good approach is wish fulfillment.

- we specify the kind of code we wish we could write

Playing card classes: The specification

We are wishing to be able to do the followings:

```
Get a deck of cards
Shuffle the deck
Display the deck
Create a hand of cards for roxx
Create a hand of cards for chris
Deal five cards to roxx
Display roxx's cards
Deal five cards to chris
Display chris' cards
How many cards are left in the deck?
If roxx has a flush
    congratulate him
```

Playing card classes: The specification

Using OOP, the corresponding Python code will be as easy to read

```
d = Deck()
d.shuffle()
print('d after shuffling =', d)
print('d has', d.cards_left(), 'cards')
roxx = Hand()
chris = Hand()
for card in range(5):
    roxx.add(d.deal())
print('Your hand of', roxx.size(), 'cards contains:', roxx)
chris.add(d.deal(5))
print('Your hand of', chris.size(), 'cards contains:', chris)
print('There are', d.cards_left(), 'cards left in the deck.')
if roxx.is_flush():
    print('roxx rocks!')
```

This code will serve as our specification.
We'll know we're done when we've added sufficient code above this that it runs correctly.

Identifying the necessary classes

- We can learn a lot about what code we have to write by carefully examining our specification code.
- First let's look for calls to constructors.
- A constructor uses a class name as a function.
  - We've seen examples of this with Python's built-in classes, e.g

```
>>> num = int(43.72)
>>> num
43
>>> lst = list('Tim')
>>> lst
['T', 'i', 'm']
>>>
```

Identifying the necessary classes

- Now we want to look for similar syntax in our specification code,

```
d = Deck()
d.shuffle()
print('d after shuffling =', d)
print('d has', d.cards_left(), 'cards')
roxx = Hand()
chris = Hand()
for card in range(5):
    roxx.add(d.deal())
print('Your hand of', roxx.size(), 'cards contains:', roxx)
chris.add(d.deal(5))
print('Your hand of', chris.size(), 'cards contains:', chris)
print('There are', d.cards_left(), 'cards left in the deck.')
if roxx.is_flush():
    print('roxx rocks!')
```

Three calls to two different constructors means our code requires two classes for sure: **Hand** and **Deck**

Identifying the classes' methods

```
d = Deck()
d.shuffle()
print('d after shuffling =', d)
print('d has', d.cards_left(), 'cards')
roxx = Hand()
chris = Hand()
for card in range(5):
    roxx.add(d.deal())
print('Your hand of', roxx.size(), 'cards contains:', roxx)
chris.add(d.deal(5))
print('Your hand of', chris.size(), 'cards contains:', chris)
print('There are', d.cards_left(), 'cards left in the deck.')
if roxx.is_flush():
    print('roxx rocks!')
```

Deck class must provide:

- shuffle()
- cards_left()
- deal()
- deal() should default to 1 card if no parameter is given

Hand class should provide

- add()
- size()
- is_flush() etc

Program skeleton 1

Notes:

We're using **pass** statement as placedholder

__init__ is required for each class:
it will contain code necessary to construct objects of the type specified by the class

```
1  # playing_cards_skeleton_1.py
2  class Deck:
3      def __init__(self):
4          pass
5      def shuffle(self):
6          pass
7      def cards_left(self):
8          pass
9      def deal(self, n=1):
10         pass
11
12 class Hand:
13     def __init__(self):
14         pass
15     def add(self, cards):
16         pass
17     def size(self):
18         pass
19     def is_flush(self):
20         pass
21
22 d = Deck()
23 d.shuffle()
24 print('d after shuffling =', d)
25 print('d has', d.cards_left(), 'cards')
26 roxx = Hand()
27 chris = Hand()
28 for card in range(5):
29     roxx.add(d.deal())
30 print('Your hand of', roxx.size(), 'cards contains:', roxx)
31 chris.add(d.deal(5))
32 print('Your hand of', chris.size(), 'cards contains:', chris)
33 print('There are', d.cards_left(), 'cards left in the deck.')
34 if roxx.is_flush():
35     print('roxx rocks!')
```

An invisible method

```
In [102]: runfile('/home/sbulut/github/cpsc128/code/python3/
playing_cards_skeleton1.py', wdir='/home/sbulut/github/cpsc128/code/python3')
d after shuffling = <__main__.Deck object at 0x7f94e0442400>
d has None cards
Your hand of None cards contains: <__main__.Hand object at 0x7f94e0442860>
Your hand of None cards contains: <__main__.Hand object at 0x7f94e0442a90>
There are None cards left in the deck.

In [103]:
```

Program Skeleton 2

Note:
we can't use **pass** as placeholder in __str__
since the method is required to return a string

```
In [103]: runfile('/home/sbulut/github/cps
playing_cards_skeleton2.py', wdir='/home/s
d after shuffling =
d has None cards
Your hand of None cards contains:
Your hand of None cards contains:
There are None cards left in the deck.

In [104]:
```

```
1  # playing_cards_skeleton_2.py
2  class Deck:
3      def __init__(self):
4          pass
5      def shuffle(self):
6          pass
7      def cards_left(self):
8          pass
9      def deal(self, n=1):
10         pass
11     def __str__(self):
12         return ''
13
14 class Hand:
15     def __init__(self):
16         pass
17     def add(self, cards):
18         pass
19     def size(self):
20         pass
21     def is_flush(self):
22         pass
23     def __str__(self):
24         return ''
25
26 d = Deck()
27 d.shuffle()
28 print('d after shuffling =', d)
29 print('d has', d.cards_left(), 'cards')
30 roxx = Hand()
31 chris = Hand()
32 for card in range(5):
33     roxx.add(d.deal())
34 print('Your hand of', roxx.size(), 'cards contains:', roxx)
35 chris.add(d.deal(5))
36 print('Your hand of', chris.size(), 'cards contains:', chris)
37 print('There are', d.cards_left(), 'cards left in the deck.')
38 if roxx.is_flush():
39     print('roxx rocks!')
```

A new class: Card

- as far as our specs go, we have done well so far
- but have to realize that Deck and Hand are both a collection of cards
- instead of both classes having duplicate code for manipulating and/or displaying cards
- we can create a new object/class called “Card”

Program Skeleton 3

```
# playing_cards_skeleton_3.py
class Card:
    pass #not sure what to put here yet

class Deck:
    def __init__(self):
        self.cards = []
        for cardnum in range(52):
            self.cards.append(Card(cardnum))
    def shuffle(self):
        pass
    def cards_left(self):
        pass
    def deal(self, n=1):
        pass
    def __str__(self):
        return '' #can't use pass here

class Hand:
    def __init__(self):
        pass
    def add(self, cards):
        pass
    def size(self):
        pass
    def is_flush(self):
        pass
    def __str__(self):
        return '' #can't use pass here

d = Deck()
d.shuffle()
print('d after shuffling =', d)
print('d has', d.cards_left(), 'cards')
roxx = Hand()
chris = Hand()
for card in range(5):
    roxx.add(d.deal())
print('Your hand of', roxx.size(), 'cards contains:', roxx)
chris.add(d.deal(5))
print('Your hand of', chris.size(), 'cards contains:', chris)
print('There are', d.cards_left(), 'cards left in the deck.')
if roxx.is_flush():
    print('roxx rocks!')

```

Deck methods

```
20 class Deck:
21     def __init__(self):
22         self.cards = []
23         for cardnum in range(52):
24             self.cards.append(Card(cardnum))
25 
26     def shuffle(self):
27         #Knuth's swap shuffle: random swap from ordered part to shuffled one
28         ncards = len(self.cards)
29         for swaps in range(ncards):
30             #loop from ncards-1 to 0 instead of 0 to ncards-1
31             swaps = ncards -1 - swaps
32             posn1 = random.randint(0, swaps)
33             self.cards[posn1], self.cards[swaps] = self.cards[swaps], self.cards[posn1]
34 
35     def cards_left(self):
36         return len(self.cards)
37 
38     def deal(self, n=1):
39         c = self.cards[-n:]
40         del(self.cards[-n:])
41         return c
42 
43     def __str__(self):
44         s = ''
45         for card in self.cards:
46             s = s + str(card) + ', '
47         return s
```

- A natural choice for Deck is a list
- Initialize a deck with 52 cards

Hand Methods

```
49 class Hand:
50     def __init__(self):
51         self.cards = []
52
53     def add(self, cards):
54         self.cards.extend(cards)
55
56     def size(self):
57         return len(self.cards)
58
59     def is_flush(self):
60         pass
61
62     def __str__(self):
63         s = ''
64         for card in self.cards:
65             s = s + str(card) + ', '
66         return s
67
```

Card methods

```
 4 class Card:
 5     FACE_VALUES = ['A','2','3','4','5','6','7','8','9','T','J','Q','K']
 6     SUITS = ['Clubs','Diamonds','Hearts','Spades']
 7
 8     def __init__(self, cardnum):
 9         self.number = cardnum
10
11     def __str__(self):
12         return self.face_value() + ' of ' + self.suit()
13
14     def face_value(self):
15         return Card.FACE_VALUES[self.number % 13]
16
17     def suit(self):
18         return Card.SUITS[self.number / 13]
19
```

Putting it all together [demo]

```python
# playing_cards_4_v2.py
import random

class Card:
    FACE_VALUES = ['A','2','3','4','5','6','7','8','9','T','J','Q','K']
    SUITS = ['Clubs','Diamonds','Hearts','Spades']

    def __init__(self, cardnum):
        self.number = cardnum

    def __str__(self):
        return self.face_value() + ' of ' + self.suit()

    def face_value(self):
        return Card.FACE_VALUES[self.number % 13]

    def suit(self):
        return Card.SUITS[self.number // 13]

class Deck:
    def __init__(self):
        self.cards = []
        for cardnum in range(52):
            self.cards.append(Card(cardnum))

    def shuffle(self):
        #Knuth's swap shuffle: random swap from ordered part to shuffled one
        ncards = len(self.cards)
        for swaps in range(ncards):
            #loop from ncards-1 to 0 instead of 0 to ncards-1
            swaps = ncards -1 - swaps
            posn1 = random.randint(0, swaps)
            self.cards[posn1], self.cards[swaps] = self.cards[swaps], self.cards[posn1]

    def cards_left(self):
        return len(self.cards)

    def deal(self, n=1):
        c = self.cards[-n:]
        del(self.cards[-n:])
        return c

    def __str__(self):
        s = ''
        for card in self.cards:
            s = s + str(card) + ', '
        return s

```

```python
class Hand:
    def __init__(self):
        self.cards = []

    def add(self, cards):
        self.cards.extend(cards)

    def size(self):
        return len(self.cards)

    def is_flush(self):
        pass

    def __str__(self):
        s = ''
        for card in self.cards:
            s = s + str(card) + ', '
        return s

if __name__ == '__main__':
    d = Deck()
    d.shuffle()
    print('d after shuffling =', d, '\n')
    print('d has', d.cards_left(), 'cards.\n')

    roxx = Hand()
    chris = Hand()

    for card in range(5):
        roxx.add(d.deal())
    print('Your hand of', roxx.size(), 'cards contains:', roxx)

    chris.add(d.deal(5))
    print('Your hand of', chris.size(), 'cards contains:', chris)

    print('\nThere are', d.cards_left(), 'cards left in the deck.')
    if roxx.is_flush():
        print('roxx rocks!')
```

Running the whole thing

```
In [3]: runfile('/home/sbulut/github/cpsc128/code/python3/playing_cards_4_ttoper_v2.py', wdir='/home/sbulut/
github/cpsc128/code/python3')
d after shuffling = 4 of Clubs, 3 of Spades, 2 of Hearts, 2 of Spades, 5 of Hearts, 7 of Diamonds, 9 of
Spades, T of Diamonds, 6 of Spades, K of Clubs, 7 of Hearts, T of Hearts, T of Spades, K of Hearts, 2 of
Diamonds, 5 of Clubs, 8 of Hearts, 3 of Diamonds, J of Hearts, J of Spades, 8 of Clubs, 3 of Hearts, 8 of
Spades, 6 of Clubs, Q of Clubs, 6 of Diamonds, J of Diamonds, 7 of Clubs, 2 of Clubs, Q of Diamonds, 9 of
Diamonds, 3 of Clubs, 5 of Diamonds, A of Clubs, 4 of Spades, 8 of Diamonds, 6 of Hearts, Q of Spades, 7 of
Spades, A of Hearts, K of Diamonds, 9 of Clubs, A of Spades, T of Clubs, K of Spades, 4 of Hearts, J of
Clubs, 9 of Hearts, 5 of Spades, 4 of Diamonds, A of Diamonds, Q of Hearts,

d has 52 cards.

Your hand of 5 cards contains: Q of Hearts, A of Diamonds, 4 of Diamonds, 5 of Spades, 9 of Hearts,
Your hand of 5 cards contains: A of Spades, T of Clubs, K of Spades, 4 of Hearts, J of Clubs,

There are 42 cards left in the deck.

In [4]:
```

Programming observation

Note how we worked backwards

- from the desired code use

- to required classes

As we started implementing the specs,

we've realized the limitations of specs and new requirements