# June 18: Polymorphism & Inheritence

# Inheritence & Polymorhism

**- three key conceps in OOP are:**

- Encapsulation
- inheritence
- polymorhism

**- so far we've focused on <u>encapsulation</u>**

**- inheritence enables one class to inherit properties (attributes, methods) of another one**

**- polymorhism: ability of same operation to trigger different behaviours in different contexts**

# Inheritance example: Playing Cards

**- Recall <u>Deck</u> and <u>Hand</u> playing card classes**

**- Deck and Hand are obviously different**

**- What about similarities:**

- both are collections of the same type of object: playing cards
- both need to be displayed
- at times we need to remove cards from both of them
- at other times we need to add cards to them
- Finally, we query how many cards are in them

**- OOP recommends:**

**(i) removing common functionality from both Deck and Hand to a <u>seperate base class</u>,**

**<u>(ii)</u> and Deck and Hand inherit functionality from it**

**- a consequence: when improvements made to the base class, they will automatically propagate to the descendants, Deck and Hand, which is efficient**

# class CardCollection

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

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

    def add(self, card):
        self.cards.append(card)

    def remove(self):
        return self.cards.pop()

    def __str__(self):
        return ', '.join( str(card) for card in self.cards )
```

everything here should be familiar: see previous version of Deck and Hand classes

# class Deck

This example shows quite a few things we can do with inheritance

```python
class Deck(CardCollection):
    # Override ancestor's constructor, i.e. replace the default.
    def __init__(self):
        self.cards = []
        for cardnum in range(52):
            self.add( Card(cardnum) )

    # Alias the inherited method "size" as "cards_left",
    # because we usually ask how many cards are left in a
    # deck rather than asking about its size.
    def cards_left(self):
        return self.size()

    # Another alias. When using a deck of cards we talk about "dealing"
    # cards not "removing" them from the deck.
    def deal(self):
        return self.remove()

    # Add a new method, shuffle, that does not exist in ancestor class.
    def shuffle(self):
        ncards = len(self.cards)
        for swaps in range(ncards):
            swaps = ncards -1 - swaps
            posn1 = random.randint(0, swaps)
            self.cards[posn1], self.cards[swaps] = self.cards[swaps], self.cards[posn1]
```

# class Hand

The code for class Hand may surprise you:

```
class Hand(CardCollection):
    pass
```

**Hand** objects:

- don’t have any extra methods that **CardCollection**s do not
- nor do they customize any **CardCollection** methods

Effectively with the code above, **Hand** is an alias for **CardCollection**

# putting the pieces together

```python
# playing_cards_5.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 CardCollection:
    def __init__(self):
        self.cards = []

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

    def add(self, card):
        self.cards.append(card)

    def remove(self):
        return self.cards.pop()

    def __str__(self):
        return ', '.join( str(card) for card in self.cards )
```

```python
class Deck(CardCollection):
    # Override ancestor's constructor, i.e. replace the default.
    def __init__(self):
        self.cards = []
        for cardnum in range(52):
            self.add( Card(cardnum) )

    # Alias the inherited method "size" as "cards_left",
    # because we usually ask how many cards are left in a
    # deck rather than asking about its size.
    def cards_left(self):
        return self.size()

    # Another alias. When using a deck of cards we talk about "dealing"
    # cards not "removing" them from the deck.
    def deal(self):
        return self.remove()

    # Add a new method, shuffle, that does not exist in ancestor class.
    def shuffle(self):
        ncards = len(self.cards)
        for swaps in range(ncards):
            swaps = ncards -1 - swaps
            posn1 = random.randint(0, swaps)
            self.cards[posn1], self.cards[swaps] = self.cards[swaps], self.cards[posn1]


class Hand(CardCollection):
    pass

```

[demo]

# Polymorphism

**- Polymorphism allows same syntax to be used with objects of different types**

**- Equivalently, the meaning of a given method or operator depends on the type of object applied**

```
x = y + 5
salutation = 'Mr.' + last_name
t = [3, 7] + [6, 2]
```

- Can we make our custom object types trigger appropriate behaviour?
- Can we give our objects same status and abilities as Python’s built-in types?

# Polymorphism

All we need to do is to define some special methods!

When Python encounters an operator in an expression:

- it identifies the objects around it (for binary op.)
- then calls a method from their class definitions (if provided)

Example: When Python encounters ‘+’,

- it checks if object on the left hand side has **__add__()**
- if so, it is invoked

Example:

- **print** looks for **__str__()** method to invoke
- square brackets for list indexing, x[2], look for **__getitem__()**

# Polymorphism

- Python provides about a dozen of these special methods for operations
- The more of them we define, the more our custom type will seem like
  a built-in type
- a complete list is given at:
  https://docs.python.org/3/reference/datamodel.html#specialnames

- today we will focus on a few of these to get an idea about how they work

# Example: Fractions

- Suppose we are working on a suite of programs for carpenters who want to work with fractions, e.g 3/4 of an inch, 5/16 of an inch and so on.
- Not surprisingly our design of this suite calls for a Fraction class.
- We don't want to just handle the fractions common in carpentry we want a general Fraction class we will be able to reuse in many settings.
- We'd like to be able to do things like:

```
>>> d1 = Fraction(2,5)
>>> print(d1)
2/5
>>> d2 = Fraction(4)
>>> print(d2)
4/1
>>> d3 = d1 + d2
>>> print(d3)
22/5
```

```
>>> if d1 <= d2:
        print('d1 is less than or equal to d2')
else:
        print('d2 is greater than d1')
d2 is greater than d1
>>> print( d1[0], d1[1] )
2 5
>>> print( d1 + 2 )
12/5
>>> print( d1 - 2 )
-8/5
>>>
```

# Example: Fractions
# Reverse engineering specs

```
>>> d1 = Fraction(2,5)
>>> print(d1)
2/5
>>> d2 = Fraction(4)
>>> print(d2)
4/1
>>> d3 = d1 + d2
>>> print(d3)
22/5
```

- object constructor takes two arguments:
  numerator denominator
- it defines an __str__ method
- constructor can hand a single input:
  second input defaults to 1
- it defines an __add__ method

```
>>> if d1 <= d2:
        print('d1 is less th
else:
        print('d2 is greater
d2 is greater than d1
>>> print( d1[0], d1[1] )
2 5
>>> print( d1 + 2 )
12/5
>>> print( d1 - 2 )
-8/5
>>>
```

- it defines an __le__ method
- it defines __getitem__ method
- __add__method works when second object it **int**
- it defines __sub__ method
- works with **int** type

# class Fraction

fraction.py

```
class Fraction:
    def __init__(self, n, d = 1):
        self.num = n # numerator
        self.den = d # denominator

    def __str__(self):
        return str(self.num) + '/' + str(self.den)

    def __add__(self, other):
        if isinstance(other, Fraction):
            bottom = self.den * other.den
            top = (self.num * other.den) + (other.num * self.den)
            return Fraction(top, bottom)
        elif isinstance(other, int):
            other = Fraction(other)
            bottom = self.den * other.den
            top = (self.num * other.den) + (other.num * self.den)
            return Fraction(top, bottom)

```

__str__ has to return str type:
so use str() to convert int to str

__add__ checks the type of thing being added to using **isinstance**
it converts other from **int** to **Fraction** when necessary

# class Fraction

```
    def __le__(self, other):
        return self.num*other.den <= other.den*self.num

    def __getitem__(self, key):
        if key == 0:
            return self.num
        elif key == 1:
            return self.den

    def __sub__(self, other):
        if isinstance(other, int):
            other = Fraction(other)
        if isinstance(other, Fraction):
            bottom = self.den * other.den
            top = (self.num * other.den) - (other.num * self.den)
            return Fraction(top, bottom)
```

a simpler way to handle **int**
compare with __add__

__getitem__ is the special method called when the interpreter encounters []s, i.e. when it encounters the expression obj[key] it calls obj.__getitem__(key).

# Programming practice: Vector class

pull git repo, & complete this task via following the example in fraction.py

**v1 = Vector(xlen,ylen)**

**print(v1)**

yields “xlen i + ylen j” or “(xlen,ylen)”, requires __str__ method

**v1.norm or .norm() should yield**

sqrt(xlen**2 + ylen**2)

**v1 <= v2 True if v1.norm < v2.norm**

requires __le__ method

**v1 + v2**

requires __add__ method

**2*v1**

yields xlen -> 2*xlen, ylen -> 2*ylen

requires __mul__ method

Extra: define __neg__ which does

v1 -> -v1:
xlen -> -xlen, ylen -> -ylen

Extra: define __sub__

# Vector class: solution

vector.py

```python
from math import sqrt

class Vector:
    def __init__(self,xlen,ylen):
        self.x = xlen
        self.y = ylen
        self.norm = sqrt(self.x**2 + self.y**2)

    def __str__(self):
        return str(self.x) + 'i + ' + str(self.y) + 'j'

    def __add__(self,other):
        if isinstance(other,Vector):
            return Vector(self.x + other.x, self.y + other.y)

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

    def __mul__(self,other):
        if isinstance(other,int):
            return Vector(other*self.x, other*self.y)

if __name__ == '__main__':
    v1 = Vector(1,2)
    print("v1 is: ", v1)
    print("v1.norm is: ", v1.norm)
    v2 = Vector(1,3)
    print("v2 is: ",v2)
    print("is v1 <= v2?: ", v1 <= v2)
    print("v1+v2 is: ", v1+v2)
    print("v1*2 is ", v1*2)
```

# Example: playing cards

- Our first example of polymorphism dealt with Fractions, an immutable type like other numbers and strings.
- There are some subtle issues that arise when dealing with mutable objects like container types.
- Our card classes provide a familiar container type for us to work with

# Example: playing cards

We would like to be able to do:

```
cc = CardCollection()
card = Card(42)
cc = cc + card
```

As before we define __add__

```
class CardCollection:
    ...
    def __add__(self, other):
        self.cards.append(other)
        return self
```

Why do we need a **return** statement above?
self.cards is mutable, so why bother?

# Example: playing cards

We would like to be able to do:

```
cc = CardCollection()
card = Card(42)
cc = cc + card
```

As before we define __add__

```
class CardCollection:
    ...
    def __add__(self, other):
        self.cards.append(other)
        return self
```

Why do we need a **return** statement above?
self.cards is mutable, so why bother?

This is because assignment operator above is expecting something on RHS.
If __add__ doesn't have **return** statement, it will return **None** by default.
Hence, cc is assigned **None** value, and is destroyed.

# Example: playing cards

Thanks to inheritance, we can use + on **Hand** and **Deck** objects

```
roxx = Hand()
roxx = roxx + Card(42)
roxx = roxx + Card(18)
print(roxx)
```

```
>>>
4 of Spades, 6 of Diamonds
>>>
```

Python's internals are well enough designed that we can chain + operations without having to write any additional code

```
roxx = Hand()
roxx = roxx + Card(42) + Card(13) + Card(2)
print(roxx)
```

```
>>>
4 of Spades, A of Diamonds, 3 of Clubs
>>>
```

# Example: playing cards

So far we are adding a **Card** to a **CardCollection**,
but what if we also wanted to be able to add two **CardCollections** together?

```
def __add__(self, other):
    if isinstance(other, Card):
        self.cards.append(other)
        return self
    elif isinstance(other, CardCollection):
        self.cards.extend(other.cards)
        return self
    else:
        print('You can only add cards ...')
```

```
roxx = Hand()
roxx = roxx + Card(42) + Card(13) + Card(2)
print('roxx:', roxx)
chris = Hand()
chris = chris + Card(3) + Card(4)
print('chris:', chris)
new = roxx + chris
print('new:', new)
```

```
>>>
roxx: 4 of Spades, A of Diamonds, 3 of Clubs
chris: 4 of Clubs, 5 of Clubs
new: 4 of Spades, A of Diamonds, 3 of Clubs, 4 of Clubs, 5 of Clubs
>>>
```

# Example: playing cards
# A subtle point

```
print('Before:')
roxx = Hand()
roxx = roxx + Card(42) + Card(13) + Card(2)
print('roxx:', roxx)
chris = Hand()
chris = chris + Card(3) + Card(4)
print('chris:', chris)
print('After:')
new = roxx + chris
print('new:', new)
print('roxx:', roxx)
print('chris:', chris)
```

```
>>>
Before:
roxx: 4 of Spades, A of Diamonds, 3 of Clubs
chris: 4 of Clubs, 5 of Clubs
After:
new: 4 of Spades, A of Diamonds, 3 of Clubs, 4 of Clubs, 5 of Clubs
roxx: 4 of Spades, A of Diamonds, 3 of Clubs, 4 of Clubs, 5 of Clubs
chris: 4 of Clubs, 5 of Clubs
>>>
```

Notice any problems?

# Example: playing cards
# A subtle point

```python
print('Before:')
roxx = Hand()
roxx = roxx + Card(42) + Card(13) + Card(2)
print('roxx:', roxx)
chris = Hand()
chris = chris + Card(3) + Card(4)
print('chris:', chris)
print('After:')
new = roxx + chris
print('new:', new)
print('roxx:', roxx)
print('chris:', chris)
```

```
>>>
Before:
* roxx: 4 of Spades, A of Diamonds, 3 of Clubs
chris: 4 of Clubs, 5 of Clubs
After:
new: 4 of Spades, A of Diamonds, 3 of Clubs, 4 of Clubs, 5 of Clubs
* roxx: 4 of Spades, A of Diamonds, 3 of Clubs, 4 of Clubs, 5 of Clubs
chris: 4 of Clubs, 5 of Clubs
>>>
```

- **roxx** is changed by what we've done in creating **new**
- The problem: we're <u>building</u> return value from an existing object
- hence
  - existing object is modified
  - new and existing object now referring to the same modified object
- Mutability and shared references can be serious problems

# Example: playing cards
# A subtle point

Solution: create a new **CardCollection** obj in __add__ method, return this obj

```
def __add__(self, other):
    # Create a new CardCollection.
    new_cc = CardCollection()
    # Put a copy of self's cards into it.
    new_cc.cards = self.cards[:]
    if isinstance(other, Card):
        new_cc.cards.append(other)
        return new_cc
    elif isinstance(other, CardCollection):
        new_cc.cards.extend(other.cards[:])
        return new_cc
    else:
        print('You can only add Cards to CardCollections!')
```

```
Before:
roxx: 4 of Spades, A of Diamonds, 3 of Clubs
chris: 4 of Clubs, 5 of Clubs
After:
new: 4 of Spades, A of Diamonds, 3 of Clubs, 4 of Clubs, 5 of Clubs
roxx: 4 of Spades, A of Diamonds, 3 of Clubs
chris: 4 of Clubs, 5 of Clubs
>>>
```

# Conclusion: adding new types

Using classes to add new types to a programming language is a very powerful technique

it makes it easier to write programs in domains the language does not support natively.

This is a big achievement of object-oriented programming.

Adding new types requires paying close attention to the specifics of the situation you are modelling, especially where mutable types are used.