

# Introduction to computer science in **Python**

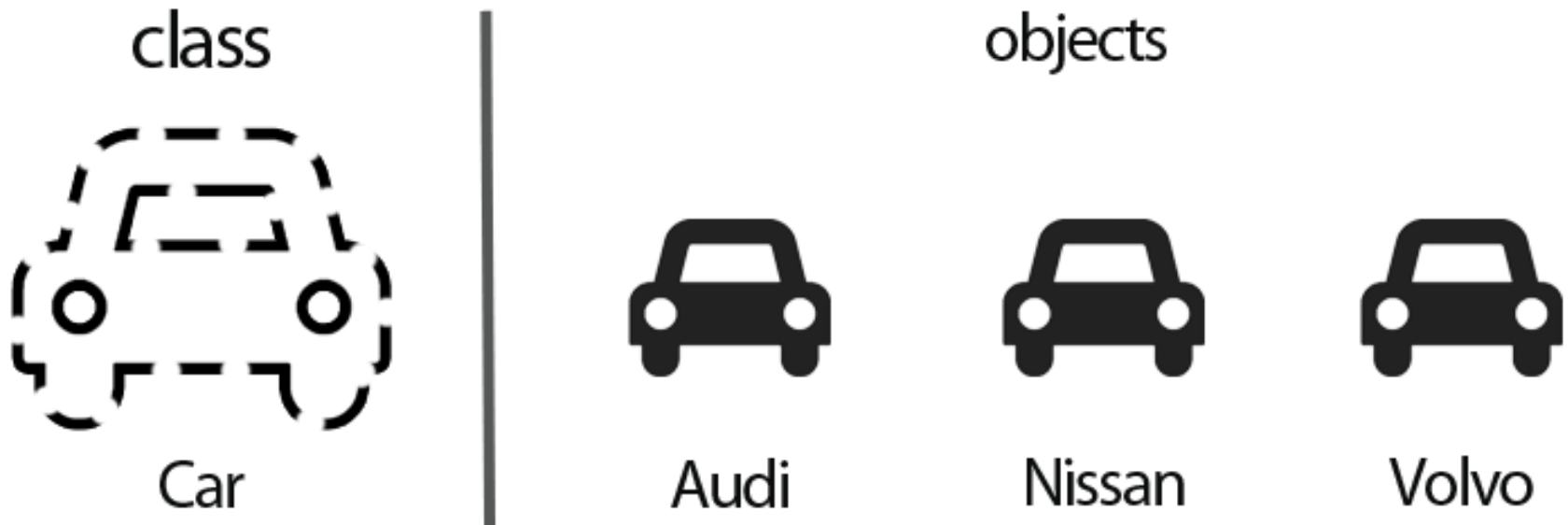
**Fall 2021-22**

**Department of Software and Information  
Systems Engineering**

**Ben-Gurion University of the Negev**

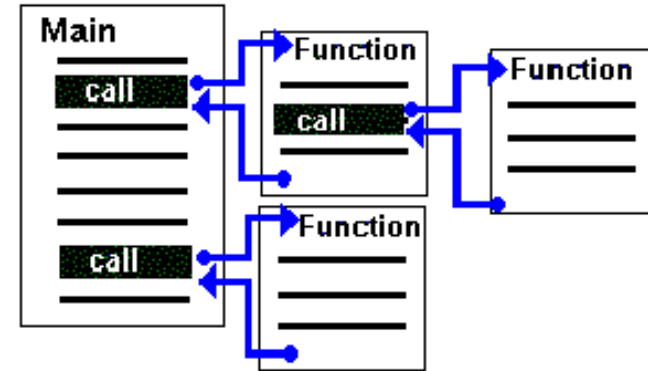
**Topic 9: Object oriented programming  
(OOP)**

# Object Oriented Programming (OOP)

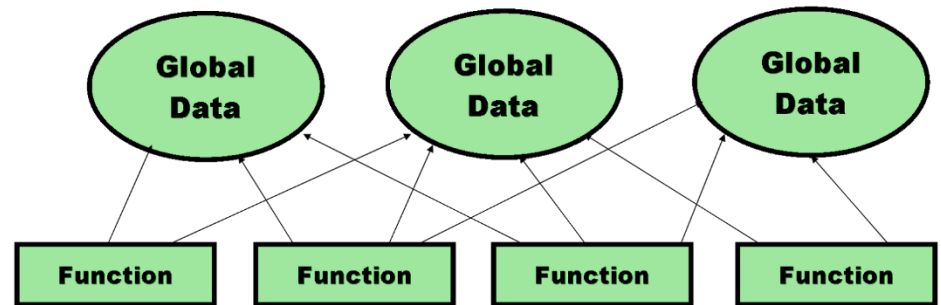


# Procedural programming

- Up until now, we used *functions* to organize our code into modular code blocks.

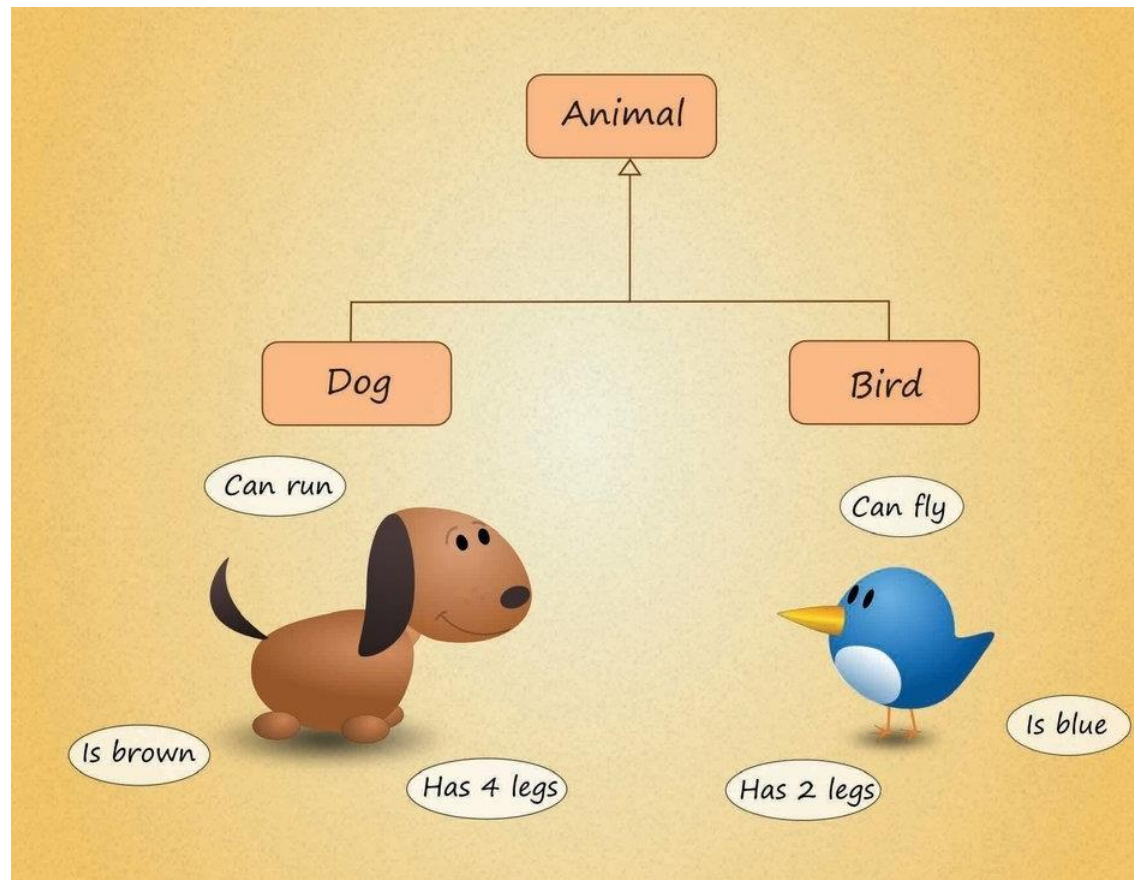


- How would we implement a software system for managing students' grades?
- Store data in global structures and accessed by many different functions.



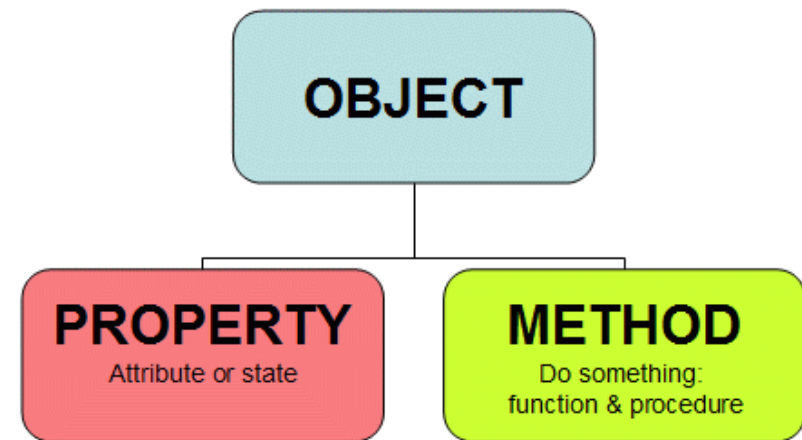
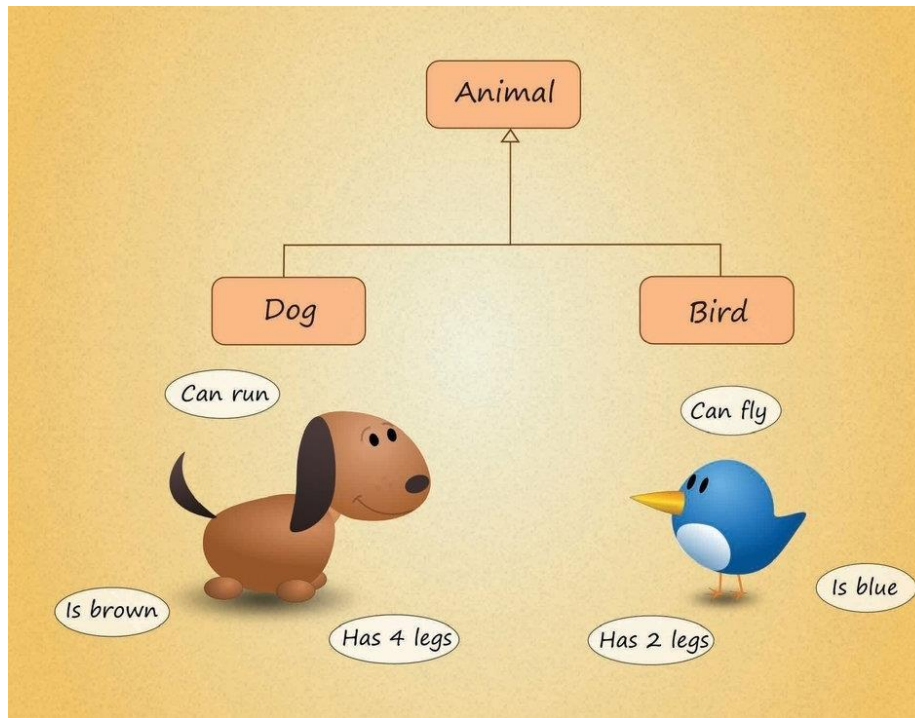
# Object oriented programming

- In Object Oriented Programming, we model the entities of the real world using **objects**.



# Object oriented programming

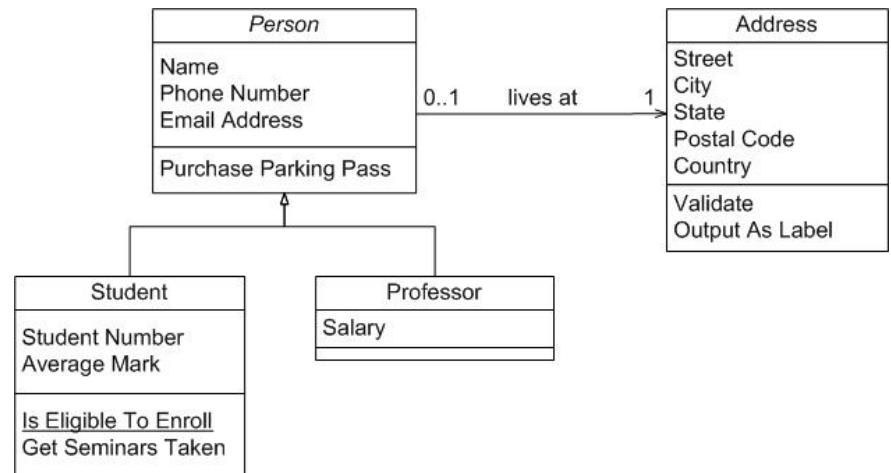
- In Object Oriented Programming, we model the entities of the real world using **objects**.
- Objects holds both **code** (functions) and **data** for the entities they represent.



# Object oriented programming

*Wikipedia:*

“An **object-oriented** program may be viewed as a **collection of interacting objects**, as opposed to the **conventional** model, in which a program is seen as **a list of tasks** (subroutines) to perform.”



# Characteristics of the OOP paradigm



# Example: an object that represents a student

<b>Student</b>
name phoneNumber address
enrollCourse() payTuition() dropCourse() getName()

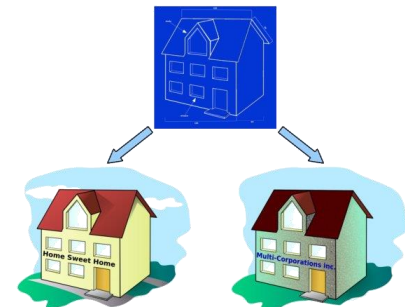
**Data:** Attributes (==Variables/Fields)

**Functionality:** Methods (==Functions)



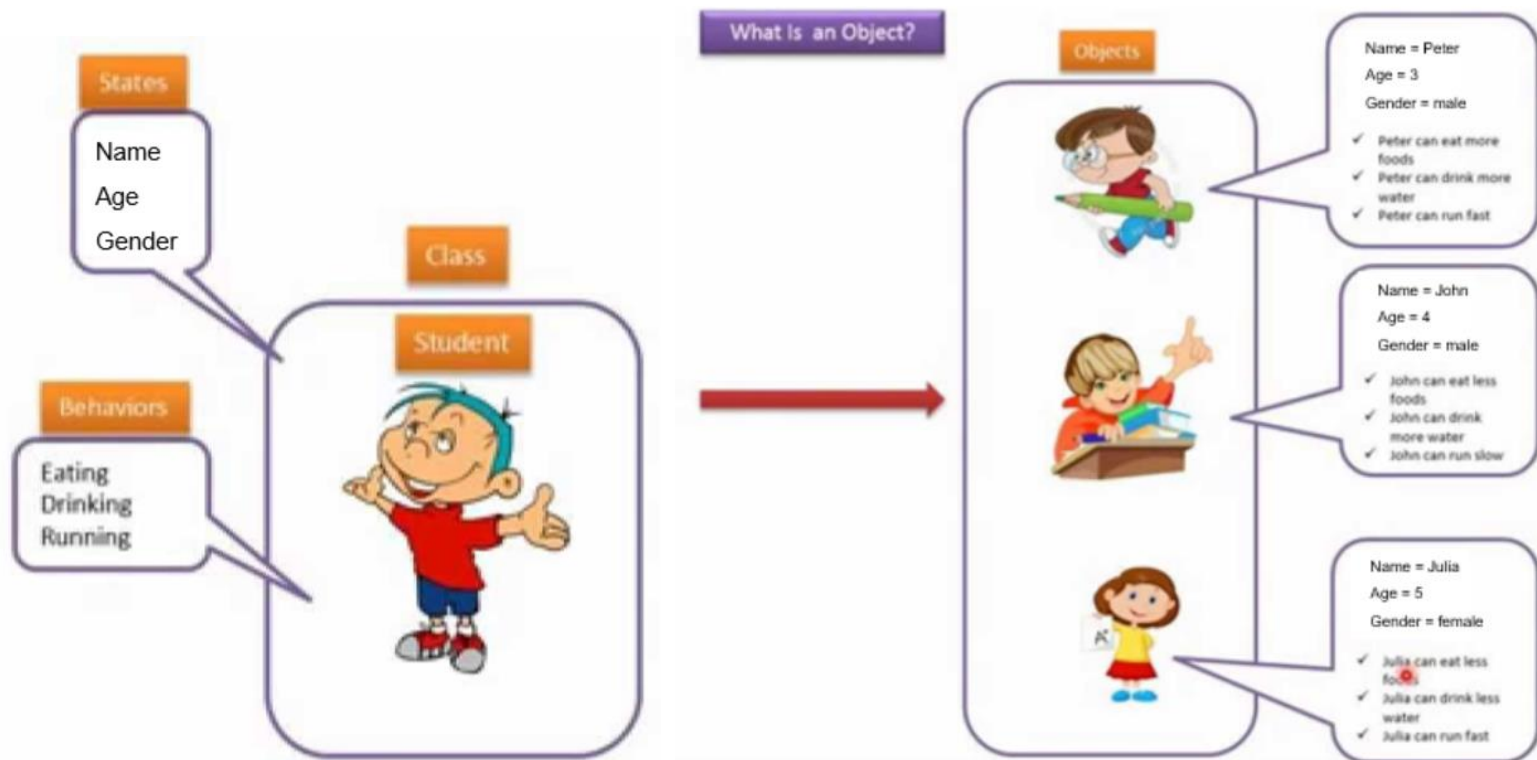
# Classes are user-defined types

- In OOP, a **Class** is a piece of code that defines a new object **type**.
  - A class defines which **attributes (fields)** will be allocated in memory for each object (instance) of the class.
  - A class contains the code for the **functions** (called methods) of a given object type.
- Classes can viewed as a blueprint for creating objects.



# Classes and objects

- A class defines a new type of which many objects (instances) can be created.
  - Each object may hold different values for the class fields.



# Classes as data types

Classes **define** types that are:

- a **composition** of other types
- have unique **functionality**

class  
Car



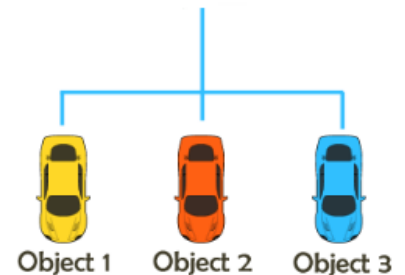
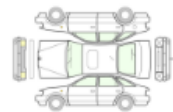
Car  
objects



Every class may contain:

- Attributes (fields)
- Methods
- Constructor (Initialization function)

Car class



# Car example

**Attributes** 4 wheels, steering wheel, horn, color etc.

- Unique for every car instance

**Methods** drive, turn left, honk, repaint etc.

**Constructor** by color, by engine type etc..



# Classes in Python

- Class definition looks like this:

```
class ClassName:
    """documentation string"""

    def __init__(self):
        # constructor

    def method_name(self, arg1, arg2):
        # method code

    def method_name(self, arg1, arg2):
        # method code

    ...
```

# How to represent a point in 2D?

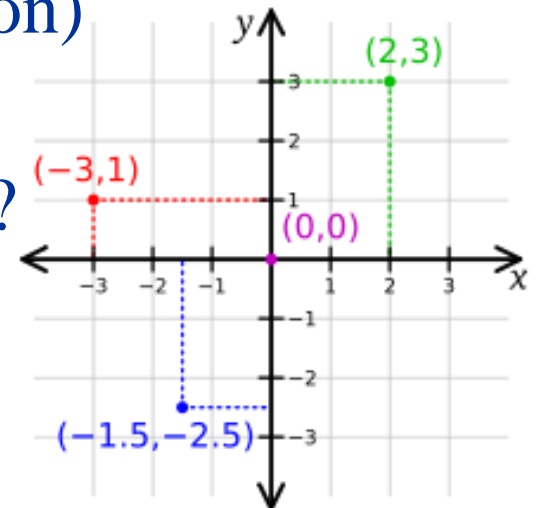
## Alternatives

- Two variables  $x$ ,  $y$
- Elements in a list / tuple
- A new data type

Creating a new type is a (little) more complicated,

But has advantages (to be apparent soon)

So - How should we represent a point?



# Creating a new Point (instantiation)

```
class Point:  
    """represents a point in 2D space.  
    Attributes: x, y"""
```

```
blank = Point() # an instance of a point
```

```
print(blank)
```

```
<__main__.Point object at 0x0000002078063588>
```

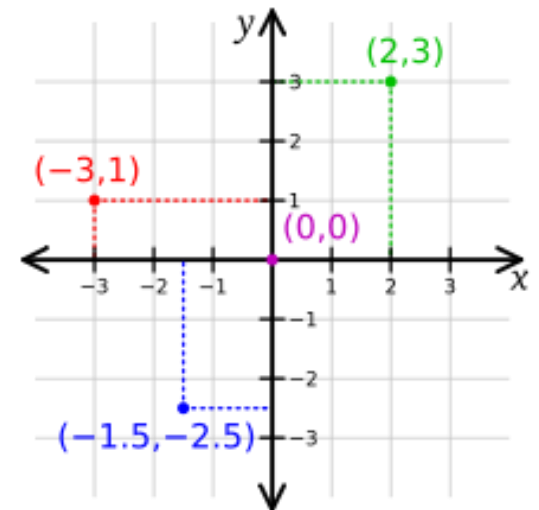
# But where is the Point?

```
blank = Point()  
blank.x
```

```
-----  
AttributeError                                Traceback (most recent  
<ipython-input-35-6f6f7681fda7> in <module>()  
      1 blank = Point()  
----> 2 blank.x
```

```
AttributeError: 'Point' object has no attribute 'x'
```

- Currently, blank does not hold any data, so x does not exist (nor does y)
- We want to be able to **initialize** and **access** x, y via Point instance





# Object initialization

- There are two ways to initialize an object:
  - Explicitly

```
blank = Point()  
blank.x = 3.0  
blank.y = 4.0
```

```
blank.x
```

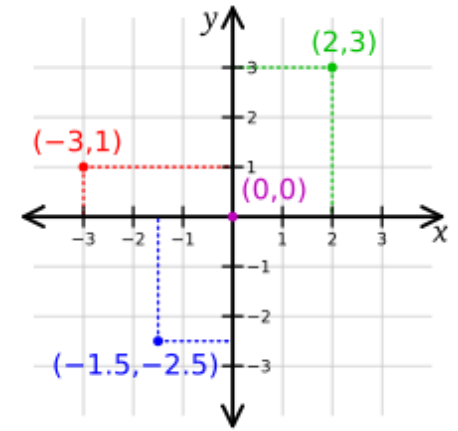
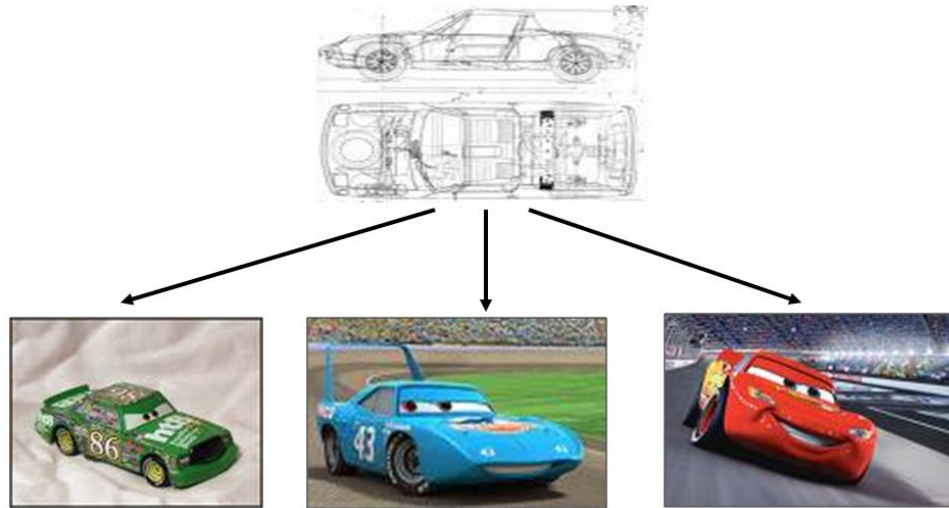
```
3.0
```

```
blank.y
```

```
4.0
```

- Using a **constructor**, which is a special method in charge of initializing the object's variables

# Constructors



- **Definition** in the class' code that “produces” instances
- **Invoked** automatically when an object is created
- **Input:** whatever is required to produce a new instance
- What would a Point constructor accept as input?

# Constructors – more technically

```
class Point:
    """represents a point in 2D space.
       Attributes: x, y"""
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

Definition within the class's scope:

- Always named `__init__`: 2 underscores, followed by 'init', followed by 2 underscores

# Constructors – more technically

```
class Point:
    """represents a point in 2D space.
       Attributes: x, y"""
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

- The first parameter is *self*
  - *self* appears in **all** non-static\* class methods, as opposed to global functions
  - *Self* is a *reference* to the current instance of the class.
  - When calling the constructor, *self* is **not** passed, it is created automatically

\* Static methods will be explained later.

# Constructors – more technically

```
class Point:
    """represents a point in 2D space.
       Attributes: x, y"""
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

## Default constructor

- If a constructor was not implemented for a class, Python assumes that the class has the default constructor
- The default constructor is an empty constructor (no arguments, no attribute initialization, no code essentially)

# Attributes

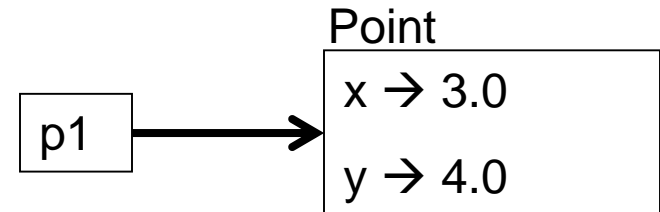
```
p1 = Point(3.0, 4.0)
```

```
p1.x
```

```
3.0
```

```
p1.y
```

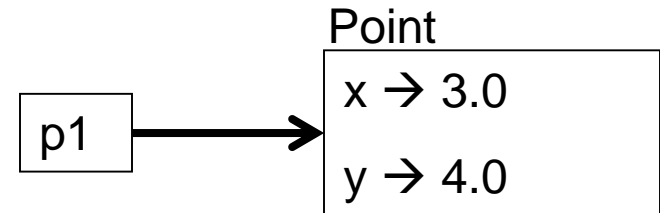
```
4.0
```



- The variable **p1** refers to a Point object
- **p1.x** means “get the value of *x* from object *p1*”

# Attributes

```
p1 = Point(3.0, 4.0)
```



- No conflict between a variable  $x$  and the attribute  $x$

```
x = p1.y  
print(x)  
print(p1.x)
```

4.0

3.0

# Instances and functions

Objects can be passed as arguments to functions:

```
def print_point(point):  
    print("<" + str(point.x) + "," + str(point.y) + ">")
```

```
print_point(p1)
```

```
<3.0,4.0>
```

Objects are mutable:

```
def move_point(point,dx,dy):  
    point.x += dx  
    point.y += dy
```

```
move_point(p1,2,1)  
print_point(p1)
```

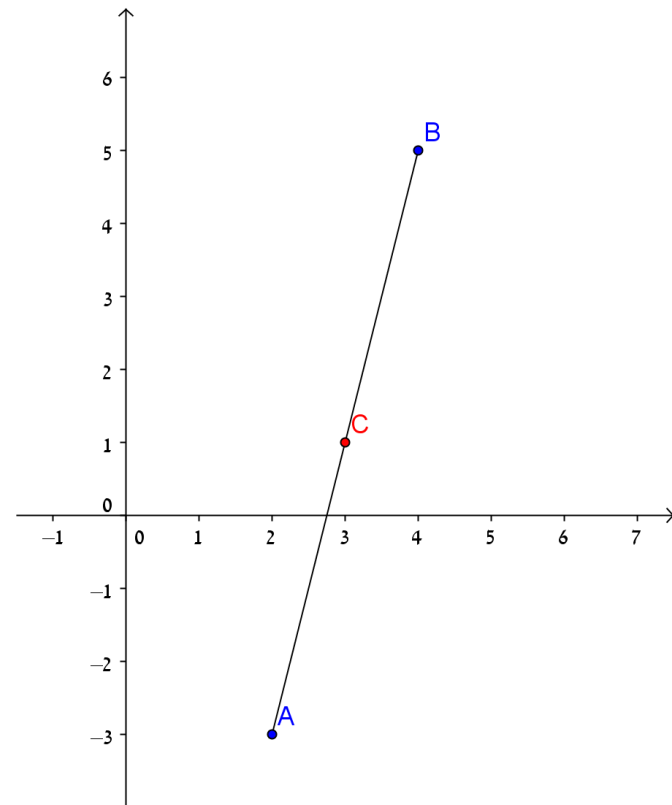
```
<5.0,5.0>
```



# Instances and functions

Objects can be returned by functions:

```
def middle_point(p1,p2):  
    return Point((p1.x+p2.x)/2.0,(p1.y+p2.y)/2.0)
```



```
p1 = Point(1,1)  
p2 = Point(3,5)  
p3 = middle_point(p1,p2)
```

p3.x

2.0

p3.y

3.0

# Object oriented programming

**Programs** are made of

- **object** definitions
- **function** definitions
- Most of the computation is in operations on objects
- An **object definition** corresponds to objects or concepts in the real world
- The **functions** that operate on that objects correspond to the ways real-world objects interact

# Methods

## Method

a function that is associated with a particular class.

Examples in strings, lists, dictionaries, tuples:

`list.append()`, `str.upper()`, `dict.items()`

Difference between **methods** and **functions**:

- Methods are defined inside a class definition
- The syntax for invoking a method is different:
  - A method is called through an instance

# Methods

1. The first argument of each method is **self**. It's not passed as an argument in the call – **self** is actually the object that invoked the method.
2. Access the attributes of the class – only with self.

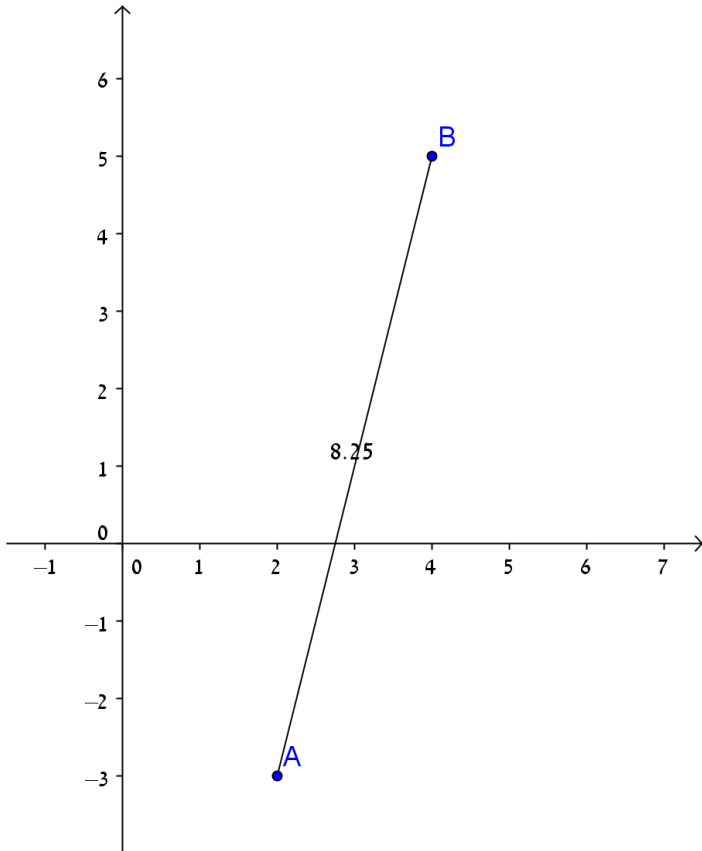
```
class New_Class:
    """documentation string"""

    def __init__(self, att1):
        self.att1 = att1;

    def method_name(self, arg1, arg2):
        do_something_with(self, att1)
```

# Example: distance between two points (method)

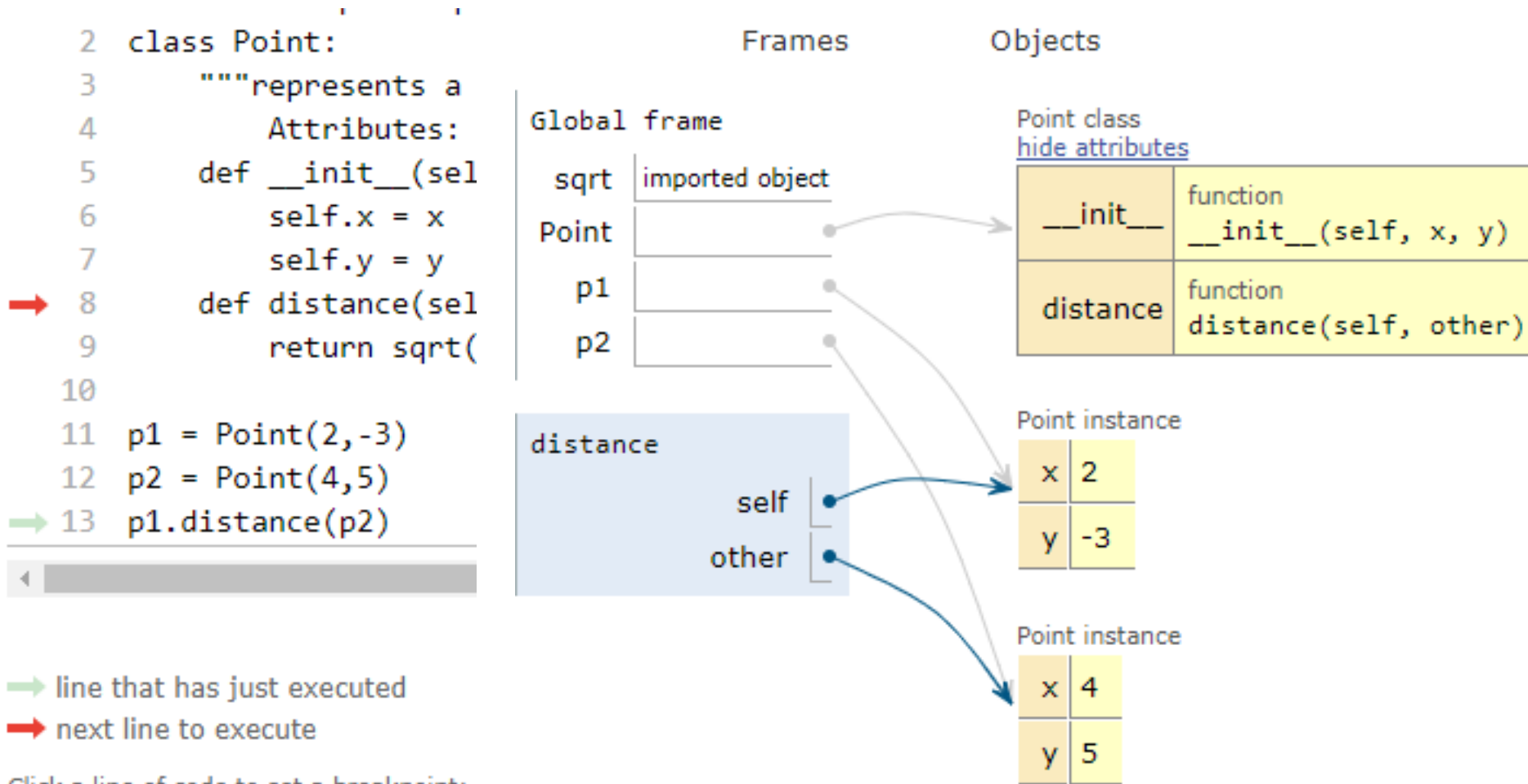
```
def distance(self, other):  
    return sqrt((self.x-other.x)**2 + (self.y-other.y)**2)
```



```
p1 = Point(2, -3)  
p2 = Point(4, 5)  
p1.distance(p2)
```

8.246211251235321

# Example: distance between two points (method) - abstraction



# Example: distance between two points (method vs. function)

```
from math import sqrt

class Point:
    .....
    def distance(self, other):
        return sqrt((self.x-other.x)**2 + (self.y-other.y)**2)
    .....

def distance(p1, p2):
    return sqrt((p1.x - p2.x)**2 + (p1.y - p2.y)**2)
```

```
>>> p1 = Point(2,-3)
```

```
>>> p2 = Point(4,5)
```

```
>>> p1.distance(p2)
```

```
8.246211251235321
```

Method call

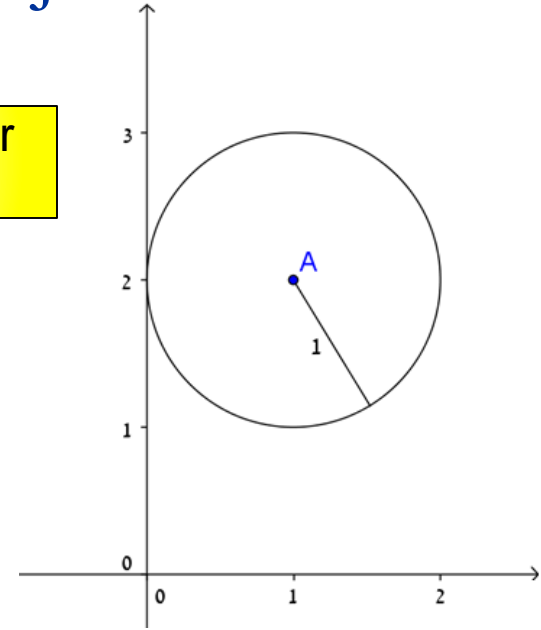
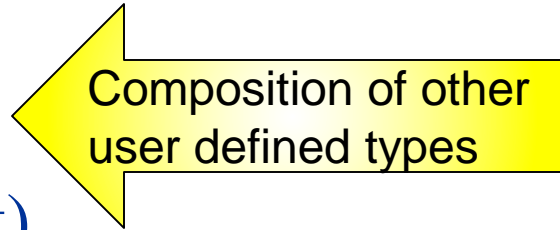
```
>>> distance(p1,p2)
```

```
8.246211251235321
```

function call

# A Circle

- How would you represent a circle object?
- Attributes:
  - center (Point)
  - radius (int/float)
- Methods:



Method name	description	arguments	return value
<code>__init__</code>	The constructor	Center(Point), radius(int/float)	-
<code>print_circle</code>	Prints circle	-	-
<code>in_circle</code>	Checks if a point is located inside the circle	a <i>point</i> object	True/False



# A Circle

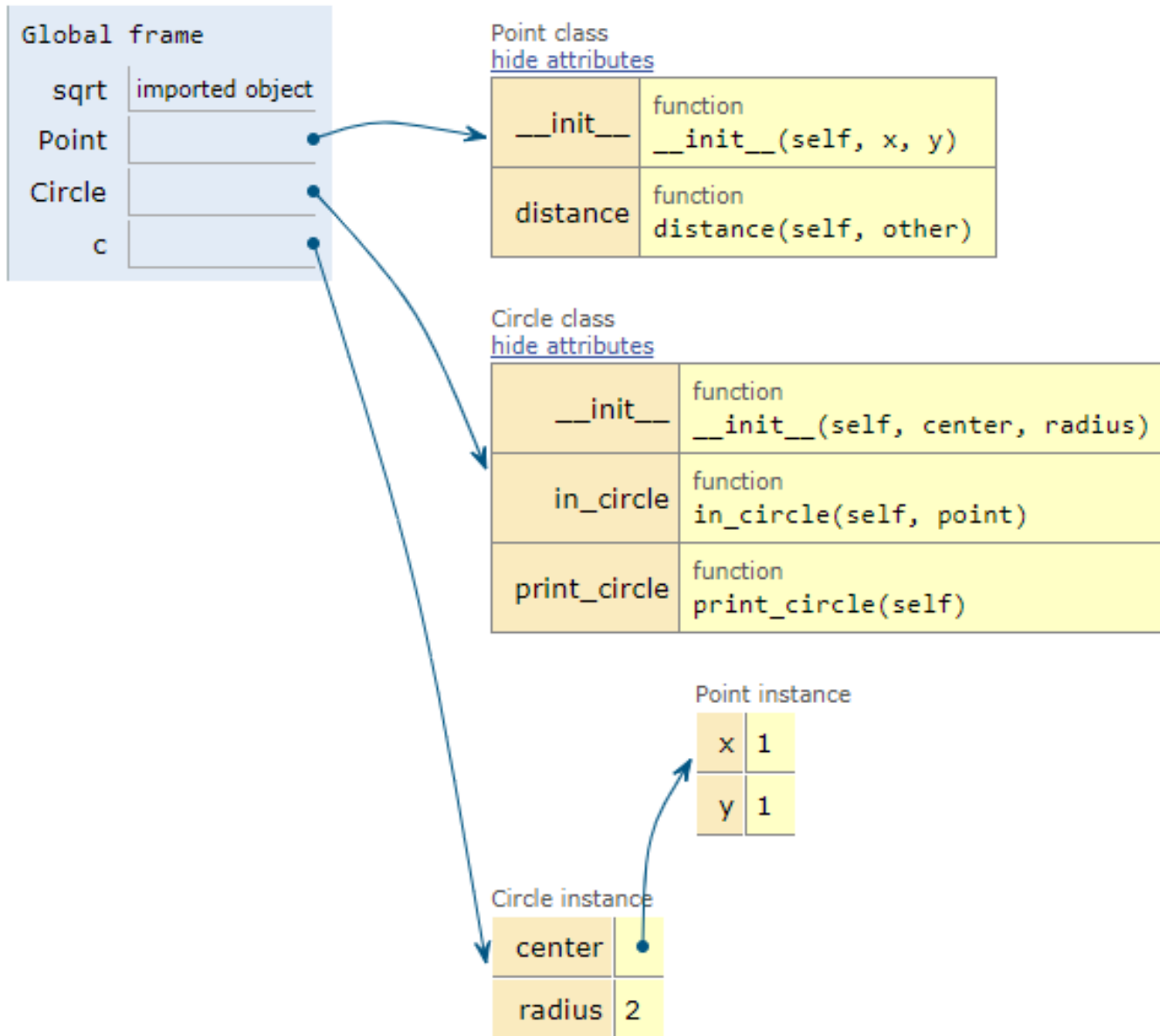
```
class Circle:
    """represents a circle shape
    attributes: center, radius"""
    def __init__(self, center, radius):
        self.center = center
        self.radius = radius

    def print_circle(self):
        print ("center:", self.center.x, self.center.y, ", radius:", self.radius)
```

```
c = Circle(Point(1,2),1)
c.print_circle()
```

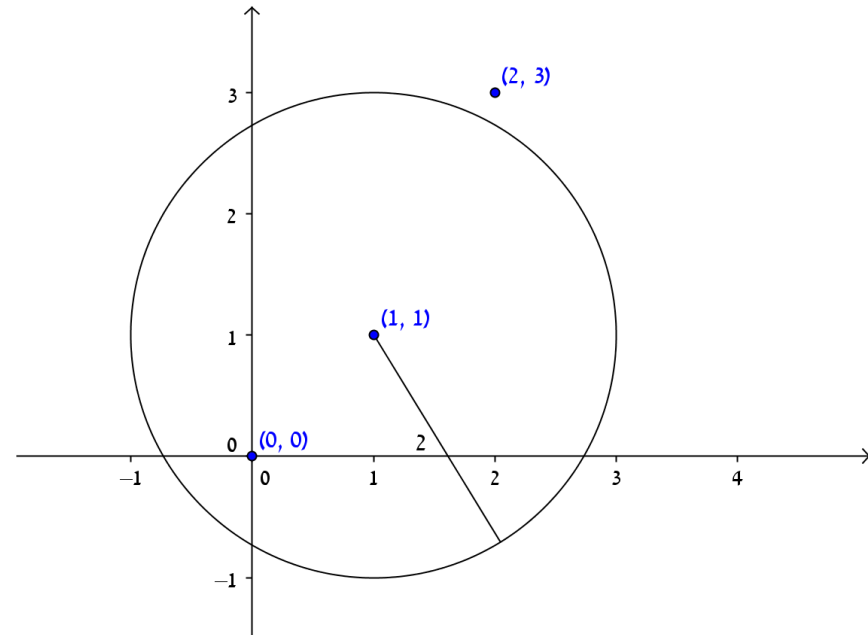
```
center: 1 2 , radius: 1
```

# A circle's memory view



# in\_circle method

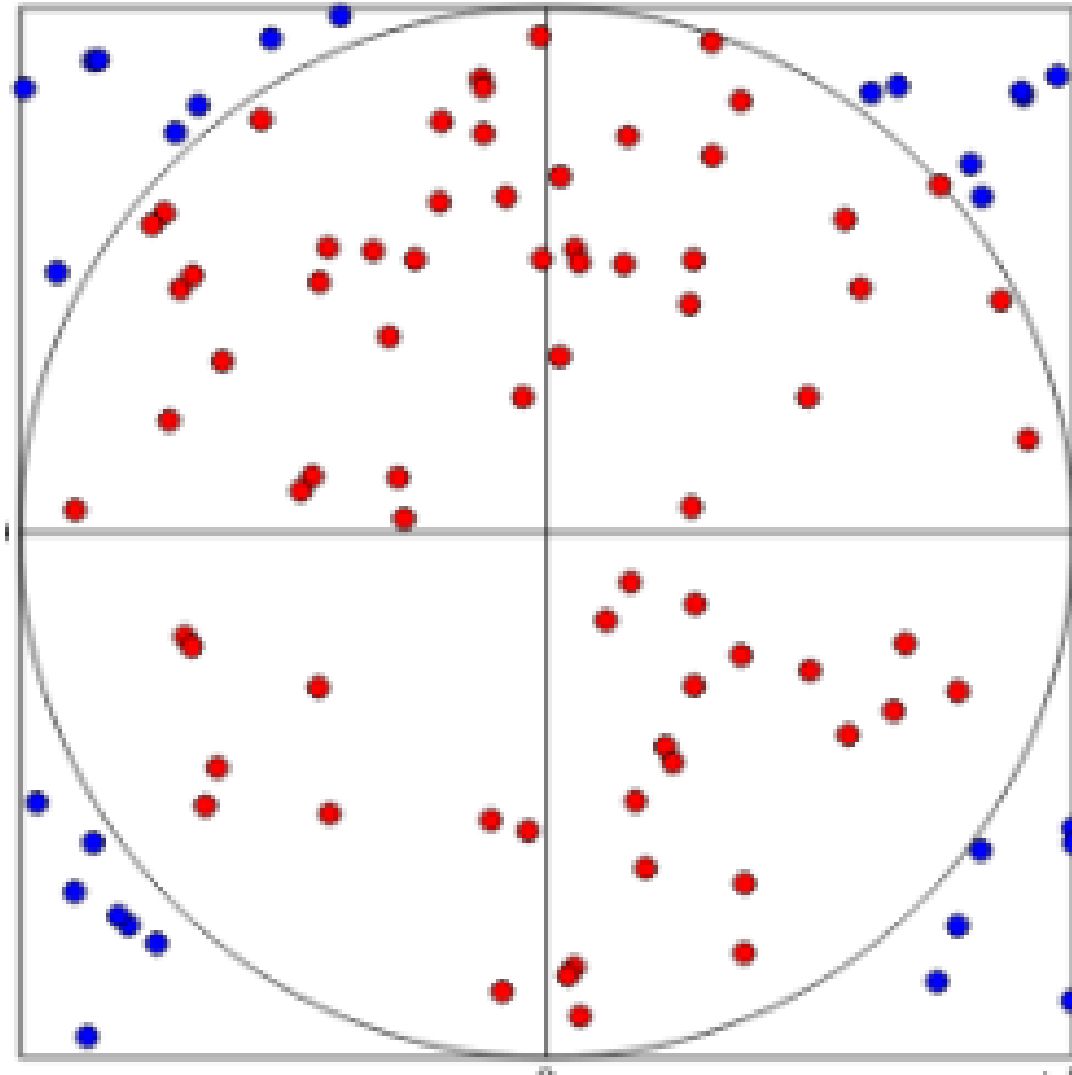
```
class Circle:
    ....
    def in_circle(self, point):
        return self.center.distance(point) < self.radius
```



```
c = Circle(Point(1,1),2)
print(c.in_circle(Point(0,0)))
print(c.in_circle(Point(2,3)))
```

True  
False

# Estimating $\pi$ with circles and points (with Monte Carlo simulation)



# Estimating $\pi$ with circles and points (with Monte Carlo simulation)

```
import random

unitCircle = Circle(Point(0,0),1)

N = [10,100,1000,10000,100000,1000000,10000000]
for n in N:
    count = 0
    for i in range(n):
        # points in [1,1]
        randPoint = Point(random.uniform(-1,1),random.uniform(-1,1))
        if unitCircle.in_circle(randPoint):
            count += 1
    print("n = " + str(n) + ", pi ~ " + str(4*count/n))
```

n = 10, pi ~ 3.2

n = 100, pi ~ 3.12

n = 1000, pi ~ 3.224

n = 10000, pi ~ 3.142

n = 100000, pi ~ 3.1404

n = 1000000, pi ~ 3.142068

n = 10000000, pi ~ 3.1423048

# Point and circle - recap

```
from math import sqrt

class Point:
    """represents a point in 2D space.
    Attributes: x, y"""
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def print_point(self):
        print("<", self.x, ",", self.y, ">")

    def distance(self, other):
        return sqrt((self.x-other.x)**2 + (self.y-other.y)**2)

class Circle:
    """represents a circle shape
    attributes: center, radius"""
    def __init__(self, center, radius):
        self.center = center
        self.radius = radius

    def print_circle(self):
        print ("center:", self.center.x, self.center.y, ", radius:", self.radius)


    def in_circle(self, point):
        return self.center.distance(point) < self.radius
```

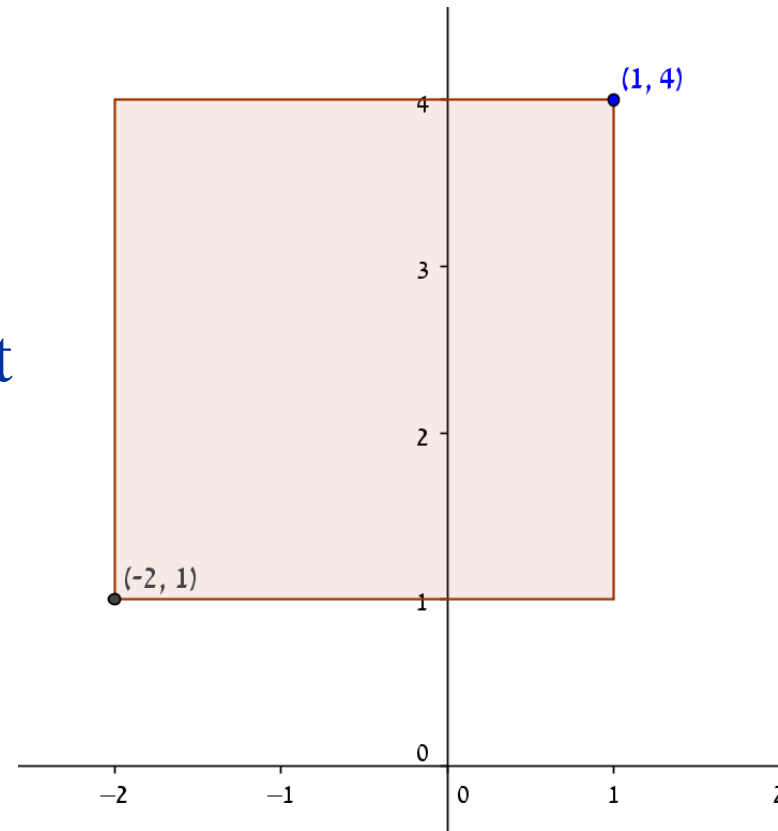
# A Rectangle (design options)

## How would you represent a rectangle?

For simplicity ignore angle, assume the rectangle's edges are parallel to the axes.

Several possibilities:

- Two opposing corners
-  One corner + width and height
- Center + Width and Height



# Rectangle - design

- Attributes:
  - width (int/float)
  - height (int/float)
  - corner (Point)
- Methods

Method name	description	arguments	return value
<code>__init__</code>	The constructor	width, height, corner	-
<code>print_rec</code>	Prints the rectangle	-	-
<code>get_center</code>	Returns the center of the rectangle	-	A <i>Point</i> object that represents the center.



# A Rectangle - implementation

```
class Rectangle:
    """represents a rectangle in 2D space.
    attributes: width, height, lower-left corner"""
    def __init__(self, width, height, corner):
        self.width, self.height = width, height
        self.corner = corner

    def print_rec(self):
        print("Width:", self.width, ", Height:",
              self.height, ", Lower-left corner:",
              self.corner.x, self.corner.y)
```

```
rect = Rectangle(100,200,Point(0,0))
rect.print_rec()
```

Width: 100 , Height: 200 , Lower-left corner: 0 0

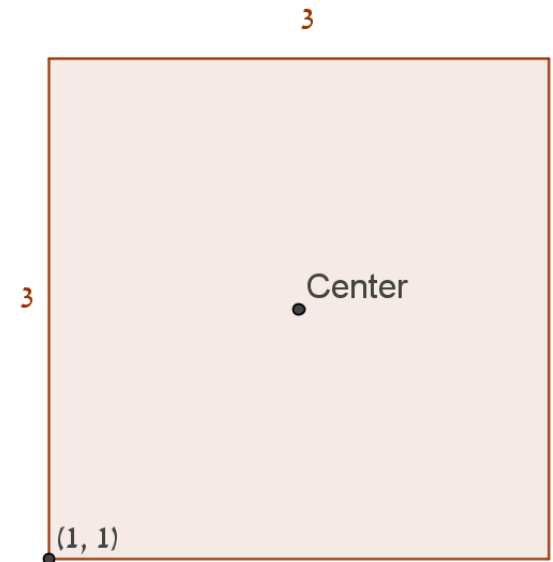
# Find rectangle center

```
class Rectangle:
    ....

    def get_center(self):
        center_x = self.corner.x + self.width/2.
        center_y = self.corner.y + self.height/2.
        return Point(center_x, center_y)
```

```
center = rect.get_center()
center.print_point()
```

```
< 50.0 , 100.0 >
```



# Inflate rectangle

```
class Rectangle:
```

```
....
```

```
def inflate_rectangle(self, factor):  
    self.width *= factor  
    self.height *= factor
```

```
rect = Rectangle(100, 200, Point(0, 0))  
rect.inflate_rectangle(1.2)  
print(rect.width)  
print(rect.height)
```

120.0

240.0

# Polymorphism

## Definition

Objects of various types define a common **interface** of operations for users.



# Polymorphism

**Enables working uniformly with objects of different types.**

- Executing the same method on objects of different types will invoke the right version of the method!



# Polymorphism - example

```
class Cat:
    def talk(self):
        return 'Meow!'

class Dog:
    def talk(self):
        return 'Woof! Woof!'

animals = [Cat(), Dog()]
for anim in animals:
    print(anim.talk())
```

Meow!

Woof! Woof!

# Shapes area

## Circle

```
def area(self):  
    return pi*self.radius **2
```

## Rectangle

```
def area(self):  
    return self.width * self.height
```

```
shapes = [Circle(Point(1,1),2),Rectangle(100,200,Point(0,0))]  
for s in shapes:  
    print(str(type(s)) + " area " + str(s.area()))
```

```
<class '__main__.Circle'> area 12.566370614359172  
<class '__main__.Rectangle'> area 20000
```

# Histogram (polymorphism)

```
def histogram(iterable):  
    hist = {}  
    for elem in iterable:  
        hist[elem] = hist.get(elem, 0) + 1  
    return hist
```

This function works for lists, tuples, strings and dictionaries as long as the elements of *iterable* can be used as keys in hist

```
>>> words = ['spam', 'egg', 'spam', 'bacon', 'spam']  
>>> histogram(words)  
{'bacon': 1, 'egg': 1, 'spam': 3}
```



# Polymorphism – the Mavs version!

<https://www.youtube.com/watch?v=iK0CH3hA0Go>



# Shallow vs deep copy

# Shallow vs deep copy

- It is sometimes required to make copies of objects.
- In simple types this is easy:
  - int, float, etc. – atomic value is copied
  - string is immutable –a new copy is made
  - List is mutable – depends on method of copy  
(`lst2 = lst1` vs `lst2 = lst1[:]`)
- User defined objects are mutable, and there are different ways to copy them.

# Reminder: identity and equality (*is* and *==*)

*is* will return True if two variables point to the same object.

*==* will return True if the objects referred to by the variables are equal

```
>>> a = [1, 2, 3]
```

```
>>> b = a
```

```
>>> b is a
```

```
True
```

```
>>> b == a
```

```
True
```

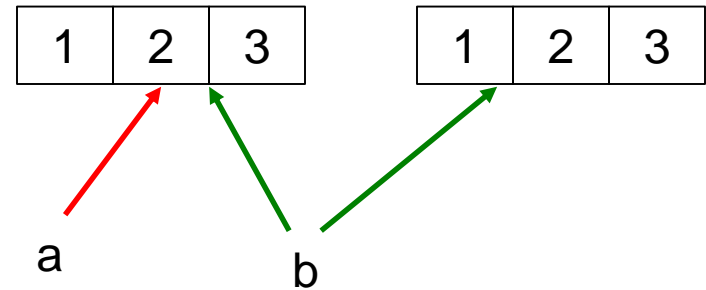
```
>>> b = a[:]
```

```
>>> b is a
```

```
False
```

```
>>> b == a
```

```
True
```



# Copying objects

```
import copy
p1 = Point(3,5)
p2 = copy.copy(p1)
print_point(p1)
print_point(p2)
```

```
< 3 , 5 >
< 3 , 5 >
```

```
p1 is p2
```

False

**This is the programmer's responsibility (we'll see soon)!**

```
p1 == p2
```

False

```
p1.x = 1
p2.x
```

3

- *is* operator indicates that p1 and p2 are not the same object
- The default behavior of the *==* operator is the same as the *is* operator

# Shallow versus deep copy

```
p1 = Point(1,1)
c1 = Circle(p1,2)
c2 = copy.copy(c1)
c1.center is c2.center
```

True

```
c3 = copy.deepcopy(c1)
c1.center is c3.center
```

False

- `copy.copy(obj)` makes a copy of `obj` by copying the value each attribute of `obj` to a new object and returning it.
- `copy.deepcopy(obj)` makes a copy of `obj` by **recursively deep copying** each attribute of `obj` to a new object and returning it

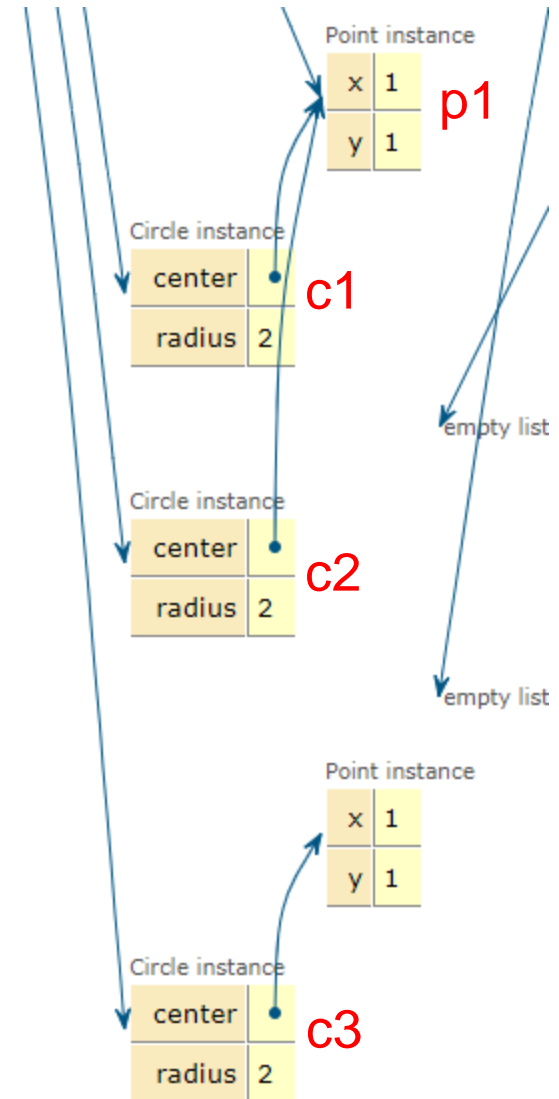
# Shallow versus deep copy - memory

```
p1 = Point(1,1)
c1 = Circle(p1,2)
c2 = copy.copy(c1)
c1.center is c2.center
```

True

```
c3 = copy.deepcopy(c1)
c1.center is c3.center
```

False



# Recap

- Object oriented programming
  - Introduction to object oriented programming
  - 2D geometry: Point, Circle, Rectangle
  - Constructors, attributes, methods, self, memory view
  - Monte Carlo estimation of pi with circles and points
  - Polymorphism
  - Shallow versus deep copy

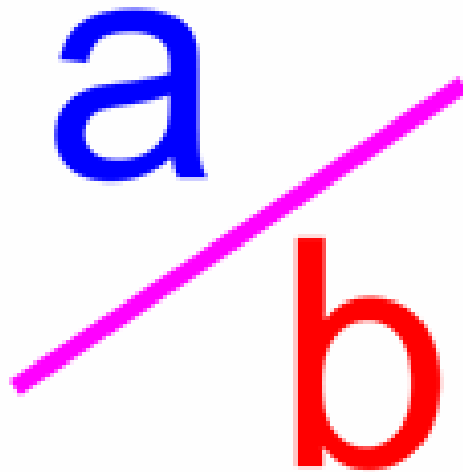


# **OOP class design:**

Implementing Rational numbers as  
new data types

# Rational numbers as “built-in” data types in Python

- Implementing a new data type
- Usage should feel like native language support



# Rational Numbers

- A rational number is a number that can be expressed as a ratio  $n/d$  ( $n, d$  integers,  $d$  not 0)
- Examples:  $1/2$ ,  $2/3$ ,  $112/239$ ,  $2/1$
- Not an approximation!  $1/1$   $1/2$   $1/3$   $1/4$   $1/5$   $1/6$   $1/7$   $1/8$

[illegible]

# Specification

- print should work smoothly
- Add, subtract, multiply, divide
- Immutable
- It should feel like native language support

```
one_half = Rational(1,2)  
two_thirds = Rational(2,3)
```

```
one_half / (2 + two_thirds)
```

3/16

```
sum([1,Rational(2,3),2])
```

11/3

# Constructing a Rational

- What are the attributes?
- How should a client programmer create a new Rational object?

```
class Rational:
    """represents a rational number
    attributes: n,d"""
    def __init__(self,n,d):
        """n: numerator
        d: denominator"""
        self.numer = n
        self.denom = d
```

# Default arguments to constructor

- Constructors other than the primary?
- Example: a rational number with a denominator of 1 (e.g.,  $5/1 \rightarrow 5$ )
- We would like to enable: **Rational(5)**

```
>>> x = Rational(5)
```

```
Traceback (most recent call last):
```

```
  File "<pysHELL#2>", line 1, in <module>
```

```
    x = Rational(5)
```

```
TypeError: __init__() takes exactly 3 arguments (2 given)
```

# Default arguments to constructors

```
class Rational:
    """represents a rational number
    attributes: n,d"""
    def __init__(self, n, d=1):
        """n: numerator
        d: denominator"""
        self.numer = n
        self.denom = d
```

```
p = Rational(5) # == Rational(5,1)
q = Rational(5,2)
```

# Checking preconditions

Ensure the arguments are **valid** when the object is initialized:

```
q = Rational(5,0)
```

```
class Rational:
    """represents a rational number
    attributes: n,d"""
    def __init__(self,n,d=1):
        """n: numerator
        d: denominator"""
        if d == 0:
            print("dominator cannot be zero!")
        self.numer = n
        self.denom = d
```



# Checking preconditions

Ensure the arguments are **valid** when the object is initialized:

```
q = Rational(5,0)
```

```
dominator cannot be zero!
```

```
print(q.numer)  
print(q.denom)
```

```
5
```

```
0
```

# Throw an exception..

(we'll learn more about it in recitations)

```
class Rational:
    """represents a rational number
    attributes: n,d"""
    def __init__(self,n,d=1):
        """n: numerator
        d: denominator"""
        if d == 0:
            raise ZeroDivisionError("Denominator cannot be zero")
        self.numer = n
        self.denom = d
```

```
q = Rational(5,0)
```

-----  
ZeroDivisionError Traceback (most recent call last):

<ipython-input-30-7d4dcd2d6eb9> in <module>()

----> 1 q = Rational(5,0)

<ipython-input-29-59f4a46311a8> in \_\_init\_\_(self, n, d)

```
      6         d: denominator"""
      7         if d == 0:
----> 8             raise ZeroDivisionError("Denominator cannot be
      9         self.numer = n
     10         self.denom = d
```

ZeroDivisionError: Denominator cannot be zero

# Printing a Rational

```
q = Rational(2,3)
```

```
q.numer
```

```
2
```

```
q.denom
```

```
3
```

```
q
```

```
<__main__.Rational
```

```
0x00000080DAEB9A58>
```

```
print(q)
```

```
<__main__.Rational object at 0x00000080DAEB9A58>
```



# Implementing `__repr__`

**`__repr__` method:**

returns the official **string representation** of an object

In class `Rational`:

```
def __repr__(self):  
    return str(self.numer) + '/' + str(self.denom)
```

Using it:

```
q = Rational(2,3)
```

```
q
```

```
2/3
```

```
print(q)
```

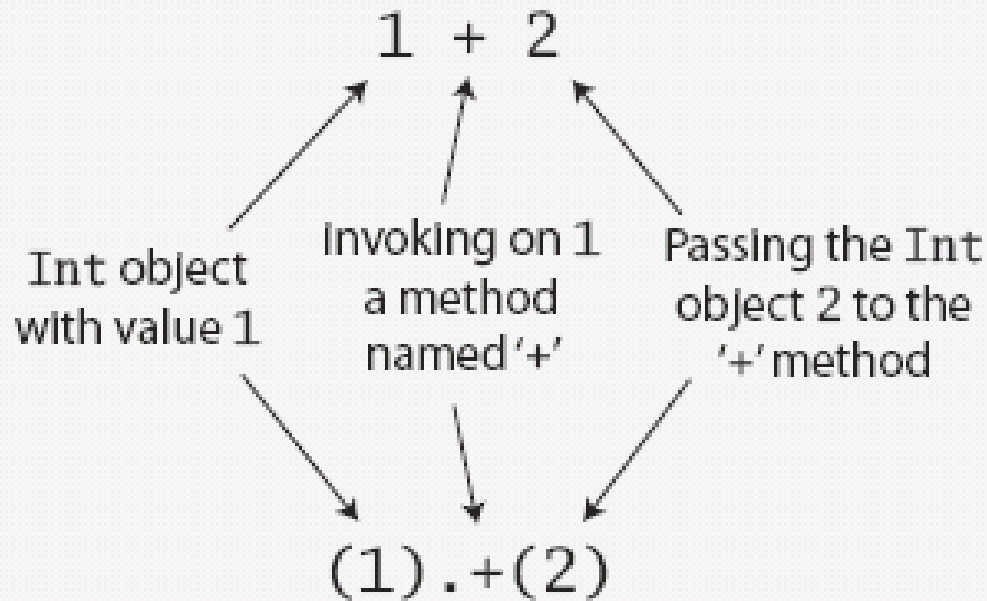
```
2/3
```

# Implementing `__str__`

- `__str__` method returns a **readable** string representation of an object
  - `__repr__` goal is to be unambiguous
  - `__str__` goal is to be readable
- The default implementation of `__str__` returns the return value of `__repr__`

# Defining operators

- Why not use natural arithmetic operators?
- Operator precedence will be kept
- All operations are method calls



From the book Programming in Scala

# Adding rational numbers

```
p = Rational(1,2)
q = Rational(2,3)
p+q
```

```
-----
TypeError                                Traceback (most recent call last)
<ipython-input-3-b3c77ca81e60> in <module>()
      1 p = Rational(1,2)
      2 q = Rational(2,3)
----> 3 p+q
```

```
TypeError: unsupported operand type(s) for +: 'Rational' and 'Rational'
```

# Operator overriding

- By defining some **special methods**, we can specify the behavior of operators on user defined types
- $+$ ,  $-$ ,  $*$ ,  $/$ ,  $<$ ,  $>$ ,  $==$ , ...



# Special methods

- Special methods, begin and end with `__`. These methods are invoked (called) when specific operators or expressions are used.
- Full list: <http://getpython3.com/diveintopython3/special-method-names.html>

You Want...	So You Write...	And Python Calls...
to initialize an instance of class MyClass	<code>x = MyClass()</code>	<code>x.__init__()</code>
the “official” representation as a string	<code>print(x)</code>	<code>x.__repr__()</code>
addition	<code>x + y</code>	<code>x.__add__(y)</code>
subtraction	<code>x - y</code>	<code>x.__sub__(y)</code>
multiplication	<code>x * y</code>	<code>x.__mul__(y)</code>
equality	<code>x == y</code>	<code>x.__eq__(y)</code>
less than	<code>x &lt; y</code>	<code>x.__lt__(y)</code>
for collections: to know whether it contains a specific value	<code>k in x</code>	<code>x.__contains__(k)</code>
for collections: to know the size	<code>len(x)</code>	<code>x.__len__()</code>
... (many more)		

# Arithmetic operators overriding

```
>>> q = Rational(1,2)
>>> p = Rational(2,3)
```

Operator	Method
p+q	<code>__add__(self,other)</code>
-p	<code>__neg__(self)</code>
p-q	<code>__sub__(self,other)</code>
p*q	<code>__mul__(self,other)</code>
p/q	<code>__truediv__(self,other)</code>

# Define `__add__` method

$$\frac{a}{b} + \frac{c}{d} = \frac{a*d+c*b}{b*d}$$

```
def __add__(self, other):  
    # immutable!  
    return Rational(self.numer * other.denom +  
                     other.numer * self.denom,  
                     self.denom * other.denom)
```

**Immutable!**

```
p = Rational(1,2)  
q = Rational(2,3)  
p+q
```

# Other arithmetic operations implementation

```
def __neg__(self):  
    return Rational(-self.numer, self.denom)  
def __sub__(self, other):  
    return self + (-other)  
def __mul__(self, other):  
    return Rational(self.numer * other.numer, self.denom * other.denom)  
def __truediv__(self, other):  
    return Rational(self.numer * other.denom, self.denom * other.numer)
```

# Other arithmetic operations

```
p = Rational(1,2)  
q = Rational(2,3)
```

```
q/p
```

4/3

```
-q
```

-2/3

```
q*(p+q)
```

14/18

```
q-p
```

1/6

```
q*p+q
```

```
q*p
```

2/6

# Arithmetics for mixed types

- Now we can add and multiply rational numbers!
- What about mixed arithmetic: Rational+int or int+Rational?

```
p = Rational(2,3)
p + 2
```

```
-----
AttributeError                                Traceback (most recent call last)
<ipython-input-28-c8e62c959e6e> in <module>()
      1 p = Rational(2,3)
----> 2 p + 2

<ipython-input-21-2b5379ea61f7> in __add__(self, other)
     14 def __add__(self, other):
     15     # immutable!
----> 16     return Rational(self.numer * other.denom +
     17                       other.numer * self.denom,
     18                       self.denom * other.denom)
```

```
AttributeError: 'int' object has no attribute 'denom'
```

# Arithmetics for mixed types

Let's try **Rational** + **int** first:

- first attempt:

$\text{Rational} + \text{int} \Rightarrow \text{Rational} + \text{Rational}(\text{int})$

```
p = Rational(2,3)
p + Rational(2)
```

8/3

Works but not elegant or comfortable:

- Add new methods for mixed addition and multiplication
- Will work thanks to polymorphism

# Arithmetics for mixed types

Intuition: converting int to Rational is a good idea, we just need to figure out where to perform the conversion.

Inside `__add__` method!

```
def __add__(self, other):  
    other = Rational(other)  
    return Rational(self.numer * other.denom  
                    + other.numer * self.denom,  
                    self.denom * other.denom)
```

Will this work?



# Arithmetics for mixed types

```
>>> p = Rational(2,3)
>>> p + 2
8/3
```



```
>>> p = Rational(2,3)
>>> p+p
```

```
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    p+p
```



What happened?

Inside `__add__` we convert *other* to Rational – but *other* is already a Rational.

Invoking Rational constructor with a Rational argument generates an unexpected object:

```
Rational(p)
```

```
2/3/1
```

Solution: convert only  
for *other* of type int!

# Revised `__add__`

- The built-in function `isinstance` takes a value and a class object, and returns True if the value is an instance of the class (False otherwise)
- **We only convert the argument is of type int!**

```
def __add__(self, other):  
    if isinstance(other, int):  
        other = Rational(other)  
    # immutable!  
    return Rational(self.numer * other.denom +  
                    other.numer * self.denom,  
                    self.denom * other.denom)
```

# Revised \_\_add\_\_

```
p = Rational(2,3)  
p + 2
```

8/3



```
p = Rational(2,3)  
p + p
```

12/9

Will  $p + 2.1$  work?

If we want Rational to be used just like any other numeric data type in respect to arithmetic operations, we must implement this behavior for **every operator** and **every data type** that is supported.

# Implicit Conversions

- Now lets make `int + Rational` work:
- When we perform `2 + p`, which `__add__` method invoked?
- `2 + p`  $\rightarrow$  `2.+(p)`: (`int`)  $\rightarrow$  `int` class contains no `__add__` method that takes a `Rational` argument ☹
- The problem: `int` class does not contains an `__add__` method that takes a `Rational` argument ☹
- We can not change the implementation of `int`, so we should revise the class `Rational`.
- Is there a method that is invoked when `Rational` is on the **right** side of the operator?

# `__radd__` - right side add

`__radd__` is invoked when a Rational object appears on the right side of the `+` operator

```
def __radd__(self, other):  
    return self + other
```

Rational + int works ☺

```
>>> p = Rational(2,3)  
>>> 2+p  
8/3
```

Implementing `__radd__` using `__add__` is correct because addition is commutative:  **$a+b = b+a$**

# Arithmetic with Rational numbers

- We now support both `int + Rational` and `Rational + int`, so we may execute complex arithmetic operations and use built-in functions:

```
1 + Rational(1,2) + Rational(3,4) + 3
```

```
42/8
```

```
sum([1,Rational(1,2),Rational(3,4),3])
```

```
42/8
```

# Comparisons

```
x = Rational(1,2)
y = Rational(1,3)
x < y
```

```
-----
TypeError                                Traceback (most recent call last)
<ipython-input-60-426754ac2c97> in <module>()
      1 x = Rational(1,2)
      2 y = Rational(1,3)
----> 3 x < y
```

TypeError: '<' not supported between instances of 'Rational' and 'Rational'

What methods are used for comparison?

Operator	Method
>>>x == y	<code>__eq__(self,other)</code>
>>>x < y	<code>__lt__(self,other)</code>
>>>x <= y	<code>__le__(self,other)</code>
>>>x > y	<code>__gt__(self,other)</code>
>>>p >= q	<code>__ge__(self,other)</code>

# Comparisons

```
def __eq__(self, other):  
    return self.numer == other.numer and self.denom == other.denom  
def __lt__(self, other):  
    return self.numer * other.denom < self.denom * other.numer  
def __le__(self, other):  
    return self.numer * other.denom <= self.denom * other.numer  
def __gt__(self, other):  
    return self.numer * other.denom > self.denom * other.numer  
def __ge__(self, other):  
    return self.numer * other.denom >= self.denom * other.numer
```



# Equality: Rational and int

```
r1 = Rational(6,3)
```

```
r1==2
```

Traceback (most recent call last):

File "<pyshell#2>", line 1, in <module>

```
r1==2
```

File "D...", line 27, in \_\_eq\_\_

```
    return self.numer == other.numer and self.denom ==  
other.denom
```

AttributeError: 'int' object has no attribute 'n'

This does not surprise us anymore...

# Comparisons

```
x = Rational(1,2)  
y = Rational(1,3)  
x > y
```

True

```
x < y
```

False

```
x >= y
```

True

```
x <= y
```

False

```
max(x,y)
```

1/2



Why does `max` work?

# Internal representation

```
Rational(8,6) == Rational(4,3)
```

False

- But 8/6 is equal to 4/3!
- We need to normalize the rational numbers to be able to compare them properly
- To normalize we divide the numerator and denominator by their *greatest common divisor* (*gcd*)
- $\text{gcd}(8,6) = 2 \rightarrow (8/2)/(6/2) = 4/3$

No need for Rational users to be aware - *Encapsulation*

# Revised Rational

```
def __init__(self,n,d=1):  
    """n: numerator  
    d: denominator"""  
    if d == 0:  
        raise ZeroDivisionError("Denominator cannot be zero")  
    gcd = math.gcd(n,d)  
    self.numer = n//gcd  
    self.denom = d//gcd
```

```
Rational(8,6) == Rational(4,3)
```

True



The change in the constructor influences more than just comparisons

```
Rational(8,6)
```

4/3

How can we maintain both the un-normalized representation and the correct comparison?

# Rational numbers in Python

- Actually, there is a Python implementation of Rational numbers
- It is called fractions

<http://docs.python.org/library/fractions.html>

## fractions — Rational numbers

Source code: [Lib/fractions.py](#)

---

The `fractions` module provides support for rational number arithmetic.

A `Fraction` instance can be constructed from a pair of integers, from another rational number, or from a string.

```
class fractions.Fraction(numerator=0, denominator=1)
class fractions.Fraction(other_fraction)
class fractions.Fraction(float)
class fractions.Fraction(decimal)
class fractions.Fraction(string)
```

# Recap

- We implemented a new class that naturally represents Rational numbers
  - Attributes, methods, constructor
  - Method overriding
  - Encapsulation
  - Define operators as method
  - Polymorphism
  - Automatic type conversion

# **Different comparison criteria**

The most awesome Beatle

# Class Beatle

```
class Beatle:
    def __init__(self, name, popularity, awesomeness):
        self.name = name
        self.popularity = popularity
        self.awesomeness = awesomeness
    def __repr__(self):
        return self.name + " (" + \
            str(self.popularity) + "," + \
            str(self.awesomeness) + ")"
beatles = []
beatles.append(Beatle('John', 10, 7))
beatles.append(Beatle('Paul', 9, 8))
beatles.append(Beatle('George', 7, 10))
beatles.append(Beatle('Ringo', 5, 5))
```

```
beatles
```

```
[John (10,7), Paul (9,8), George (7,10), Ringo (5,5)]
```

How can we find out who is the most popular Beatle and who is the most awesome?



# Who is the most popular Beatle? The most awesome?

- Easy! We can add `__lt__` implementation and use `max`
- Is it good enough?
- No, when we implement `__lt__` we commit to one comparison/order criterion

# key argument for max

- `max(iterable[, key])`

*key* – an optional argument for max function.

If specified, *key* is applied on each element of *iterable*

*max* returns the element *elem* such that `key(elem)` is the maximum value among the other elements in *iterable*.

# key argument for max

```
lst = ['abc', 'c', 'bb']  
max(lst)
```

'c'


```
max(lst, key=len)
```

'abc'

```
int_lst = [1, 12, 3]  
max(int_lst, key=len)
```

```
-----  
TypeError                                Trace  
<ipython-input-108-06e610b31212> in <module>()  
      1 int_lst = [1, 12, 3]  
----> 2 max(int_lst, key=len)
```

TypeError: object of type 'int' has no len()



**key** is applied on  
each list **element**

# Compare according to different criteria

```
def popularity_key(beatle):  
    return beatle.popularity
```

```
def awesomeness_key(beatle):  
    return beatle.awesomeness
```

```
max(beatles, key=popularity_key)
```

John (10,7)



The most popular Beatle

```
max(beatles, key=awesomeness_key)
```

George (7,10)



The most awesome Beatle

# Also works for min and sort!

```
min(beatles,key=popularity_key)
```

```
Ringo (5,5)
```

```
min(beatles,key=awesomeness_key)
```

```
Ringo (5,5)
```

```
sorted(beatles,key=popularity_key)
```

```
[Ringo (5,5), George (7,10), Paul (9,8), John (10,7)]
```

```
sorted(beatles,key=awesomeness_key)
```

```
[Ringo (5,5), John (10,7), Paul (9,8), George (7,10)]
```

**Encapsulation – hiding the  
unnecessary**

# Encapsulation – hiding the unnecessary

- Preventing accident erroneous modifications by restricting access to methods and variables
- Class Car has two methods: drive() and update\_software()
  - update\_software will be implemented as a **private** method **\_\_update\_software()**, which is invoked only when a new car is created, and not from the object directly

# A private method

```
class Car:
    def __init__(self):
        self.__update_software()

    def drive(self):
        print('driving')

    def __update_software(self):
        print('updating software')

redcar = Car()
redcar.drive()
```

updating software  
driving



# Private methods are not accessible from the object

```
redcar.__updateSoftware()  # not accesible from object.
```

```
-----  
AttributeError                                Traceback (most recent call last)  
<ipython-input-47-32cfbc497406> in <module>  
----> 1 redcar.__updateSoftware()  # not accesible from object.  
  
AttributeError: 'Car' object has no attribute '__updateSoftware'
```

# Encapsulation prevents from accessing accidentally, but not intentionally

```
redcar._Car__update_software()
```

updating software

# A private field

Objects can hold crucial data for your application and you do not want that data to be changeable from anywhere in the code. Can only be changed within a class method and not outside of the class.

```
class Car:

    __maxspeed = 0
    __name = ""

    def __init__(self):
        self.__maxspeed = 200
        self.__name = "Supercar"

    def drive(self):
        print('driving. maxspeed ' + str(self.__maxspeed))

redcar = Car()
redcar.drive()
redcar.__maxspeed = 10  # will not change variable because its private
redcar.drive()
```

```
driving. maxspeed 200
driving. maxspeed 200
```

Source: <https://pythonspot.com/encapsulation/>

# Encapsulation prevents from accessing accidentally, but not intentionally

```
redcar._Car__maxspeed = 20  
redcar.drive()
```

```
driving. maxspeed 20
```

# setter - a method that sets the value of a private variable

```
class Car:

    __maxspeed = 0
    __name = ""

    def __init__(self):
        self.__maxspeed = 200
        self.__name = "Supercar"

    def drive(self):
        print('driving. maxspeed ' + str(self.__maxspeed))

    def set_max_speed(self, speed):
        self.__maxspeed = speed

redcar = Car()
redcar.drive()
redcar.set_max_speed(320)
redcar.drive()
```

```
driving. maxspeed 200
driving. maxspeed 320
```

# Encapsulating the implementation of a list-like data structure with a dictionary

```
lst1 = SecretList()
lst1[0] = 'zero'
lst1[1] = 'one'
lst1[7] = 'seven'
lst1[5] = 'five'
print(lst1)
lst1[5]
```

```
['zero', 'one', , , , 'five', , 'seven']
```

```
'five'
```

```
(lst1[-1], lst1[-3], len(lst1))
```

```
('seven', 'five', 4)
```

# SecretList implantation

```
class SecretList:
    def __init__(self):
        self.__list = {}

    def __getitem__(self, key):
        if type(key) == int:
            if key < 0:
                if len(self.__list) > 0:
                    key = max(self.__list) + key + 1
            if key in self.__list:
                return self.__list[key]
            else:
                print(key)
                raise IndexError ("list index undefined")
        else:
            raise TypeError ("list indices must be integers or slices, not", type(key))
```

# SecretList implantation

```
def __setitem__(self, key, val):
    if type(key) == int:
        if key < 0:
            if len(self.__list) > 0:
                key = max(self.__list) + key + 1
                if key < 0:
                    raise IndexError("list assignment out of range")
            self.__list[key] = val
        else:
            raise TypeError ("list indices must be integers or slices, not", type(key))

def __delitem__(self, key):
    if self[key]:
        del self.__list[key]
```



# SecretList implentation

```
def __repr__(self):
    ret_val = '['
    if len(self.__list) > 0:
        for i in range(max(self.__list)+1):
            if i in self.__list:
                ret_val += self.__list[i].__repr__()
            ret_val += ', '
        return ret_val[:-2] + ']'
    else:
        return '[]'

def __len__(self):
    return len(self.__list)
```

# Characteristics of the Object-Oriented Paradigm



# Inheritance

# Use case: secret agents

We will design and develop a system to manage information about secret agents over the world

- Classes: types of agents
- Objects/instances: individual agents

# Agency requirements I

- An agent:
  1. Name
  2. Age
  3. List of visas to enable travel
  4. Current location (HQ as default)
- An agent can:
  - travel on assignments to other countries (upon visa)

# Class agent: design

- **Attributes:** name, age, visas list, location
  - Default location: headquarters (HQ).
- **Constructor:** name and age.
- **Methods:**
  1. *add\_visa(self, country)*: add *country* to the list of visas in case it is not already there.
  2. *send\_to(self, country)*: check if agent can be sent to *country*. If so, change location and return *True*. Otherwise, return *False*.
  3. *\_\_repr\_\_(self)*

# Class Agent: implementation

```
class Agent:
    def __init__(self, name, age):
        self.name = name
        self.age = age
        self.visas = []
        self.location = 'HQ'
```

```
    def __repr__(self):
        result = self.name + " , " + str(self.age) + "\n"
        result += "["
        for country in self.visas:
            result += country + " "
        result += "]\n"
        result += "Now in: " + self.location + "\n"
        return result
```

# Class Agent – Implementation (Cont'd)

```
def add_visa(self, country):  
    if country not in self.visas:  
        self.visas.append(country)
```

```
def send_to(self, country):  
    if country in self.visas:  
        self.location = country  
        return True  
    return False
```



# Using the class Agent

```
a=Agent("David",25)  
print(a)
```

```
David , 25  
[]  
Now in: HQ
```

```
a.add_visa("Russia")  
a
```

```
David , 25  
[Russia ]  
Now in: HQ
```

```
a.send_to("Brazil")
```

```
False
```

```
a.send_to("Russia")
```

```
True
```

```
a
```

```
David , 25  
[Russia ]  
Now in: Russia
```

# Is that all?

- Create an agent
- Test representation string
- Add visas
- Send between countries
- Send to and from HQ

**Sending an agent to HQ is always valid!**

# Adding return to HQ

```
def send_to(self, country):  
    if country=='home' or country=='HQ':  
        self.location = 'HQ'  
        print("Returning to HQ")  
        return True  
    if country in self.visas:  
        self.location = country  
        return True  
    return False
```

```
a=Agent("David",25)  
print(a)
```

```
David , 25  
[]  
Now in: HQ
```

```
a.add_visa("Russia")  
a
```

```
David , 25  
[Russia ]  
Now in: HQ
```

```
a.send_to("Russia")
```

```
True
```

```
a
```

```
David , 25  
[Russia ]  
Now in: Russia
```

```
a.send_to('HQ')  
print(a)
```

```
Returning to HQ  
David , 25  
[Russia ]  
Now in: HQ
```

# Requirements II: Special Agents

- Not all agents are the same
- Some are **Special Agents**
  - Special agents have a **rank** (attribute)
  - Special agents can be **promoted** (method)
- Regular agents:
  - Have no rank
  - Cannot be promoted

# Solution 1: Dummy Attributes

- Add attribute *rank* to **every** Agent
- Assign dummy value (-1) for regular agents
- Verify *rank* validity before every access to it

For example:

```
def promote(self):  
    if self.rank < 0:  
        raise Exception("...")  
    self.rank += 1
```

# Dummy Attributes

## Pros

- Easy to implement

## Cons

- Add code to check validity of rank
- What happens if more attributes are added?
- Bad design



## Solution 2: new class, duplicated code

- Create a new class called **SpecialAgent**
  1. **Copy all code** from Agent Class
  2. Then:
    - Add *rank* variable
    - Update `__repr__` method
    - Add *promote* method

# Class SpecialAgent (1)

```
class SpecialAgent:                                #Added:
    def __init__(self, name, age, rank):
        self.name = name
        self.age = age
        self.visas = []
        self.location = 'HQ'
        ##### ADDED: #####
        self.rank = rank
        #####

    def __repr__(self):
        result = self.name + " , " + str(self.age) + "\n"
        result += "["
        for country in self.visas:
            result += country + " "
        result += "]\n"
        result += "Now in: " + self.location + "\n"
        ##### ADDED: #####
        result += "Rank:" + str(self.rank) + "\n"
        #####
        return result

    def add_visa(self, country):
        if country not in self.visas:
            self.visas.append(country)

    def send_to(self, country):
        if country in self.visas:
            self.location = country
            return True
        return False

    ##### ADDED: #####
    def promote(self):
        self.rank+=1
        #####
```



# Special Agent demo

```
english = SpecialAgent("Johnny English", 43, 7)  
english
```

```
Johnny English , 43  
[]  
Now in: HQ  
Rank:7
```

```
english.promote()  
english
```

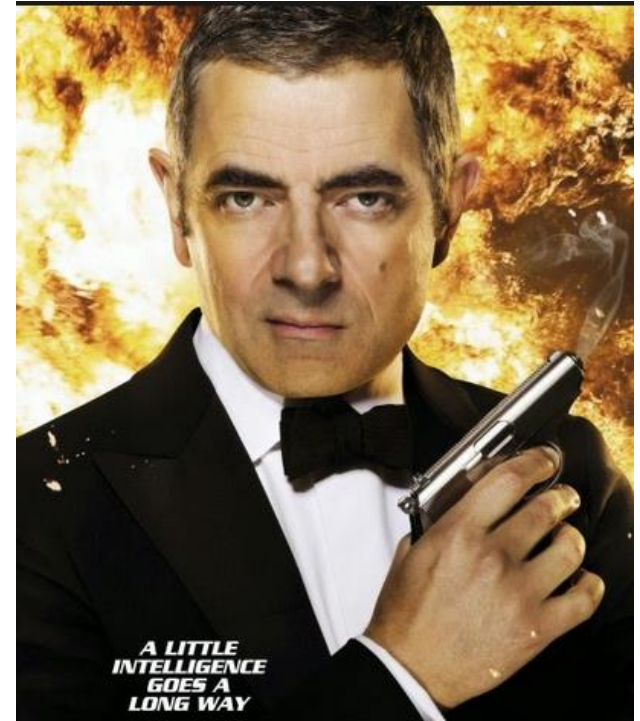
```
Johnny English , 43  
[]  
Now in: HQ  
Rank:8
```

## Pros

- Very easy to implement

## Cons

- Code duplication is bad!



# Bad: code duplication

- *SpecialAgent* is almost identical to *Agent*
- Code duplication – **very bad practice!**
  - Bug → multiple bugs
  - Change → multiple changes
  - Change → potential new bugs

# Making changes in duplicated code: example

- Please change the word “now” to “currently”:

```
>>> a = Agent("Sarah", 30)
```

```
>>> a
```

```
Sarah, 30
```

```
[]
```

```
Currently in: HQ
```

```
def __repr__(self):  
    result = self.name + " , " + str(self.age) + "\n"  
    result += "["  
    for country in self.visas:  
        result += country + " "  
    result += "]\n"  
    result += "Now in: " + self.location + "\n"  
    return result
```

change  
here

# Encapsulation

Implementation details are hidden

Is it sufficient to change Agent class and update its `__repr__` method?

# Testing New Agent `__repr__`

```
>>> a = Agent("Sarah", 40)
```

```
>>> a
```

```
Sarah, 40
```

```
[]
```

```
Currently in: HQ
```



# What about Special Agents?

```
>>> sa = SpecialAgent("Commander Bond", 37, 9)
```

```
>>> sa
```

```
Commander Bond, 37
```

```
[]
```

```
Now in: HQ
```

```
Level: 9
```



Wrong!!

# Code duplication vs. reuse

- Duplication: **hard to maintain**
  - Fixing bugs: in several places
  - Adding new features: in several places
- **Code reuse** makes software development **easier**:
  - Code is written once
  - Bugs are fixed **in one place only**
  - Features are added **in one place**
  - Code is easier to understand

# Idea: class containment

- A special agent contains an attribute of type agent
- All operations are done through the agent:
  - Implement the same methods in every class
- Extensive code duplication



# Special agent *is an* agent

- A special agent is a type of an agent
- A special agent **is-an** agent

*called: an **is-a** relationship*

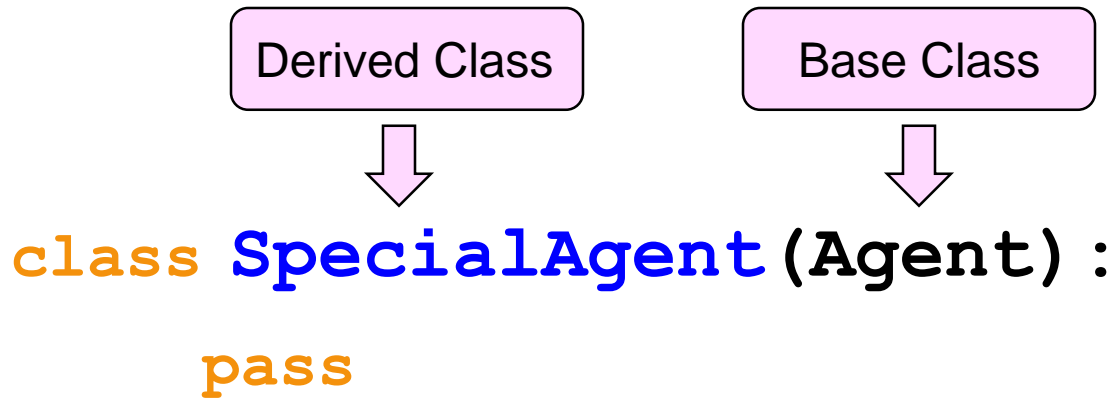
- Not all classes relate likewise:
  - Point is not a Circle
    - a **has-a relationship**: Circle **has-a** Point (containment)
  - A car is not an agent

# Implementation: Class inheritance

we use Inheritance to reflect an *is-a* relationship

- Abstractly:
  - B *is-a* type of A
  - B is a subtype of A
- In program design:
  - Class B *extends* class A
  - Class B **inherits** from class A
  - Class A is said to be a **base/parent/super** class
  - Class B is said to be a **derived/inherited** class

# Inheritance: class *SpecialAgent*



Special Agent is a derived class of Agent

# Code reuse

Before adding new code, all code is inherited:

```
>>> sa = SpecialAgent("Sarah", 40)
```

```
>>> sa
```

```
Sarah, 40
```

```
[ ]
```

```
Now in: HQ
```

# Overriding methods

In order to add the *rank* attribute we redefine the constructor `__init__` in *SpecialAgent* class:

```
class SpecialAgent(Agent):  
    def __init__(self, name, age, rank):  
        self.name = name  
        self.age = age  
        self.visas = [ ]  
        self.location = "HQ"  
        self.rank = rank
```

The new implementation overrides the constructor in the base class *Agent*

# Code reuse and overriding

Problem: the constructor still duplicates code!

Solution: call the *Agent*'s constructor, with changes as required

# Special Agent Constructor

```
class SpecialAgent (Agent) :
```

```
    def __init__(self, name, age, rank) :
```

```
        Agent.__init__(self, name, age)
```

```
        self.rank = rank
```

SpecialAgent  
specific code

Explicit call  
to Agent *init*

# Adding functionality

Derived classes can have additional attributes and methods that base classes do not have.

For example, the SpecialAgent class will have a method to promote an agent.



# Rules for subclass constructor

Do one of the following:

## 1. Use parent's constructor

```
class SpecialAgent (Agent) :  
    pass
```

- Don't write any code
- Parent's constructor invoked automatically

## 2. Write a new constructor

- First invoking the parent's constructor
- Then add fields, changes, etc.

```
class SpecialAgent (Agent) :  
    def __init__ (self, name, age, rank) :  
        Agent.__init__ (self, name, age)
```

# Rules for inherited methods

**Always** use inherited methods when possible

```
class SpecialAgent( Agent ):  
  
    def __init__( self, name, age, rank ):  
        Agent.__init__(self, name, age)  
        self.rank=rank  
  
    def __repr__( self ):  
        res = Agent.__repr__(self)  
        res += "Rank: " +str(self.rank) + "\n"  
        return res  
  
    def promote(self):  
        self.rank += 1
```

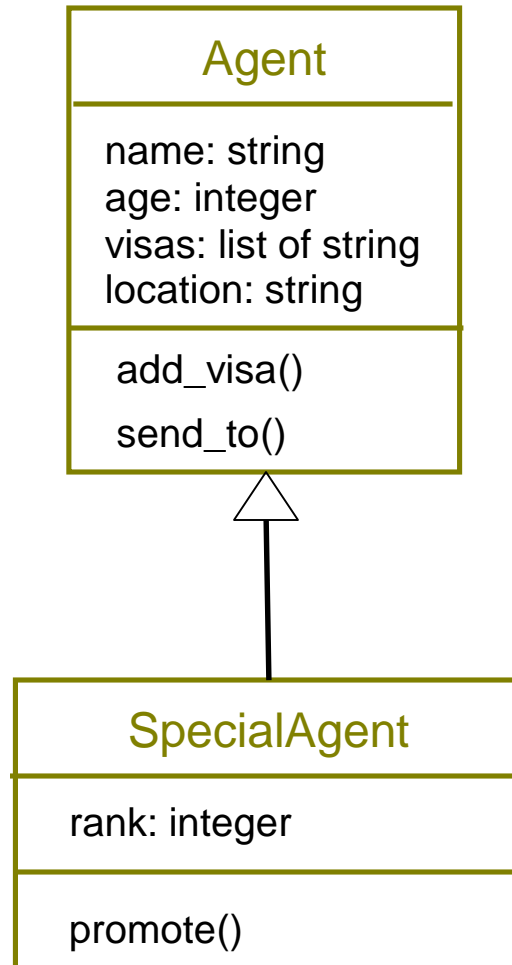
Call for **Agent's**  
class methods

# Special Agent Demo

```
>>> sa = SpecialAgent("Ronit", 21, 8)
>>> sa
Ronit , 21
[]
Now in: HQ
Rank: 8

>>> sa.add_visa("Israel")
>>> sa.send_to("Israel")
True
>>> sa.promote()
>>> sa
Ronit , 21
[Israel ]
Now in: Israel
Rank: 9
```

# Class Diagram



# Remember Polymorphism?

```
class Cat:
    def talk(self):
        return 'Meow!'

class Dog:
    def talk(self):
        return 'Woof! Woof!'

animals = [Cat(), Dog(), Cat()]
for anim in animals:
    print anim.talk()
```



# Polymorphism revisited

- All agents (including special) can be treated the same way:

```
bond = SpecialAgent("James Bond", 40, 7)
english = SpecialAgent("Johnny English", 43, 6)
cohen = Agent("Nissim Cohen", 25)
```

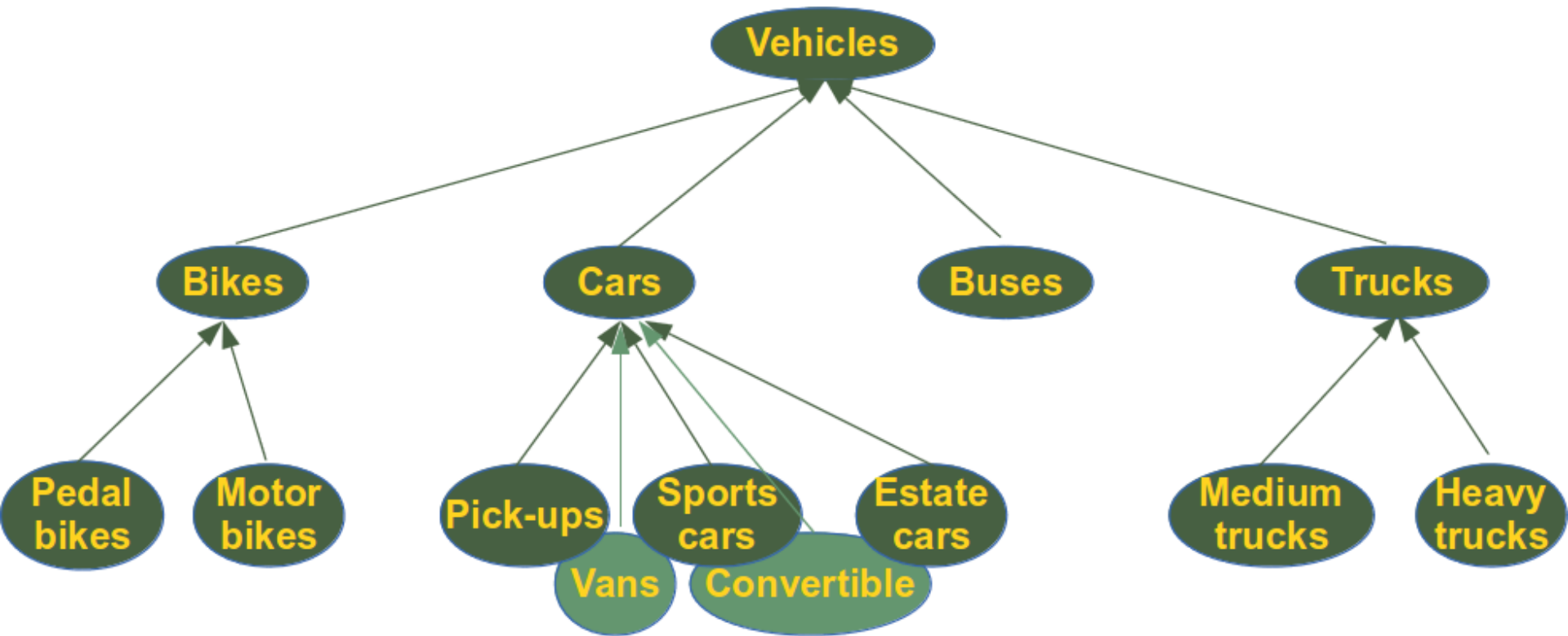
```
agents = [bond, english, cohen]
for agent in agents:
    agent.add_visa("Israel")
    agent.send_to("Israel")
    print(agent)
```

```
James Bond , 40
[Israel ]
Now in: Israel
Rank: 7
```

```
Johnny English , 43
[Israel ]
Now in: Israel
Rank: 6
```

```
Nissim Cohen , 25
[Israel ]
Now in: Israel
```

# Inheritance



# More Requirements

- Some agents are **Restricted Agents**
- Cannot have more than 5 visas
- How should we add this functionality?



# Overriding

**Overriding** can be used also to modify class behavior completely (rather than just adding)

```
class RestrictedAgent (Agent):  
  
    def add_visa(self, country):  
        if country in self.visas: return  
        if len(self.visas) < 5:  
            Agent.add_visa(self, country)  
        else:  
            print("Cannot add", country, ":Visa limit exceeded!")
```

# Overriding: example

```
class RestrictedAgent (Agent):  
  
    def add_visa(self, country):  
        if country in self.visas: return  
        if len(self.visas) < 5:  
            Agent.add_visa(self, country)  
        else:  
            print("Cannot add", country, ":Visa limit exceeded!")
```

```
jason = RestrictedAgent("Jason Bourne", 50)  
jason
```

Jason Bourne , 50

[]

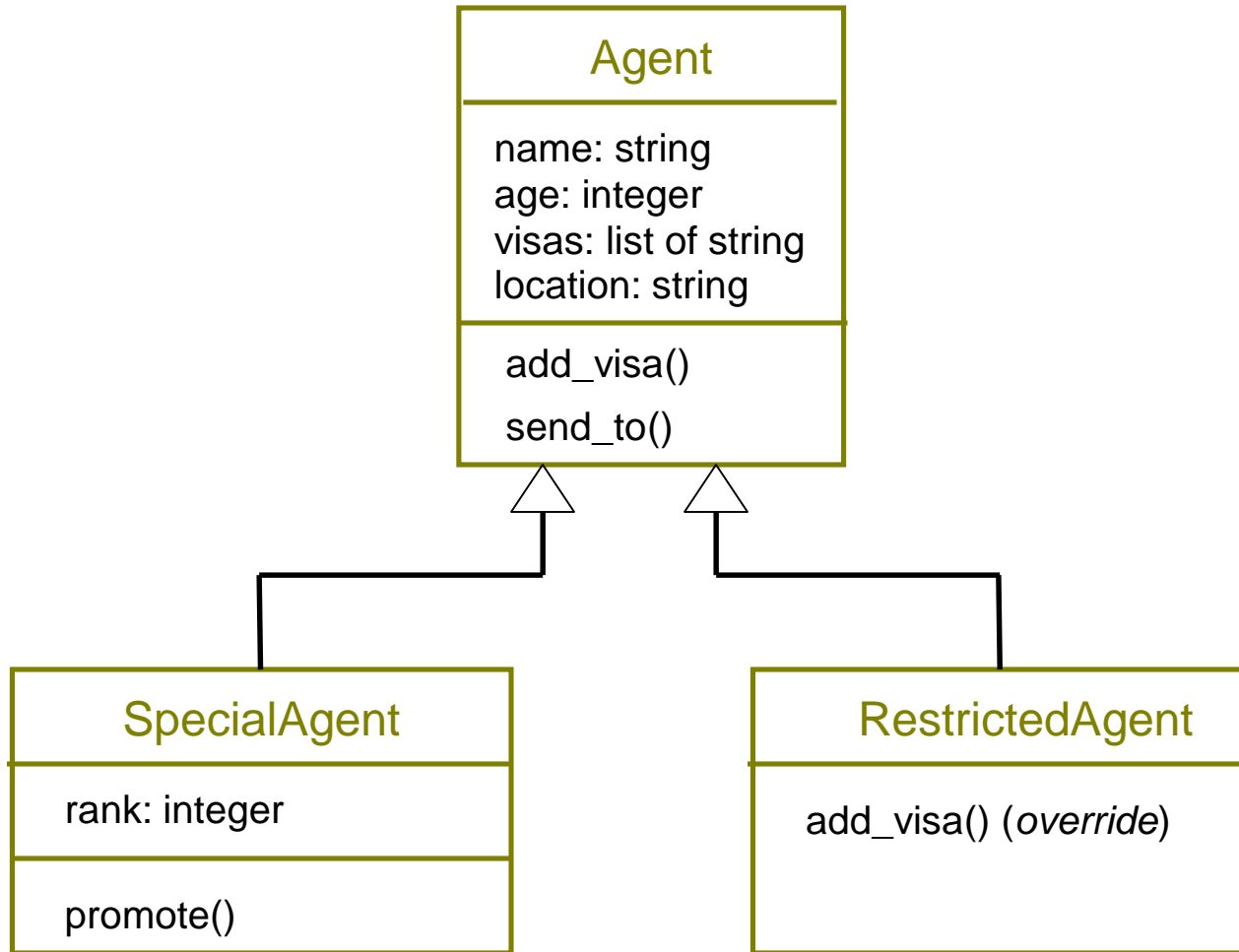
Now in: HQ

```
countryList = "Israel Japan Russia Sweden Iraq".split(" ")  
[jason.add_visa(country) for country in countryList]  
jason.add_visa("China")  
jason.add_visa("Israel")  
jason.add_visa("Finland")
```

Cannot add China :Visa limit exceeded!

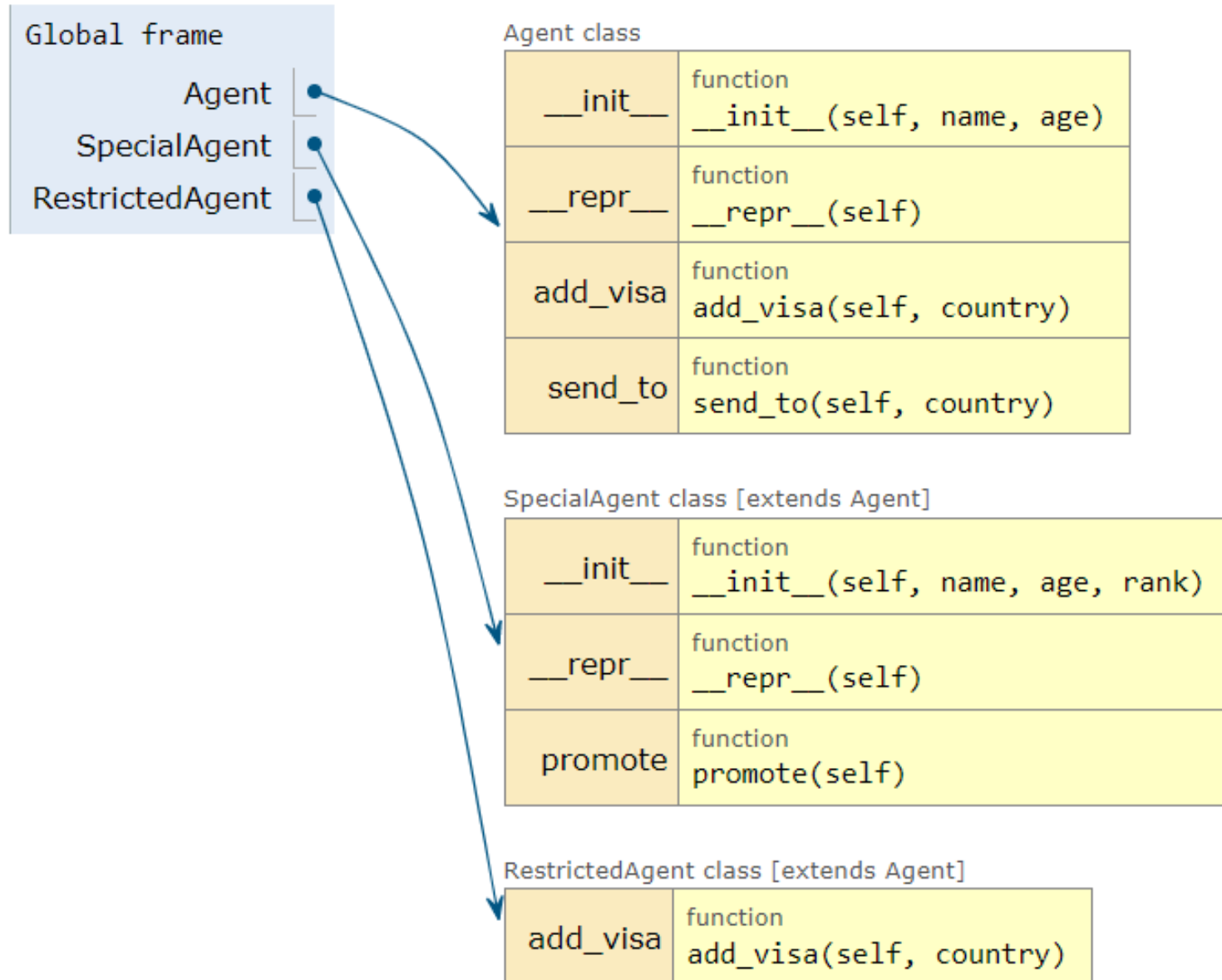
Cannot add Finland :Visa limit exceeded!

# Class diagram



## Frames

## Objects



# Task force

- Establish a **task force** of agents
- Several agent types can be in one task force
- All task force agents receive visas together
- All task force agents are sent together



# Task force

```
deadpool = Agent("Wade Wilson", 30)
bond = SpecialAgent("James Bond", 40, 7)
english = SpecialAgent("Johnny English", 43, 6)
jason = RestrictedAgent("Jason Bourne", 25)
```

Special Agent **bond**

*send\_to(state)*

Special Agent **english**

*send\_to(state)*

Restr. Agent **jason**

*send\_to(state)*

Agent **deadpool**

*send\_to(state)*

# Agents polymorphism

```
class TaskForce():
    def __init__(self, agents):
        self.agents = agents
        print("TaskForce established. Agent names:")
        for agent in self.agents:
            print("\t", agent.name)

    def add_visas(self, country):
        for agent in self.agents:
            agent.add visa(country)

    def send_agents(self, country):
        for agent in self.agents:
            print("sending agent " + agent.name)
            agent.send_to(country)
            if agent.location == country:
                print("...location verified")
            else: print("Error")

    def deploy(self, country):
        print("Deploying to " + country)
        print("Adding Visas...")
        self.add_visas(country)
        print("Sending Agents...")
        self.send_agents(country)
        print("Team Deployed.")
```

# Deploy task force

```
ateam = TaskForce([bond, english, deadpool, jason])
```

TaskForce established. Agent names:

James Bond  
Johnny English  
Wade Wilson  
Jason Bourne

```
1 ateam.deploy("Iraq")
```

Deploying to Iraq  
Adding Visas...  
Sending Agents...  
sending agent James Bond  
...location verifyied  
sending agent Johnny English  
...location verifyied  
sending agent Wade Wilson  
...location verifyied  
sending agent Jason Bourne  
...location verifyied  
Team Deployed.



# Villains



Pretend to be Agents, without the methods

```
#First Villain - onlt data, no send_to or add_visa
class Villain1:
    def __init__(self, name, age, hometown):
        self.name = name
        self.age = age
        self.visas = []
        self.location = hometown
    def __repr__(self):
        result = self.name + " , " + str(self.age) + "\n"
        result += "[I am everywhere!]\n"
        result += "Now in: " + self.location + "\n"
        return result
```

# Can vilans infiltrate the task force?

```
theJoker = Villain1( 'Joker', "Old", "Gotham" )  
theJoker
```

```
Joker , Old  
[I am everywhere!]  
Now in: Gotham
```



```
ateam = TaskForce([bond, deadpool, jason, theJoker])
```

TaskForce established. Agent names:

James Bond  
Wade Wilson  
Jason Bourne  
Joker

# Can the Joker join the mission?

```
ateam.deploy("Hawaii")
```

```
Deploying to Hawaii  
Adding Visas...
```

```
-----  
AttributeError                                Traceback (most recent call last)  
<ipython-input-165-0d1adab62174> in <module>()  
----> 1 ateam.deploy("Hawaii")
```

```
<ipython-input-145-4cbdda67764a> in deploy(self, country)  
    21     print("Deploying to " + country)  
    22     print("Adding Visas...")  
--> 23     self.add_visas(country)  
    24     print("Sending Agents...")  
    25     self.send_agents(country)
```

```
<ipython-input-145-4cbdda67764a> in add_visas(self, country)  
     8     def add_visas(self, country):  
     9         for agent in self.agents:  
--> 10             agent.add_visa(country)  
    11  
    12     def send_agents(self, country):
```

```
AttributeError: 'Villain1' object has no attribute 'add_visa'
```

# Villains – take II



Pretending to be an Agent, including the methods

```
class Villain2:
    def __init__(self, name, age, hometown):
        self.name = name
        self.age = age
        self.visas = []
        self.location = hometown
    def __repr__(self):
        result = self.name + " , " + str(self.age) + "\n"
        result += "I am Evil Bwhaha...\n"
        result += "Now in: " + self.location + "\n"
        return result
    def add_visa(self, country):
        pass
    def send_to(self, country):
        self.location = country
        return True
```

# Can Voldemort join a mission?

```
voldemort = Villain2("Tom M. Riddle", "Who Knows?", "London")  
voldemort
```

```
Tom M. Riddle , Who Knows?  
I am Evil Bwhaha...  
Now in: London
```

```
bteam = TaskForce([bond, deadpool, jason, voldemort])
```

```
TaskForce established. Agent names:  
    James Bond  
    Wade Wilson  
    Jason Bourne  
    Tom M. Riddle
```

```
bteam.deploy("Caribbean")
```

```
Deploying to Caribbean  
Adding Visas...  
Sending Agents...  
sending agent James Bond  
...location verified  
sending agent Wade Wilson  
...location verified  
sending agent Jason Bourne  
...location verified  
sending agent Tom M. Riddle  
...location verified  
Team Deployed.
```



# How to control such cases?

- Python does not offer static typing
  - The compiler doesn't catch villains...
- We have to check types in runtime
  - So we can control our program's execution
- Task Force example:
  - Do not allow non-agents to join

# *isinstance* and *issubclass*

- *isinstance* checks the type of an instance
  - Also succeeds if the instance belongs to a subclass (i.e., inherited)
  - Accepts an object and a class as arguments
- *issubclass* checks class inheritance
  - Accepts two classes as arguments

# isinstance

```
deadpool = Agent("Wade Wilson", 30)
bond = SpecialAgent("James Bond", 40, 7)
jason = RestrictedAgent("Jason Bourne", 25)
theJoker = Villain1( 'Joker', "Old", "Gotham" )
```

```
isinstance(deadpool, Agent)
```

True

```
isinstance(deadpool, SpecialAgent)
```

False

```
[isinstance(bond, SpecialAgent), isinstance(bond, Agent)]
```

[True, True]

```
[isinstance(jason, SpecialAgent), isinstance(jason, RestrictedAgent)]
```

[False, True]



# issubclass

```
[issubclass(Agent, SpecialAgent), \  
issubclass(SpecialAgent, Agent), \  
issubclass(RestrictedAgent, Agent), \  
issubclass(RestrictedAgent, SpecialAgent)]
```

```
[False, True, True, False]
```

# Protecting the task force I

```
class TaskForce():
    def __init__(self, agents):
        self.agents=[]
        for new_agent in agents:
            if isinstance(new_agent, Agent):
                self.agents.append(new_agent)
            else:
                print(new_agent.name + " is not an agent!")

        print("TaskForce established. Agent names:")
        for agent in self.agents:
            print("\t",agent.name)
```

# Protecting the task force II

```
class TaskForce():
    def __init__(self, agents):
        self.agents=[]
        for new_agent in agents:
            assert(isinstance(new_agent, Agent))

        self.agents = agents

        print("TaskForce established. Agent names:")
        for agent in self.agents:
            print("\t",agent.name)
```

```
bteam = TaskForce([bond, deadpool, jason, voldemort])
```

---

```
AssertionError                                Traceback (most recent call 1
<ipython-input-205-0ad47c1a26df> in <module>()
----> 1 bteam = TaskForce([bond, deadpool, jason, voldemort])

<ipython-input-204-e9788760e5c0> in  init  (self, agents)
```

# Agents: more requirements

Given an agent, get the list of permitted visas.

That's easy! Just access the *visas* attribute!

```
>>> a = Agent("Josh", 22)
```

```
>>> a.visas
```

```
[]
```

# Encapsulation violation

- Separate functionality from implementation!
- The user should not be aware of internal changes of the class implementation.
- Encapsulation

# Encapsulation: private attributes

- To hide attributes, use the `__` prefix:
- For example, let's make visa and location private

```
class PrivateAgent:
    def __init__(self, name, age):
        self.name = name
        self.age = age
        self.__visas = []
        self.__location = 'HQ'
    def __repr__(self):
        result = self.name + " , " + str(self.age) + "\n"
        result += "["
        for country in self.__visas:
            result += country + " "
        result += "]\n"
        result += "Now in: " + self.__location + "\n"
        return result
```

(there would not be two classes *Agent* and *PrivateAgent*. *Agent* would already use encapsulation.)

# Private attributes

- Cannot directly access `__visas` from outside:

```
james = PrivateAgent("James Bond", 45)
```

```
james.add_visa("USA")  
james
```

```
James Bond , 45  
[USA ]  
Now in: HQ
```

```
james.visas
```

```
-----  
AttributeError                                Trace  
<ipython-input-235-d520def960d4> in <module>()  
----> 1 james.visas
```

```
AttributeError: 'PrivateAgent' object has no at
```

```
james.__visas
```

```
-----  
AttributeError                                Trace  
<ipython-input-236-1a3638fafa2a> in <module>()  
----> 1 james.__visas
```

```
AttributeError: 'PrivateAgent' object has no at
```

# Workaround

- Actually this can be “hacked”:

```
james = PrivateAgent("James Bond", 45)
```

```
james.add_visa("USA")  
james
```

```
James Bond , 45  
[USA ]  
Now in: HQ
```

```
james._PrivateAgent__visas
```

```
['USA']
```



# The right way

- Add a *getter* Method:

```
def send_to(self, country):  
    if country=='home' or country=='HQ':  
        self.__location = 'HQ'  
        print("Returning to HQ")  
        return True  
    if country in self.__visas:  
        self.__location = country  
        return True  
    return False  
def get_visas(self):  
    return self.__visas
```

```
james = PrivateAgent("James Bond", 45)
```

```
james.get_visas()
```

```
['USA']
```

# Changing visas externally

- Can change the visas outside the class:

```
james.get_visas()
```

```
['USA']
```

```
visas=james.get_visas()
```

```
#We can ruin the visas
```

```
visas[0] = "Gotcha!"
```

```
print(james.get_visas())
```

```
['Gotcha!']
```

- How to solve this? Copy..

# Protecting private data

- Return a copy of the data

```
import copy
class PrivateAgent:
    ...
    def get_visas(self):
        return copy.copy(self.__visas)
```

- And now:

```
visas = james.get_visas()
visas[0]="Bwhaha"
print(james)
```

```
James Bond , 45
[USA ]
Now in: HQ
```

# Inheriting a private class

- Let's define RestrictedPrivateAgent, inheriting from PrivateAgent

```
class RestrictedPrivateAgent(PrivateAgent):  
  
    def add_visa(self, country):  
        if country in self.__visas: return  
        if len(self.__visas) < 5:  
            PrivateAgent.add_visa(self, country)  
        else:  
            print("Cannot add", country, ":Visa limit exceeded!")
```

```
jason = RestrictedPrivateAgent("Jason Bourne", 50)  
jason
```

Jason Bourne , 50

[]

Now in: HQ

# Inheriting a private class

- Let's invoke `add_visa()`
  - this method tries to access `self.__visas`

```
jason.add_visa("USA")
```

-----  
**AttributeError** Traceback (most recent call last):

<ipython-input-277-98d0f8ede9d2> in <module>()  
----> 1 jason.add\_visa("USA")

<ipython-input-274-025b6acd4215> in add\_visa(self, country)  
2

3 def add\_visa(self, country):  
----> 4 if country in self.\_\_visas: return  
5 if len(self.\_\_visas) < 5:  
6 PrivateAgent.add\_visa(self, country)

**AttributeError:** 'RestrictedPrivateAgent' object has no attribute '\_\_visas'

# Inheriting a private class

- private attributes are not inherited!
- we can still use getters or the workaround:

```
class RestrictedPrivateAgent(PrivateAgent):  
  
    def add_visa(self, country):  
        visas = self.get_visas()  
        if country in visas: return  
        if len(visas) < 5:  
            PrivateAgent.add_visa(self, country)  
        else:  
            print("Cannot add", country, ":Visa limit exceeded!")
```

```
jason = RestrictedPrivateAgent("Jason Bourne", 50)  
jason
```

```
Jason Bourne , 50  
[]  
Now in: HQ
```

```
jason.add_visa("USA")  
jason
```

```
Jason Bourne , 50  
[USA ]  
Now in: HQ
```

# List comprehension

Given a Task Force, how can we choose a subset of the agents according to some criteria?

- Lets use *list comprehension*
- Something like this:

```
[f(i) for i in lst if condition(i)]
```

# List comprehension

```
a1 = Agent("a1", 20)
a2 = Agent("a2", 21)
a3 = SpecialAgent("a3", 22, 20)
a4 = RestrictedAgent("a4", 23)

l = [a1, a2, a3, a4]

tf = TaskForce(l)

subset = [agent for agent in tf.agents if agent.age > 22]
print subset
```

```
>>>
[a4 , 23]
[]
Now in: HQ
]
```

The same as:

```
subset = []
for agent in tf.agents:
    if agent.age > 22:
        subset.append(agent)
```

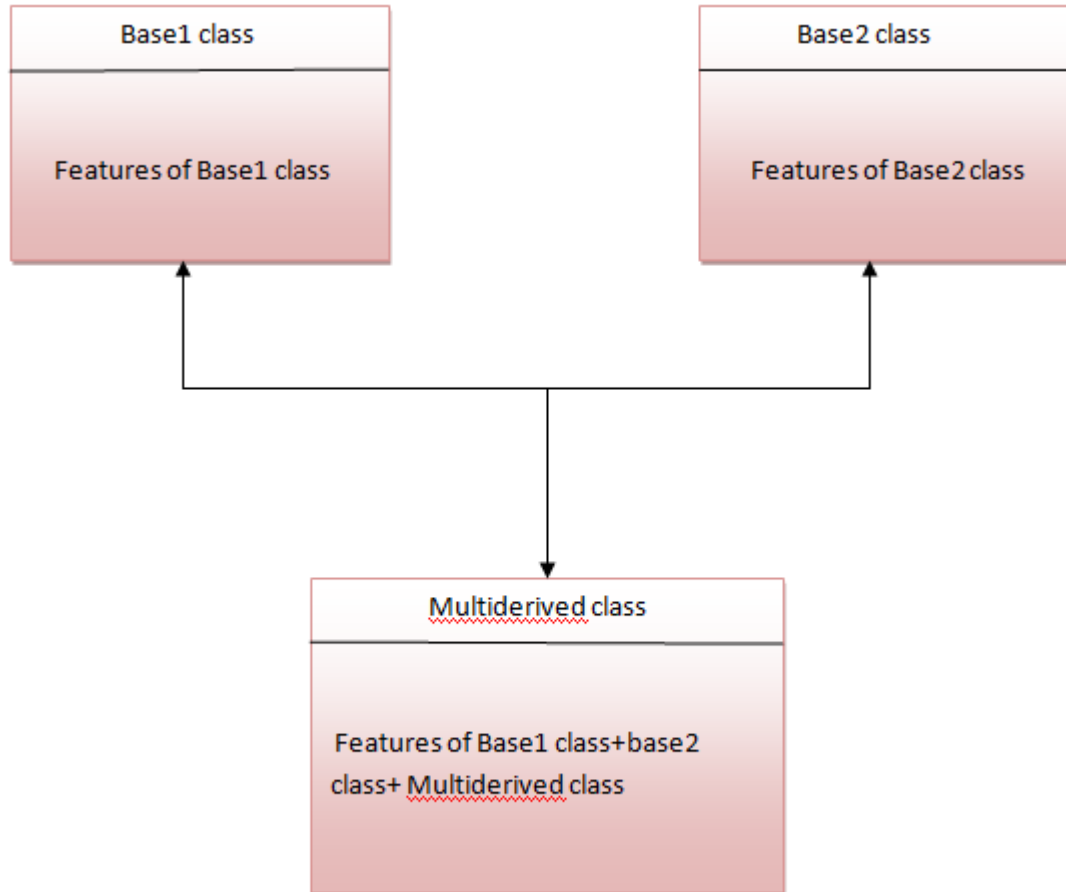


# Software design

- Design should be:
  - Intuitive (It makes sense that it's like that)
  - Efficient (minimal/no code duplication).
- Sometimes, this requires defining classes that do not represent real life entities

# Multiple inheritance

Sometimes we want to combine classes, to get functionality of both methods



More reading: <https://www.programiz.com/python-programming/multiple-inheritance>

# SuperHero class

```
class SuperHero:
    def __init__(self, real_name, secret_identity):
        self.name = secret_identity
        self.real_name = real_name
        self.secret_identity = secret_identity
    def reveal(self):
        if self.name==self.real_name:
            print(self.name + " already revealed...")
            return
        print(self.real_name + " Appears!")
        self.name = self.real_name
    def __repr__(self):
        if not self.name==self.real_name:
            return "This is just " + self.name + " :("
        else:
            return "This is... " + self.name + " !!!"
```

```
superman = SuperHero("Superman", "Clark Kent")
superman
```

This is just Clark Kent :(

```
superman.reveal()
```

Superman Appears!

```
superman
```

This is... Superman !!!

# Enlisting Superman to our task force

- We can not add Superman to our task force (he is not an Agent)
- Create a new class: *AgentHero*

```
#Note 2 classes in parentheses
class AgentHero(Agent, SuperHero):
    def __init__(self, name, age, secret_identity):
        Agent.__init__(self, name, age)
        SuperHero.__init__(self, name, secret_identity)

#Superman Becomes a secret Agent
supermanAgent = AgentHero("Superman", 38, "Clark Kent")
```

```
[isinstance(supermanAgent, Agent), isinstance(supermanAgent, SuperHero)]
```

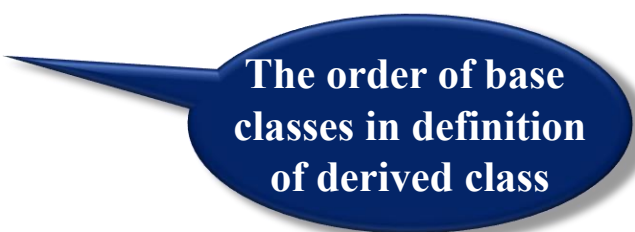
```
[True, True]
```

# Working with AgentHeroes

- Creating and Printing

```
supermanAgent
```

```
Clark Kent , 38  
[]  
Now in: HQ
```



The order of base  
classes in definition  
of derived class

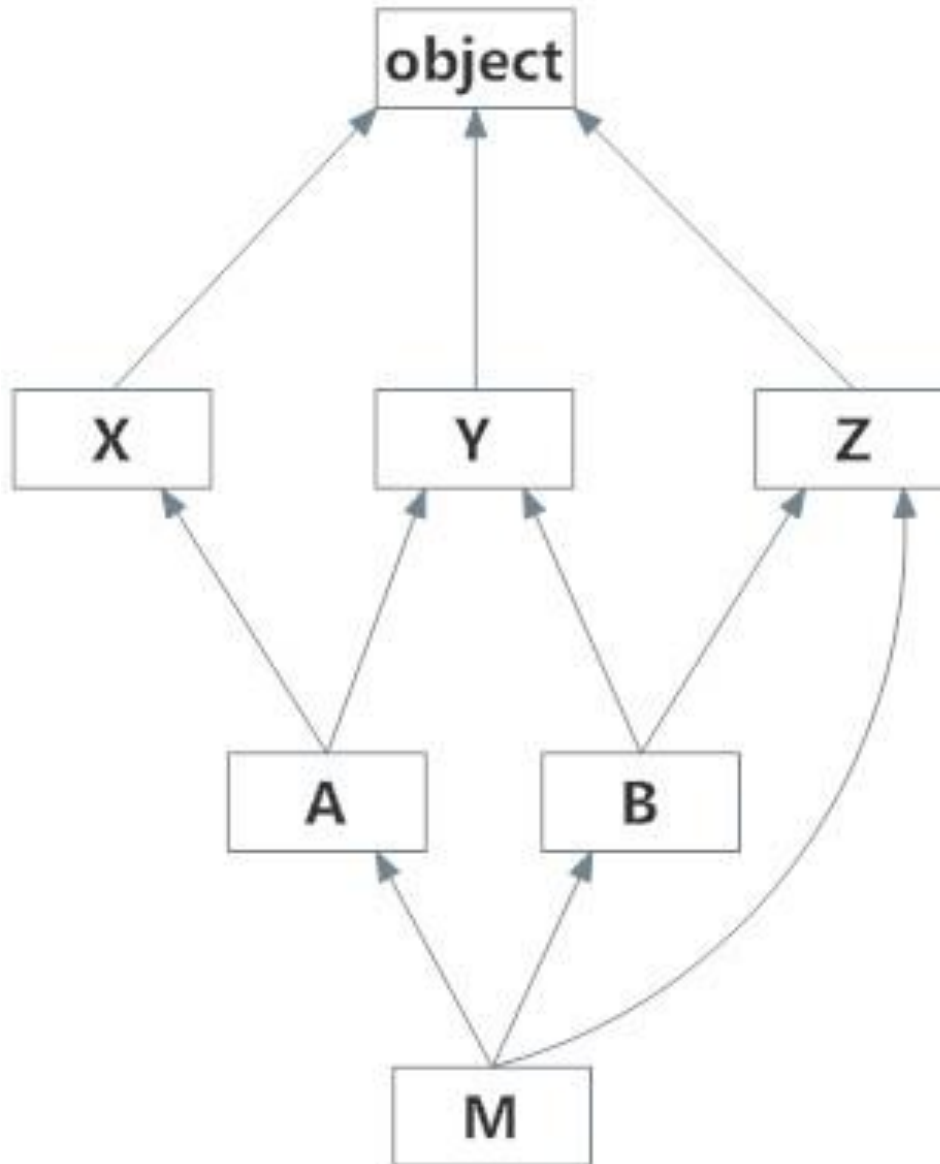
```
class AgentHero(Agent, SuperHero):
```

- Now Superman can join the team:

```
cteam = TaskForce([deadpool, supermanAgent])
```

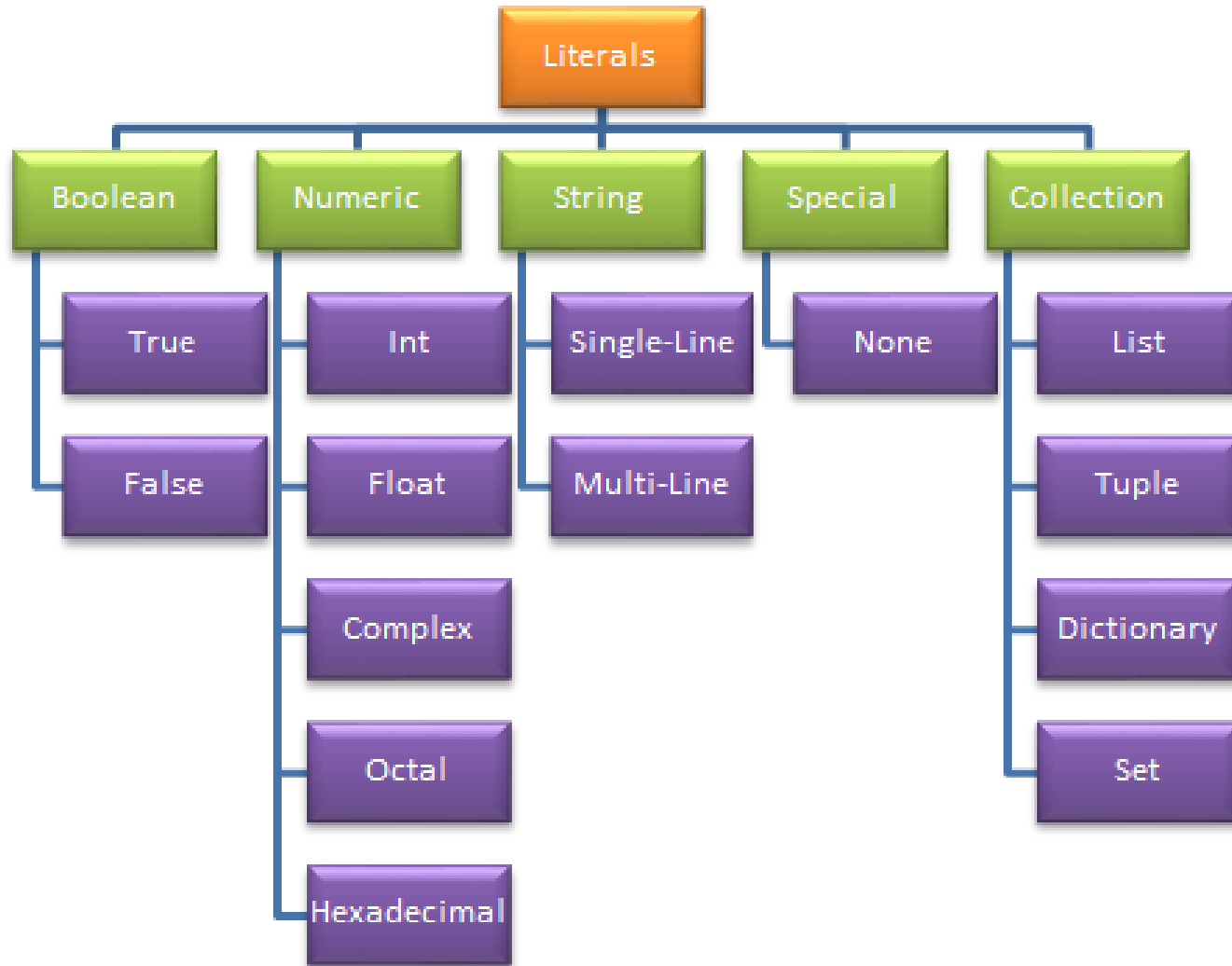
```
TaskForce established. Agent names:  
    Wade Wilson  
    Clark Kent
```

# The class object in Python



# Literals

Notation for representing fixed values (still OOP)



# Abstract classes



TETRAHEDRON



HEXAGONAL PRISM



CONE



PARALLELEPIPED



CUBE



HEXAGONAL PYRAMID



SPHERE



ICOSAHEDRON



CYLINDER



ELLIPSOID



TRIANGULAR PRISM



PENTAGONAL PRISM



DODECAHEDRON



OCTAHEDRON



CUBOID



SQUARE PYRAMID



# Abstract class

- A class containing one or more abstract methods.
- Abstract method is a method that is declared but not implemented.
- We cannot create direct instances of an abstract class only of its derived classes that implement the abstract method.
- We will use abstract classes, when two or more derived classes share the same methods but different behaviors.

# Abstract Base Class (ABC)

## abc — Abstract Base Classes

Source code: [Lib/abc.py](#)

---

This module provides the infrastructure for defining [abstract base classes](#) (ABCs) in Python, as outlined in [PEP 3119](#); see the PEP for why this was added to Python. (See also [PEP 3141](#) and the [numbers](#) module regarding a type hierarchy for numbers based on ABCs.)

- Abstract class must extend ABC
- *abstractmethod* decorator indicates that a method is abstract

# Abstract class Shape

```
from abc import ABC, abstractmethod

class Shape(ABC):

    @abstractmethod
    def calc_area(self):
        pass

    @abstractmethod
    def calc_perimeter(self):
        pass
```

# No instances of an Abstract class

```
s = Shape()
```

```
-----  
TypeError                                Traceback (most recent call last)  
<ipython-input-148-add4c52145cc> in <module>  
----> 1 s = Shape()
```

```
TypeError: Can't instantiate abstract class Shape with abstract methods calc_
```

# Class Circle

```
from math import pi

class Circle(Shape):

    def __init__(self, radius):
        self.radius = radius

    def calc_area(self):
        return pi * self.radius**2

    def calc_perimeter(self):
        return 2 * pi * self.radius
```

# Class Rectangle

```
class Rectangle(Shape):  
  
    def __init__(self,width, height):  
        self.width = width  
        self.height = height  
  
    def calc_area(self):  
        return self.width * self.height  
  
    def calc_perimeter(self):  
        return 2 * self.width + 2 * self.height
```

# Polymorphism

```
for shape in [Circle(3),Rectangle(4,2)]:  
    print(shape)
```

Circle

area: 28.274333882308138

perimeter: 18.84955592153876

Rectangle

area: 8

perimeter: 12

Q: where was the `__repr__` implemented?

# **Overloading and overriding methods**



# Overloading vs. overriding

- Overloading: several ways to call a method
- Overriding: implementation in the child class of a method that is provided by the parent class

<https://pythonspot.com/method-overloading/>

<https://www.geeksforgeeks.org/difference-between-method-overloading-and-method-overriding-in-python/>