9 Classes

Friday, April 12, 2024 1:59 PM

OOP :- Object Oriented Programming.

- A form of code organization that couples data (variables) and their operations (functions)
- Terminology varies

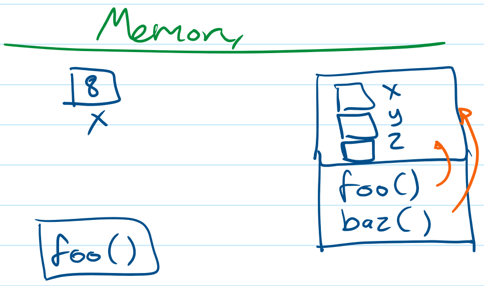
History:

"Simula" '80

"Small talk" by Alan key.

and later

C++, Java, C#.....



Pillars of OOP

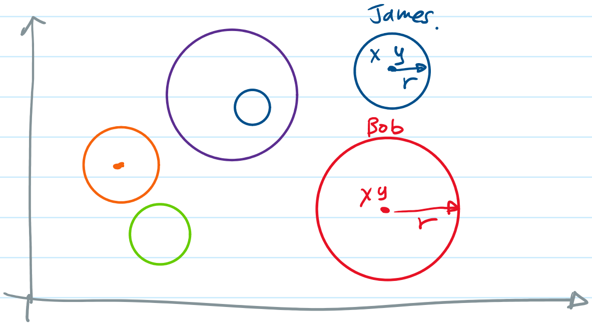
- Abstraction :- The ability to create new types.
- Encapsulation :- types keep their data hidden.
- Inheritance :- The ability to define a type as an extension of another type.
- Polymorphism :- A type may behave differently according to its internal circumstances.

Python :-

- Abstraction ☆
- ~~Encapsulation~~
- Inheritance ok
- Polymorphism fine.

Motivating example:

Circles:

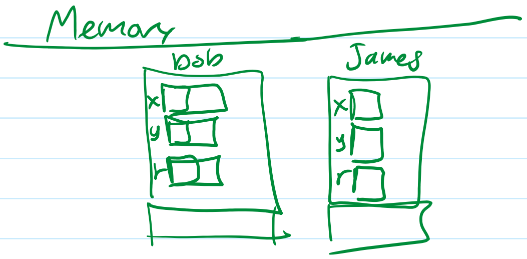


The OOP approach is to create a new type "circle" to represent these:

class.

the new type is a class
Bob and James are instances/objects of this class

Syntax class classname :
} variables and functions.

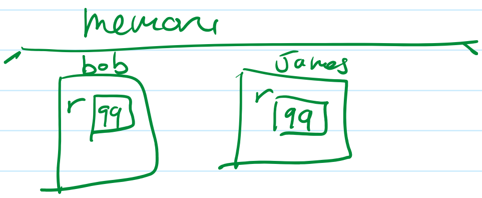


- "Constructor" special function inside a class

Syntax def __init__(self, parameters)
} Body of function.

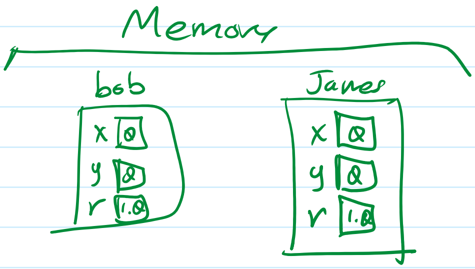
E.G.

```
class circle :
    def __init__(self) :
        self.r = 99


bob = circle()
james = circle()
```



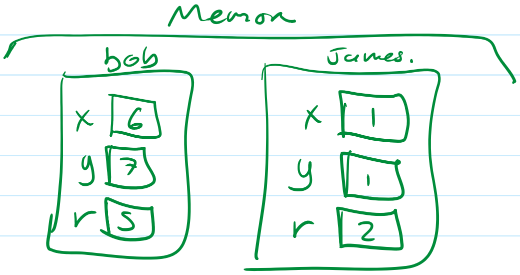
E.G.

```
class circle :
    def __init__(self) :
        self.x = 0
        self.y = 0
        self.r = 1.0


bob = circle()
james = circle()
```



```
class circle :
```

```
class circle :
    def __init__(self, a=0, b=0, c=1.0) :
        self.x = a
        self.y = b
        self.r = c

bob = circle(6,7,5)
james = circle(1,1,2)
```

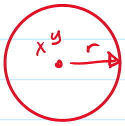


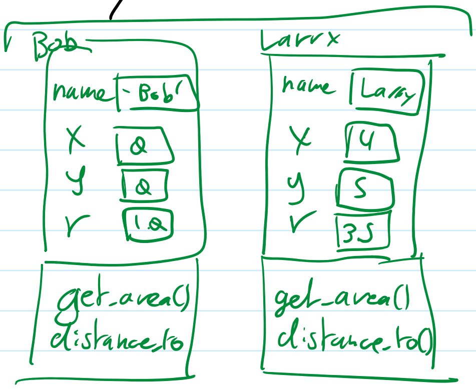
- Class "attributes:"
  - A data member that is shared by all objects of the same class
    instances.
  - declared directly in the class.
  - Usually intended for values that should not be changed.

- Class "Interface"

  the collection of member variables and member functions
  "fields" "methods"
  expected in an object of a particular class.

- Member functions / Methods.





- Introspection.

  You can check for a variables class

  isinstance(var, type)

- Special methods

  __str__() print variables
  should return a string.

- Operator Overloading.

__lt__() implements < operator.

e.g bob < larry.

⇓

bob.__lt__(larry)

- Other Operators you can overload.

>	__gt__	
or	__or__	
and	__and__	
int	__int__	convert object to int.
float	__float__	convert object to float.

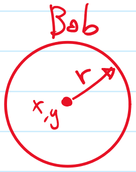
Using objects as mathematical entities:

==

+ − * /

example:

*



bob * 5 → a circle with five times the radius of bob.

__mul__()

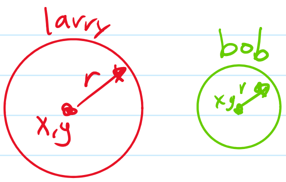
bob * 5

⇓

bob.__mul__(5)

example.

+



bob + larry →

⇓

bob.__add__(larry)

__add__ ()

example

== __eq__()

- an illustration of OOP:

```
class Queen
class Bishop
class Knight
class Roook
```

```
Class Queen
    self.r
    self.c
```

```
class Piece
    self.type = `Q"
    self.r
    self.c
```

class Board.

- SAMPLE CODE

```
import math

class Circle :
   def __init__(self, name, a=0, b=0, c=1.0) :
      self.name = name
      self.x = a
      self.y = b
      self.r = c

   def get_area(self) :
      return 3.14159 * ( self.r ** 2 )

   def distance_to(self, other) :
      xs = self.x - other.x
      ys = self.y - other.y
      d = math.sqrt( (xs**2) + (ys**2) )
      return d

   def is_collision(self, otherCircle) :
      sumr = self.r + otherCircle.r
      d = self.distance_to(otherCircle)
      if sumr > d :
         return True
      return False

   def __str__(self) :
      return self.name + ':(' + str(self.x) + ',' + str(self.y) + '-' + str(self.r) + ')'

   def __lt__(self, other) :
      return self.r < other.r

   def __mul__(self, n) :
      new_r = n * self.r
      new_x = self.x
```

```
        new_y = self.y
        new_name = "new " + self.name
        return Circle(new_name, new_x, new_y, new_r)

    def __add__(self, rhs) :
        new_r = self.r + rhs.r
        new_x = self.x + rhs.x
        new_y = self.y + rhs.y
        new_name = self.name + rhs.name
        return Circle(new_name, new_x, new_y, new_r)

    def __eq__(self, rhs) :
        return self.r == rhs.r
```