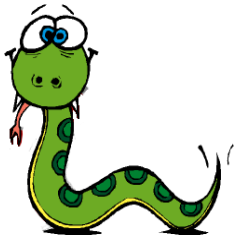
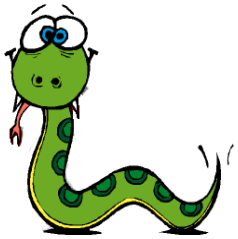


Classes and Objects

Chapter 11

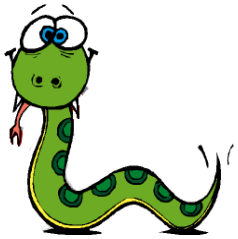
Prof. Mauro Gaspari: gaspari@cs.unibo.it

Defining new data types



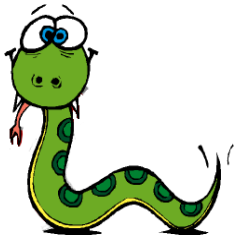
- We have used many of Python's built-in types: Strings, Tuples, Lists, Dictionaries.
- However, applications often have specific requirements that cannot be modelled well with these data types.
- Programming languages (Python) provide mechanisms to define new data types.
- Creating a new type is (a little) more complicated than the other options, but it has advantages that will be apparent soon.

Defining a point type



- We will create a type called Point that represents a point in two-dimensional space.
- Notation: (x, y) represents the point x units to the right and y units up from the origin.
- There are several ways we might represent points in Python:
  - store the coordinates separately in two variables, 1 and 2.
  - store the coordinates as elements in a list or tuple.
  - create a new type to represent points as objects.

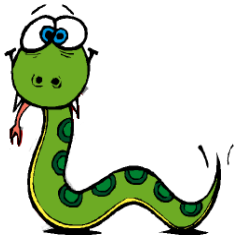
Example



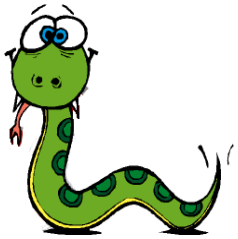
- Implementing a point with a lists.
- The type include a function that is able to move points.

```
>>> point = [0,0]
>>> def move(p,x,y):
...     p[0] = x
...     p[1] = y
...
>>> move(point,4,5)
>>> point
[4,5]
```

Problems

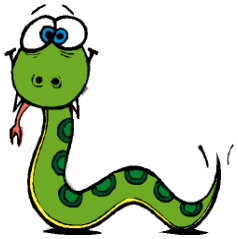


- This implementation has several problems:
  - We can use the len function on a point .
  - We can change the coordinates of point without using the move function.
  - We can modify the internal structure of a point, for example deleting a field or adding more values with the append function.
  - The type function returns 'list' as a type.
- In summary it does not behave like a built-in data types.



Example

```
>>> len(punto)
2
>>> type(punto)
<class 'list'>
>>> punto[1:1] = [5,6]
>>> punto
[4, 5, 6, 5]
>>>
```

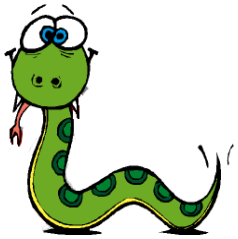


Types as Classes

- A user-defined type is also called a **class**.
- A class definition looks like this:

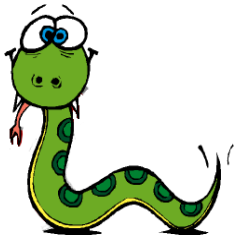
```
class Point:

  def __init__(self, x, y):
    self.x = x
    self.y = y
```

Defining classes

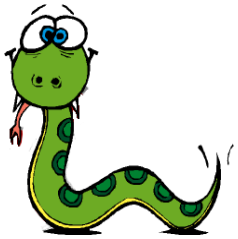


- This header indicates that the new class is a Point, which is a kind of object, which is a built-in type.
- Defining a class named Point creates a class object. Then you can define variables and functions inside a class definition.
- The init method (short for “initialization”) is a special method that gets invoked when an object is instantiated (created). Its full name is __init__ (two underscore characters, followed by init, and then two more underscores).
- For a good programming style I recommend to define init as a first method of the class: this makes the structure of objects clear.

__init__ method.



- It is common for the parameters of init to have the same names as the attributes.
- The parameter **self** refers to the created object and must be the first argument.
- The init indicates that the point class has two attributes.
- The syntax self.x and self.y denotes object attributes. The notation adopted is similar to that adopted for modules.

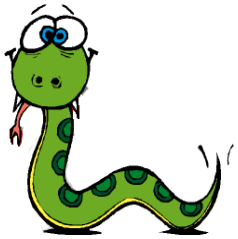


Objects and instances

- To create a Point, you call Point as if it were a function. This function calls the __init__ method automatically when this is defined.

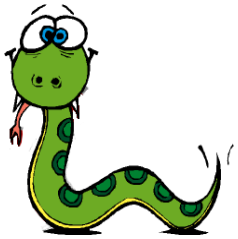
```
blank = Point(0,0)
```

The reference to the new object is assigned to variable blank.



Example

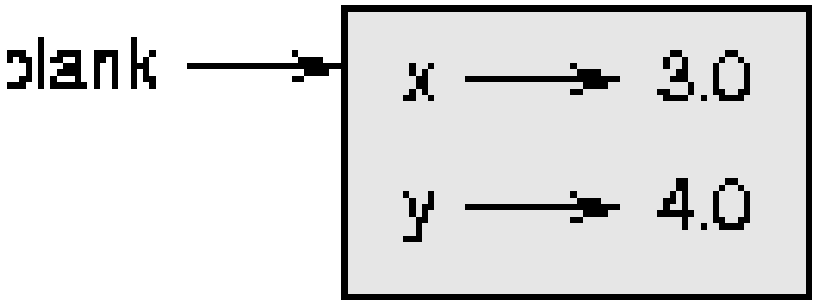
```
>>> blank.x
0
>>> blank.y
0
>>> blank
<__main__.Point instance at 0xb7f75c2c>
>>>
>>> type(blank)
<class '__main__.Point'>
>>>
```

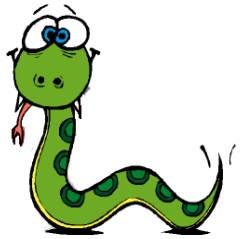


Example

- Objects are mutable:

```
>>> blank.x = 3.0
>>> blank.y = 4.0
```



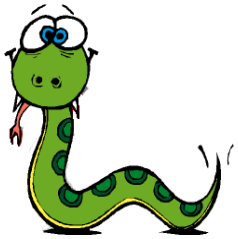


Example

Using the dot notation.

```
>>> print(blank.y)
4.0
>>> x = blank.x
>>> print(x)
3.0
```

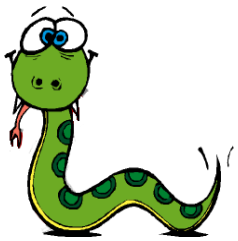
Note that variable x and attribute X are different.



Printing an object

```
>>> blank
<__main__.Point instance at 0xb7f75c2c>

# this is just a reference

def printPoint(p):
  print('(' + str(p.x) + ', ' + str(p.y) + ')')
```

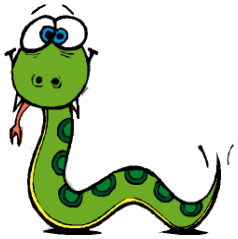
Note that when an object is given as an argument to a function the reference is passed.



Default values

- We can assign default values to parameters of functions.
- The same is true for methods.

```
def alwaysprint(x=0):
...    print(x)
...
>>> alwaysprint(1)
1
>>> alwaysprint()
0
```

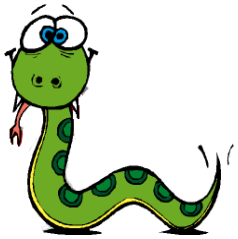


Examples

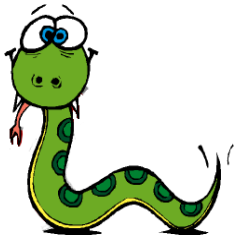
```
def find(str, ch):
  index = 0
  while index < len(str):
    if str[index] == ch:
      return index
    index = index + 1
  return -1
```

```
def find(str, ch, start=0):
  index = start
  while index < len(str):
    if str[index] == ch:
      return index
    index = index + 1
  return -1
```

Improving the point definition



```
class Point:
  def __init__(self, x=0, y=0):
    self.x = x
    self.y = y

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

  def move(self,a,b):
    self.x = a
    self.y = b
```



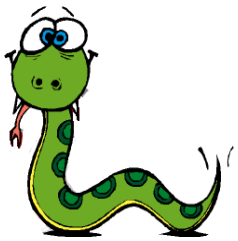
Example

```
>>> p = Point(3, 4)
>>> str(p)
'(3, 4)'
```

Note that: __str__ overrides the standard behavior of str

```
>>> p = Point(3, 4)
>>> print(p)
(3, 4)
```

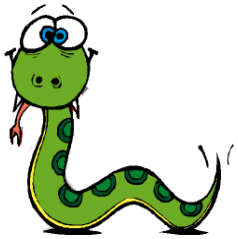
The str function is called while printing an object



Executing a script

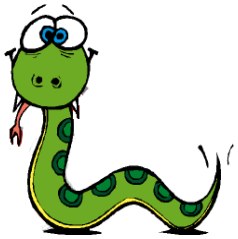
```
class Point:
  def __init__(self, x=0, y=0):
    self.x = x
    self.y = y

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

  def muovi(self,a,b):
    self.x = a
    self.y = b

b = Point(4,5)
b.move(6,7)
print(b)
```

Objects are mutable!



Example

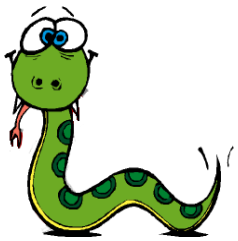
```
>>> p1 = Point(3,4)
>>> p2 = Point(3,4)
>>> p1 == p2
False
>>> p2 = p1
>>> p1 == p2
True
```



Example

- deep equality for points.

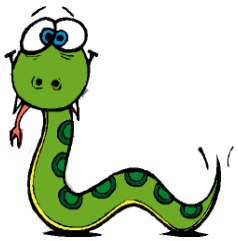
```
def samePoint(p1, p2) :
  return (p1.x == p2.x) and (p1.y == p2.y)
```



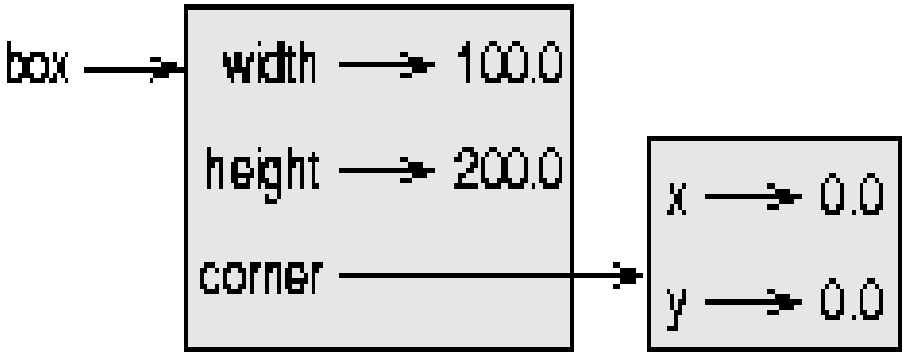
A rectangle object

```
class Rectangle:
   def __init__(self,width,height,x,y):
        self.corner = Point(x,y)
        self.width = width
        self.height = height


box = Rectangle(100.0,200.0,0.0,0.0)
```

NB. the point object is part of the rectangle object.



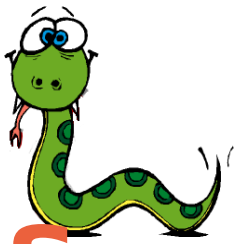
Compound Objects



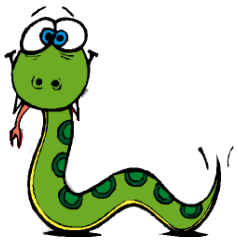
Access to compound objects:

```
>>> print(box.corner.x)
```

Instances as return values



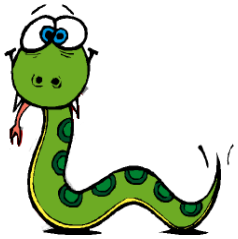
```
def findCenter(box):
  p = Point()
  p.x = box.corner.x + box.width/2.0
  p.y = box.corner.y + box.height/2.0
  return p

>>> center = findCenter(box)
>>> printPoint(center)
(50.0, 100.0)
```



Object Cloning

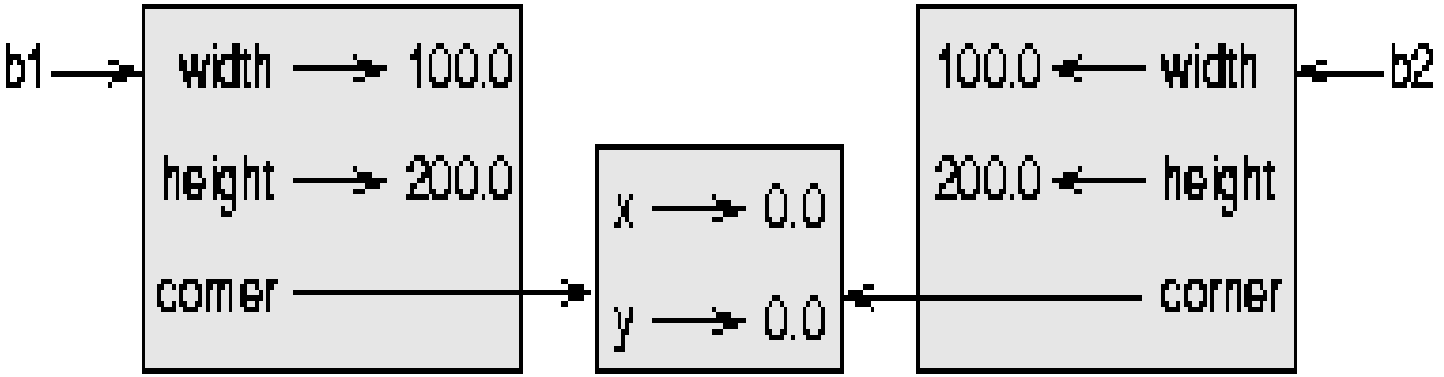
- Aliasing can also occur for objects.
- It is recommended to copy objects to avoid aliasing.
- The copy module provide this functionality.

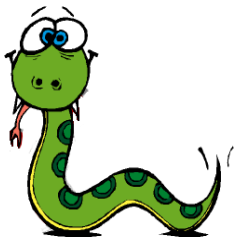
```
>>> import copy
>>> p1 = Point(3,4)
>>> p2 = copy.copy(p1)
>>> p1 == p2
False
>>> samePoint(p1, p2)
True
```



Example

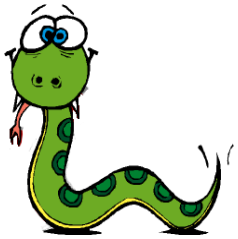
- Warning the function copy.copy (the function copy of module copy) implements shallow copyng only.





Example 2

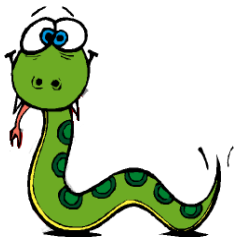
```
def growRect(box, dwidth, dheight) :
  box.width = box.width + dwidth
  box.height = box.height + dheight


def moveRect(box,nx,ny):
   box.corner.x = nx
   box.corner.y = ny
```



Deep Copy

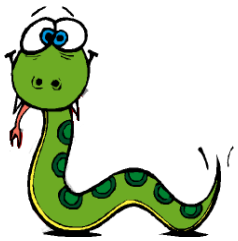
```
>>> b2 = copy.deepcopy(b1)

# and now no aliasing arises between b1 and b2

def growRect(box, dwidth, dheight) :
  import copy
  newBox = copy.deepcopy(box)
  newBox.width = newBox.width + dwidth
  newBox.height = newBox.height + dheight
  return newBox
```

Example a Time object



```
class Time:
  def __init__(self,hours=0,minutes=0,seconds=0):
     self.hours = hours
     self.minutes = minutes
     self.seconds = seconds
  def __str__(self):
     temp = str(self.hours)+':'+str(self.minutes)
     return temp+':'+str(self.seconds)
```

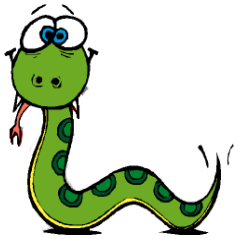
```
>>> time = Time(11,59,30)
>>> print(time)
11:59:30
```



Operator overloading

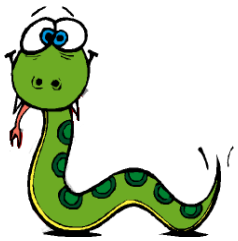
- The ability to change built-in operators to adapt them to user defined data types (str is an example). +,*,==,>,< can be overloaded too.

```
class Point:
  # previously defined methods here...

  def __add__(self, other):
    return Point(self.x + other.x, self.y + other.y)
```



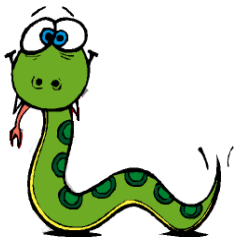
Example

```
>>>   p1 = Point(3, 4)
>>>   p2 = Point(5, 7)
>>>   p3 = p1 + p2
>>>   print(p3)
(8, 11)
```



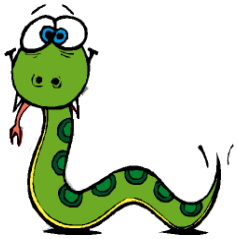
More overloading

- `p1 + p2` is equivalent to `p1.__add__(p2)`
- It means to apply the method __add__ to object p1.
- Infix notation is more declarative and convenient.
- Other overloaded operators are:
  - __mul__
  - __rmul__
  - __lt__, __gt__, __le__, __ge__, __eq__, and __ne__



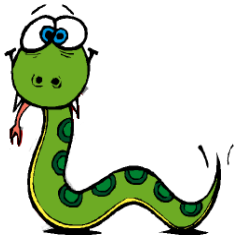
Example

```
def __mul__(self, other):
  return self.x * other.x + self.y * other.y

def __rmul__(self, other):
  return Point(other * self.x,  other * self.y)
```



Exercise 1

- Define a student Class with name, surname and matricola data.
- Define the __str__ method for the student.



Exercise 2

- Define a circle object.
- Two circles are equal if they have the same radius define the != and == operators for circle defining the __eq__ and __neq__ methods.

```
bank = [['Mauro','1111',700],['Johanna','2222',40000]]

def create_account(b):
   name = raw_input('Account name: ')
   psw = raw_input('Password: ')
   amount = input('Euro: ')
   b.append([name,psw,amount])
   print "crated an account for ",name
   return None

def get_user_index(b,n):
   i = 0
   while i < len(b):
       if b[i][0] == n:
           return i
       i = i + 1
   print('user ', n , ' is not allowed toaccess this bank')
   return None

def check_psw(account):
   for j in range(3):
       psw = input('Give me your pin: ')
       if psw == account[1]:
           return True
   return False

def cash_dispenser(b):
    while True:
        card = input('Give me your card name: ')
        i = get_user_index(b,card)
        if i == None:
            continue
        if check_psw(b[i]):
            amount = int(input('Give me how much you want to withdraw : '))
            if b[i][2] >= amount:
                b[i][2] = b[i][2] - amount
                print('here are your ', amount, ' Euro')
            else:
                print('Operation not allowed')
```