Clases y Objetos en Python

Introducción a la Computación

Clase 23

Patricia Borensztejn

Tipos en Python

- Ya vimos que tenemos un conjunto importante de tipos básicos y compuestos en Python definidos por el lenguaje.
- Por ejemplo:
  - los tipos básicos: enteros, reales, complejos, booleanos (ver Clase 4)
  - Las secuencias: cadenas, listas (ver clase 11) , tuplas (ver clase 14)
  - Otros tipos estructurados: diccionarios (clase 14) , sets, registros (ver clase 12)

Tipos definidos por el usuario: clases

- Vamos el crear un tipo nuevo: el tipo punto.
- Un punto está definido por sus dos coordenadas: (x,y)
- Podemos usar uno de los tipos compuestos existentes, como una tupla o bien una lista.
- Pero tambien podemos crear un nuevo tipo compuesto también llamado una clase.

```
class Punto:
  pass
```

Sentencia class

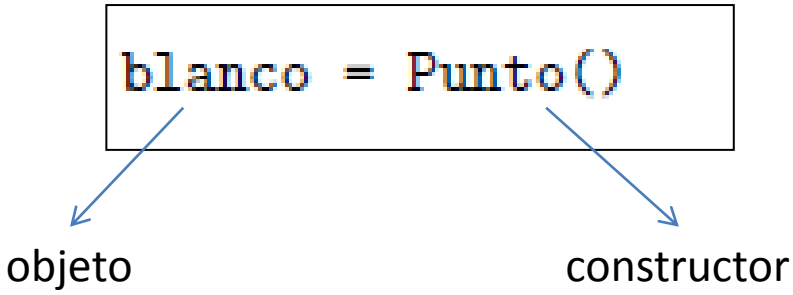
- La sentencia class es una sentencia compuesta, que, como toda sentencia compuesta está formada:
  - por una cabecera seguida por los dos puntos y
  - un cuerpo o bloque de sentencias
- En el caso de la clase Punto, como no hay cuerpo, ponemos la sentencia **pass** que no tiene ningún efecto.

```
class Punto:
  pass
```

```
if x > 0:
  print "x es positivo"
```

Instancias o Objetos

- Al crear la clase hemos creado un nuevo tipo llamado Punto.
- Para crear un objeto de este nuevo tipo debemos instanciar la clase.

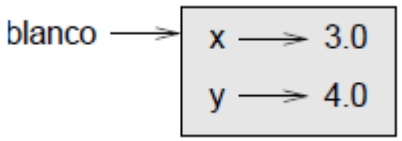


Atributos

- Podemos añadir nuevos datos a la instancia usando el operador . (punto)

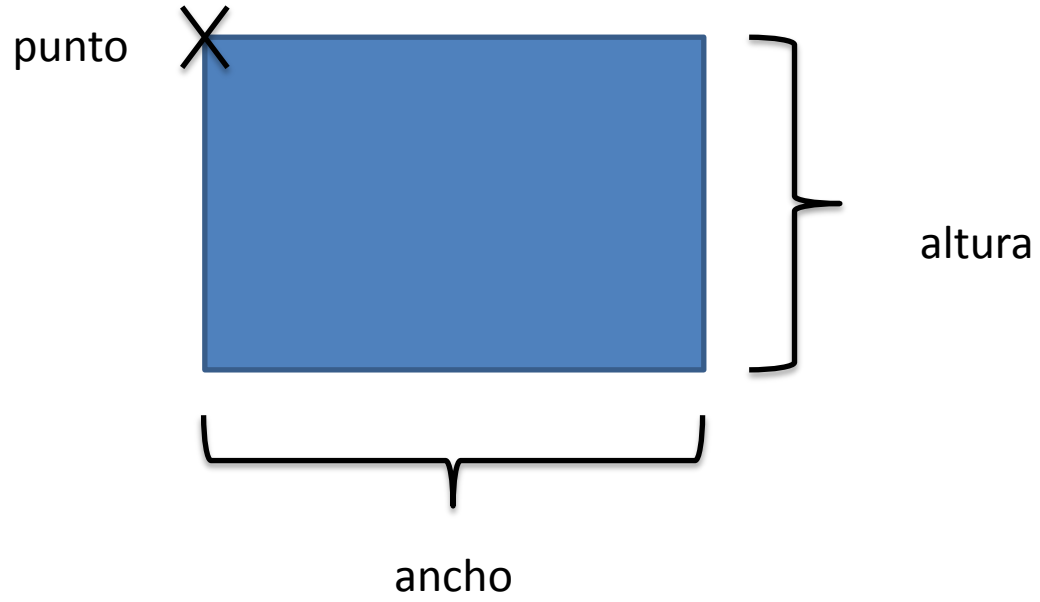
```
>>> blanco.x = 3.0
>>> blanco.y = 4.0
```

- La variable blanco apunta a un objeto Punto que contiene dos atributos. Cada atributo apunta a un número en coma flotante.



Otro ejemplo: rectángulo

- Un rectángulo puede definirse así:



- Es decir: el Punto de la esquina superior izquierda, y sus dimensiones: ancho y altura

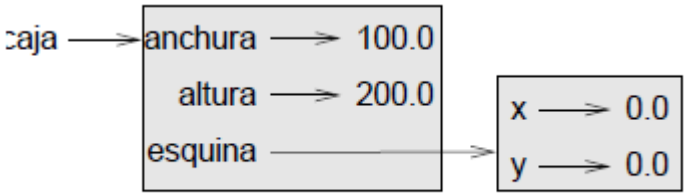
Rectángulo

- Es decir:

```
class Rectangulo
    pass
```

```
caja = Rectangulo()
caja.anchura = 100.0
caja.altura = 200.0
```

```
caja.esquina = Punto()
caja.esquina.x = 0.0;
caja.esquina.y = 0.0;
```



Copia de objetos

```
class Punto:
    pass

class Rectangulo:
    pass

caja=Rectangulo()
caja.anchura=100.0
caja.altura=200.0
caja.esquina=Punto()
caja.esquina.x =0.0
caja.esquina.y=0.0

1   print caja
    otracaja=caja
2   print otracaja
    otracaja=Rectangulo()
    otracaja.esquina=Punto()
3   print otracaja
4   print caja.esquina
5   print otracaja.esquina
```

```
>>>
1   <__main__.Rectangulo instance at 0x02A29058>
2   <__main__.Rectangulo instance at 0x02A29058>
3   <__main__.Rectangulo instance at 0x02A23BE8>
4   <__main__.Punto instance at 0x02A29080>
5   <__main__.Punto instance at 0x02A23E90>
```

Copia con Módulo Copy

- Hay un módulo llamado copy que tiene dos métodos: copy() y deepcopy(). Los probamos.

```
import copy
b1=copy.copy(caja)
1 print caja
2 print caja.esquina
3 print b1
4 print b1.esquina
```

```
1 <__main__.Rectangulo instance at 0x02AC9058>
2 <__main__.Punto instance at 0x02AC9080>
3 <__main__.Rectangulo instance at 0x02AC9418>
4 <__main__.Punto instance at 0x02AC9080>
```

- Son distintos objetos pero comparten la misma referencia al objeto Punto (esquina)

Copia con Módulo Copy: deepcopy()

- Probamos con deepcopy().

```
   import copy
   b1=copy.deepcopy(caja)
1  print caja
2  print caja.esquina
3  print b1
4  print b1.esquina
```

```
1  <__main__.Rectangulo instance at 0x029FA058>
2  <__main__.Punto instance at 0x029FA030>
3  <__main__.Rectangulo instance at 0x029FA418>
4  <__main__.Punto instance at 0x029EE7D8>
```

- Son distintos objetos, y también es distinto el objeto interno.

Otro ejemplo

- Definimos una clase llamada Hora:

```
class Hora:
  pass
```

- Y ahora una función para imprimir la hora:

```
def imprimeHora(hora):
  print str(hora.horas) + ":" +
        str(hora.minutos) + ":" +
        str(hora.segundos)
```

- Y ahora instanciamos la clase e invocamos la función:

```
>>> horaActual = Hora()
>>> horaActual.horas = 9
>>> horaActual.minutos = 14
>>> horaActual.segundos = 30
>>> impriemHora(horaActual)
```

Otra forma de hacerlo: Métodos

- Python nos permite definir funciones dentro de la clase: esas funciones se llaman métodos

```
class Hora:
  def imprimeHora(hora):
    print str(hora.horas) + ":" +
          str(hora.minutos) + ":" +
          str(hora.segundos)
```

- Y ahora la invocamos así:

```
>>> horaActual.imprimeHora()
```

- Hemos convertido a la función en un método de la clase.

Métodos

- Fijémonos que el método imprimeHora tiene un parámetro llamado hora, y que al invocar el método lo hacemos sin el parámetro.
- Es el objeto sobre el que se invoca el método el parámetro del método.

```
class Hora:
  def imprimeHora(hora):
    print str(hora.horas) + ":" +
          str(hora.minutos) + ":" +
          str(hora.segundos)
```

```
>>> horaActual.imprimeHora()
```

Métodos: parámetro self

- Por convención, el primer parámetro de todos los métodos que definamos dentro de una clase, se llama self.
- Self, hace referencia al objeto (instancia) que invoca al método.

```
class Hora:
    def imprimeHora(self):
        print str(self.horas) + ":" + str(self.minutos) + ":" + str(self.segundos)
```

Métodos:parámetro self

```
class Hora:
    def imprimeHora(self):
        print str(self.horas) + ":" + str(self.minutos) + ":" + str(self.segundos)
    def incremento(self, segundos):
        self.segundos = segundos + self.segundos
        while self.segundos >= 60:
            self.segundos = self.segundos - 60
            self.minutos = self.minutos + 1
        while self.minutos >= 60:
            self.minutos = self.minutos - 60
            self.horas = self.horas + 1

horaActual=Hora()
horaActual.horas=12
horaActual.minutos=5
horaActual.segundos=30
horaActual.imprimeHora()
horaActual.incremento(500)
horaActual.imprimeHora()
```

Una idea distinta

- La sintaxis para la llamada a una función,
  imprimeHora(horaActual)

  sugiere que la función es el agente activo.

  Dice algo así como :

  "Oye imprimeHora! Aquí hay un objeto para que lo imprimas".

- En programación orientada a objetos, los objetos son los agentes activos. Una invocación como
  horaActual.imprimeHora()

  dice:

  "Oye horaActual! Imprímete!"

Inicialización: método Init

- Hay un método especial que se llama __init__ por convención, que se ejecuta cuando instanciamos un objeto de la clase.

```
class Hora:
  def __init__(self, horas=0, minutos=0, segundos=0):
    self.horas = horas
    self.minutos = minutos
    self.segundos = segundos
```

```
>>> horaActual = Hora(9, 14, 30)
>>> horaActual.imprimeHora()
>>> 9:14:30
```

Método __Init__

- Ejemplos de uso del constructor

```
>>> horaActual = Hora()
>>> horaActual.imprimeHora()
>>> 0:0:0
```

```
>>> horaActual = Hora (9, 14)
>>> horaActual.imprimeHora()
>>> 9:14:0
```

```
>>> horaActual = Hora (9)
>>> horaActual.imprimeHora()
>>> 9:0:0
```

```
>>> horaActual = Hora(segundos = 30, horas = 9)
>>> horaActual.imprimeHora()
>>> 9:0:30
```

Objetos

- Veamos que es un objeto:

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

```
>>> p=Punto(3,4)
>>> p
<__main__.Punto instance at 0x023238A0>
>>> print p
<__main__.Punto instance at 0x023238A0>
>>> print p.x
3
>>>print p.y
4
```

Imprimiendo objetos

- Cuando hacemos print p, la verdad es que no queremos imprimir la referencia al objeto, sino su contenido.
- Esto se hace, definiendo o mejor dicho, redefiniendo el método *str* dentro de la nueva clase. Porque print llama a *str*.
- Recordamos que str convierte a una representación en forma de cadena cualquier tipo de objeto.

Redefiniendo __str__

- Si una clase ofrece un método llamado __str__ , éste se impone al comportamiento por defecto de la función interna str de Python.

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

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

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

Sobrecarga de operadores

- Ahora, que ya tenemos el tipo Punto, lo que quisiéramos es sumar dos puntos. Pero lo que queremos es hacer esto:

```
>>> p1=Punto(2,3)
>>> p2=Punto(3,1)
>>> p3=p1+p2
>>> print p3
(5,4)
```

- Es decir, queremos utilizar el operador + para realizar una suma definida por nosotros!

Sobrecarga de operadores

- Se hace así:

```
class Punto:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y
    def __str__(self):
        return '(' + str(self.x) + ', ' + str(self.y) + ')'
    def __add__(self, otro):
        return Punto(self.x + otro.x, self.y + otro.y)
```

- Es decir, hay que redefinir el método llamado __add__
- La expresión p1 + p2 equivale a p1. add (p2), pero es obviamente más elegante.

Sobrecarga de operadores

- Para el caso de la multiplicación , redefinimos __mul__ como el producto interno de dos puntos:

```
class Punto:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y
    def __str__(self):
        return '(' + str(self.x) + ', ' + str(self.y) + ')'
    def __add__(self, otro):
        return Punto(self.x + otro.x, self.y + otro.y)
    def __mul__(self, otro):
        return self.x * otro.x + self.y * otro.y
    def __rmul__(self, otro):
        return Punto(self.x * otro + self.y * otro)
```

Sobrecarga de operadores

- __mul__ funciona si el objeto que está a la izquierda de la * es un Punto.
- __rmul__ funciona si el objeto que está a la izquierda de la * es un escalar

```
>>> p1 = Punto(3, 4)
>>> p2 = Punto(5, 7)
>>> print p1 * p2
43
>>> print 2 * p2
(10, 14)
```

Usando Puntos

- Escribimos la clase en un archivo punto.py y lo importamos.

```
>>> from punto import Punto
>>> p1 = Punto(3, 4)
>>> p2 = Punto(5, 7)
>>> print p1 * p2
43
>>> print 2 * p2
(10, 14)
```

Ejercicio: fracciones

- Implementar la clase de los números fraccionarios. El constructor debe, por defecto, poner el denominador =1. Eso permite representar los números enteros como fracciones.
- Implementar la suma, resta, multiplicación y división.
- Si el resultado de las operaciones es «simplificable», aplicar el algoritmo de Euclides para obtener el resultado simplificado.

Método de Euclides para el cálculo del MCD

- Reducir la fracción es encontrar el Máximo Común Divisor entre el numerador y el denominador y dividirlos a ambos por ese números.
- Algoritmo de Euclides (recursivo) para encontrar el MCD(m,n)

Si n divide a m sin resto, entonces n es el MCD. De lo contrario, el MCD es el MCD de n y el resto de dividir m entre n.

Multisuma (MAC)

- Hay una operación común en algebra lineal (si?) llamada multisuma (existe en algunos ISA con el nombre de MAC: multiplica y acumula), que tiene tres entradas (a,b,c) , y cuya salida es : a*b+c (multiplica a*b y acumula sobre c)

```
def multisuma (x, y, z):
  return x * y + z
```

```
>>> multisuma (3, 2, 1)
7
```

Multisuma es una función Polimórfica

- Porque funciona con enteros

```
>>> multisuma (3, 2, 1)
7
```

- ¡Y funciona con Puntos!

```
>>> p1 = Punto(3, 4)
>>> p2 = Punto(5, 7)
>>> print multisuma (2, p1, p2)
(11, 15)
>>> print multisuma (p1, p2, 1)
44
```

- Una función como ésta que puede tomar parámetros de distintos tipos se llama *polimórfica*

Polimorfismo

- Para determinar si una función se puede aplicar a un nuevo tipo, aplicamos la regla fundamental del polimorfismo:
  - *Si todas las operaciones realizadas dentro de la función se pueden aplicar al tipo, la función se puede aplicar al tipo.*
- En el caso de multisuma, tanto la multiplicación como la suma están definidas para el tipo Punto, por lo tanto multisuma se puede aplicar al tipo Punto.

Polimorfismo: Otro Ejemplo

- Supongamos que hemos escrito la siguiente función para imprimir una lista del derecho y del revés:

```
def delDerechoYDelReves(derecho):
  import copy
  reves = copy.copy(derecho)
  reves.reverse()
  print str(derecho) + str(reves)
```

```
>>> miLista = [1,2,3,4]
>>> delDerechoYDelReves(miLista)
>>>
[1, 2, 3, 4][4, 3, 2, 1]
>>>
```

delDerechoYDelReves

- Para saber si podemos utilizar esa función para Puntos, miramos las funciones que utiliza:
  - copy: es un método del Módulo copy que ya vimos que se puede usar para todo tipo de objetos
  - str: hemos redefinido __str__ dentro de la clase Punto
  - Reverse: es un método de las listas que los Puntos no tienen➔ lo hacemos.

Haciendo que delDerechoYDelReves sea polimórfica

- Definimos para la clase Punto la función reverse así:

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

- Ahora podemos usar delDerechoYDelReves para Puntos.

```
>>>   p = Punto(3, 4)
>>>   delDerechoYDelReves(p)
(3, 4)(4, 3)
```

Polimorfismo

- El mejor tipo de polimorfismo es el que no se busca: cuando usted descubre que una función que había escrito se puede aplicar a un tipo para el que nunca la había planeado.

Mas cosas: la herencia

- Supongamos que queremos definir un nuevo tipo, el tipo instrumento musical. Lo haríamos así:

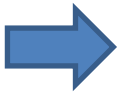
```
class Instrumento:
        def __init__(self, precio):
                self.precio = precio
        def tocar(self):
                print “Estamos tocando musica”
        def romper(self):
                print “Eso lo pagas tu”
                print “Son”, self.precio, “$$$”
```

```
guitarra=Instrumento(2000)
guitarra.tocar()
guitarra.romper()
```

```
Estamos tocando musica
Eso lo pagas tu
Son 2000 $$$
>>>
```

Herencia

- Ahora vamos a decir que la guitarra, el piano, la batería, y cualquier otro instrumento que se me ocurra, es UN NUEVO TIPO que hereda todos los atributos y todas las funciones del tipo instrumento. Y lo hacemos así:
- Se llama herencia….



```
class Instrumento:
        def __init__(self, precio):
                self.precio = precio
        def tocar(self):
                print “Estamos tocando musica”
        def romper(self):
                print “Eso lo pagas tu”
                print “Son”, self.precio, “$$$”

class Guitarra(Instrumento):
          pass
class Piano(Instrumento):
          pass
class Batería(Instrumento):
          pass
```

Herencia

- Lo interesante de la herencia, es que las clases “hijas” o subclases, pueden definir métodos propios que no existen en el “padre”, y también pueden redefinir métodos que sí existen en el “padre”
- Por ejemplo: podríamos redefinir la clase tocar de cada uno de los instrumentos.

```
class Bateria(Instrumento):
    def tocar(self):
        print "Estamos tocando batería"

class Guitarra(Instrumento):
    def tocar(self):
        print "Estamos tocando guitarra"

guitarra=Guitarra(2000)
guitarra.tocar()
guitarra.romper()
batería=Batería(10000)
batería.tocar()
batería.romper()
```

Herencia

- También, podría suceder que dentro de una subclase (guitarra) querramos sobreescribir el método de la clase padre (Instrumento) pero para agregarle algunas sentencias. En ese caso, se puede llamar al metodo del padre dentro del método (del mismo nombre) de la subclase.

```
class Guitarra(Instrumento):
    def __init__(self, precio, tipo_cuerda):
        Instrumento.__init__(self,precio)
        self.cuerda=tipo_cuerda
        print "Las cuerdas de mi guitarra son de"
        print self.cuerda
    def tocar(self):
        print "Estamos tocando guitarra"

guitarra=Guitarra(2000,"metal")
```

```
>>>
Las cuerdas de mi guitarra son de
metal
```

Herencia Múltiple

- La clase Cocodrilo hereda de dos superclases : Terrestre y Acuático:

```
class Terrestre:
    def desplazar(self):
        print "El animal anda"

class Acuatico:
    def desplazar(self):
        print "El animal nada"

class Cocodrilo(Terrestre, Acuatico):
    pass

c = Cocodrilo()
c.desplazar()
```

Resumen

- Hemos visto que podemos definir tipos nuevos mediante las clases
- Hemos visto que las clases contienen atributos y métodos
- Hemos visto que los métodos pueden sobrecargar operadores (redefinir operadores)
- Hemos visto que podemos construir una clase y subclases que heredan sus método y atributos
- Hemos visto que podemos definir operaciones que son válidas para distintos tipos

Paradigma orientado a objetos

- Todas las características que vimos en esta clase corresponden al paradigma “objetos”
- Es decir, Python es un lenguaje que puede ser utilizado con el modelo de orientación a objetos, aunque también puede ser utilizado con el paradigma modular y estructurado con el que venimos trabajando…

POO: programación orientada a objetos

- Es un paradigma de programación (modelo) que usa objetos y clases para diseñar aplicaciones.
- Las características mas importantes de este paradigma son:
  - Encapsulamiento
  - Herencia
  - Polimorfismo

¿Que es un paradigma de programación?

- Es un modelo. Una manera de ver y organizar las cosas, es decir las funciones y las variables (no hay mas cosas, en el fondo , o sos instrucción o sos dato)
- Pero antes, estaba el paradigma spaguetti, ¿se acuerdan? (lo expliqué cuando introduje el tema de estructuras de control) Y para resolver las cosas, apareció el famoso artículo *no hay nada peor que un goto* (debe haber muchas cosas peores…) y con ese artículo Dijkstra dio origen a la Programación estructurada. Año 1968
- Con el tiempo, los lenguajes fueron incorporando mas niveles de abstracción
- Actualmente, coexisten todos los paradigmas.
  - Assembler x86: Paradigma Spaguetti
  - C: Programación Estructurada y Modular
  - Python: Multiparadigna

Python es un lenguaje…

- Multiparadigma.
- Eso quiere decir que permite trabajar en varios paradigmas de programación al mismo tiempo:
  - Programación estructurada (de la forma en que lo venimos usando) y programación modular
  - Programación Orientada a Objetos
  - Programación Funcional

Lenguajes POO

C++
Java
Smalltalk
Eiffel
Lexico (en castellano)
Ruby
Python
OCAML
Object Pascal
CLIPS

Visual .net
Actionscript
COBOL
Perl
C#
Visual Basic.NET
PHP
Simula
Delphi
PowerBuilder

Ejercicio 2: El tipo lista encadenada

- ¿Se animan a crear el tipo lista encadenada con sus operaciones de: crear lista, insertar elemento, borrar elemento e imprimir elemento?
- ¿Se animan a crear una clase matriz dispersa implementada como lista enlazada, con sus operaciones de sumar y multiplicar?