

```

import threading, socket, time
class sock(threading.Thread):
    def __init__(self):
        self.sck=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        threading.Thread.__init__(self)
        self.flag=1
    def connect(self, addr, port, func):
        try:
            self.sck.connect((addr, port))
            self.handle=self.sck
            self.todo=2
            self.func=func
            self.start()
        except:
            print "Error: Could not connect"
    def listen(self, host, port, func):
        try:
            self.sck.bind((host, port))
            self.sck.listen(5)
            self.todo=1
            self.func=func
            self.start()
        except:
            print "Error: Could not bind"
    def run(self):
        while self.flag:
            if self.todo==1:
                x, ho=self.sck.accept()
                self.todo=2
                self.client=x
                self.handle=x
            else:
                dat=self.handle.recv(4096)
                self.data=dat
                self.func()
    def send(self, data):
        self.handle.send(data)
    def close(self):
        self.flag=0
        self.sck.close()

```

DAM/DAW

PROGRAMACIÓN

06.3

Introducción a las clases en Python

Indice

- [Objetos en Python](#)
- [Definición de Clases](#)
- [Creando Objetos](#)
- [Añadiendo Métodos](#)
- [Atributos de Clase](#)
- [Atributos de tipos mutables](#)
- [Copia de objetos](#)
- [Propiedades y Decoradores](#)
- [Métodos estáticos](#)
- [Herencia](#)

```
import threading, socket, time
class sock(threading.Thread):
    def __init__(self):
        self.sock=socket(socket.AF_INET,socket.SOCK_STREAM)
        threading.Thread.__init__(self)
        self.flag=1
    def connect(self,addr,port,func):
        try:
            self.sock.connect((addr,port))
            self.handle=self.sock
            self.todo=2
            self.func=func
            self.start()
        except:
            print "Error:Could not connect"
            def listen(self,host,port,func):
                try:
                    self.sock.bind((host,port))
                    self.sock.listen(2)
                    self.todo=1
                    self.func=func
                    self.start()
                except:
                    print "Error:Could not bind"
            def run(self):
                while self.flag:
                    if self.todo==1:
                        x,ho=self.sock.accept()
                        self.client=ho
                        self.handle=x
                    else:
                        dat=self.handle.recv(4096)
                        self.data=dat
                        self.func()
                def send(self,data):
                    self.handle.send(data)
            def close(self):
                self.flag=0
            self.sock.close()
```

Objetos en Python (I)

- Ya sabemos que Python soporta diferentes tipos de datos:

1234 3.14159 "Hola" [1, 2, 3, 5, 7, 11]
{ "GZ": "Galicia", "MD": "Madrid", "AS": "Asturias" }

- Cada uno de estos **objetos** o **instancias** se caracterizan por:
 - un **tipo** (clase)
 - una **representación interna de datos** (mediante tipos primitivos o por composición de objetos)
 - un conjunto de procedimientos para **interactuar** con el objeto
- Cada **instancia** es un tipo particular de objeto:
 - 1234 es una instancia de un *int* (**class 'int'**)
 - "Hola" es una instancia de un *string* (**class 'str'**)
 - 3.14159 es una instancia de un *float* (**class 'float'**)

Objetos en Python (II)

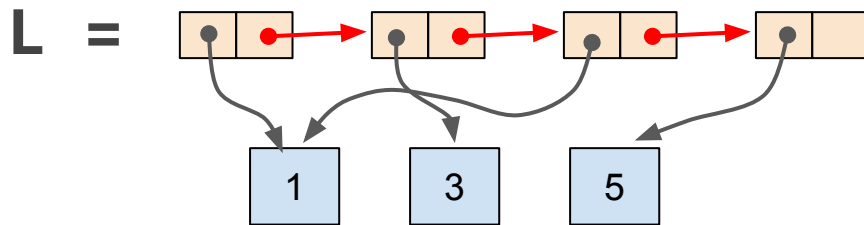
- En Python **todo** son objetos pertenecientes a un tipo (clase) determinado
- Estos objetos son **abstracciones** que encapsulan:
 - una **representación interna** mediante atributos de datos
 - un **interfaz** para interactuar con los objetos a través de métodos (procedimientos). Define el comportamiento pero oculta la implementación
- Se pueden **crear nuevas instancias** u objetos
- Se pueden **destruir objetos**:
 - Explícitamente, usando el comando **del**, o simplemente “olvidándonos” de ellos
 - Python dispone de un **“garbage collector”** (recolector de basura) que “eliminará” definitivamente aquellos objetos destruidos o inaccesibles (objetos cuya cuenta de referencias está a 0)

Objetos en Python (y III)

- Veamos un caso particular:

`[1, 3, 1, 5]` es un objeto de tipo **lista** (class 'list')

- ¿Cuál es su **representación interna**?



listas enlazadas
(realmente, nos da igual)

- ¿Cómo **manipulamos** la lista?

- `L[i]`, `L[i:j]`, `L[i,j,k]`, `+`
- `len()`, `min()`, `max()`, `del(L[i])`
- `L.append()`, `L.extend()`, `L.count()`, `L.index()`, `L.insert()`, `L.pop()`, `L.remove()`, `L.reverse()`, `L.sort()`

- La representación interna debe ser **privada** (oculta). Se interactúa con el objeto a través de **interfaces definidos**

Definición de Clases (I)

- En general, la creación de una nueva clase supone:
 - definir el **nombre** de la clase
 - definir sus **atributos**
 - definir e implementar sus **métodos**
- Usaremos la palabra reservada **class** para definir una nueva clase:

```
      nombre clase   clase padre
class  Coordinada(object):
    <definición de atributos y métodos>
```

- De igual modo al resto de estructuras de Python, tendremos que indentar el código para delimitar aquel que pertenece a la definición de la clase
- **object** indica que la clase deriva de ella. En Python 3 todas las clases derivan por defecto de object (se puede omitir de la definición). Es decir, object es una **superclase** de todas las clases en Python 3

Definición de Clases (II)

- Para poder crear objetos de la clase, necesitamos que la clase defina un método especial denominado **constructor**. Este método se invoca al crear el nuevo objeto y, a través de él, podemos pasarle argumentos a la clase para parametrizar la creación del objeto.
- A través de la clase **object**, Python proporciona un método constructor a todas las clases, llamado **__init__**, que nosotros podremos sobrescribir.
- Normalmente, los atributos de la clase se definen en el método **__init__** y se les asignan sus valores iniciales.
- Todos los métodos de las clases en Python, incluido **__init__**, tienen como primer argumento una variable denominada (por convención) **self**. Esta variable contiene una referencia al propio objeto y se emplea para poder “*identificar*” los atributos propios del objeto y diferenciarlos de cualquier otra variable local del mismo nombre.

Definición de Clases (y III)

- Siguiendo con el ejemplo anterior:

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

Coordenada
- x
- y

método
constructor
heredado
de la clase
object

datos de **inicialización** de
estado del nuevo objeto

parámetro con la **referencia** a
la nueva instancia

- La clase define dos nuevos **atributos**, de nombres **x** e **y**
- Estos atributos se inicializan a los **argumentos** **x** e **y** del constructor
- El uso de **self** permite distinguir entre los atributos del objeto y los argumentos del constructor

Creando objetos

- Vamos a crear objetos de la clase Coordenada anterior:

```
c = Coordenada(3, 4)
origen = Coordenada(0, 0)
print(c.x)
print(origen.x, origen.y)
```

Creamos nuevos objetos (instancias) de la clase `Coordenada` utilizando su nombre de clase

Proporcionamos los parámetros necesarios según la definición del constructor. El argumento para `self` no se proporciona, es proporcionado automáticamente por Python

- Empleamos el **operador "."** para acceder a los miembros del objeto
- **`c.x`** se interpreta como *"accede al valor asociado a `x` en el contexto (frame) `c`", es decir, la **variable de instancia `x`** del objeto `c`*

La función **`isinstance(obj, class)`** nos permite chequear si un objeto concreto es una instancia de una clase determinada. Ej:

```
isinstance(c, Coordenada)
```

Añadiendo métodos (I)

- Con la salvedad de que sólo actúan dentro de la clase, los métodos son equivalentes a las funciones: reciben parámetros, realizan operaciones devuelven valores.
- Python siempre pasa una referencia del objeto como primer argumento, por lo que todos los métodos deberán tener como mínimo un parámetro. Por convención, el nombre de ese parámetro es *self*
- Al igual que con los atributos, emplearemos el **operador “.”** para acceder a los diferentes métodos proporcionados por la clase
- Una clase heredará todos los métodos de su *superclase*. Opcionalmente, podrá proporcionar su propia definición de dichos métodos de forma que adapte su comportamiento a sus necesidades (*overriding*)
- Cualquier método puede ser invocado como **método de clase** (eq. Java *static*). Recuerda que debe recibir, al menos, un argumento

Añadiendo métodos (II). Métodos getter/setter

- Con objeto de garantizar la **encapsulación**, no deberíamos poder acceder directamente a la **estructura interna** de los objetos sino a través del **interfaz** correspondiente. Así, el diseñador es libre de modificarla.
- Todas las variables miembro deberían permanecer **privadas**, es decir, ocultas al exterior. Los lenguajes POO suelen incluir modificadores para configurar el acceso a las variables (*public*, *private*, *protected*)
- Para aquellas que precisen ser “accedidas” para leer o modificar su valor, la clase debe proporcionar los métodos de interfaz correspondientes
- Tradicionalmente, estos métodos reciben el nombre de **getters** (lectura) y **setters** (modificación). Su nombre se forma a partir de los prefijos **get** o **set**, y del nombre de la variable miembro. Por ej: **get_var()** o **getVar()**
- Python no dispone de modificadores de acceso para las variables. La ocultación de las variables miembro y la definición de sus métodos **get/set** se implementa mediante la definición de **propiedades**

Añadiendo métodos (III). Métodos getter/setter

- Continuemos con el ejemplo de la clase Coordenada. Vamos a añadir los *getter* y *setter* para modificar los valores *x* e *y* de los objetos.
- Vamos a implementarlos del modo tradicional (Java, C++,...). Más adelante veremos el *"pythonic way"* de hacerlo, sin duda, más “elegante”

```
class Coordenada:  
    def __init__(self, x, y):  
        self.x = x  
        self.y = y  
    def get_x(self):  
        return self.x  
    def set_x(self, x):  
        self.x = x  
    def get_y(self):  
        return self.y  
    def set_y(self, y):  
        self.y = y
```

Coordenada
- x - y
get_x() set_x() get_y() set_y()

Fíjate que todos los métodos incluyen el argumento *self* como primer parámetro para poder acceder a las variables miembro de la instancia

Añadiendo métodos (IV). Métodos getter/setter

- La definición de *getters* y *setters* independendiza la estructura de datos interna del acceso al estado de los objetos. De este modo se pueden hacer cambios en la estructura (cambio de nombres de variables, de tipos,...) sin afectar a los programas existentes que usen dichos objetos
- Debemos tener cuidado en Python ya que, al no disponer de modificadores de acceso, podremos acceder directamente a dichas variables miembro

```
c = Coordinada(3, 4)
c.x = 5           # set
print(c.x)       # get
```
- En Python se emplean diversas convenciones como:
 - Uso de _ al inicio del nombre para indicar que es una variable interna
 - Uso de __ al inicio provoca “cierta” ocultación de la variable al modificar su nombre interno anteponiendo el de la clase
- Volveremos sobre ello cuando hablemos de las **propiedades**

Añadiendo métodos (V). Más métodos

- Continuemos añadiendo métodos a la clase Coordenada. En este caso, uno que nos devuelva la distancia entre dos coordenadas:

Coordenada
-x -y
get_x() set_x() get_y() set_y() distance()

```
import math
class Coordenada:
    . . .
    def distance(self, coord):
        x_diff = self.x - coord.x
        y_diff = self.y - coord.y
        return math.sqrt(x_diff**2 + y_diff**2)
```

otra instancia de la clase Coordenada

- El método lo podremos invocar desde un objeto o desde la propia clase:

```
p1 = Coordenada(3, 4)
p2 = Coordenada(-3, -4)
print(p1.distance(p2))
```

```
p1 = Coordenada(3, 4)
p2 = Coordenada(-3, -4)
print(Coordenada.distance(p1, p2))
```

Añadiendo métodos (VI). Métodos especiales

- ¿Qué pasaría si, del mismo modo que hacemos con las listas, los strings o los diccionarios, quisieramos imprimir nuestro objetos?

```
p1 = Coordinada(3, 4)
print(p1)
<__main__.Coordinada object at 0x7f659b14bfd0>
```

- Python nos informa de que p1 es un objeto de la clase Coordinada y nos devuelve una referencia interna a su posición en memoria. Seguramente no la respuesta que esperábamos...
- Como sabemos, nuestra clase deriva de **object**, de la que hereda varios métodos especiales como, por ejemplo, el constructor **__init__**. Otro de ellos es **__str__**, que se invoca automáticamente cuando llamamos a la función `print()`. Igual que con el constructor, podemos **sobreescribirlo** para adaptarlo a nuestras necesidades. Tiene que devolver un **string**

Añadiendo métodos (VII). Métodos especiales

- Vamos a incluir el método `__str__` para que nos devuelva un **string** con el siguiente formato: “< coord_x, coord_y >”

```
class Coordenada:  
    . . .  
    def __str__(self):  
        return '<{}, {}>'.format(self.x, self.y)
```

- Ahora el resultado sería:

```
p1 = Coordenada(3, 4)  
print(p1)  
<3, 4>
```


Añadiendo métodos (VIII). Métodos especiales

- ¿Y si quisiéramos comparar dos coordenadas?

```
p1 = Coordenada(3, 4)
p2 = Coordenada(3, 4)
print(p1 == p2)
```

False

Son objetos diferentes por lo que la comparación, por defecto, nos devolverá un valor Falso

- El mismo concepto visto anteriormente, se aplica con otros **operadores** (+,-,==,<,>,<=,>=,...) y sus **métodos especiales** asociados. Algunos son:

operador	método
+	<code>__add__(self, other)</code>
-	<code>__sub__(self, other)</code>
==	<code>__eq__(self, other)</code>
<	<code>__lt__(self, other)</code>
len()	<code>__len__(self)</code>

¿Cómo modificaríamos la clase para que comparara coordenadas?

<https://docs.python.org/3/reference/datamodel.html#basic-customization>

Añadiendo métodos (y IX). Métodos especiales

- Vamos a incluir el método `__eq__` para comparar coordenadas. El valor de retorno será un **boolean**

```
class Coordenada:  
    . . .  
    def __eq__(self, other):  
        return self.x == other.x and self.y == other.y
```

- Ahora el resultado sería:

```
p1 = Coordenada(3, 4)  
p2 = Coordenada(3, 4)  
print(p1 == p2)  
True
```

Atributos de Clase (I)

- Un atributo de clase es un atributo **común** para todas las instancias de dicha clase. Dicho atributo podrá ser **accedido** y **modificado** desde cualquier objeto de la clase, así como desde la propia clase (en cierto modo, similar a los miembros *static* de Java o C++)

Fichero: u03_02.py

```
class MyClass:
    class_var = 1
    def __init__(self, val):
        self.inst_var = val
```

```
>>> from u03_02 import MyClass
>>> obj1 = MyClass(3)
>>> obj2 = MyClass(4)
>>> obj1.class_var, obj1.inst_var
(1, 3)
>>> obj2.class_var, obj2.inst_var
(1, 4)
>>> MyClass.class_var
1
>>> MyClass.class_var = 99
>>> obj2.class_var
99
>>> MyClass.class_var
99
```

Atributos de Clase (y II)

- Como regla general, no deberíamos usar atributos de clase salvo en contados casos:
 - Definición de constantes o valores por defecto

```
class Circle:
    _pi = 3.14159
    def __init__(self, radius):
        self.radius = radius
    def area(self):
        return Circle._pi * self.radius**2
```

- Gestionar cierta información común a todos los objetos

```
class Person:
    _all_people = []
    def __init__(self, name):
        self.name = name
        Person._all_people.append(self)
    def remove(self):
        Person._all_people.remove(self)
```

Atributos de tipos mutables (I)

- Tenemos que tener especial cuidado cuando nuestros atributos son de tipos mutables. Recordemos que tanto en asignaciones, como en argumentos o retornos de funciones, se emplean **referencias** a dichos objetos, no los valores concretos a los que “apuntan”.
- Esto puede hacer que nos encontremos con problemas inesperados al modificar inadvertidamente un **atributo mutable** de un objeto, cuya referencia fue propagada al exterior, desde fuera de dicho objeto
- La clase de ejemplo que se muestra a continuación, tiene un atributo de tipo **lista** y, por tanto, mutable. Fíjate como en el **getter** correspondiente se devuelve la propia referencia a dicha lista, y no una referencia a una copia de la lista (como debería ser). Esto va a provocar que cambios “externos” en dicha lista afecten al atributo del objeto

Atributos de tipos mutables (II)

Fichero: u03_02.py

```
class Temp:
    def __init__(self, temp):
        self.temps = []
        self.set_temp(temp)
    def set_temp(self, temp):
        self.temp = temp
        self.temps.append(temp)
    def get_temp(self):
        return self.temp
    def get_temps(self):
        return self.temps
```

Obtenemos una copia de la lista de t^a

En realidad, la clase **no** nos devuelve una **copia**, devuelve la **misma lista** (fíjate que tiene el **mismo id**)


Si hacemos cambios en la "copia" (que no es tal), nos afecta al propio objeto

```
>>> from u03_02 import Temp
>>> t1 = Temp(21)
>>> t1.set_temp(23)
>>> print("Ta actual:", t1.get_temp())
Ta actual: 23
>>> print("Ta registradas:", t1.get_temps())
Ta registradas: [21, 23]
>>> lista_t = t1.get_temps()
>>> id(t1.temps), id(lista_t)
(139869576302280, 139869576302280)
>>> lista_t.extend([18, -2, 35])
>>> print("Ta registradas:", t1.get_temps())
Ta registradas: [21, 23, 18, -2, 35]
```

Atributos de tipos mutables (y III)

- Deberemos garantizar que, cuando se trata de atributos mutables, nuestro *getter* devuelva una **copia** del mismo
- Aplicaremos lo mismo que ya vimos respecto a las copias de variables de tipos estructurados (*swallow-copy* y *deep-copy*)
- Nuestro caso lo podríamos resolver fácilmente de la forma siguiente:

```
class Temp:
    def __init__(self, temp):
        self.temps = []
        self.set_temp(temp)
    def set_temp(self, temp):
        self.temp = temp
        self.temps.append(temp)
    def get_temp(self):
        return self.temp
    def get_temps(self):
        return self.temps[:]
```



```
>>> from u03_02 import Temp
>>> t1 = Temp(21)
>>> t1.set_temp(23)
>>> print("Tª actual:", t1.get_temp())
Tª actual: 23
>>> print("Tª registradas:", t1.get_temps())
Tª registradas: [21, 23]
>>> lista_t = t1.get_temps()
>>> id(t1.temps), id(lista_t)
(139863355187016, 139863355293896)
>>> lista_t.extend([18, -2, 35])
>>> print("Tª registradas:", t1.get_temps())
Tª registradas: [21, 23]
```

Copia de objetos (I)

- Similares problemas se nos plantean cuando queremos hacer la **copia** de un objeto. Podemos abordar la copia de objetos de diversas maneras:
 - Crear nuestro propio **método** de copia:

```
class Temp:
    def __init__(self, temp):
        self.temps = []
        self.set_temp(temp)
    def set_temp(self, temp):
        self.temp = temp
        self.temps.append(temp)
    def get_temp(self):
        return self.temp
    def get_temps(self):
        return self.temps[:]

    def copy(self):
        new_temp = Temp(self.temp)
        new_temp.temps = self.temps[:]
        return new_temp
```

```
>>> from u03_02 import Temp
>>> t1 = Temp(21)
>>> t1.set_temp(23)
>>> print("Tª registradas:", t1.get_temps())
Tª registradas: [21, 23]
>>> t2 = t1.copy()
>>> t1
<U03_02.u03_02.Temp object at 0x7f79dfee4c88>
>>> t2
<U03_02.u03_02.Temp object at 0x7f79dfee4e10>
>>> id(t1.temps), id(t2.temps)
(140161424711496, 140161424710152)
>>> t2.set_temp(30)
>>> print("Tª registradas T2:", t2.get_temps())
Tª registradas: [21, 23, 30]
>>> print("Tª registradas T1:", t1.get_temps())
Tª registradas: [21, 23]
```


Copia de objetos (y II)

- Emplear las funciones *copy()* y *deepcopy()* del módulo *copy*, para realizar una *swallow-copy* ó *deep-copy* de nuestro objeto. El emplear una función u otra dependerá de si nuestros objetos tienen atributos mutables susceptibles verse modificados a través del **objeto copia**.
 - Tenemos la posibilidad sobrescribir (*overriding*) los métodos *__copy__* y *__deepcopy__* si queremos implementar nuestras propias versiones de dichos procesos de copia.
 - <https://docs.python.org/3/library/copy.html>

```
>>> import copy
>>> t1 = Temp(21)
>>> t2 = copy.copy(t1)
>>> t1
<U03_02.u03_02.Temp object at 0x7f79dfec4c88>
>>> t2
<U03_02.u03_02.Temp object at 0x7f79df838080>
>>> id(t1.temps), id(t2.temps)
(140161424711496, 140161424711496)
```

```
>>> import copy
>>> t1 = Temp(21)
>>> t2 = copy.deepcopy(t1)
>>> t1
<U03_02.u03_02.Temp object at 0x7f79dfec4c88>
>>> t2
<U03_02.u03_02.Temp object at 0x7f79df848438>
>>> id(t1.temps), id(t2.temps)
(140161424711496, 140161366461576)
```

Propiedades y Decoradores (I)

- A diferencia de otros lenguajes como Java ó C++, Python no proporciona modificadores de acceso a los miembros de la clase de forma que podamos ocultar la estructura interna de nuestros objetos.
- Como ya comentamos previamente en nuestro ejemplo de la clase *Coordenada*, podemos acceder directamente a sus atributos (x, y) usando el operador punto (.) sin necesidad de usar los *getter* o *setter* definidos.

```
c = Coordenada(3, 4)
c.x = 5      # equivale a c.set_x(5)
print(c.x)   # equivale a print(c.get_x())
```

- Python nos proporciona mecanismos como las **propiedades** y los **decoradores**, que nos permiten “reescribir” nuestras clases de forma que podamos acceder a sus atributos con el operador punto (que es más “*pythónico*”) pero a través de *getters* y *setters* (obteniendo cierta ocultación)

Propiedades y Decoradores (II)

❑ Propiedades

- La función *property()* nos permite definir un **atributo** de la clase y asociarle una serie de métodos que se invocarán al acceder (*get*) y modificar (*set*) su valor. Es lo que se denominan **atributos gestionados**.
- El uso de la función *property()* es el siguiente:

```
atributo = property(fget=None, fset=None, fdel=None, doc=None)
```

- donde,
 - **atributo**, nombre del nuevo atributo de la clase
 - *fget*, función que devuelve el valor del atributo (*getter*)
 - *fset*, función que permite modificar el valor del atributo (*setter*)
 - *fdel*, función que gestiona la eliminación del atributo del objeto
 - *doc*, texto para documentación (*docstring*)

Propiedades y Decoradores (III)

- El valor retornado por la función *property()* será el nuevo atributo gestionado de la clase al que accederemos con el operador punto.
- Si leemos el valor del atributo mediante *obj.atributo*, Python invocará al método que hayamos definido en *fget*.
- De igual modo, si modificamos el valor del atributo mediante *obj.atributo = valor*, Python invocará al método que hayamos definido en el parámetro *fset* de la propiedad.
- Python invocará el método que hayamos definido en el parámetro *fdel* de la propiedad cuando hagamos un *del obj.atributo* con la intención de eliminar el atributo del objeto.
- Al nombre de la variable y métodos de la clase empleados por el atributo gestionado, les antepondremos en el nombre un guión bajo () con objeto de resaltar su carácter **interno** y evitar que sean invocados

Propiedades y Decoradores (IV)

- Nuestra clase Coordenada, podría quedar así:

```
class Coordenada:
    def __init__(self, x, y):
        self._x = x
        self._y = y
    def _get_x(self):
        return self._x
    def _set_x(self, x):
        self._x = x
    def _get_y(self):
        return self._y
    def set_y(self, y):
        self._y = y

# atributos
x = property(fget=_get_x, fset=_set_x, doc="La propiedad x.")
y = property(fget=_get_y, fset=_set_y, doc="La propiedad x.")
```

Fíjate como los nombre de las variables miembro `_x` e `_y`, así como de los métodos `get/set` correspondientes, llevan un `_` delante

Ahora, podemos acceder a los valores de los objetos `Coordenada` usando sus nuevos atributos `x` e `y`:

```
Coordenada c1(3, 4)
print(c1.x, c1.y) # --> imprime 3 4
c1.y = 7
print(c1.x, c1.y) # --> imprime 3 7
```

Propiedades y Decoradores (V)

Decoradores

- En esencia, los decoradores son funciones que aceptan otra función como argumento. En general, añaden alguna nueva funcionalidad a la función recibida pero no modifican la función original.
- Para aplicar un decorador a una función, usamos la siguiente sintaxis:
`@nombre_decorador`
- La línea anterior se escribirá justo antes de la definición de la función a la que se lo queremos aplicar.
- Podemos aplicar varios decoradores a una misma función, simplemente “apilando” dichos decoradores antes de la definición de la función.
- Python nos proporciona diferentes decoradores (como *`@staticmethod`* ó *`@property`*) para diferentes tareas y podemos crearnos los nuestros (https://python101.pythonlibrary.org/chapter25_decorators.html)

Propiedades y Decoradores (VI)

➤ *@property*

- El decorador *@property* nos permite **simplificar** la definición de un **atributo** y su **getter** simplemente decorando un método (que actuará como *getter*) y cuyo nombre será el de la nueva propiedad.
- Por ejemplo, volviendo a nuestra clase *Coordenada*:

```
class Coordenada:
    def __init__(self, x, y):
        self._x = x
        self._y = y
    @property
    def x(self):
        return self._x
    @property
    def y(self):
        return self._y
```

Los métodos **x()** e **y()** son ahora **propiedades** que funcionan como *getters*

Podemos acceder a los valores de los objetos *Coordenada* usando sus nuevos atributos (propiedades) **x** e **y**:

```
Coordenada c1(3, 4)
print(c1.x, c1.y) # --> imprime 3 4
```

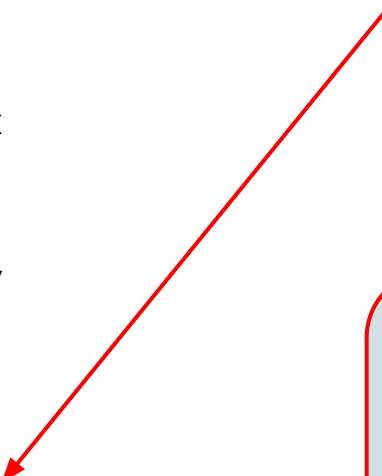
Propiedades y Decoradores (y VII)

- El decorador `@nombre_propiedad.setter` nos permite **simplificar** la definición de un **setter** para nuestras nuevas propiedades:

```
class Coordenada:
    def __init__(self, x, y):
        self._x = x
        self._y = y

    @property
    def x(self):
        return self._x
    @property
    def y(self):
        return self._y

    @x.setter
    def x(self, y):
        self._x = x
    @y.setter
    def y(self, y):
        self._y = y
```



Usando los decoradores `@x.setter` e `@y.setter` añadimos métodos *setter* a los atributos (propiedades) **x** e **y**

Fíjate que estos métodos reciben un **valor** para actualizar la variable miembro interna de la clase

Ahora podemos modificar los valores de los atributos (propiedades) **x** e **y**:

```
Coordenada c1(3, 4)
print(c1.x, c1.y) # --> imprime 3 4
c1.y = 7
print(c1.x, c1.y) # --> imprime 3 7
```


Métodos estáticos (I)

- Para definir un método **estático** de una clase, es decir, un método que podemos invocar sin necesidad de crear una instancia de la misma, no tenemos más que usar el decorador ***@staticmethod***
- Al ser un método estático que se invoca usando directamente la clase y no una instancia de la misma (aunque está permitido), no incluiremos en su definición el parámetro ***self*** que referencia al objeto desde el que se llamaron los métodos de la clase.
- Alternativamente, disponemos de otro decorador, ***@classmethod***, que también nos permitiría definir un método como estático. En este caso, debemos incluir un primer argumento, habitualmente denominado ***cls***, que recogerá el nombre de la clase desde la que invocamos dicho método. Esto tiene cierta utilidad, por ejemplo, cuando queremos saber desde qué **subclase** se invocó un método heredado de este tipo.

Métodos estáticos (y II)

- Vamos a añadir a nuestra clase un método estático para calcular la distancia entre dos objetos Coordena

```
import math

class Coordinada:
    def __init__(self, x, y):
        self._x = x
        self._y = y

    ...

    @staticmethod
    def distance(obj1, obj2):
        x_diff = obj1.x - obj2.x
        y_diff = obj1.y - obj2.y
        return math.sqrt(x_diff**2 + y_diff**2)
```

Fíjate que el método estático de la clase no tiene parámetro *self*

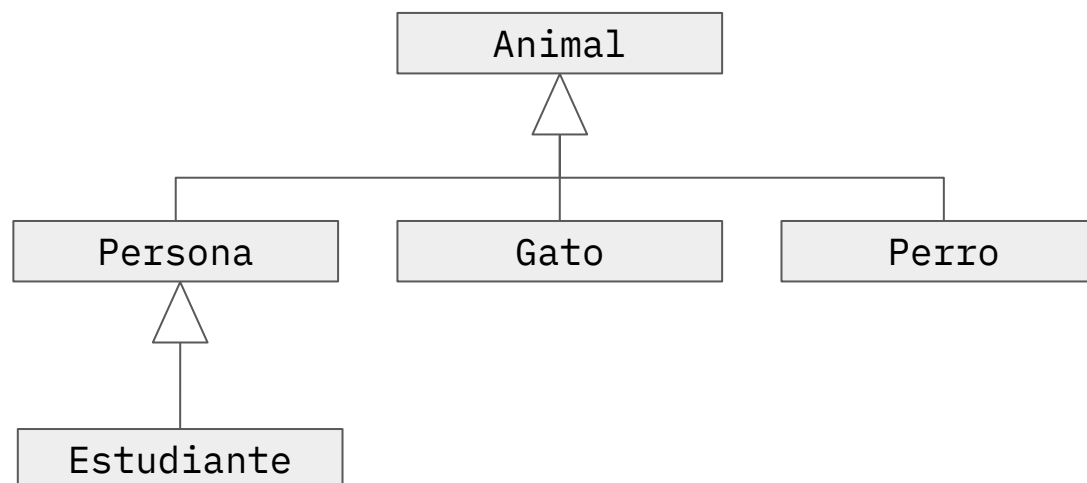
Invocamos el método estático usando la clase:

```
Coordinada c1(3, 4)
Coordinada c2(-2, 5)

print(Coordinada.distance(c1, c2)
# --> imprime 5
```

Herencia (I)

- La herencia nos permite crear **jerarquías** de clases
- Estas jerarquías nos permiten agrupar en las clases padre (**superclase**) características y comportamientos comunes de sus hijos (**subclase**), al tiempo que vamos refinando los comportamientos (**especialización**) a medida que descendemos por los diferentes niveles de la jerarquía.
- Vamos a ver cómo maneja la herencia en Python. Para ello, implementaremos las clases de la siguiente jerarquía:



Herencia (II)

```
class Animal:
```

```
    def __init__(self, edad):  
        self.edad = edad  
        self.nombre = None
```

```
    def get_edad(self):  
        return self.edad
```

```
    def get_nombre(self):  
        return self.nombre
```

```
    def set_edad(self, edad):  
        self.edad = edad
```

```
    def set_nombre(self, nombre = ""):  
        self.nombre = nombre
```

```
    def __str__(self):  
        return "animal: {}: {}".format(self.nombre, self.edad)
```

getters

setters

`__init__` y `__str__`
se heredan de `object`

Hereda de `Animal`Usa su método `__init__` y sobrescribe el método `__str__`

```
class Gato(Animal):
```

```
    def habla(self):  
        print("miau!!")
```

```
    def __str__(self):  
        return "gato: {}: {}".format(self.nombre, self.edad)
```

Añade nueva funcionalidad a través de un nuevo método

Herencia (III)

```
>>> import u03_02
>>> mi_animal = u03_02.Animal(3)
>>> print(mi_animal)
animal:None:3
>>> mi_animal.set_nombre("dude")
>>> print(mi_animal)
animal:dude:3
>>> mi_animal.get_edad()
3
>>> yin = u03_02.Gato(1)
>>> yin.habla()
miau!!
>>> yin.set_nombre("YinYang")
>>> yin.get_nombre()
'YinYang'
>>> print(yin)
gato:YinYang:1
>>> print(Animal.__str__(yin))
animal:YinYang:1
>>> blob = u03_02.Animal(1)
>>> blob.set_nombre()
>>> print(blob)
animal::1
```

Importamos el módulo donde están definidas las clases

Se busca el método en la clase (Gato). Como no se encuentra, se sube por la jerarquía hasta encontrarlo (Animal)

Podemos invocar el método *sobrescrito* del padre

Se usa el valor *por defecto* del método

Herencia (IV)

```
class Perro(Animal):  
    def habla(self):  
        print("guau!!")  
    def __str__(self):  
        return "perro: {}: {}".format(self.nombre, self.edad)
```

```
class Persona(Animal):  
    def __init__(self, nombre, edad):  
        Animal.__init__(self, edad)  
        self.set_nombre(nombre)  
        self.amigos = []  
    def get_amigos(self):  
        return self.amigos[:]  
    def add_amigo(self, amigo):  
        if isinstance(amigo, Persona) and amigo not in self.amigos:  
            self.amigos.append(amigo)  
    def habla(self):  
        print("hola!")  
    def __str__(self):  
        return "persona: {}: {}".format(self.nombre, self.edad)
```

Llamada al *constructor* de la *superclase*

Nuevo atributo

Añadimos nuevos objetos sólo de tipo *Persona* (incluye subclases)

Herencia (V)

```
import random

class Estudiante(Persona):
    def __init__(self, nombre, edad, estudia=None):
        Persona.__init__(self, nombre, edad)
        self.estudia = estudia
    def get_estudia(self):
        return self.estudia
    def set_estudia(self, estudia):
        self.estudia = estudia
    def habla(self):
        r = random.random()
        if r < 0.25:
            print("tengo deberes")
        elif 0.25 <= r and r < 0.5:
            print("necesito dormir")
        elif 0.5 <= r and r < 0.75:
            print("tengo hambre!")
        else:
            print("estoy viendo la tele")
    def __str__(self):
        return "estudiante:{}: {}: {}".format(self.nombre, self.edad, self.estudia)
```

Llamada al *constructor* de la *superclase*

Nuevo atributo

Genera un *float* entre *[0, 1)*

Herencia (y VI)

```
>>> import u03_02
>>> eric = u03_02.Persona("eric", 41)
>>> joe = u03_02.Persona("joe", 32)
>>> print(joe)
persona:joe:32
>>> josh = u03_02.Estudiante("josh", 19, "Biología")
>>> fred = u03_02.Estudiante("fred", 18)
>>> print(josh)
estudiante:josh:19:Biología
>>> print(fred)
estudiante:fred:18:
>>> josh.habla()
'tengo hambre!'
>>> eric.add_amigo(joe)
>>> eric.add_amigo(josh)
>>> eric.add_amigo(fred)
>>> for amigo in eric.get_amigos():
...     print(amigo.get_nombre() + "> ", end="")
...     amigo.habla()
joe> hola!
josh> necesito dormir
fred> estoy viendo la tele
```

Enlazado *dinámico* del método (*polimorfismo*)