UNIVERSIDAD DE MÁLAGA

FGUMA FUNDACIÓN GENERAL UNIVERSIDAD DE MÁLAGA

CURSOS ONLINE

FUNDACIÓN GENERAL UNIVERSIDAD DE MÁLAGA

# Objetos Python

## Unidad 11

Python para principiantes

dgm@uma.es

# Introducción

- La Programación Orientada a Objetos es la evolución de la Programación Estructurada
- Trata de modelar los programas como si fueran “Objetos” del mundo real
  - Atributos de los objetos
  - Acciones que pueden realizar
- No tendrás la visión global hasta que lo utilices en el **contexto de un problema real complejo**

# La Orientación a Objetos está en todas partes (y ya la habías estado usando)

# 5. Data Structures

This chapter describes some things you've learned about already in more detail, and adds some new things as well.

## 5.1. More on Lists

The list data type has some more methods. Here are all of the methods of list objects:

`list.`**`append`**(*x*)
Add an item to the end of the list. Equivalent to `a[len(a):] = [x]`.

`list.`**`extend`**(*L*)
Extend the list by appending all the items in the given list. Equivalent to `a[len(a):] = L`.

`list.`**`insert`**(*i*, *x*)
Insert an item at a given position. The first argument is the index of the element before which to insert, so `a.insert(0, x)` inserts at the front of the list, and `a.insert(len(a), x)` is equivalent to `a.append(x)`.

`list.`**`remove`**(*x*)
Remove the first item from the list whose value is *x*. It is an error if there is no such item.

`list.`**`pop`**([*i*])
Remove the item at the given position in the list, and return it. If no index is specified, `a.pop()` removes and returns the last item in the list. (The square brackets around the *i* in the method signature denote that the parameter is optional, not that you should type square brackets at that position. You will see this notation frequently in the Python Library Reference.)

https://docs.python.org/3/tutorial/datastructures.html

# 12.6. `sqlite3` — DB-API 2.0 interface for SQLite databases

**Source code:** Lib/sqlite3/

---

SQLite is a C library that provides a lightweight disk-based database that doesn't require a separate server process and allows accessing the database using a nonstandard variant of the SQL query language. Some applications can use SQLite for internal data storage. It's also possible to prototype an application using SQLite and then port the code to a larger database such as PostgreSQL or Oracle.

The sqlite3 module was written by Gerhard Häring. It provides a SQL interface compliant with the DB-API 2.0 specification described by PEP 249.

To use the module, you must first create a `Connection` object that represents the database. Here the data will be stored in the `example.db` file:

```
import sqlite3
conn = sqlite3.connect('example.db')
```

You can also supply the special name `:memory:` to create a database in RAM.

Once you have a `Connection`, you can create a `Cursor` object and call its `execute()` method to perform SQL commands:

```
c = conn.cursor()

# Create table
c.execute('''CREATE TABLE stocks
             (date text, trans text, symbol text, qty real, price real)''')
```

https://docs.python.org/3/library/sqlite3.html

# Comencemos con los programas

```
inp = input('Planta Europa?')
usf = int(inp) + 1
print('Planta USA', usf)
```

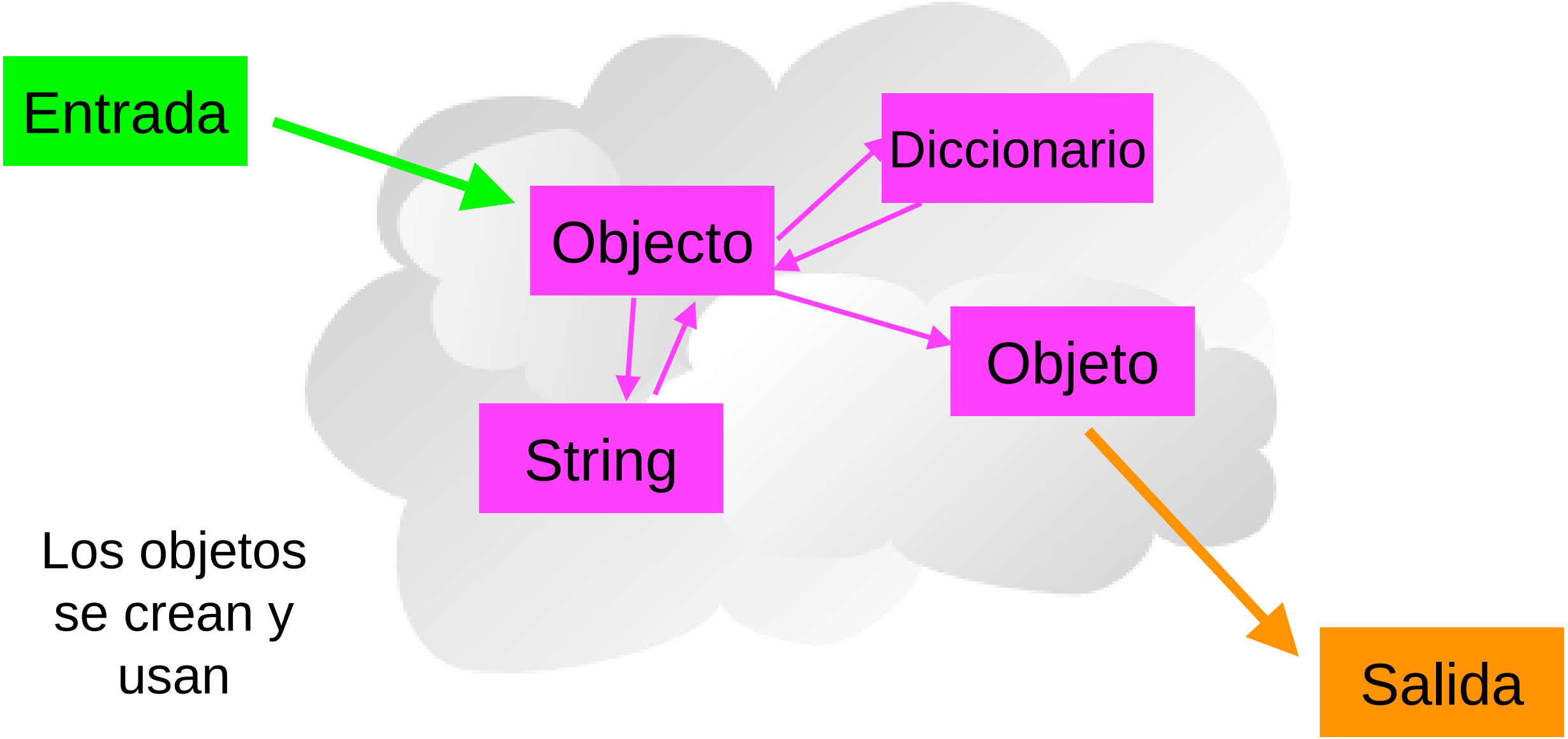
Planta Europa? 0
Planta EEUU 1

# Orientación a Objetos

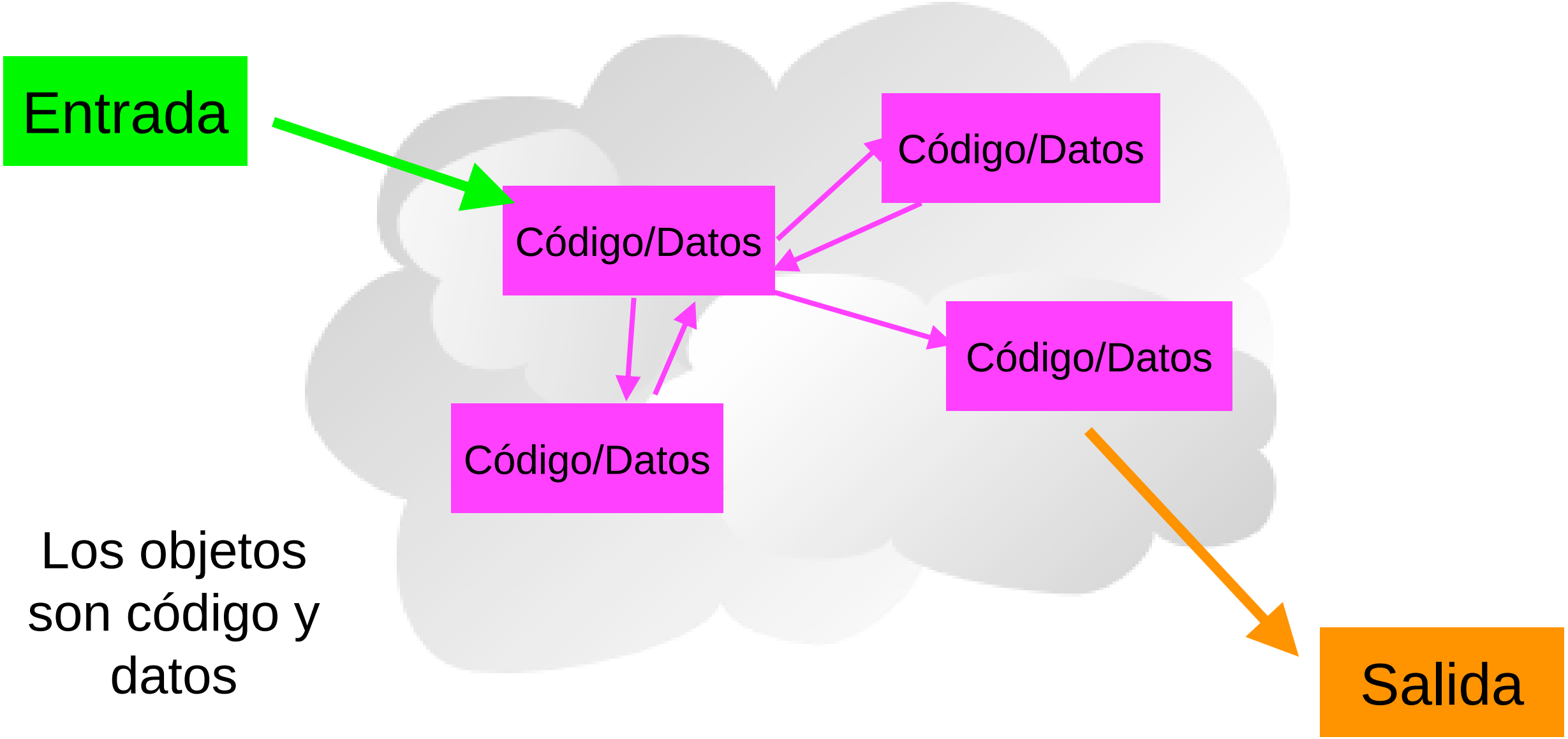
- Un programa está hecho de muchos objetos **cooperando**
- En lugar de ser  el "programa completo" – **cada objeto** es una **pequeña "isla"** dentro del programa trabajando cooperativamente con otros objetos
- Un programa está hecho de uno o más objetos trabajando juntos – los objetos hacen uso de las **capacidades de los demás**

# Objeto

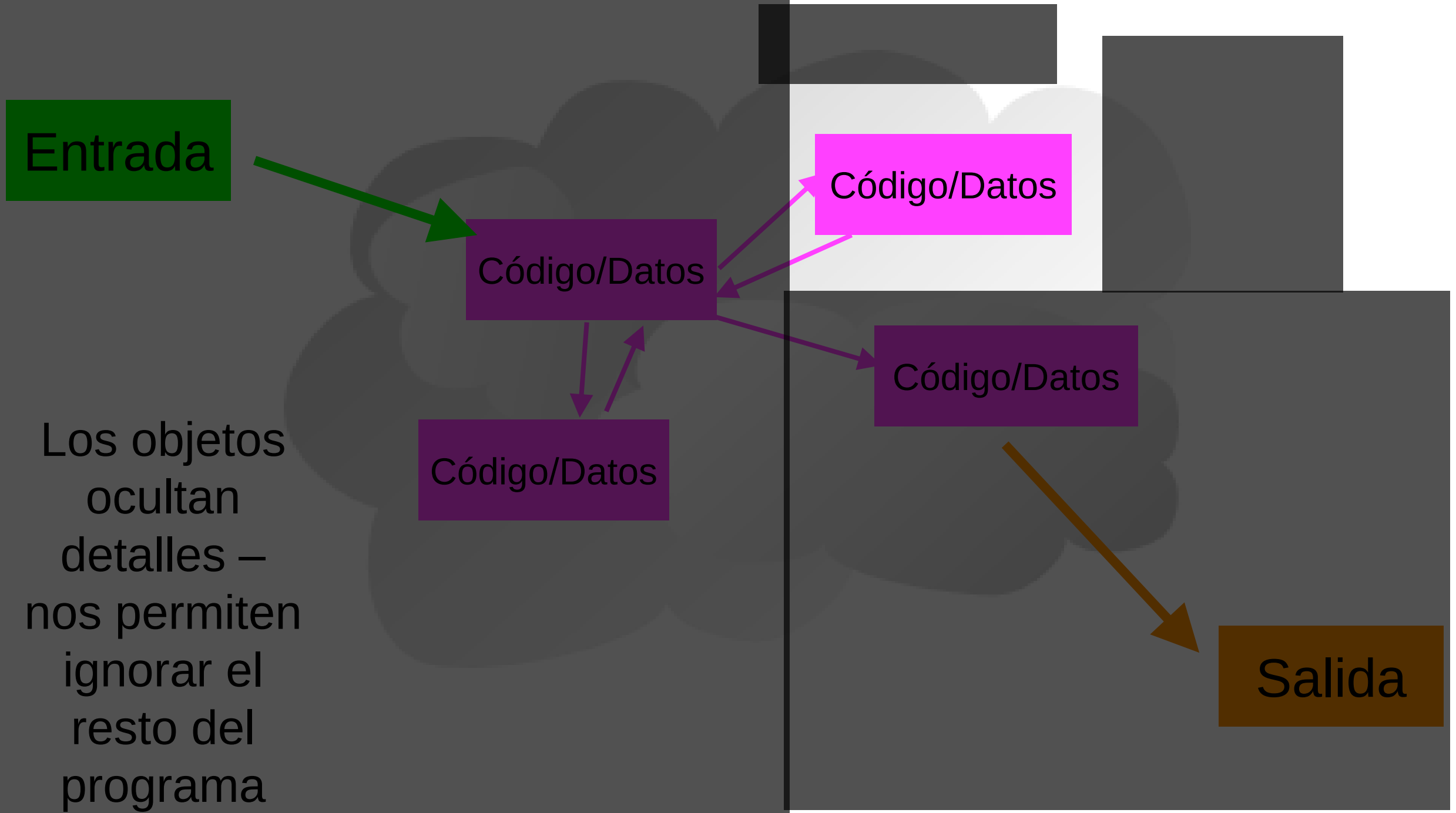
- Un Objeto incluye **Código y Datos**
- Un aspecto clave de la orientación a objetos es dividir un problema en partes más pequeñas manejables (**divide y vencerás**)
- Los objetos tienen límites/fronteras que nos permiten ignorar los detalles innecesarios
- Hemos estado usando objetos todo el rato: objetos string, objetos integer, objetos diccionarios, objetos listas...



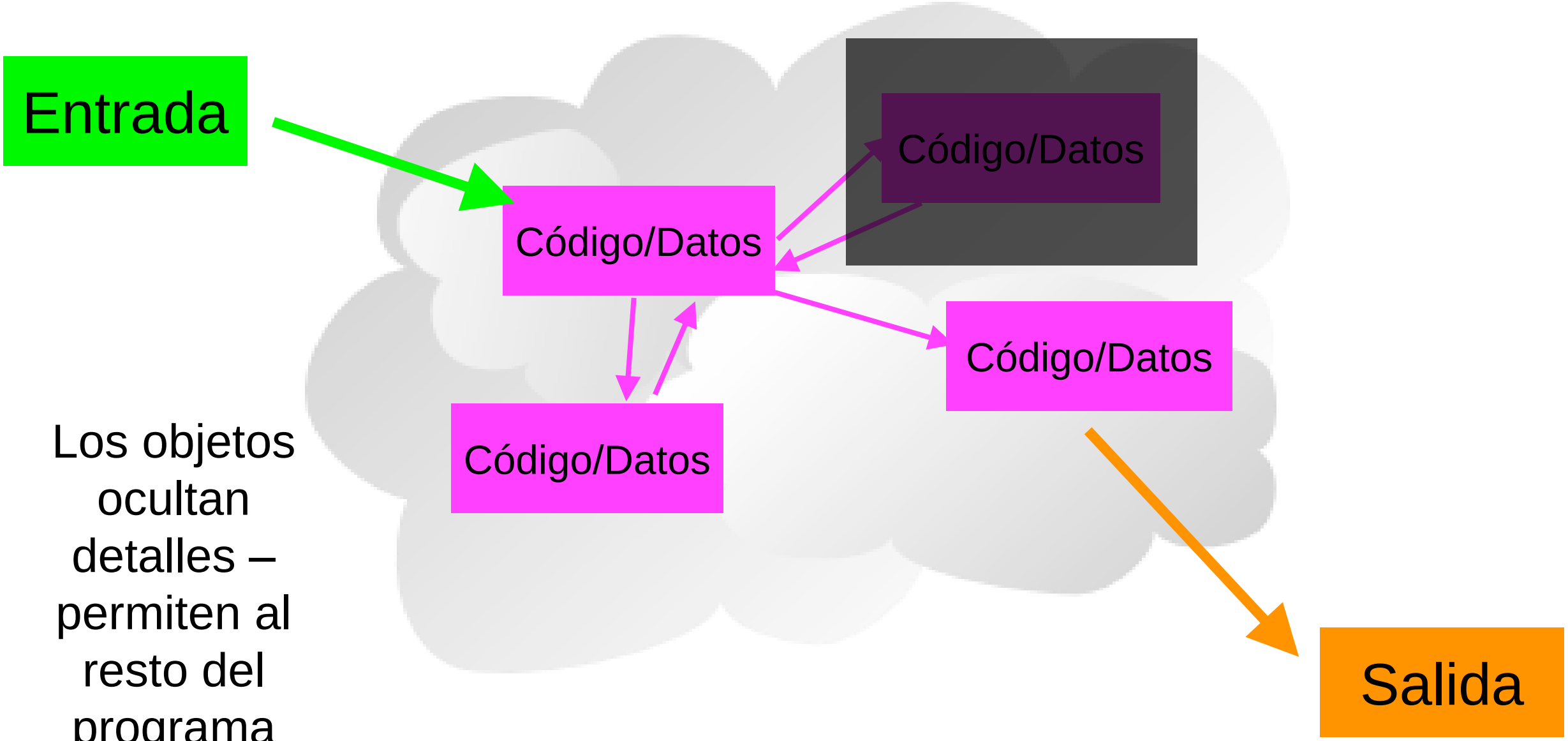

Los objetos se crean y usan




Los objetos son código y datos




Los objetos ocultan detalles – nos permiten ignorar el resto del programa




Los objetos ocultan detalles – permiten al resto del programa ignorarlos

# Definiciones

- **Clase** – una plantilla
- **Método o Mensaje** – Una capacidad definida de una clase
- **Campo o atributo** – Dato en una clase
- **Objeto o Instancia** – Una instancia particular de una clase

# Terminología: Clase

Define las características **abstractas** de una cosa (**objeto**), incluyendo sus **características (sus atributos, campos o propiedades)** y su **comportamiento (las cosas que puede hacer, o métodos u operaciones)**. Se podría decir que una clase es una plantilla o plano que describe la naturaleza de algo. Por ejemplo, la clase Perro podría consistir de rasgos compartidos por todos los perros, tales como la raza y el color (características), y la habilidad para ladrar y sentarse (comportamientos).

https://es.wikipedia.org/wiki/Programaci%C3%B3n_orientada_a_objetos

# Terminología: Instancia

Podemos tener instancias u objetos de una clase. Las instancias son los objetos reales creados durante la ejecución del programa. En "jerga" de programador, el objeto Lassie **es una instancia** de la clase Perro. El conjunto de valores de los atributos de un objeto en particular, forman su **estado**. El objeto consiste en el estado y el comportamiento definido en la clase del objeto.

Se usan los términos Objecto e Instancia como sinónimos

https://es.wikipedia.org/wiki/Programaci%C3%B3n_orientada_a_objetos

# Terminología: Método

Una habilidad de un objeto. En lenguaje, los métodos son "**verbos**". Lassie, siendo un Perro, tiene la habilidad para ladrar. Así que ladrar() es uno de los métodos de Lassie. Podría tener más métodos también, por ejemplo sentar() o comer() o pasear(). Dentro del programa, normalmente usar un método **solo afecta a un objeto en particular**; todos los Perros saben ladrar, pero solo necesitamos que lo haga uno en particular

Método y Mensaje son usados como sinónimos.

https://es.wikipedia.org/wiki/Programaci%C3%B3n_orientada_a_objetos

# Algunos objetos Python

```
>>> x = 'abc'
>>> type(x)
<class 'str'>
>>> type(2.5)
<class 'float'>
>>> type(2)
<class 'int'>
>>> y = list()
>>> type(y)
<class 'list'>
>>> z = dict()
>>> type(z)
<class 'dict'>
```

```
>>> dir(x)
[ … 'capitalize', 'casefold', 'center', 'count',
'encode', 'endswith', 'expandtabs', 'find',
'format', … 'lower', 'lstrip', 'maketrans',
'partition', 'replace', 'rfind', 'rindex', 'rjust',
'rpartition', 'rsplit', 'rstrip', 'split',
'splitlines', 'startswith', 'strip', 'swapcase',
'title', 'translate', 'upper', 'zfill']
>>> dir(y)
[… 'append', 'clear', 'copy', 'count', 'extend',
'index', 'insert', 'pop', 'remove', 'reverse',
'sort']
>>> dir(z)
[…, 'clear', 'copy', 'fromkeys', 'get', 'items',
'keys', 'pop', 'popitem', 'setdefault', 'update',
'values']
```

# Una Clase de ejemplo

**class** es una palabra reservada

Cada objeto Fiestero tiene una parte de codigo

Decirle al objeto que "ejecute" el código fiesta() dentro de él

```
class Fiestero:
    x = 0

    def fiesta(self) :
      self.x = self.x + 1
      print("Tan lejos",self.x)

obj = Fiestero()

obj.fiesta()
obj.fiesta()
obj.fiesta()
```

Esta es la plantilla para hacer objetos Fiestero

Cada objeto Fiestero tiene una parte de datos

Construye un objeto Fiestero y almacénalo en una variable

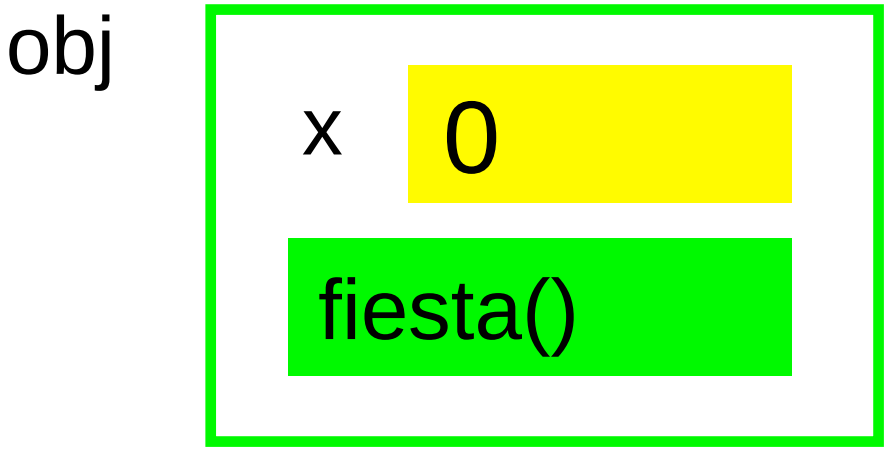
Fiestero.fiesta(obj)

```
class Fiestero:
    x = 0

    def fiesta(self) :
      self.x = self.x + 1
      print("Tan lejos",self.x)

obj = Fiestero()

obj.fiesta()
obj.fiesta()
obj.fiesta()
```

```
$ python party1.py
```

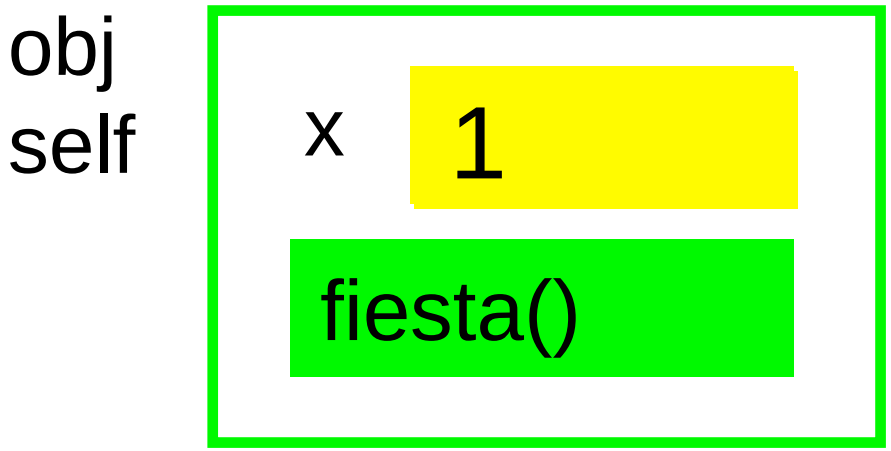
```
class Fiestero:
    x = 0

    def fiesta(self) :
      self.x = self.x + 1
      print("Tan lejos",self.x)

obj = Fiestero()

obj.fiesta()
obj.fiesta()
obj.fiesta()
```

$ python party1.py



```
class Fiestero:
    x = 0

    def fiesta(self) :
      self.x = self.x + 1
      print("Tan lejos",self.x)

obj = Fiestero()

obj.fiesta()
obj.fiesta()
obj.fiesta()
```

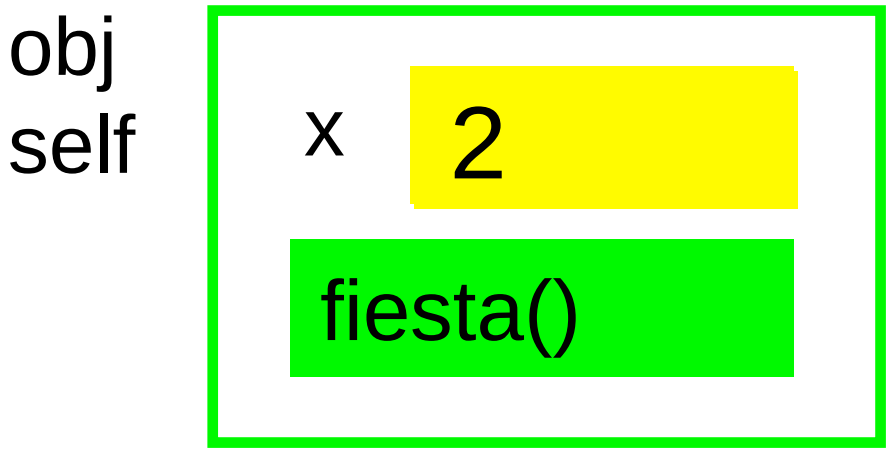
```
$ python party1.py
Tan lejos 1
```




Fiestero.fiesta(obj)

```
class Fiestero:
    x = 0

    def fiesta(self) :
      self.x = self.x + 1
      print("Tan lejos",self.x)

obj = Fiestero()

obj.fiesta()
obj.fiesta()
obj.fiesta()
```

```
$ python party1.py
Tan lejos 1
Tan lejos 2
```




Fiestero.fiesta(obj)

```
class Fiestero:
    x = 0

    def fiesta(self) :
      self.x = self.x + 1
      print("Tan lejos",self.x)

obj = Fiestero()

obj.fiesta()
obj.fiesta()
obj.fiesta()
```

```
$ python party1.py
Tan lejos 1
Tan lejos 2
Tan lejos 3
```


Fiestero.fiesta(obj)

# Jugando con dir() y type()

# Una forma sencilla de encontrar capacidades

- El comando **dir()** muestra capacidades
- Ignora las que empiezan con subrayados "_" - son usadas por Python
- El resto son operaciones reales que el objeto puede realizar
- Es como **type()** – nos dice *algo* sobre una variable

```
>>> y = list()
>>> type(y)
<class 'list'>
>>> dir(y)
['__add__', '__class__',
'__contains__', '__delattr__',
'__delitem__', '__delslice__',
'__doc__', … '__setitem__',
'__setslice__', '__str__',
'append', 'clear', 'copy',
'count', 'extend', 'index',
'insert', 'pop', 'remove',
'reverse', 'sort']
>>>
```

```
class Fiestero:
    x = 0

    def fiesta(self) :
      self.x = self.x + 1
      print("Tan lejos",self.x)

obj = Fiestero()

print("Type", type(obj))
print("Dir ", dir(obj))
```

Podemos usar dir() para encontrar las “capacidades” de nuestra nueva clase.

```
$ python party2.py
Type <class '__main__.Fiestero'>
Dir  ['__class__', ... 'fiesta', 'x']
```

# Intenta dir() con un string

```
>>> x = 'Hello there'
>>> dir(x)
['__add__', '__class__', '__contains__', '__delattr__',
'__doc__', '__eq__', '__ge__', '__getattribute__',
'__getitem__', '__getnewargs__', '__getslice__', '__gt__',
'__hash__', '__init__', '__le__', '__len__', '__lt__',
'__repr__', '__rmod__', '__rmul__', '__setattr__', '__str__',
'capitalize', 'center', 'count', 'decode', 'encode', 'endswith',
'expandtabs', 'find', 'index', 'isalnum', 'isalpha', 'isdigit',
'islower', 'isspace', 'istitle', 'isupper', 'join', 'ljust',
'lower', 'lstrip', 'partition', 'replace', 'rfind', 'rindex',
'rjust', 'rpartition', 'rsplit', 'rstrip', 'split',
'splitlines', 'startswith', 'strip', 'swapcase', 'title',
'translate', 'upper', 'zfill']
```

# Ciclo de vida de los objetos

# Ciclo de vida de los objetos

- Los objetos son **creados, usados y descartados**
- Tenemos bloques especiales de código (métodos) que son llamados
  - Cuando se crea el objeto (**constructor**)
  - Cuando se destruye el objeto (**destructor**)
- Los constructores se usan mucho
- Los destructores rara vez se usan (en Python)

# Constructor

El propósito principal del constructor es **inicializar** algunas variables de instancia (los atributos) para que tengan valores apropiados cuando se cree el objeto

https://es.wikipedia.org/wiki/Constructor_(inform%C3%A1tica)

```
class Fiestero:
    x = 0

    def __init__(self):
     print('Me construyen')

    def fiesta(self) :
      self.x = self.x + 1
      print("Tan lejos",self.x)

    def __del__(self):
     print('Me destruyen', self.x)


obj = Fiestero()
obj.fiesta()
obj.fiesta()

obj=42
print('obj contiene',obj)
```

```
$ python party3.py
Me construyen
Tan lejos 1
Tan lejos 2
Me destruyen 2
obj contiene 42
```

El constructor y el destructor son opcionales. El constructor es usado para inicializar variables. El destructor es rara vez usado

# Muchas instancias

- Podemos crear muchos objetos – la clase es la plantilla para el objeto
- Podemos almacenar cada objeto en su propia variable
- Podemos tener **muchas instancias de la misma clase**
- Cada instancia tiene **su propia copia** de las variables de instancia (atributos)

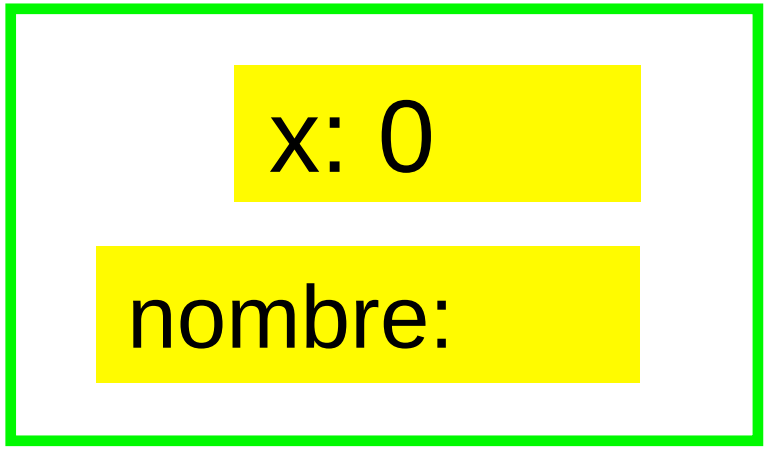
```
class Fiestero:
    x = 0
    nombre = ""

    def __init__(self,z):
        self.nombre=z
        print(self.nombre,"construido")

    def fiesta(self) :
      self.x = self.x + 1
      print(self.nombre,"contador",self.x)

s = Fiestero("Sally")
j = Fiestero("Jim")

s.fiesta()
j.fiesta()
s.fiesta()
```

Los constructores pueden tener **parámetros adicionales**. Esto sirve para inicializar los atributos de cada instancia en particular

party4.py

```
class Fiestero:
    x = 0
    nombre = ""

    def __init__(self,z):
        self.nombre=z
        print(self.nombre,"construido")

    def fiesta(self) :
      self.x = self.x + 1
      print(self.nombre,"contador",self.x)

s = Fiestero("Sally")
j = Fiestero("Jim")

s.fiesta()
j.fiesta()
s.fiesta()
```

```
class Fiestero:
    x = 0
    nombre = ""

    def __init__(self,z):
        self.nombre=z
        print(self.nombre,"construido")

    def fiesta(self) :
      self.x = self.x + 1
      print(self.nombre,"contador",self.x)

s = Fiestero("Sally")
j = Fiestero("Jim")

s.fiesta()
j.fiesta()
s.fiesta()
```



```
class Fiestero:
    x = 0
    nombre = ""

    def __init__(self,z):
        self.nombre=z
        print(self.nombre,"construido")

    def fiesta(self) :
      self.x = self.x + 1
      print(self.nombre,"contador",self.x)

s = Fiestero("Sally")
j = Fiestero("Jim")

s.fiesta()
j.fiesta()
s.fiesta()
```

x: 0

nombre: Sally

Tenemos dos instancias independientes

x: 0

nombre: Jim

```
class Fiestero:
    x = 0
    nombre = ""

    def __init__(self,z):
        self.nombre=z
        print(self.nombre,"construido")

    def fiesta(self) :
      self.x = self.x + 1
      print(self.nombre,"contador",self.x)

s = Fiestero("Sally")
j = Fiestero("Jim")

s.fiesta()
j.fiesta()
s.fiesta()
```

```
Sally construido
Jim construido
Sally contador 1
Jim contador 1
Sally contador 2
```

# Herencia

http://www.ibiblio.org/g2swap/byteofpython/read/inheritance.html

# Herencia

- Cuando hacemos una nueva clase podemos reutilizar una clase existente y **heredar todas las capacidades (y datos) que tenga**, sumando entonces algo más en nuestra nueva clase
- Otra forma de **reutilización**
- Escribir una vez – reutilizar muchas veces
- La nueva **clase (hija)** tiene todas las capacidades de la vieja **clase (padre)** – y algunas más

# Terminología: Herencia

Las ‘**subclases**’ son versiones especializadas de una clase, que hereda atributos y comportamiento de la clase padre, y puede introducir las suyas propias.

https://es.wikipedia.org/wiki/Programaci%C3%B3n_orientada_a_objetos

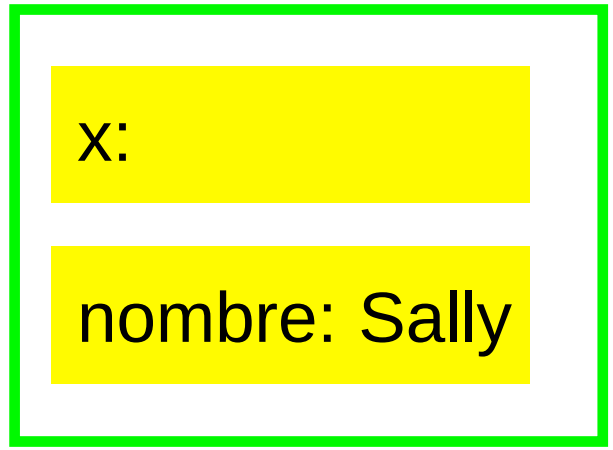
```
class Fiestero:
    x = 0
    nombre = ""

    def __init__(self,z):
        self.nombre=z
        print(self.nombre,"construido")

    def fiesta(self) :
      self.x = self.x + 1
      print(self.nombre,"contador",self.x)

class Jugador(Fiestero):
   puntos = 0
   def jugada(self):
      self.puntos = self.puntos + 1
      self.fiesta()

      print(self.nombre,"puntos",self.puntos)
```

```
s = Fiestero("Sally")
s.fiesta()

j = Jugador("Jim")
j.fiesta()
j.jugada()
```

Jugador es una clase que **"extiende"** a Fiestero. Tiene todas las capacidades de Fiestero y más aún.

```
class Fiestero:
    x = 0
    nombre = ""

    def __init__(self,z):
        self.nombre=z
        print(self.nombre,"construido")

    def fiesta(self) :
      self.x = self.x + 1
      print(self.nombre,"contador",self.x)

class Jugador(Fiestero):
   puntos = 0
   def jugada(self):
      self.puntos = self.puntos + 1
      self.fiesta()

      print(self.nombre,"puntos",self.puntos)
```

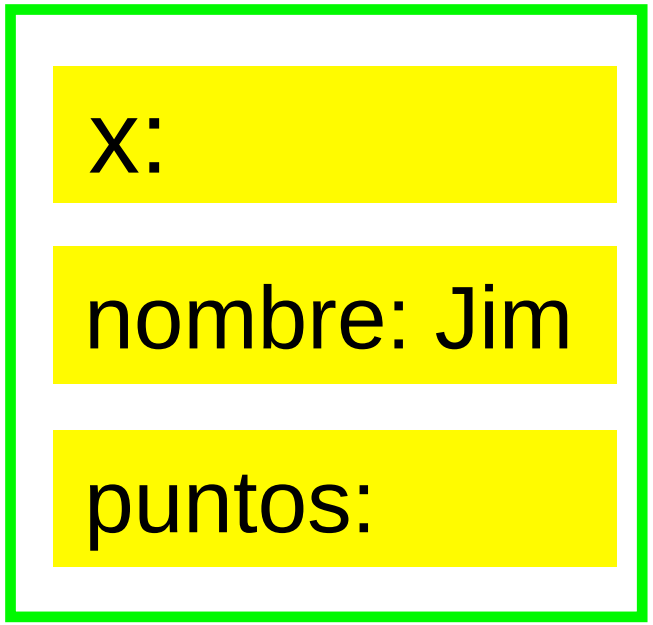
```
s = Fiestero("Sally")
s.fiesta()

j = Jugador("Jim")
j.fiesta()
j.jugada()
```



```
class Fiestero:
    x = 0
    nombre = ""

    def __init__(self,z):
        self.nombre=z
        print(self.nombre,"construido")

    def fiesta(self) :
      self.x = self.x + 1
      print(self.nombre,"contador",self.x)

class Jugador(Fiestero):
   puntos = 0
   def jugada(self):
      self.puntos = self.puntos + 1
      self.fiesta()

      print(self.nombre,"puntos",self.puntos)
```

```
s = Fiestero("Sally")
s.fiesta()

j = Jugador("Jim")
j.fiesta()
j.jugada()
```

j



# Definiciones

- **Clase** – una plantilla
- **Atributo** – Una variable dentro de una clase
- **Método** – Una función dentro de una clase
- **Objeto** – Una instancia particular de una clase
- **Constructor** – Código que se ejecuta cuando un objeto es creado
- **Herencia** – La habilidad para extender una clase y crear una nueva clase

# Resumen

- La **Programación Orientada a Objetos** es una forma muy estructurada de **reutilizar** código
- Podemos agrupar datos y funcionalidad juntos para crear muchas instancias independientes de una clase

# Acknowledgements / Contributions




Initial Development: Charles Severance, University of Michigan School of Information

… Insert new Contributors and Translators here
Spanish Version: Daniel Garrido (dgm@uma.es)

Continue…