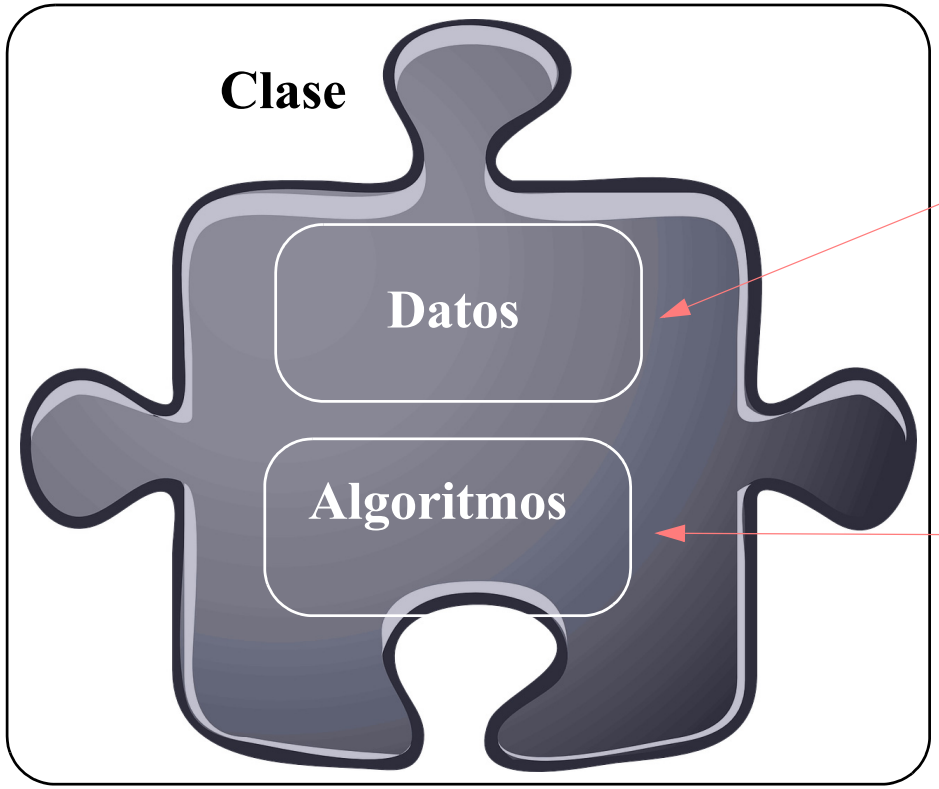
Parte I: Programación en un lenguaje orientado a objetos

1. **Introducción a los lenguajes de programación**
2. **Datos y expresiones**
3. ***Clases***
   - Concepto de clase y objeto. Definición de clases. Creación y uso de objetos. Atributos y métodos de instancia y de clase. Espacios de nombres. Módulos y paquetes. El ciclo de vida del software.
4. **Estructuras algorítmicas**
5. **Estructuras de Datos**
6. **Tratamiento de errores**
7. **Entrada/salida**
8. **Herencia y polimorfismo**

3.1. Concepto de clase y objeto

Los programas orientados a objetos se construyen mediante ***clases***

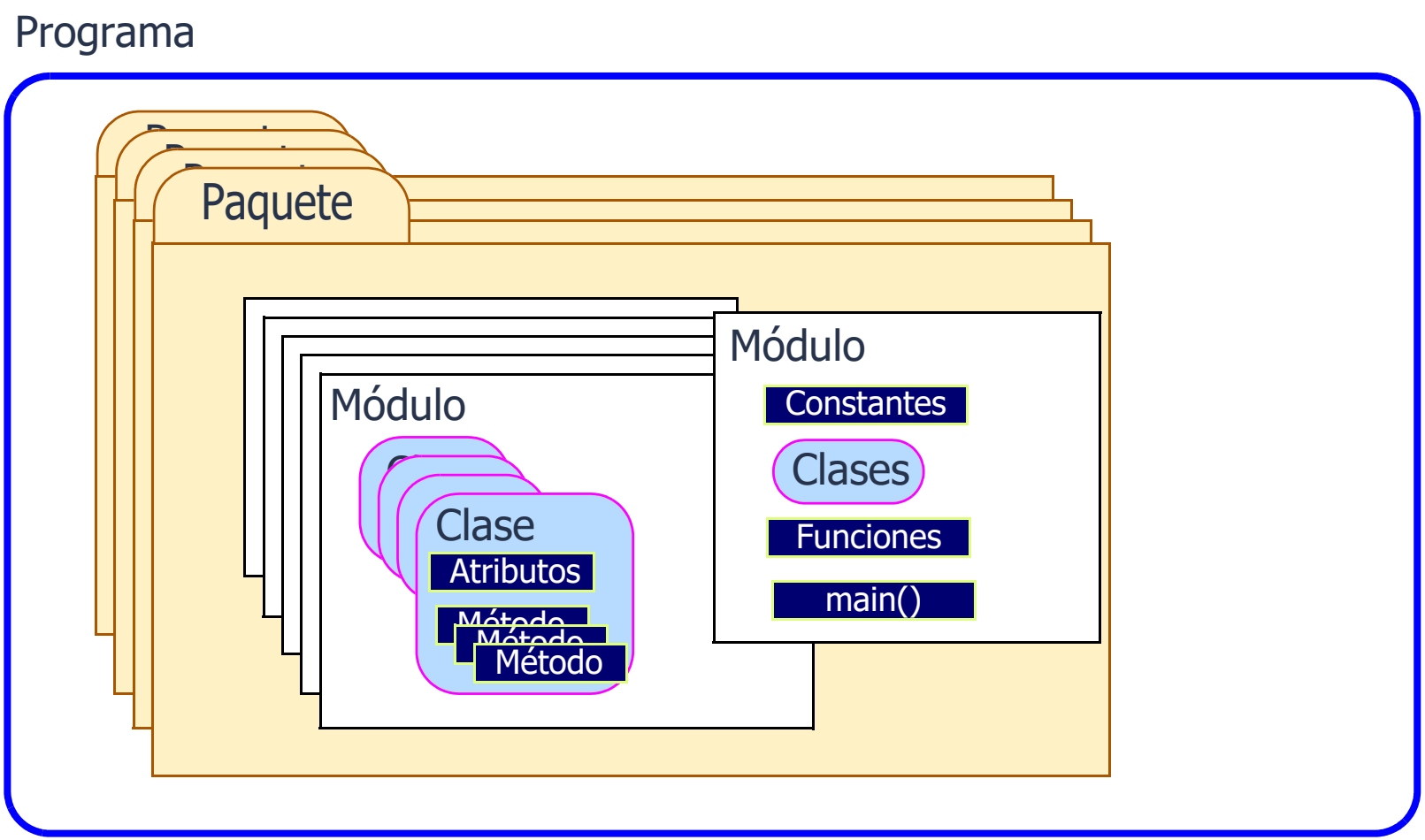
Una ***clase*** representa un elemento de programa que contiene datos y algoritmos



atributos
(variables de la clase)
tienen vida larga

métodos
u operaciones
(funciones de la clase)
sus datos tienen
vida corta

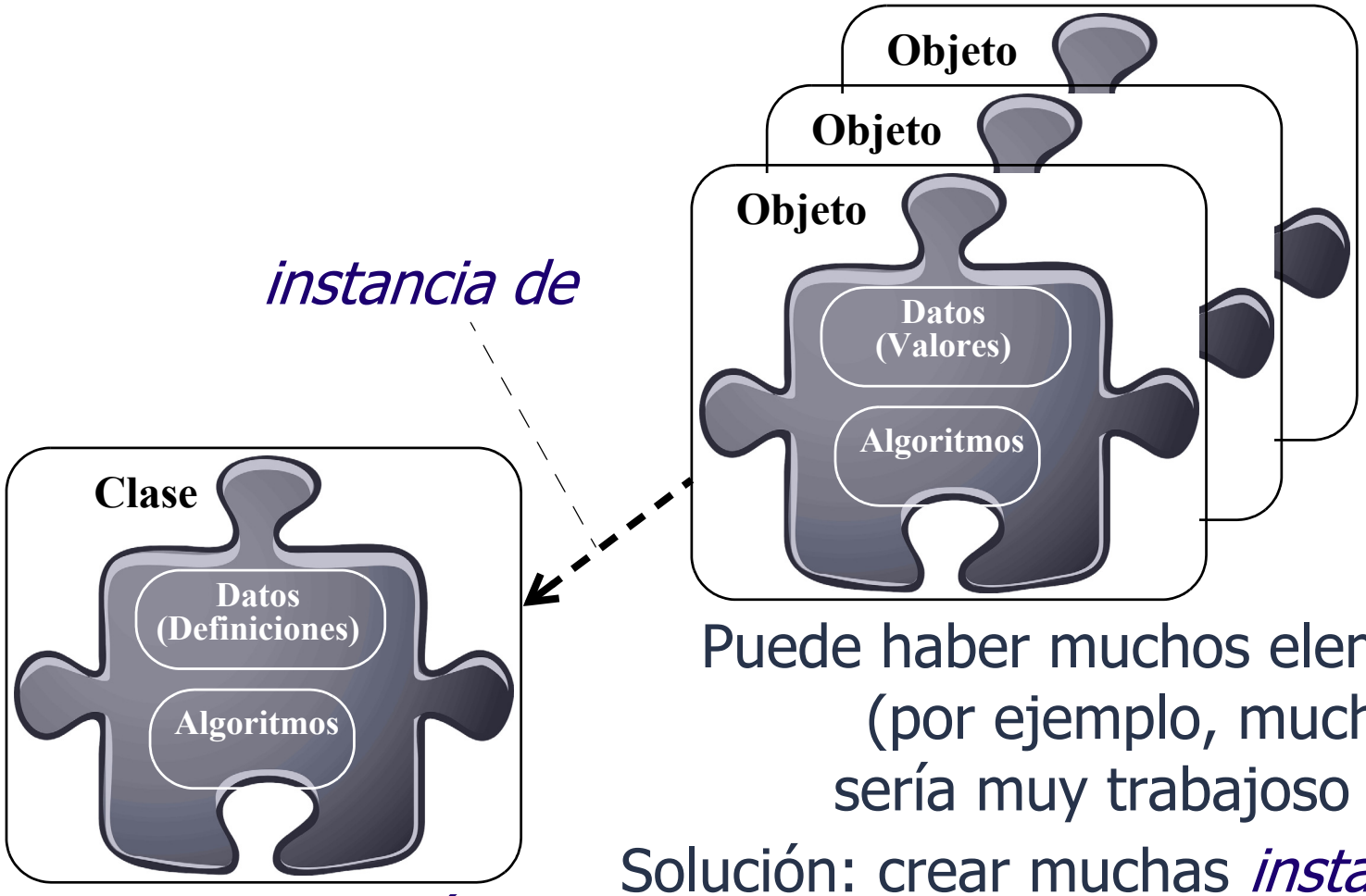
Jerarquía de elementos de programa en Python



Un principio básico

¡No te repitas!

Objetos:



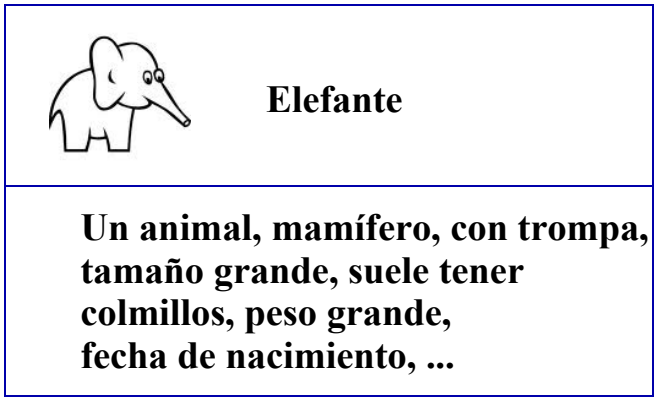
Es una *definición*

Puede haber muchos elementos parecidos (por ejemplo, muchas personas, ...) sería muy trabajoso repetir las clases

Solución: crear muchas *instancias* de la clase *!objetos!*

Diferencia entre clase y objeto

En la clase se *definen* las características comunes de un conjunto de objetos

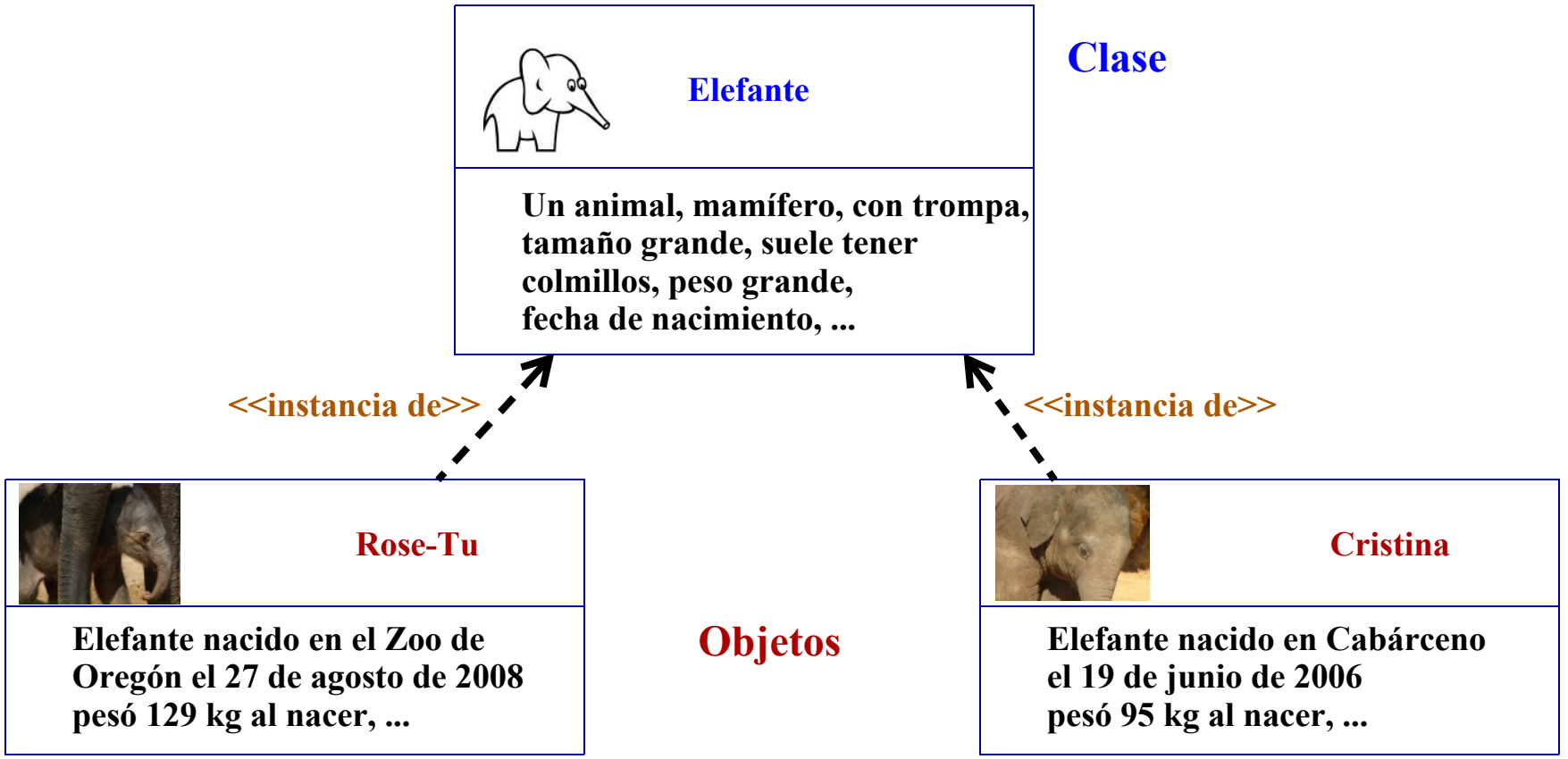


Clase

Define las características de todos los elefantes

Diferencia entre clase y objeto

Las instancias de una clase son *objetos* concretos que tienen las características de la clase y valores concretos de sus datos



Imágenes de Creative Commons por Julielangford y Superjew

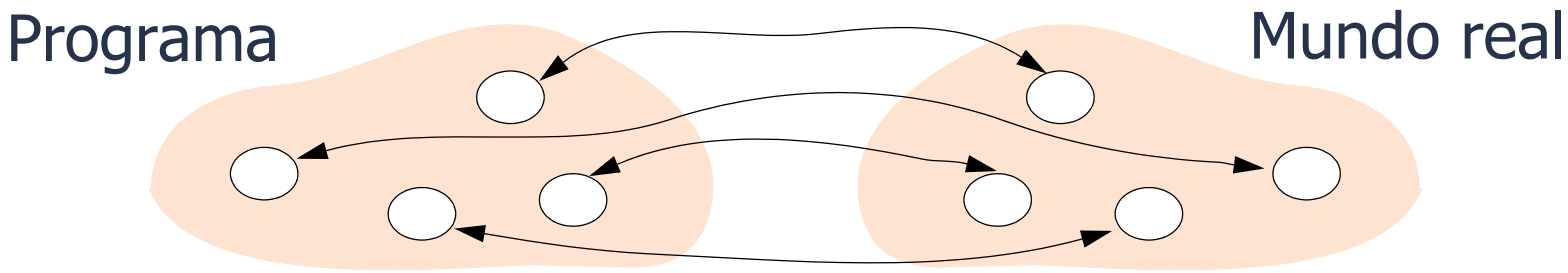
Concepto de clase y objeto

Un ***objeto*** es un elemento de programa que se caracteriza por:

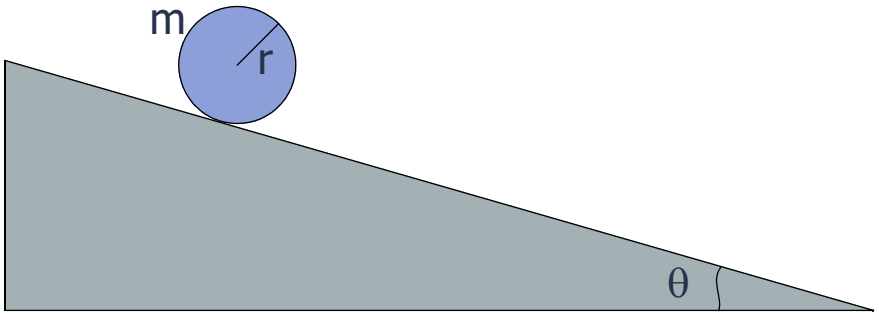
- ***atributos***: son datos contenidos en el objeto y determinan su estado
- ***operaciones*** o ***métodos***: son acciones con las que podemos solicitar información del objeto, o modificarla
  - Son secuencias de instrucciones que operan con los atributos
  - y pueden invocar operaciones de otros objetos

Ambos, atributos y métodos, se *definen* en la clase

Se intenta siempre corresponder los objetos de un programa con objetos del problema que éste resuelve



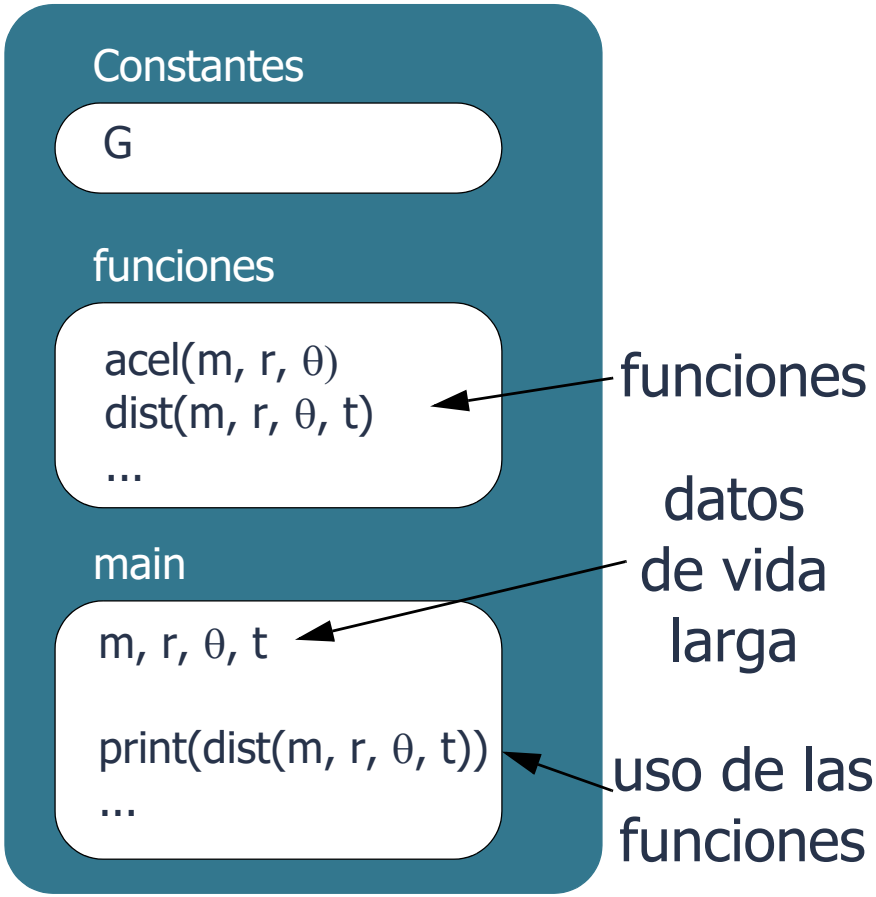
Ejemplo, esfera que rueda en un plano inclinado



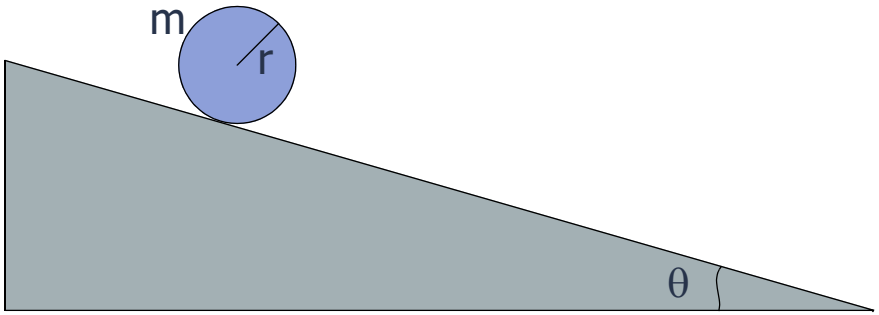
Problemas:

- Las funciones y los datos están separados
- Si hubiese cientos o miles de módulos, no sería práctico tener todos los datos en el main
- Las funciones necesitan muchos parámetros

Diseño anterior (no OO)



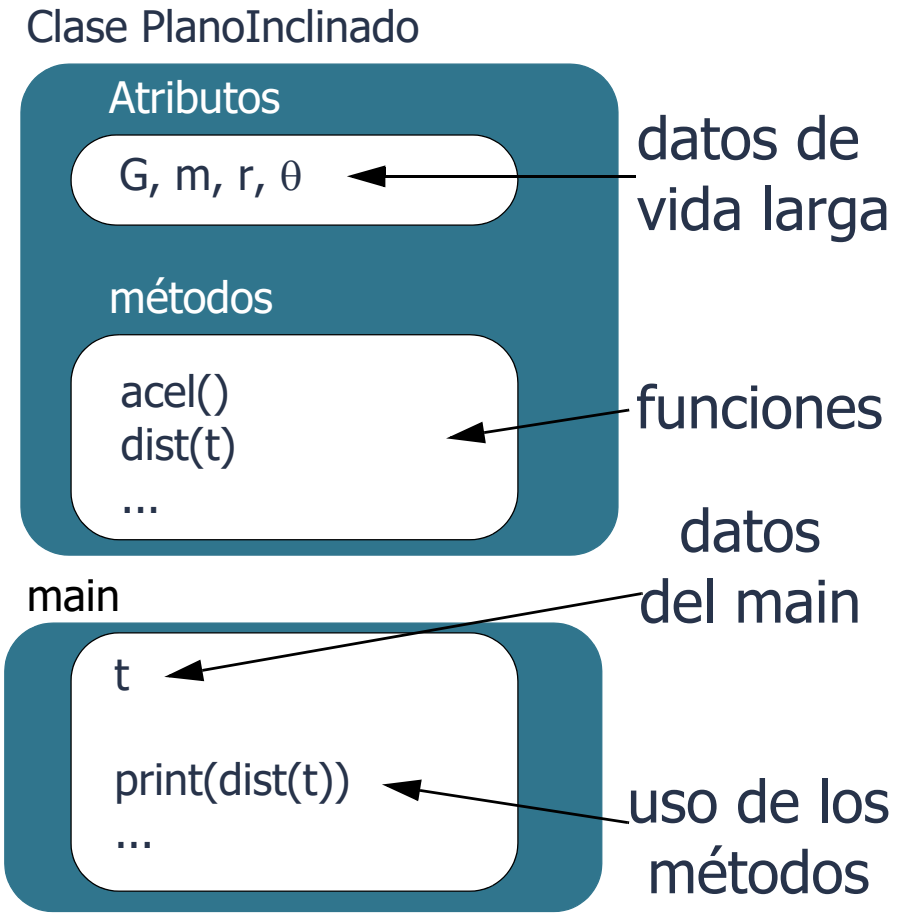
Ejemplo: Diseño orientado a objetos



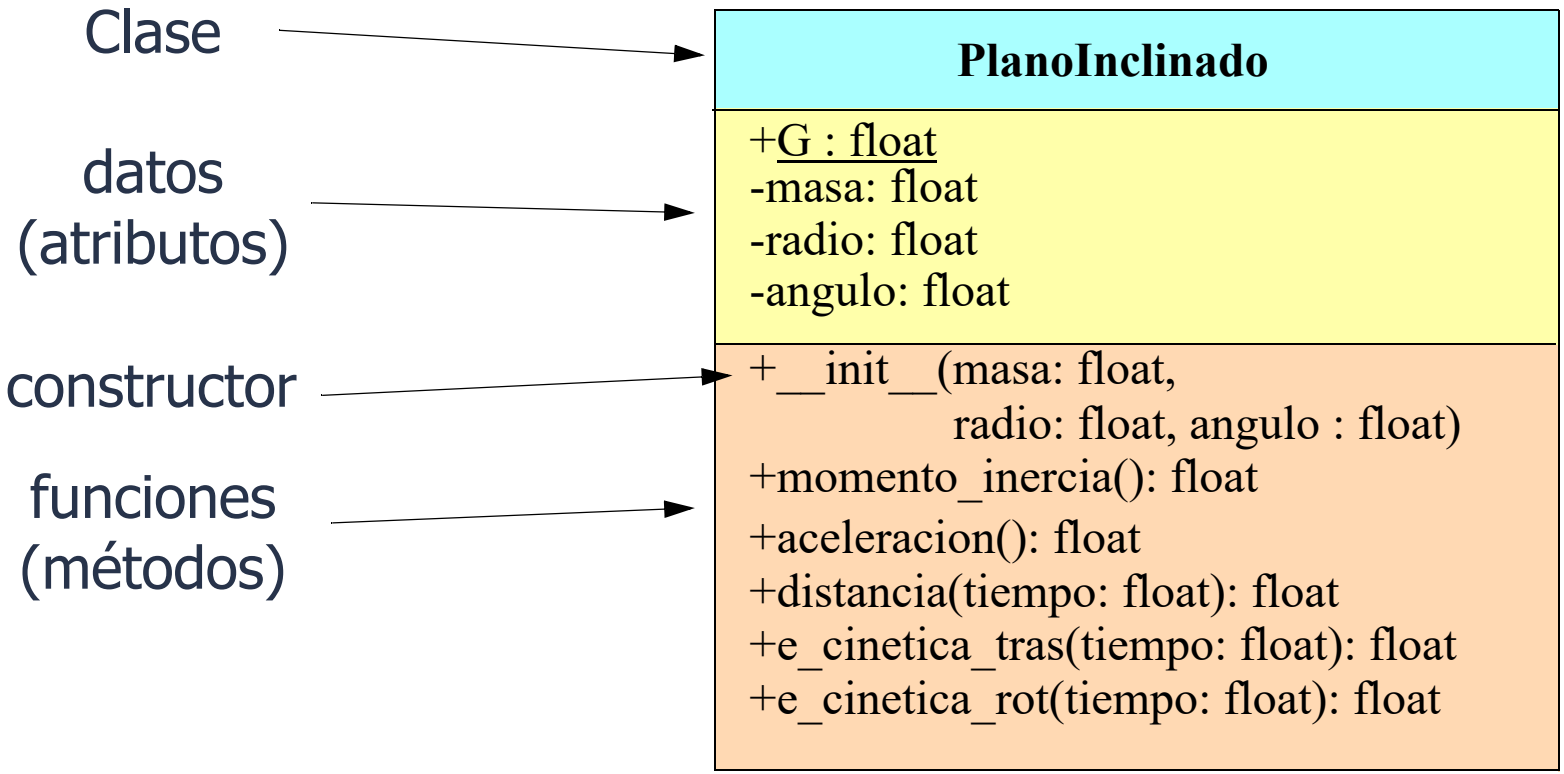
Ventajas:

- Las funciones y sus datos están juntos
- El main solo tiene los datos suyos
- Las funciones tienen pocos parámetros

Diseño OO



Ejemplo: Diagrama de clase



- elemento *privado* (recomendado para atributos)
+ elemento *público* (para métodos que se usen desde fuera de la clase)

subrayado: indica un elemento estático (o de clase)
normal: indica un elemento de instancia (de objeto)

Diagrama de clases

El diagrama de clases es una representación estándar que define:

- ***nombre*** de la clase
- ***atributos***: nombres y tipos
- ***métodos***: son funciones con su nombre, parámetros (con nombre y tipo de cada uno), y valor retornado (tipo)
  - un método puede ser un *constructor*, que se usa para crear e inicializar el objeto
  - en Python se llama `__init__`

Normas de estilo para los nombres:

- *clases*: comienzan por mayúscula, usan *CamelCase*
- *atributos* y *métodos*: comienzan por minúscula, usan *snake_case*

3.2. Definición de clases en Python

```
class NombreClase:
    """
    Comentario de documentación
    """

    constructor(self, otros parámetros)
    # crea y da valor a los atributos

    métodos(self, otros parámetros)
```

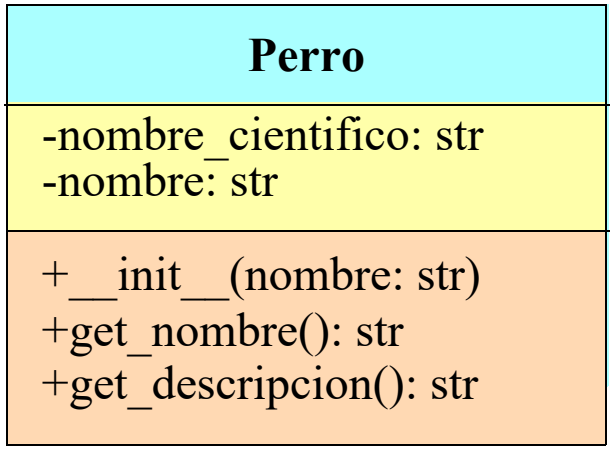
El constructor y los métodos reciben como primer parámetro `self`

- es una referencia al objeto
- da acceso a sus datos y métodos
- no se pone en el diagrama, pues en otros lenguajes no es preciso

Ejemplo de clase: Perro

Por brevedad, omitimos los comentarios de documentación

```
class Perro:

    def __init__(self, nombre: str):
        self.__nombre: str = nombre
        self.__nombre_cientifico: str = \
            "canis lupus familiaris"

    def get_nombre(self) -> str:
        return self.__nombre

    def get_descripcion(self) -> str:
        return "Perro ("+self.__nombre_cientifico + \
            ") de nombre "+self.__nombre
```



- En Python los métodos o atributos privados llevan el prefijo _ o __
- Observar el uso de un atributo desde dentro de la clase: `self.nombre_atributo`

Ejemplo: plano inclinado

```
# -*- coding: utf-8 -*-
"""
Simula el movimiento de una esfera en un plano inclinado

Se entiende que la esfera rueda sin deslizarse
Utilizamos unidades del sistema internacional (kg, m, s, J)
Para el ángulo utilizamos grados
Ecuaciones del movimiento: http://www.sc.ehu.es/sbweb/fisica/

@author: Michael
"""

import math
from typing import Final

# Constantes: (se podrían definir como atributos de clase)
# G : gravedad en m*s**2

G: Final[float] = 9.8
```

Ejemplo (cont.)

```
class PlanoInclinado:
    """
    Sistema Esfera - Plano Inclinado
    Contiene los datos de un sistema formado por
    una esfera y un plano inclinado
    y métodos para hacer cálculos de trayectorias
    y energías

    Attributes:
        __masa:   masa de la esfera, en kg
        __radio:  radio de la esfera, en m
        __angulo: ángulo del plano inclinado, en grados
    """ __
```

Ejemplo (cont.)

```
    def __init__(self, masa: float, radio: float,
                 angulo: float):
        """
        Constructor del sistema esfera - plano inclinado

        Args:
            masa:   masa de la esfera, en kg
            radio:  radio de la esfera, en m
            angulo: ángulo del plano inclinado, en grados

        """

        self.__masa: float = masa
        self.__radio: float = radio
        self.__angulo: float = angulo
```

Ejemplo (cont.)

```
    def momento_inercia(self) -> float:
        """
        Calcula el momento de inercia de la esfera

        Returns:
            el momento de inercia de la esfera, en kg*m**2
        """

        return 2.0*self.__masa*self.__radio**2/5.0


    def aceleracion(self) -> float:
        """
        Calcula la aceleración lineal del objeto
        Returns:
            la aceleración de la esfera en m/s
        """

        return (G*math.sin(math.radians(self.__angulo)) /
                (1+self.momento_inercia() /
                 (self.__masa*self.__radio**2)))
```

Ejemplo (cont.)

```
    def distancia(self, tiempo: float) -> float:
        """
        Calcula la distancia recorrida por el
        objeto en el tiempo indicado

        Args:
            tiempo: tiempo transcurrido, en s
        Returns:
            la distancia recorrida por la esfera
        """

        return self.aceleracion()*tiempo**2/2.0
```

Observar el uso de un método interno a la clase:

```
self.aceleracion()
```

Ejemplo (cont.)

```
    def e_cinetica_tras(self, tiempo: float) -> float:
        """
        Calcula la energía cinética de traslación,
        del objeto transcurrido el tiempo indicado (Jul)

        Args:
            tiempo: tiempo transcurrido, en s
        Returns:
            la energía cinética de traslación de la esfera en J
        """

        vel: float = self.aceleracion()*tiempo
        return self.__masa*vel**2/2.0
```

Ejemplo (cont.)

```
    def e_cinetica_rot(self, tiempo: float) -> float:
        """
        Calcula la energía cinética de rotación,
        del objeto transcurrido el tiempo indicado (Jul)

        Args:
            tiempo: tiempo transcurrido, en s
        Returns:
            la energía cinética de rotación de la esfera en J
        """

        vel_angular: float = \
            self.aceleracion()*tiempo/self.__radio
        return self.momento_inercia()*vel_angular**2/2.0
```

3.3. Creación y uso de objetos

Crear un objeto

```
nombre_objeto = Clase(parámetros)
```

Ejemplos

```
mi_perro = Perro("Brena")
perro_de_maria = Perro(nombre="Pombo")
plano_1 = PlanoInclinado(1.5, 0.2, 30)
plano_2 = PlanoInclinado(masa=1.1, radio=0.15,
                         angulo=35)
```

¿Qué parámetros se usan?: los del constructor

- como en toda función, podemos usar los nombres de los parámetros

¿Dónde está el parámetro `self`?: está implícito en la instrucción

Usar un objeto

Usar un atributo público:

```
objeto.atributo
```

Invocar un método público:

```
objeto.método(parámetros)
```

Ejemplos:

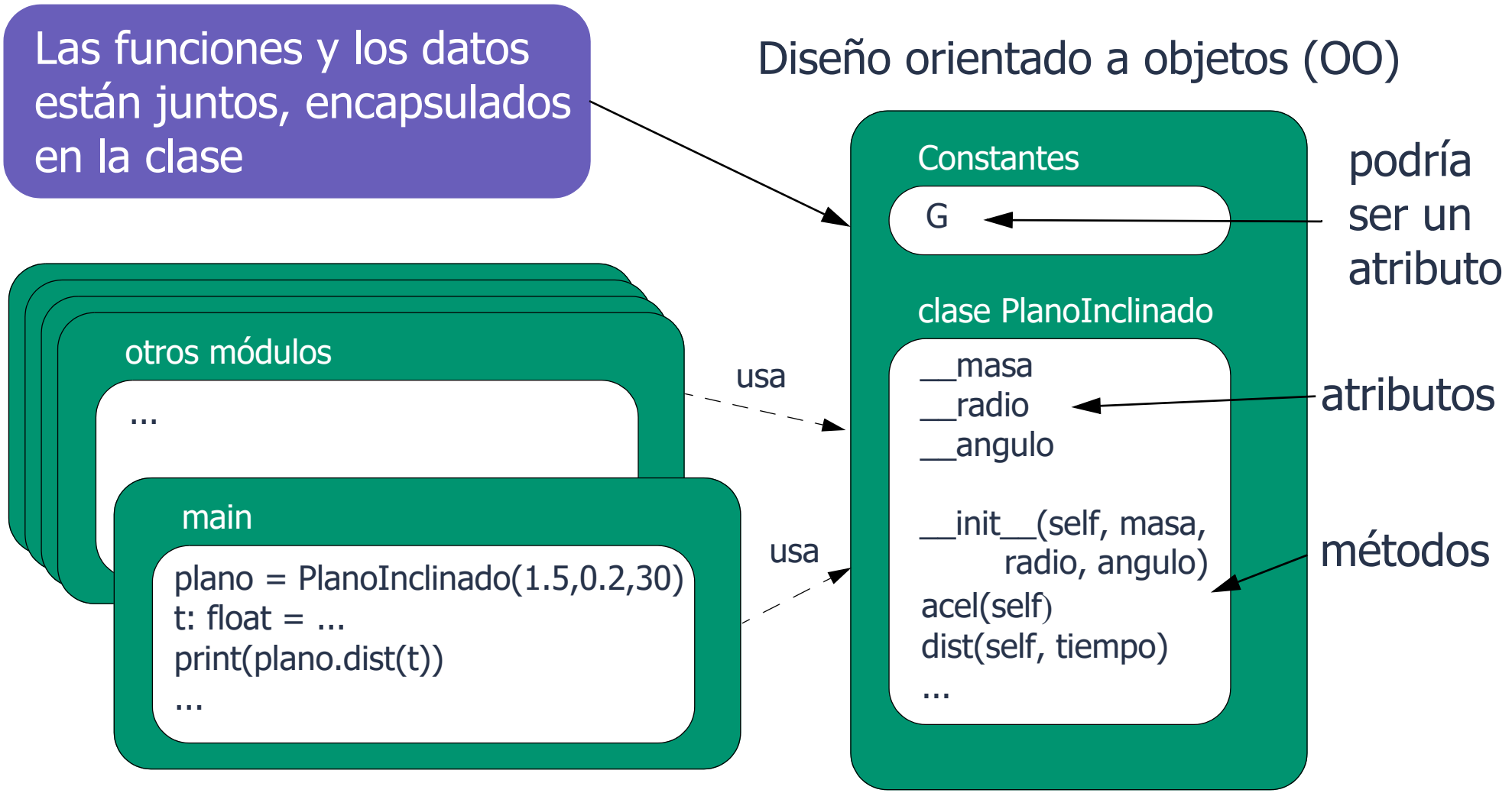
```
nombre: str = perro_de_maria.get_nombre()
dist: float = plano_1.distancia(10.0)
```

¿Dónde está el parámetro `self`?: está implícito en el objeto

Ejemplo: plano inclinado

```
def main():
    """
    Programa principal que muestra para una esfera en
    un plano inclinado la distancia y energías a los
    cuatro segundos
    """

    # Crear el sistema plano-esfera
    # Ángulo 30 grados, esfera de 1.5 kg y r=0.2 m
    plano = PlanoInclinado(masa=1.5, radio=0.2, angulo=30)

    # Mostrar resultados
    print("Dist. a los 4 seg: " +
          f"{plano.distancia(4.0)} m")
    print("E. C. tras. a 4 seg: " +
          f"{plano.e_cinetica_tras(4.0):.3f} J")
    print("E. C. rot a 4 seg: " +
          f"{plano.e_cinetica_rot(4.0):.3f} J")
```

Ejemplo: resumen del diseño



3.4. Atributos y métodos de instancia y de clase

Los atributos definidos con `self` son atributos de instancia

- pertenecen a los objetos de la clase
- cada objeto tiene los suyos

Los atributos definidos directamente en la clase son atributos de clase

- también llamados estáticos
- pertenecen a la clase
- todos los objetos de la clase los comparten

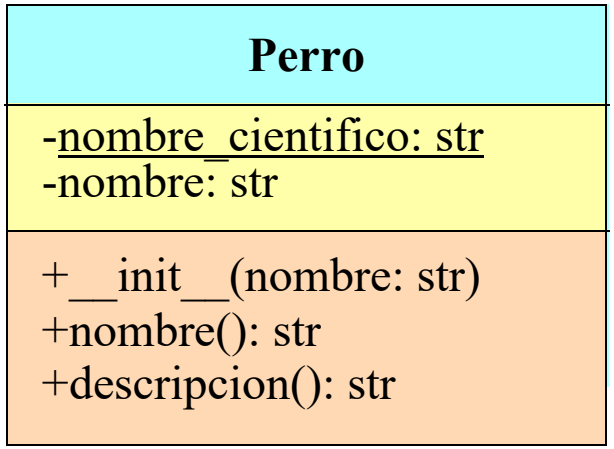
Uso

- desde dentro de la clase: `self.atributo`
- desde fuera de la clase: `objeto.atributo` o `clase.atributo`

Ejemplo: Clase Perro

Vamos a modificar la clase `Perro` para que el nombre científico sea un atributo de clase

- todos los perros lo comparten
  - así ahorramos memoria y posibles confusiones
- en cambio, cada perro tiene su propio nombre, por lo que el atributo `nombre` es de instancia



En el diagrama indicamos este hecho poniendo el atributo `nombre_cientifico` con subrayado

Ejemplo (cont.)

Por brevedad, omitimos los comentarios de documentación

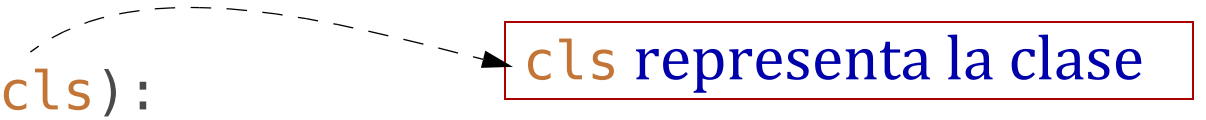
```
class Perro:

    __nombre_cientifico: str = "canis lupus familiaris"

    def __init__(self, nombre: str):
        self.__nombre: str = nombre

    def nombre(self) -> str:
        return self.__nombre

    def descripcion(self) ->) -> str:
        return "Perro ("+self.__nombre_cientifico + \
            ") de nombre "+self.__nombre
```

antes se definía en `__init__`

Se podría usar el nombre de la clase, pero es mejor usar `self`

Métodos estáticos y de clase

```
class MiClase:

    __c: str = "'soy estático'"

    def __init__(self, nombre):
        self.__nombre: str = nombre # atributo de instancia

    def metodo_de_instancia(self):
        return 'En método de inst. de nombre ' + \
            self.__nombre+'. Accede a '+self.__c

    @classmethod
    def metodo_de_clase(cls):
        return "En método de clase. Hay acceso a " + \
            cls.__c

    @staticmethod
    def metodo_estatico():
        return "En método estático. " + \
            "No se debe acceder a ningún atributo"
```



Métodos estáticos y de clase (cont.)

Este fragmento de código:

```
obj = MiClase("Pepe")
print(obj.metodo_de_instancia())
print(obj.metodo_de_clase())
print(obj.metodo_estatico())
```

Produce el siguiente resultado:

```
En método de inst. de nombre Pepe. Accede a 'soy estático'
En método de clase. Hay acceso a 'soy estático'
En método estático. No se debe acceder a ningún atributo
```

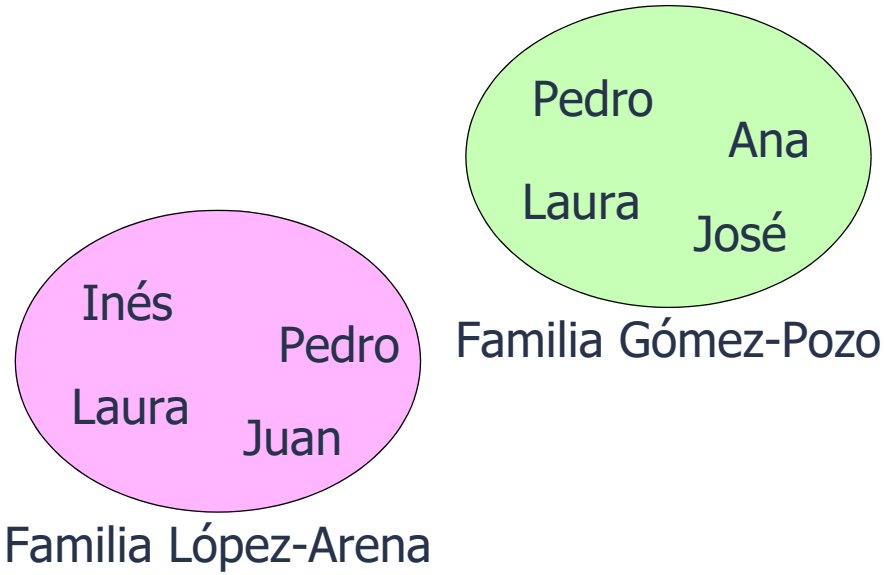
3.5. Espacios de nombres

Las siguientes entidades Python definen espacios de nombres

- el entorno Python
- módulos
- funciones
- clases

Estos espacios de nombres permiten tener nombres sin ambigüedades, aunque sean iguales

- Como en las familias de la figura



Ámbitos de los espacios de nombres

Como puede haber relaciones de inclusión entre módulos, clases y funciones,

- una entidad puede usar su espacio de nombres y los de sus contenedores
  - `f1`, `f2` y `f3` el del módulo (y el de Python)
  - `f2` también el de `f1`
- en general, si una entidad crea un nombre igual a uno externo, el nombre nuevo enmascara el externo
  - en el ejemplo hay dos variables `b`, una en `f1` y otra en `f2`
  - salida:

```
a=a
b=x
b=b
a=a
d=d
```

Módulo fichero.py

```
a = "a"

def f1():
    b="b"
    def f2():
        c="c"
        b="x"
        print("a="+a)
        print("b="+b)
    f2()
    print("b="+b)

def f3():
    d="d"
    print("a="+a)
    print("d="+d)
f1()
f3()
```

Uso de nombres de otros módulos

Para usar nombres definidos en otros módulos usamos `import`, de dos maneras:

`import módulo`

- importa el nombre del módulo, pero no los nombres contenidos en él
- para usarlos hay que escribir `módulo.elemento`

```
import math      # importamos el módulo math
x=math.sqrt(2)   # uso
```

`from modulo import elemento`

- importa el nombre del elemento, que podremos usar directamente

```
from math import sqrt  # importamos la función sqrt
x=sqrt(2)              # uso
```

Es posible importar *todos* los nombres de un módulo con `*`

`from modulo import *`

- esto está totalmente desaconsejado, ya que dificulta saber qué nombres hay definidos e introduce confusión

Renombrado

Al importar es posible renombrar los nombres que importamos

```
import matplotlib.pyplot as plt  # nombre más corto

from math import sqrt as raiz_cuadrada  # traducido

y = raiz_cuadrada(2)
plt.show()
```

3.6. Módulos y paquetes

Es posible organizar módulos relacionados entre sí incluyéndolos en *paquetes*

Facilita la organización del código

El uso es con la notación "."

```
paquete.módulo
```

Un paquete puede incluir sub-paquetes, al nivel deseado

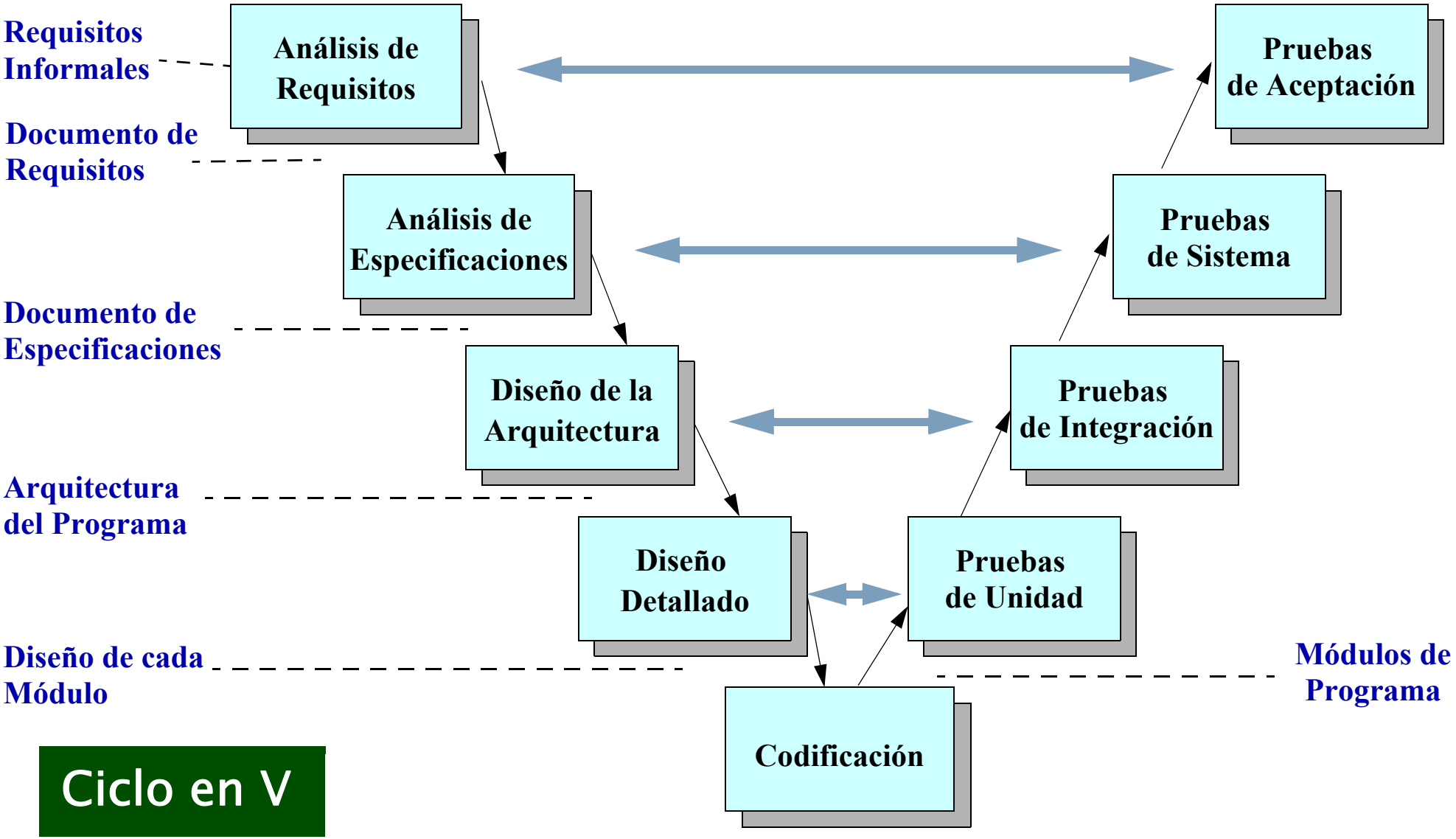
Creación de paquetes

Un paquete es una carpeta conteniendo módulos Python y además, un módulo especial llamado `__init__.py`

- este módulo puede contener instrucciones de inicialización del paquete
- o puede estar vacío si no es necesaria esta inicialización

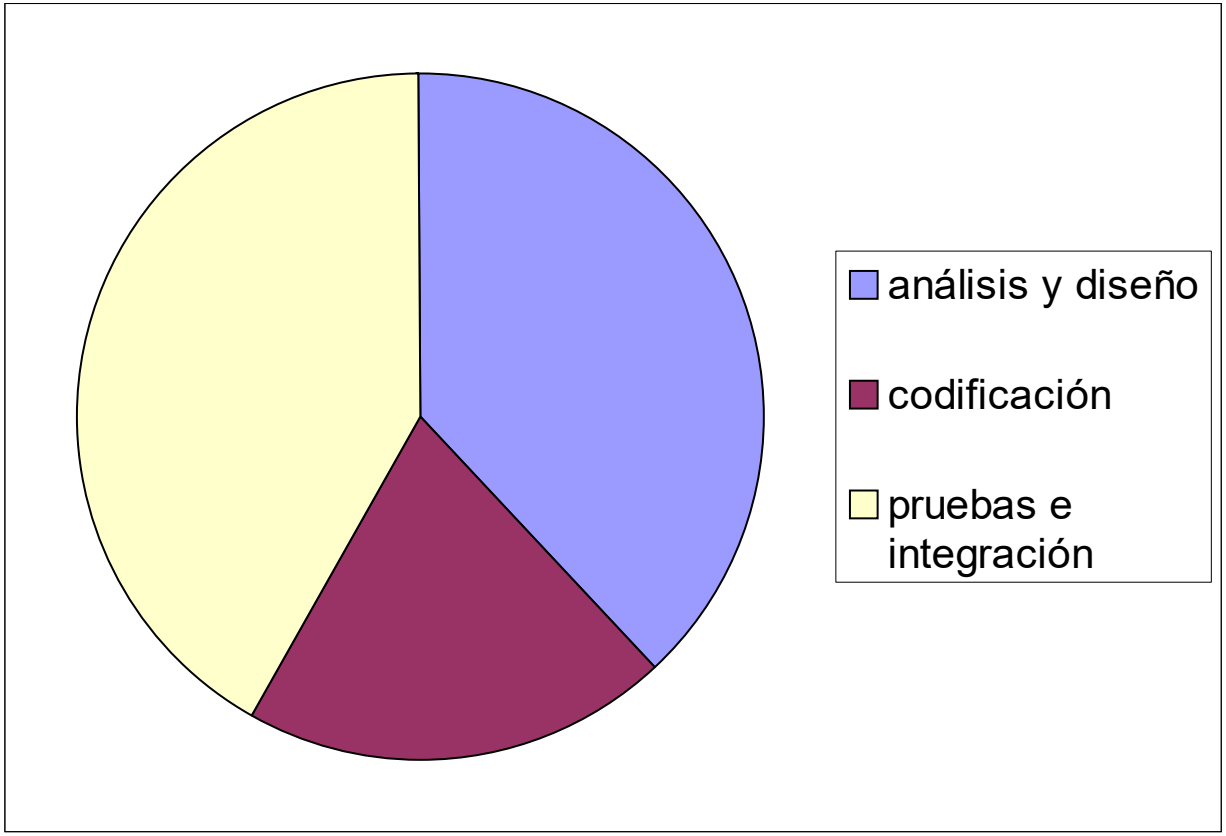
Referencia:
https://docs.python.org/3/tutorial/modules.html#packages

3.7. El ciclo de vida del software



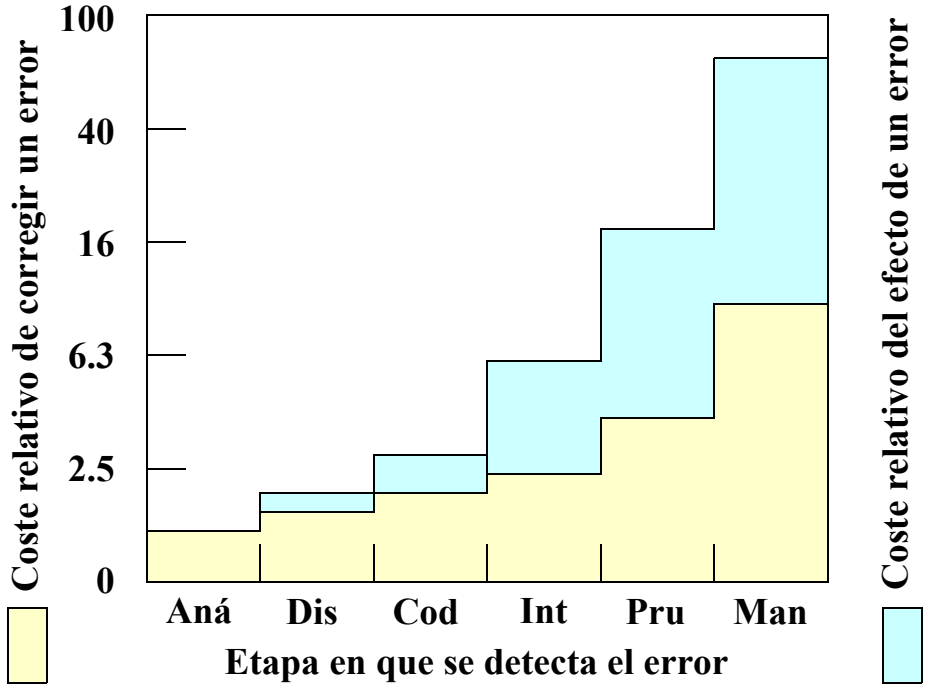
Ciclo en V

Distribución del Esfuerzo de desarrollo



Coste de los sistemas informáticos

Los errores software tienen un alto coste: efecto y corrección



Coste de los sistemas informáticos

Relación entre el coste de HW y SW