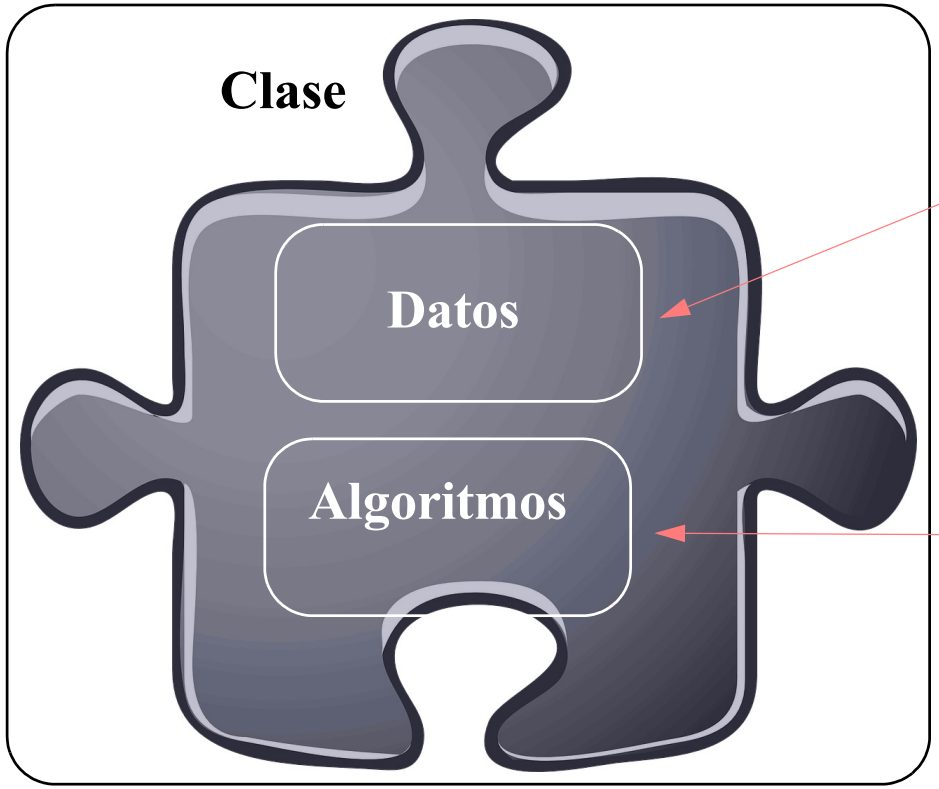
Parte I: Programación en un lenguaje orientado a objetos

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

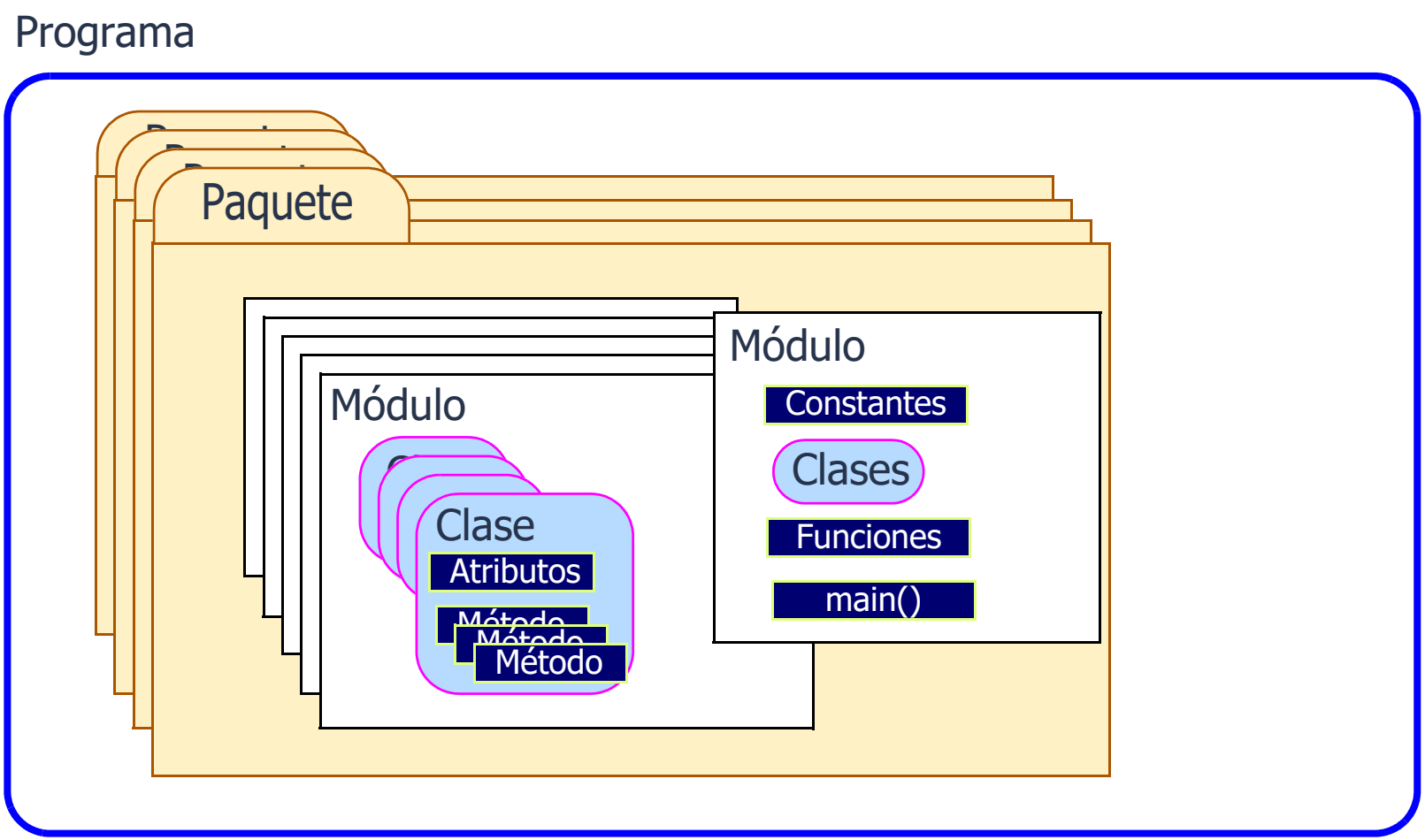
métodos
u operaciones
(funciones de la clase)
sus datos tienen
vida corta

Notas:

Una clase es un elemento de programa que permite encapsular juntos:

- Un conjunto de *funciones*, que contienen algoritmos.
  - Las funciones de una clase se llaman *métodos*
- Un conjunto de *datos de vida larga*, usados por esas funciones.
  - Los datos de una clase se llaman *atributos*

Jerarquía de elementos de programa en Python



Notas:

Los programas en Python se organizan en los siguientes elementos jerárquicos:

- *Paquete*: en esencia es una carpeta que contiene módulos relacionados entre sí, cada uno en un fichero diferente
- *Módulo*: es un *script* contenido en un fichero, y que puede contener:
  - *datos*: por seguridad normalmente restringimos los datos de un módulo solo a constantes
  - *funciones*: una secuencia de instrucciones con argumentos y variables locales, que son datos de vida corta
  - *clases*: contienen datos de vida larga, llamados atributos y métodos, que son funciones que se aplican a los objetos de la clase. A continuación comentamos el concepto de objeto

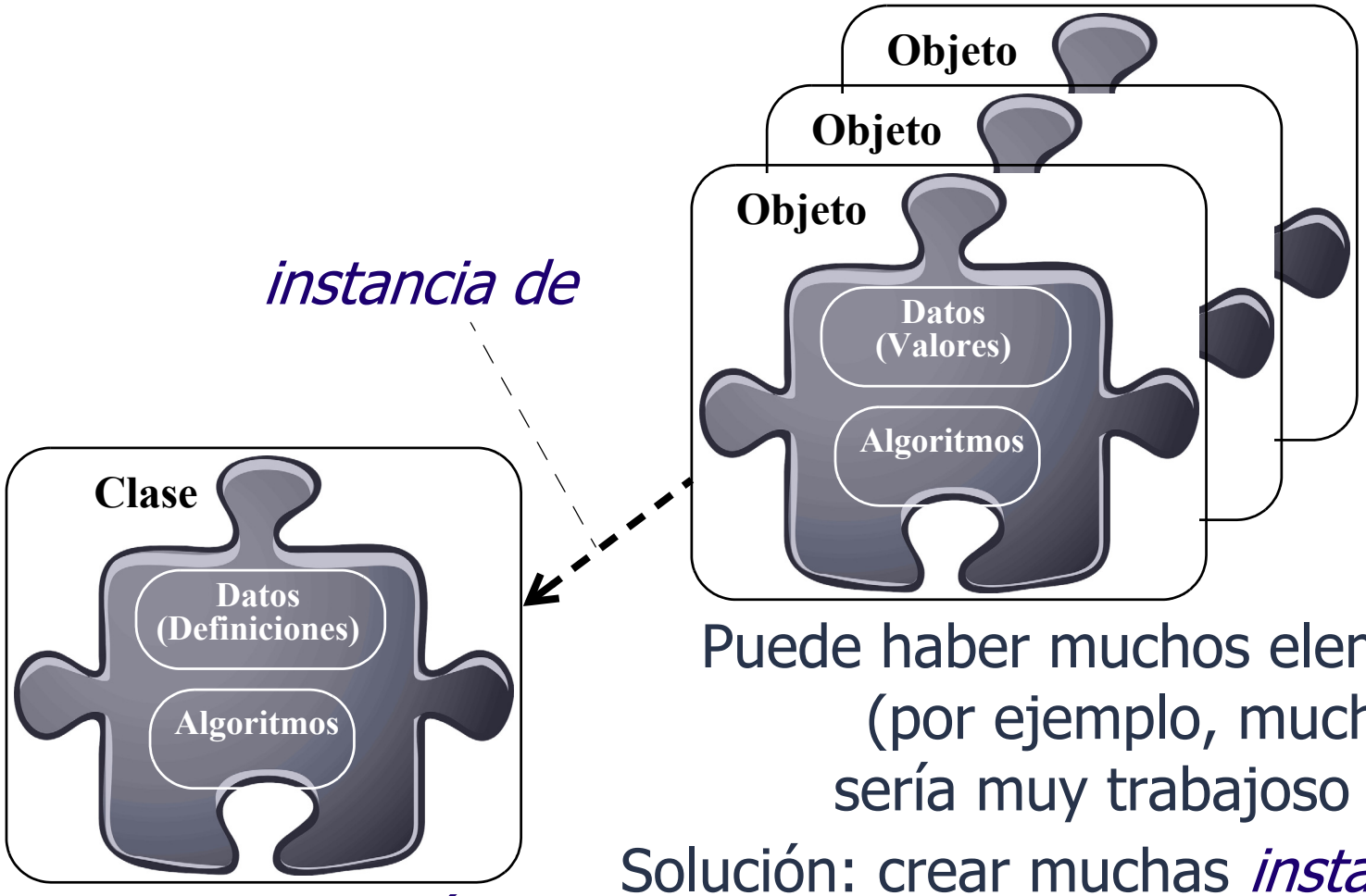
Una de las funciones se llama `main()`, y es por donde empezamos a ejecutar

Como vemos, las clases nos proporcionan la posibilidad de tener *datos de vida larga*.

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!*

Notas:

Es muy habitual tener que guardar en un programa elementos similares.

- Suponer, por ejemplo, una clase para guardar los datos de una *persona*, y algoritmos para trabajar con ellos.
- Para crear muchas personas, podríamos repetir la clase muchas veces, pero esto es ineficiente y va en contra de un principio de la programación:

> ¡No te repitas!

- Nunca debemos repetir instrucciones, para que no surjan *inconsistencias* si se hacen cambios.

La solución está en crear la clase una sola vez, y luego crear *muchas instancias* de la clase, llamadas *objetos*.

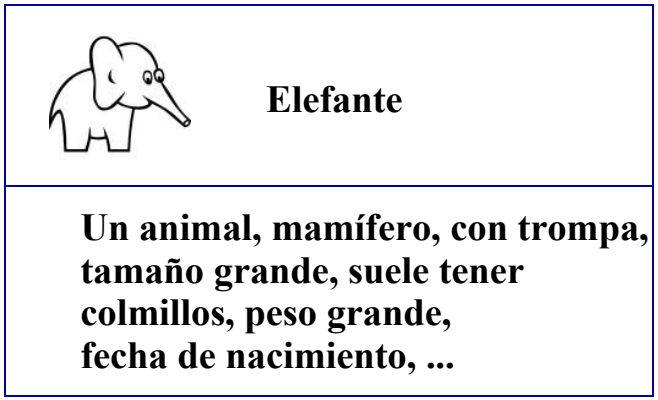
- A partir de la clase `Persona`, crearemos *muchos objetos* para representar personas concretas.
- Nunca repetiremos código.

En este contexto, la clase solo se usa como *definición* para luego crear objetos.

- Una clase sola normalmente no sirve para nada si no se crean objetos de ella.

Diferencia entre clase y objeto

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

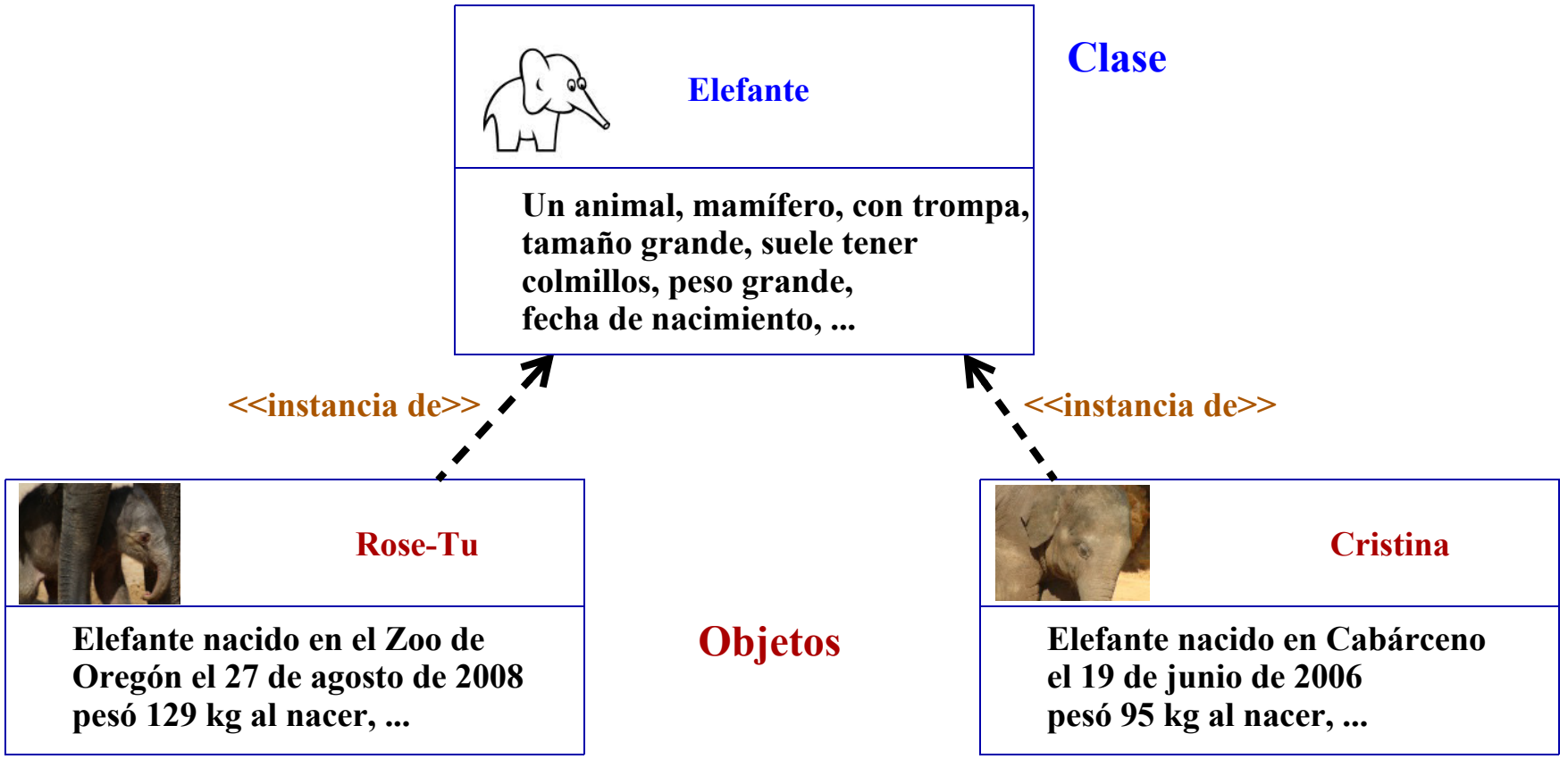


Clase

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

Notas:

En este ejemplo del mundo real mostramos las diferencias entre los conceptos de *clase y objeto*.

- No es un ejemplo del mundo informático, pero sirve para ilustrar los conceptos.

La clase `Elefante` describe las características que son comunes a todos los elefantes.

- Por ejemplo, todos tienen nombre, fecha de nacimiento, peso al nacer, ...

Los datos concretos de cada *objeto* elefante son los definidos en la clase, pero con *valores concretos*.

- Así, el primer elefante tiene como nombre "Rose-Tu", como fecha de nacimiento el 27 de agosto de 2008, etc.
- El segundo elefante tiene como nombre "Cristina", como fecha de nacimiento el 19 de junio de 2006, etc.

Decimos que los objetos como Rose-Tu o Cristina son *instancias de la clase* `Elefante`.

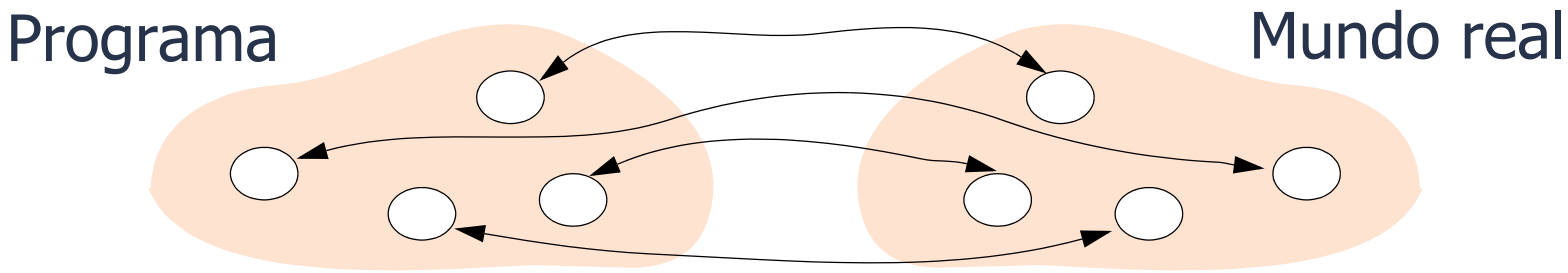
Concepto de clase y objeto

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



Notas:

Las funciones de una clase se denominan *métodos*.

- Cada una de ellas contiene una secuencia de instrucciones que forman un *algoritmo*.

Los datos de una clase se llaman *atributos*.

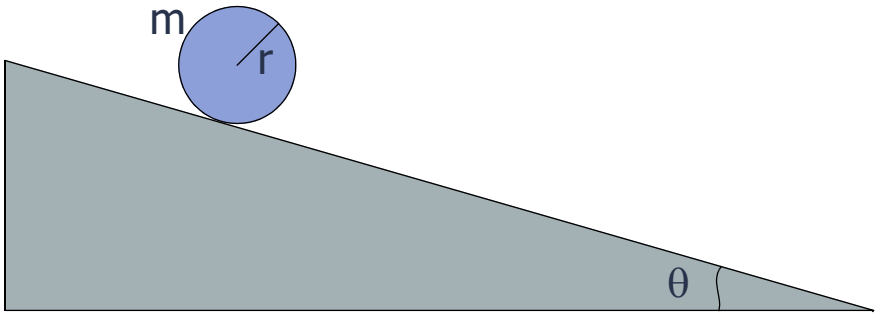
Tanto los atributos como los métodos se definen en la clase.

Luego, los objetos de la clase dan valores concretos a los atributos, y usamos sus métodos sobre ellos.

La técnica de programación orientada a objetos (OOP) divide el programa en clases, intentando que cada una de ellas se corresponda con un objeto del mundo real.

- De esta forma, es más fácil establecer los paralelismos entre la realidad y la implementación informática.

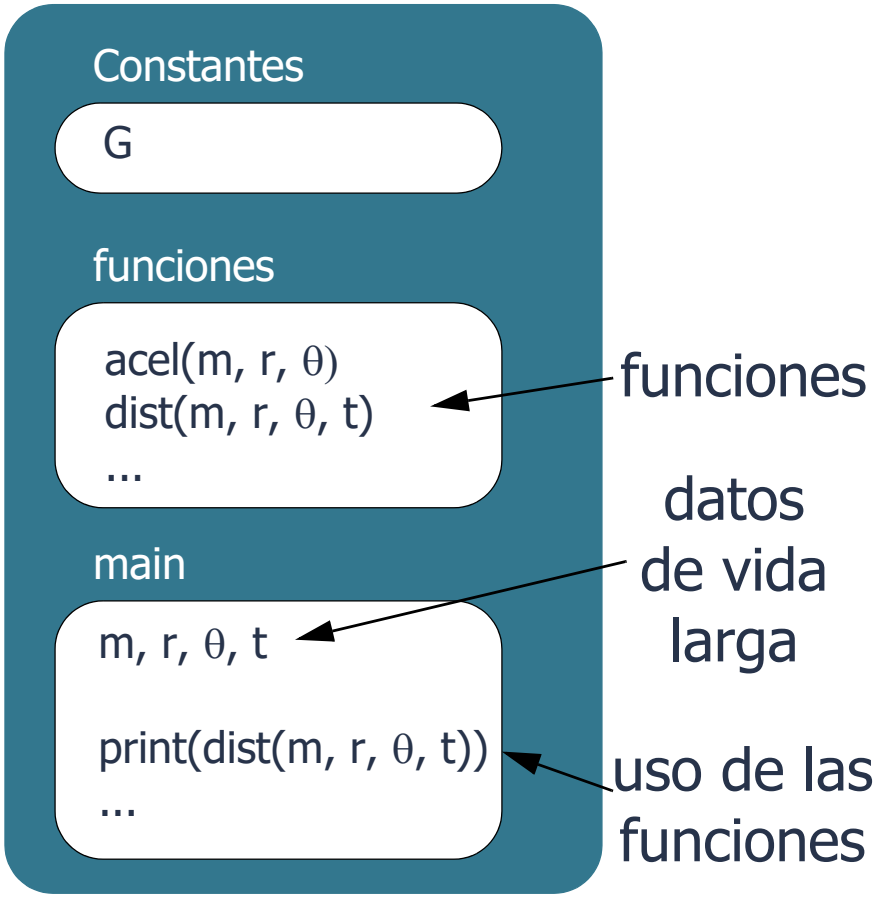
Ejemplo, esfera que rueda en un plano inclinado



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)



Notas:

Retomamos aquí el ejemplo del programa visto en el Capítulo 2, que realizaba cálculos de la trayectoria y energía de una bola que rueda sin deslizar por un plano inclinando.

En aquel ejemplo no habíamos visto las clases, y por tanto los datos de vida larga se guardaban dentro de la función `main()` y se pasaban como parámetros a las funciones que los necesitaban.

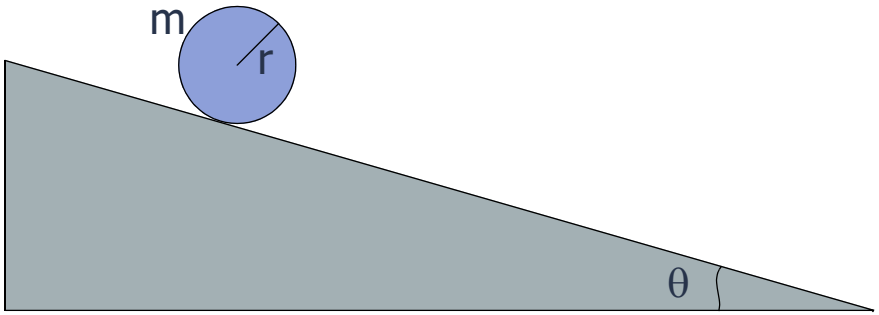
Esto presenta los tres problemas indicados arriba:

- Separación entre las funciones y sus datos.
- En programas grandes, no es práctico tener todos los datos en el `main()`.
  - Imaginemos un programa con centenares o miles de datos. Sería muy difícil de manejar y comprender.
- Muchos parámetros.

Todos estos inconvenientes se evitan con un diseño orientado a objetos, con clases y objetos.

- A continuación veremos cómo hacer un diseño orientado a objetos metiendo los elementos relacionados con el plano inclinado en la clase `PlanoInclinado`.

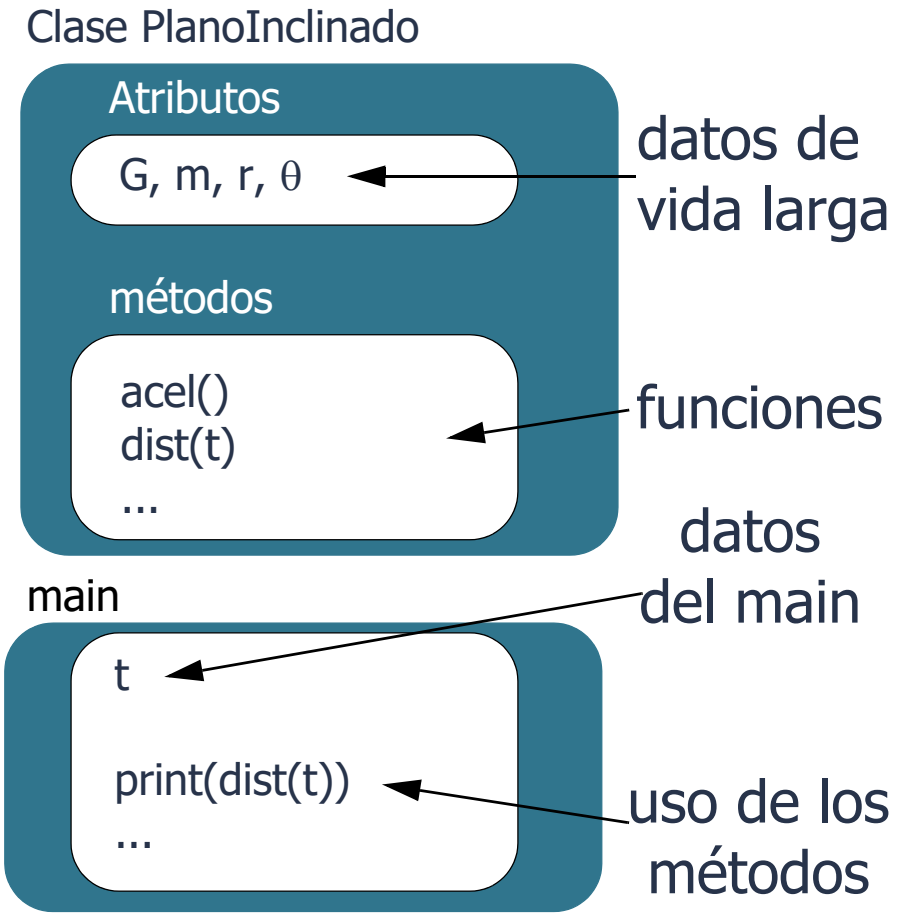
Ejemplo: Diseño orientado a objetos



Ventajas:

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

Diseño OO



Notas:

Aplicamos al ejemplo anterior un cambio de diseño, usando clases y objetos.

La clase `PlanoInclinado` contiene:

- Los datos de vida larga que antes estaban en el `main()` y que caracterizan el sistema bola-plano-inclinado, que ahora son *atributos*.
- Las funciones que hacían cálculos sobre el sistema del plano inclinado, convertidas en *métodos*.
  - Veremos más adelante que los *métodos* son funciones con un parámetro llamado `self`, que hace referencia al objeto al que pertenecen.

En el main se queda la variable `t`, que representa el tiempo y no es parte intrínseca del sistema del plano inclinado.

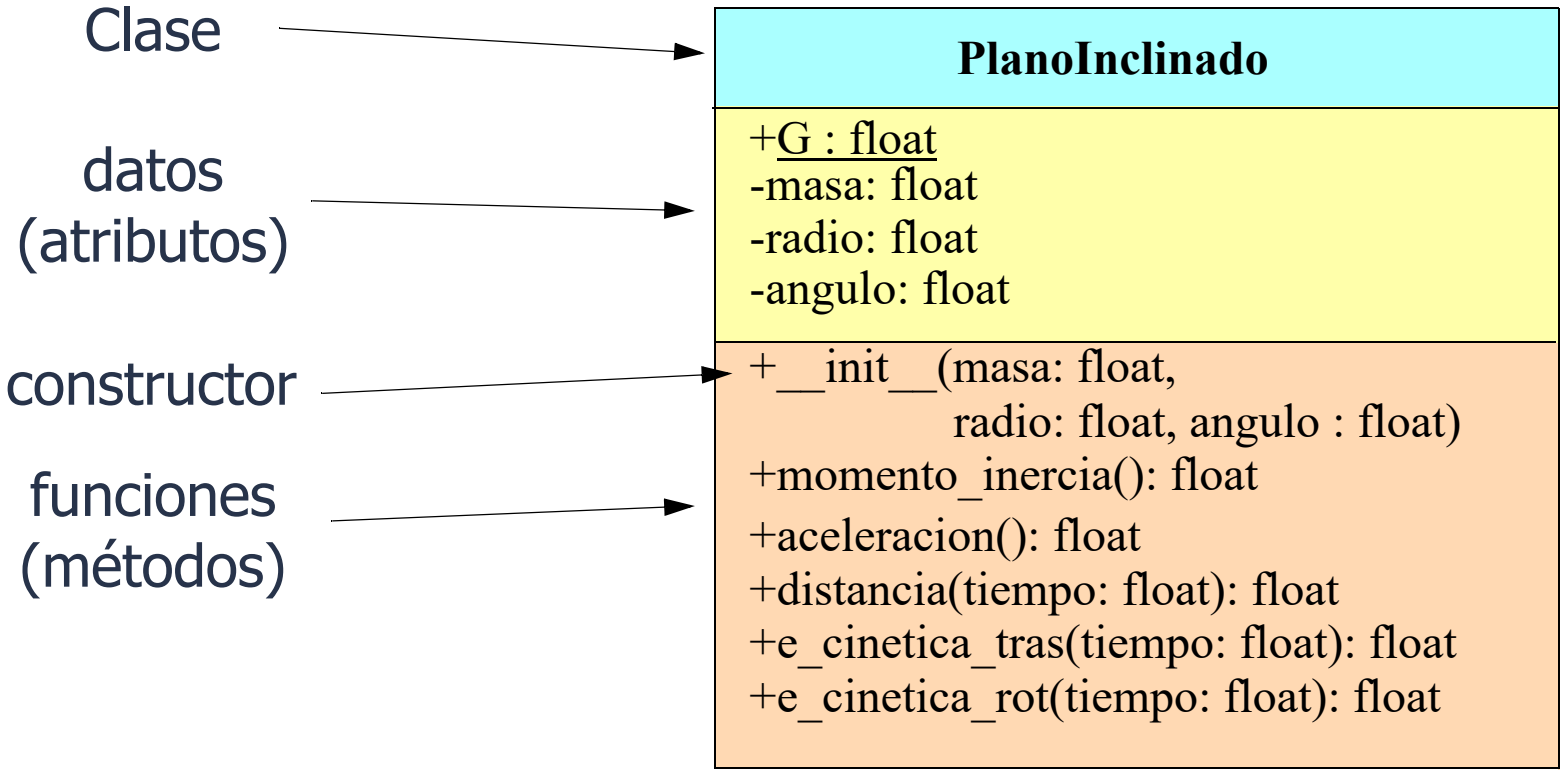
- Las funciones que necesitan este dato lo reciben como parámetro.

En el `main()` crearemos un objeto de la clase y además se organizan las llamadas a los métodos, igual que antes se invocaban las funciones.

La principal ventaja es mantener en un *mismo lugar* los datos y operaciones que los usan.

- En este caso los datos y operaciones del plan inclinado.
- Esto es especialmente útil para organizar programas grandes.

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)

Notas:

Se muestra arriba el diagrama de la clase `PlanoInclinado` según el nuevo diseño. El diagrama tiene:

- *nombre* de la clase, *atributos*, cada uno con su nombre y tipo y *métodos*, cada uno con su encabezamiento: nombre, parámetros y tipo retornado.

En los atributos y usamos los siguientes símbolos:

- El signo + indica que el atributo es *público*, es decir que se puede usar desde cualquier sitio del programa.
  - Esto es peligroso, excepto para los atributos constantes. Podría haber cambios sin control.
- El signo - indica atributo *privado*, es decir, que solo se puede usar desde la propia clase.
  - Esto será lo habitual. Queremos mantener los cambios de valor de los datos controlados y realizados solo desde dentro de la clase. Sólo los métodos cambiarán los atributos.

Cuando el atributo está subrayado, indica que es un atributo *estático*, es decir, perteneciente a la clase. Si no, es un atributo *normal*, perteneciente a los objetos.

- Los atributos de clase pertenecen a la clase y tienen un único valor compartido por todos los objetos.
- Los atributos normales pertenecen a los objetos. Cada objeto almacena en ellos un valor diferente.

Por ejemplo, si una clase define un atributo estático y uno normal, y de esta clase se hacen 1000 objetos, habrá un solo dato para el atributo estático, y 1000 datos, uno para cada uno de los atributos normales de cada 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*

Notas:

También los métodos pueden ser privados (-) o públicos (+), según queramos que solo se pueda usar desde dentro de la clase o desde cualquier lugar del programa.

- Es habitual que los métodos sean públicos.

Es muy habitual que las clases necesiten un método especial llamado constructor.

- Sirve para crear y dar valor a los atributos normales.
- En Python se llama `__init__()`.
  - Observar los dos caracteres barra baja _ antes y después de la palabra `init`.

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

Notas:

Vemos que en Python la clase comienza por un *encabezamiento* con la palabra `class` y el nombre de la clase.

A continuación viene el *comentario de documentación* de la clase, el *constructor* y los *métodos*.

- Igual que en otros elementos Python, el *alcance* de la clase se define mediante un nivel de *sangrado*.
- Al volver al sangrado normal, la clase se acaba.

Los métodos (incluido el constructor) llevan como primer parámetro la palabra `self`, que representa el objeto actual.

- Como esto es algo especial de Python y los diagramas de clases son independientes del lenguaje, este parámetro `self` no se pone en el diagrama.
- Se entiende que es implícito.
- Luego, al escribir el método en Python hay que ponerlo.

Pero ¿dónde están los *atributos*?

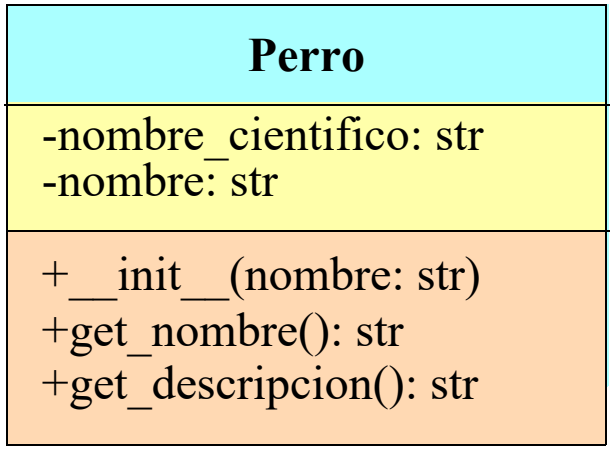
- En primer lugar, descritos en el *comentario de documentación*, tras el título `"Attributes:"`.
- Luego, se crean y definen dentro del *constructor*.

El *comentario de documentación* describe brevemente los contenidos de la clase y con más detalle los atributos, con sus unidades si es el caso.

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`

Notas:

Vemos aquí un ejemplo de una clase muy sencilla que sirve para:

- Guardar dos datos de un perro en sendos *atributos*: el nombre científico y el nombre del perro.
- Definir un *constructor* que crea y da valor a estos atributos.
- Definir también dos *métodos*, uno para obtener el nombre del perro y otro para obtener una descripción más completa que incluye el nombre y el nombre científico.

Observamos que el nombre de los atributos comienza por dos barras bajas: __

- Esto es el indicativo en Python de que un atributo es *privado*.
  - Es decir, que solo se puede usar desde dentro de la clase.
  - En el diagrama de clases las dos barras bajas no aparecían, ya que esto es algo especial de Python
  - Se suponen implícitas.

Los atributos privados también pueden comenzar por una sola barra baja _, pero esto lo usaremos para atributos heredados, que veremos en el capítulo 8.

Observamos también que para acceder a un atributo se usa una notación precedida del objeto actual, `self`.

- Por ejemplo, `self.__nombre` es el atributo `nombre`.

Vemos que ambos métodos utilizan como primer y único parámetro el objeto actual: `self`.

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
```

Notas:

Mostramos ahora la implementación en Python del ejemplo del plano inclinado visto en el capítulo 2, pero con el diseño orientado a objetos descrito en la página 16.

En este diseño, el módulo contiene:

- La constante `G` global, para la gravedad.
- La clase `PlanoInclinado` que contiene un constructor que define como atributos los datos del plano inclinado (masa, radio y ángulo) y como métodos las funciones de cálculo del momento de inercia, la aceleración, el desplazamiento y las energías cinéticas de traslación y de rotación.

Podemos ver que tanto el constructor como los métodos llevan como primer parámetro el objeto actual, `self`, que se usa como prefijo para:

- acceder a los atributos. Por ejemplo, `self.__masa`
- igualmente, para invocar a las funciones internas a la clase. Por ejemplo: `self.momento_inercia()`

Observamos que, puesto que los atributos son privados, llevan el prefijo con dos barras bajas __ delante del nombre.

Asimismo, al estar los atributos accesibles a todas las funciones, éstas no los necesitan como parámetros.

El constructor lleva como parámetros los valores iniciales de los atributos.

- Esto es habitual, pero no es imprescindible y habrá casos en que no sea apropiado.

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

Notas:

Para crear un objeto lo hacemos con su constructor, pero no ponemos su nombre (`__init__`) sino el nombre de la clase.

También es extraño que omitimos el primer parámetro del constructor (`self`). Está implícito en la instrucción.

- Veremos luego, que esto mismo ocurre cuando invocamos a un método: se omite el parámetro `self`.

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

Notas:

Como vimos anteriormente, los métodos y atributos de un objeto se usan desde dentro de la clase con el prefijo `self`.

Sin embargo, si los queremos usar desde fuera de la clase en lugar de ese prefijo se usa el nombre del objeto seguido de un punto .

Al igual que con el constructor, el parámetro `self` del método se omite. Está implícito en el objeto que se pone como prefijo.

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")
```

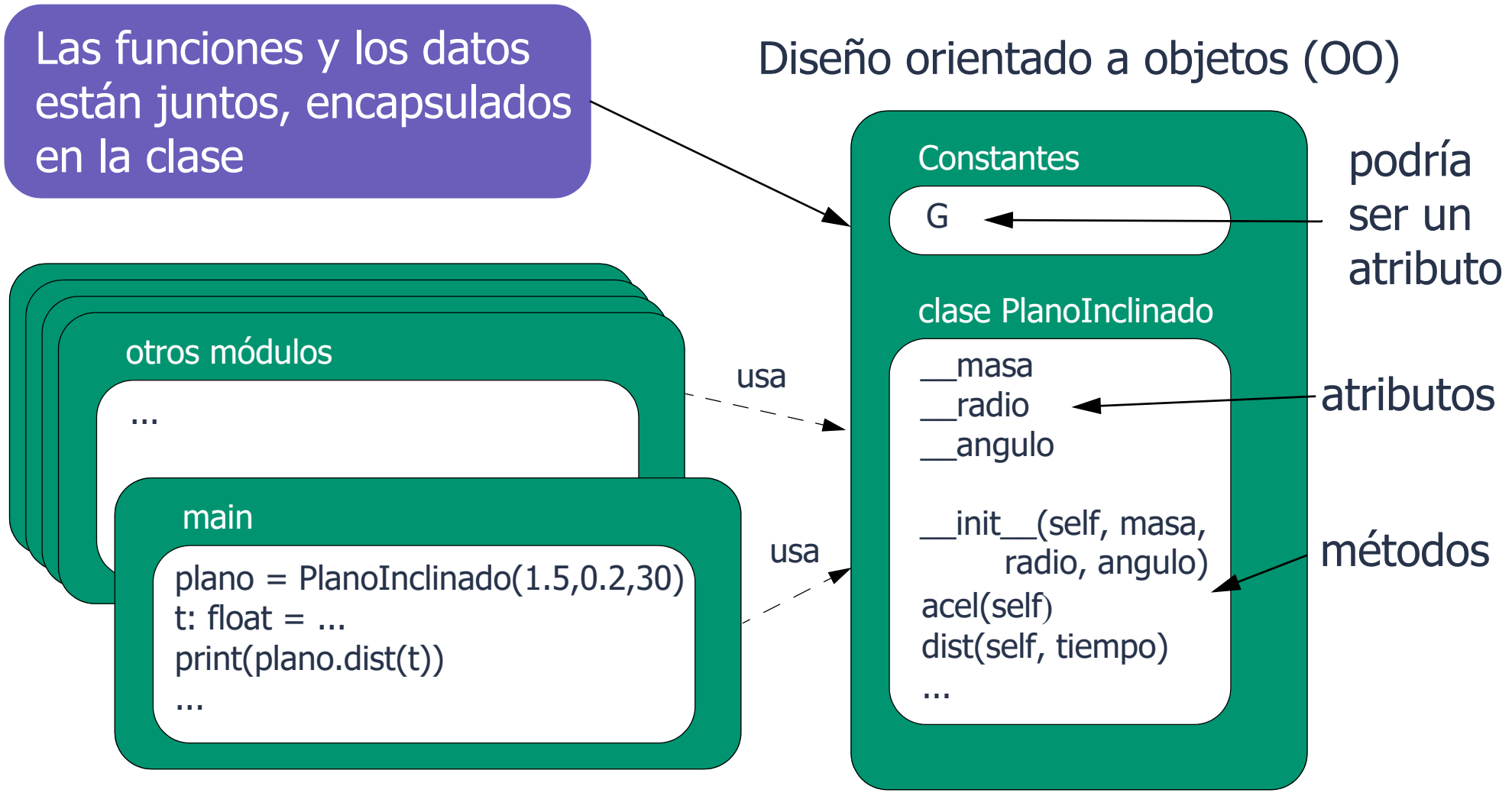
Notas:

Completamos arriba el ejemplo del plano inclinado mostrando un programa principal `main()` que crea un objeto de la clase `PlanoInclinado` y usa sus métodos.

Al crear el objeto invocando a su constructor hemos usado para cada parámetro su nombre, con el formato `nombre_parámetro = valor`

- Esta notación no es obligatoria. Podríamos usar solo el valor, pero incluir el nombre hace el programa más fácil de entender.
- Esta notación de parámetros con nombre no es exclusiva de las clases. Se puede usar con cualquier función.

Ejemplo: resumen del diseño



Notas:

En el diseño que se muestra arriba se enfatiza la ventaja de la programación orientada a objetos.

La clase engloba los datos del plano inclinado y sus métodos.

Esta clase se puede usar desde cualquier parte del programa. Por ejemplo, desde el `main()` y desde otras funciones y módulos.

- Bastará crear objetos de la clase y usar sus métodos.
- Nos despreocuparemos de dónde guardar los datos del sistema, tales como la masa, el radio o el ángulo. Los objetos creados siempre tendrán estos datos en su interior.

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`

Notas:

Sabemos que los *atributos de instancia* (los "normales") pertenecen a los objetos.

- Si, por ejemplo, tenemos una clase con 5 atributos y se crean 1000 objetos de esa clase, tendremos 5000 atributos en la memoria, 5 para cada objeto.

A veces nos interesa tener *atributos de clase*.

- Por ejemplo, para guardar una constante como la gravedad terrestre.
- No queremos que esa constante, que siempre vale lo mismo, se repita en cada objeto de la clase.
- Así ahorramos memoria.

Para crear atributos de clase simplemente los definimos dentro de la clase, pero fuera del constructor o de cualquier método.

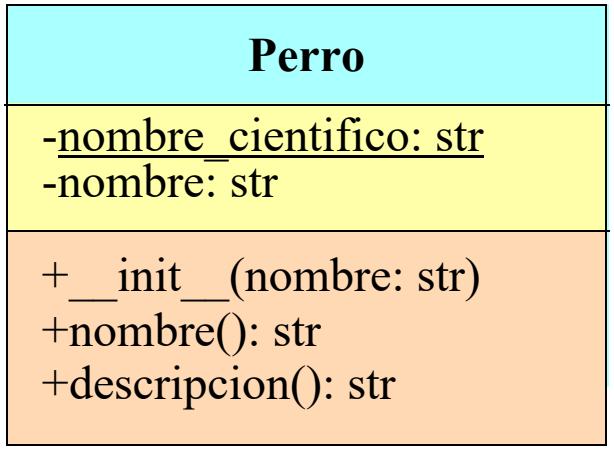
El uso es como los atributos de instancia:

- Desde dentro de la clase: con el prefijo `self`.
- Desde fuera de la clase:
  - Con un objeto cualquiera de la clase como prefijo.
  - O con el nombre de la clase como prefijo.

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

Notas:

Como vemos en el diagrama de arriba, los atributos estáticos (o de clase) se marcan con el formato 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`

Notas:

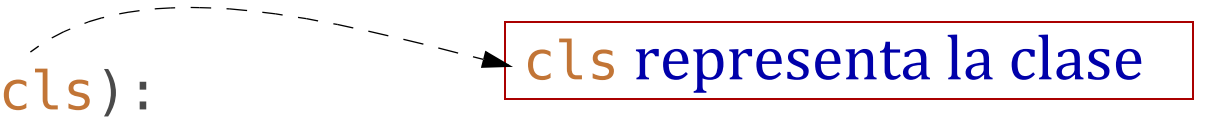
Vemos en el ejemplo que el atributo estático `__nombre_cientifico` se ha creado directamente dentro de la clase, fuera del constructor y de cualquier método.

No hay ningún otro cambio en esta clase, con respecto al ejemplo que se mostró en la página 24.

- Pero hay una *ventaja clara*. El nombre científico no se repite. Si creo 1000 objetos de la clase `Perro` en la memoria solo se almacena el nombre científico una sola vez.

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"
```



Notas:

Vemos ahora que, al igual que los atributos estáticos, podemos crear métodos estáticos y de clase.

Los métodos "normales" son de instancia y pertenecen a los objetos.

Los métodos estáticos y de clase pertenecen ambos a la clase, pero tienen ligeras diferencias:

- Métodos de clase:
  - Llevan como primer parámetro `cls`, que representa la clase.
  - Pueden usar atributos estáticos con la notación `cls.nombre_atributo`
  - No pueden usar atributos ni métodos de instancia.
  - Se identifican con el decorador `@classmethod` puesto antes del encabezamiento.
- Métodos estáticos:
  - No pueden acceder a ningún atributo. Ni estático, ni de instancia.
  - Tampoco pueden acceder a métodos de clase ni de instancia.
  - Se identifican con el decorador `@staticmethod` puesto antes del encabezamiento.

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
```

Notas:

Para invocar métodos estáticos o de clase desde fuera de la clase:

- Se pone como prefijo cualquier objeto de la clase o el nombre de la clase:
  `Clase.método(parámetros)`
  `objeto.método(parámetros)`

Para invocar métodos estáticos o de clase desde dentro de la clase, siempre desde métodos de instancia o de clase, se pone como prefijo `cls` o `self`, según estén disponibles:

- `cls.método(parámetros)`
- `self.método(parámetros)`

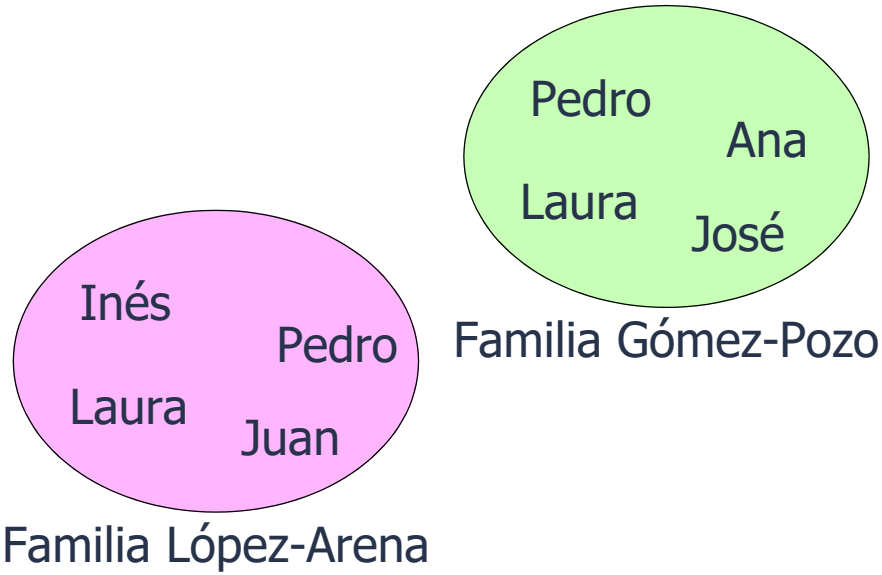
3.5. Espacios de nombres

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



Notas:

Los *ámbitos de nombres* permiten trabajar sin miedo a colisiones de nombres.

- En la vida corriente estamos muy acostumbrados a esto.
- En el ejemplo con dos familias que se muestra arriba, se repiten los nombres Pedro y Laura. Sin embargo, no hay confusión entre las personas, ya que cada una está dentro de su ámbito, en este caso dentro de su familia.

Los ámbitos de nombres permiten que las diferentes partes de un programa sean *independientes* unas de otras y facilitan la programación de grandes aplicaciones y el uso de librerías o módulos creados por otras personas.

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()
```

Notas:

Cuando un ámbito está incluido en otro, el interno puede usar los elementos del externo.

Cuando se repite un nombre, el nombre interno enmascara el externo dentro de su ámbito.

- Esto ocurre en el ejemplo anterior con la variable `b`, de la que hay dos versiones. Una en `f1` y otra en `f2`, ambas con valores diferentes.
- Desde `f2` usamos `b="x"`
- Desde `f1` usamos `b="b"`

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

Notas:

La instrucción `import` hace accesibles nombres de otros módulos Python.

Pero distinguimos dos elementos:

- El propio nombre del módulo.
- Los nombres contenidos en el módulo.

Podemos importar un elemento u otro con las dos formas de la instrucción `import`:

```
import módulo
from módulo import nombre
```

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

Notas:

Es posible agrupar módulos relacionados entre sí en *carpetas separadas*, para facilitar la organización de proyectos grandes.

A su vez, un paquete puede tener dentro subpaquetes, es decir, subcarpetas contenidas él.

Para remarcar que una carpeta es un *paquete Python* se suele poner en ella un fichero llamado __init__.py.

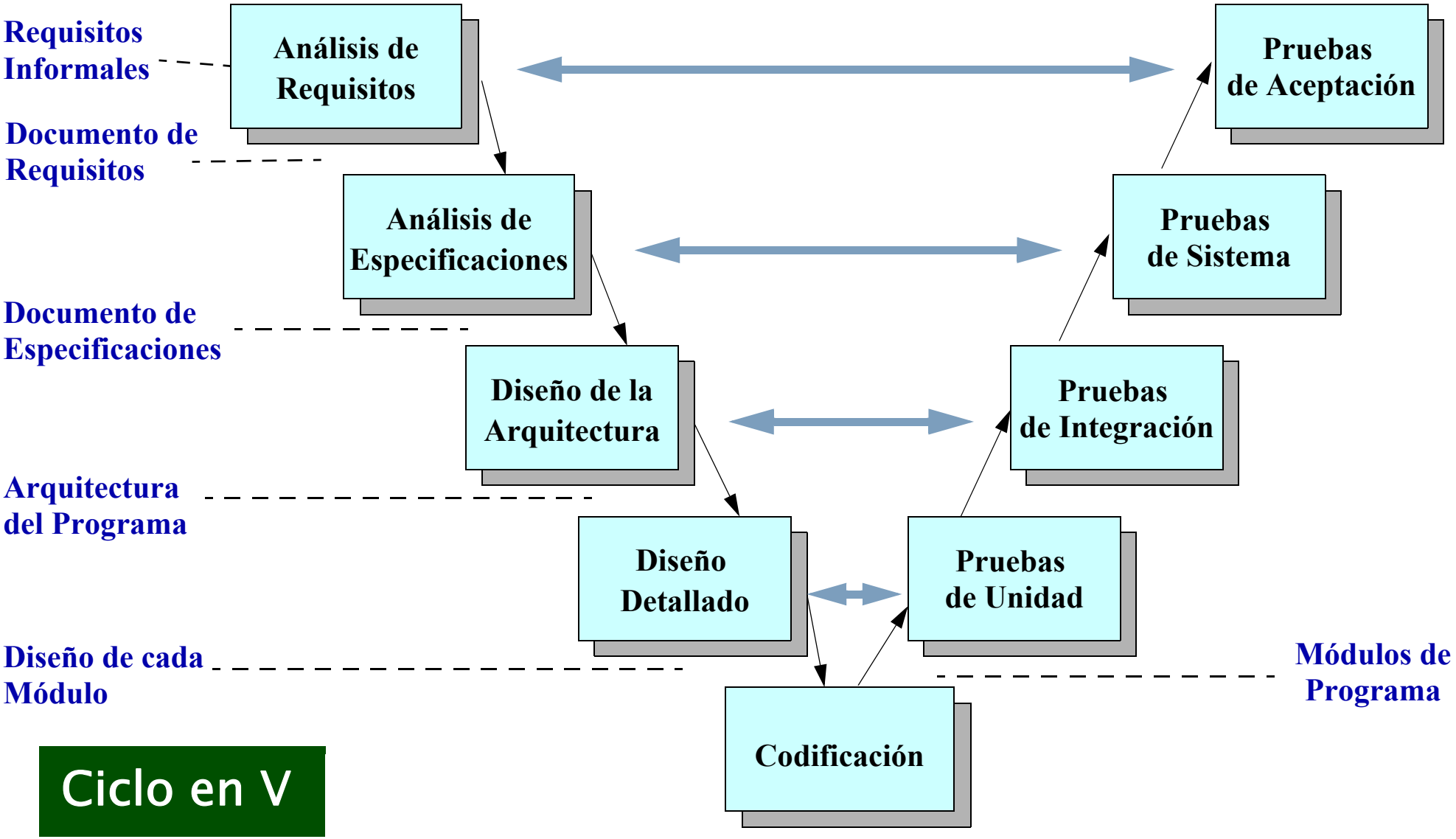
- Se puede dejar vacío, excepto por el comentario de documentación que se pone para describir los contenidos del paquete, autor y fecha, al igual que en cualquier módulo Python.

Los paquetes pueden estar contenidos directamente en la carpeta del proyecto.

Si están en otro lugar del disco duro, hay una *variable del sistema* llamada `PYTHONPATH` que almacena la lista de directorios que podrían contener paquetes, para que el intérprete Python los encuentre.

- Esta variable se puede gestionar desde Spyder en:
  `Herramientas -> Administrador del PYTHONPATH`.

3.7. El ciclo de vida del software



Ciclo en V

Notas:

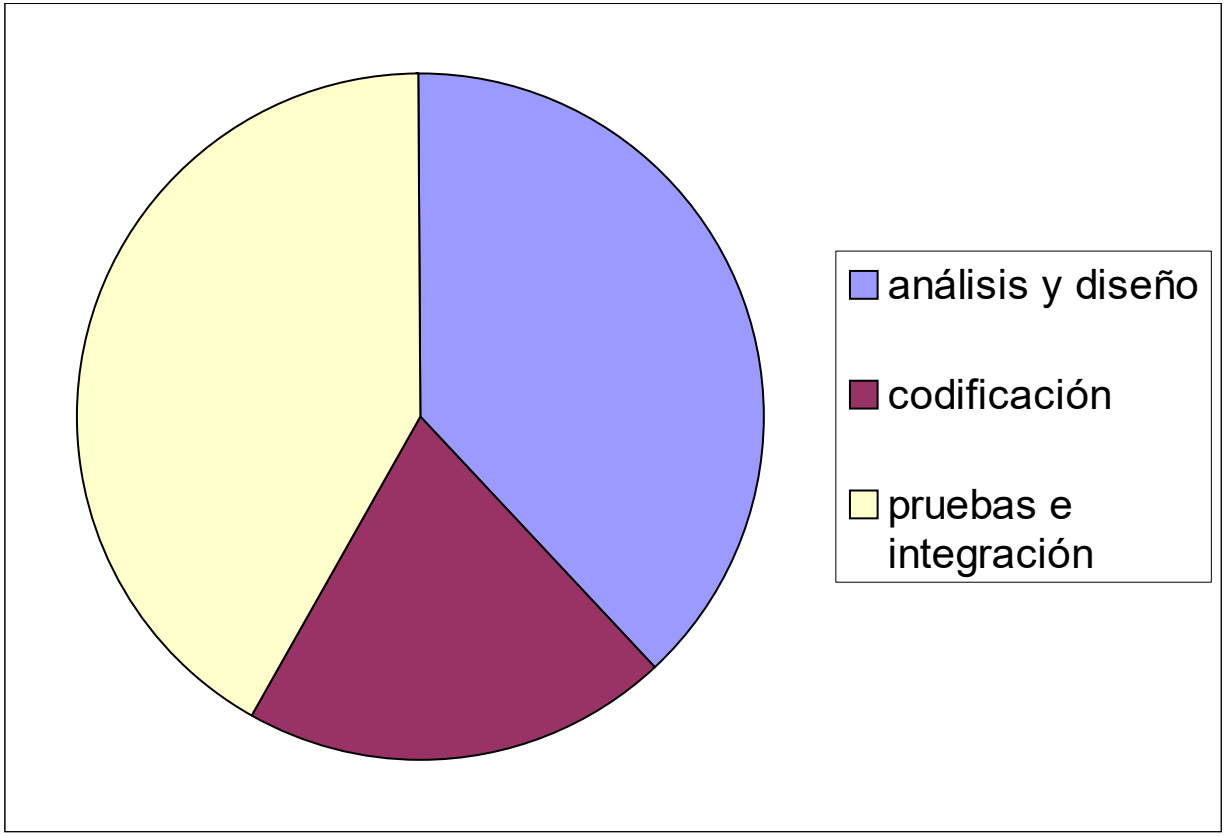
Los pasos que se siguen generalmente a la hora de desarrollar un programa son los siguientes:

- *Análisis de requisitos*: Se define el problema a resolver y todos los objetivos que se pretenden, pero sin indicar la forma en la que se resuelve.
- *Especificación*: Se determina la forma en la que se resolverá el problema, pero sin entrar aún en su implementación informática. Se determina asimismo la interfaz con el usuario.
- *Diseño de la arquitectura del programa*: Se divide el problema en módulos, se especifica lo que hace cada módulo, así como las interfaces de cada uno de ellos.
- *Diseño detallado de los módulos*: Para cada módulo se diseñan detalladamente las estructuras de datos y los algoritmos a emplear.
- *Codificación*: Se escribe el programa en el lenguaje de programación elegido.
- *Pruebas de módulos*: Se prueban los módulos del programa aisladamente y se corrigen los fallos hasta conseguir un funcionamiento correcto.
- *Integración y Prueba de sistema*: Se unen todos los módulos, y se prueba el funcionamiento del programa completo.
- *Prueba de aceptación*: Instalación en el lugar definitivo y aceptación por parte del cliente.

El *ciclo en V* recibe este nombre porque las fases del ciclo de vida se organizan en forma de V, como en la figura, y cada fase del desarrollo se hace corresponder con una fase de las pruebas.

- Por ejemplo, en la fase de pruebas de integración se comprueba que el diseño de la arquitectura es correcto. Si no fuese así, se hacen los cambios necesarios en este y tras ellos se procede a descender por la V, pasando al diseño detallado y codificación, para después ascender por la derecha de la V.

Distribución del Esfuerzo de desarrollo



Notas:

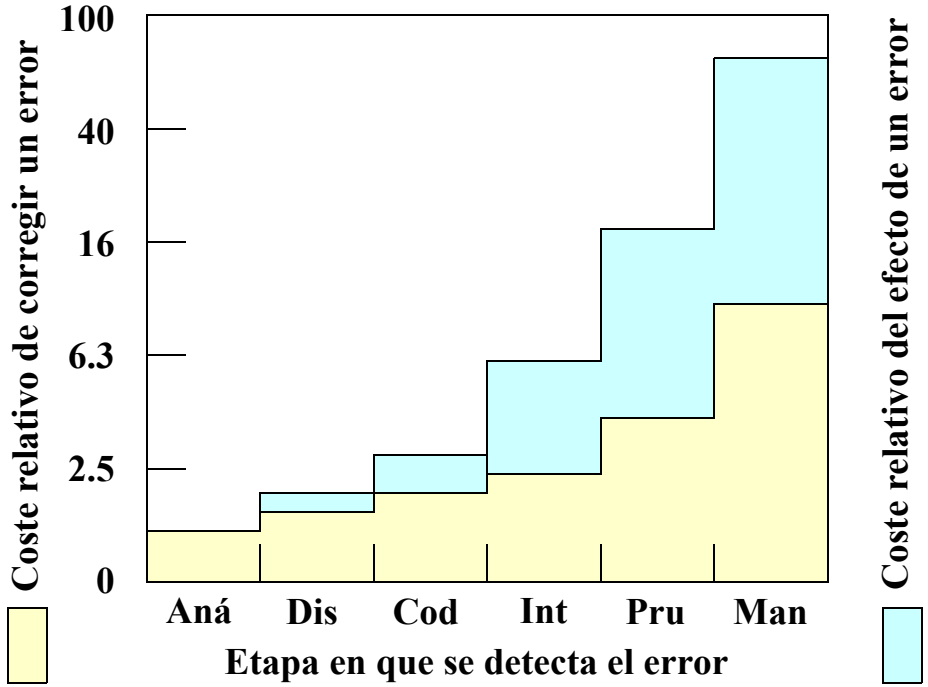
Un análisis realizado sobre un gran número de proyectos de software para estudiar el esfuerzo dedicado a las diferentes fases del ciclo de vida arroja los resultados de la figura.

Podemos ver que la codificación es la fase que menos esfuerzo lleva. Se emplea más tiempo en el análisis y diseño y también en las pruebas.

Podemos extraer una conclusión de esta figura: Si queremos reducir el coste final de un producto software, las fases donde más nos interesa invertir en tecnologías avanzadas son el análisis, diseño y pruebas.

Coste de los sistemas informáticos

Los errores software tienen un alto coste: efecto y corrección



Notas:

En la figura superior se muestra una gráfica del coste de corregir un error o del efecto causado por ese error, según la etapa del ciclo de vida en la que se detecta.

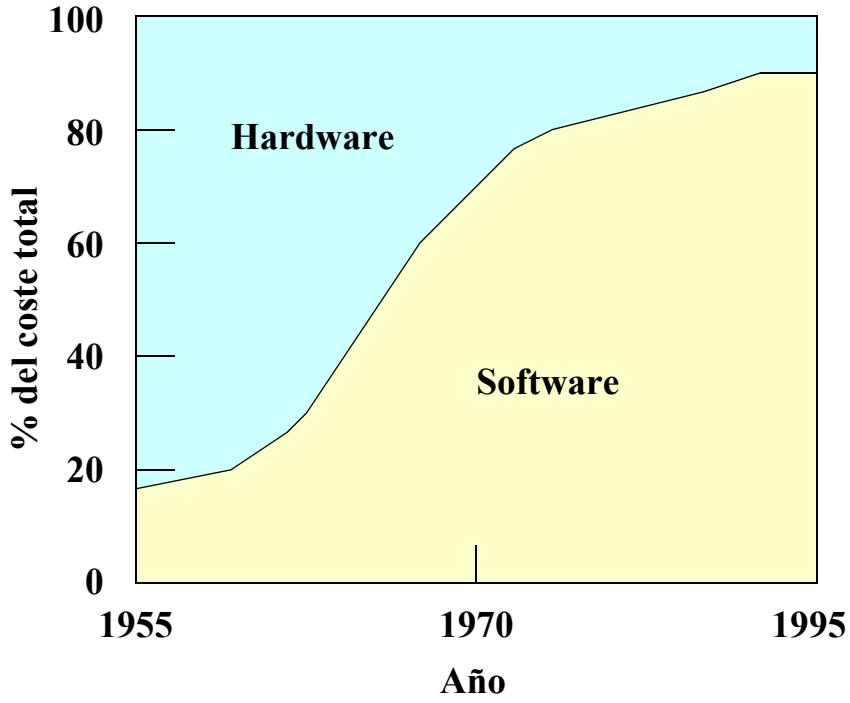
- Se ha añadido una etapa más al ciclo de vida, que es el mantenimiento (Man) del software tras su entrega y puesta en marcha.

Podemos ver que, a medida que avanzamos en las etapas del ciclo de vida, tanto la corrección como el efecto del error son más costosos.

De ello podemos sacar una conclusión interesante: merece la pena esforzarse por encontrar todos los errores posibles en las primeras etapas del desarrollo, evitando así los altos costes que tendrían los errores no detectados si se localizan más adelante.

Coste de los sistemas informáticos

Relación entre el coste de HW y SW



Notas:

La figura muestra el coste relativo del *hardware* (equipo físico) y el *software* (programas) en proyectos software.

En los inicios de la computación el hardware tenía un coste muy alto. A medida que el hardware ha ido bajando de precio con el avance en las tecnologías electrónicas el software se ha mantenido, o incluso ha aumentado su coste debido a la gran complejidad que tiene. En términos relativos ha aumentado mucho su coste.

- Aunque la gráfica llega hasta 1995, se ha seguido manteniendo la misma tendencia hasta nuestros días.

La gráfica es contraria a la percepción normal. Cuando compramos un computador personal, habitualmente gastamos más en el hardware. El software es gratis en muchos casos, o tiene un coste pequeño.

- Sin embargo, en los proyectos informáticos profesionales esto no es así.
  - El software doméstico o de oficina es barato, porque se venden muchas copias y el alto coste de desarrollo se distribuye entre muchas ventas.
  - El software profesional casi siempre es a medida y el alto coste del desarrollo no se divide en un gran número de copias.

En conclusión, para reducir el coste total de un proyecto informático hay que incidir en mejorar los procesos de la parte más costosa, que es el software.