Grado en Ciencia de Datos e Inteligencia Artificial

Asignatura: Algoritmos y Estructuras de Datos

Herencia y Polimorfismo

Profesores de Algoritmos y Estructuras de Datos

DLSIIS - E.T.S. de Ingenieros Informáticos
Universidad Politécnica de Madrid

Octubre 2020

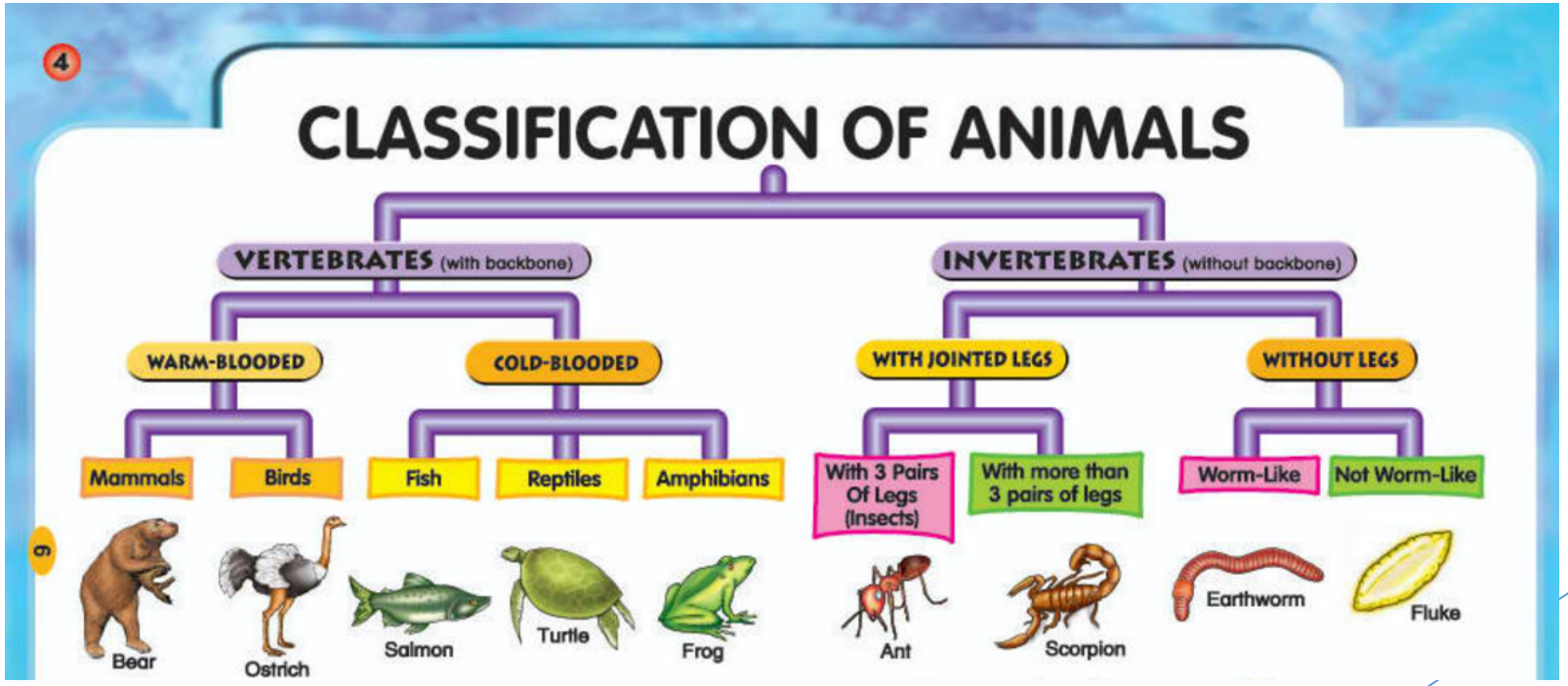
Contenido

- Introducción
- Motivación
- Herencia simple
  - Definición
  - Herencia de atributos y métodos
  - Constructores de las clases hijas
  - Referencias a objetos de subclases
  - Niveles de acceso
- Herencia múltiple
- Herencia con sobrescritura
  - Sobrescritura de métodos
  - Polimorfismo
- Clases abstractas
  - Métodos abstractos
- Interfaces
- Excepciones

Introducción

Herencia en el mundo real

- Existen **taxonomías/clasificaciones de conceptos** en multitud de áreas de conocimiento.
  - Se definen **relaciones de generalización "es un" o "es una subclase de"** entre los conceptos
  - Si B es una subclase de A, B poseerá también las características de A (**herencia**).
    - Como Mamífero es una subclase de Vertebrado de Sangre Caliente, cualquier mamífero tiene esqueleto y regula su temperatura corporal.



Herencia en la programación

- En programación, los conceptos se implementan mediante clases.
- Las características de una clase son sus:
  - atributos: datos de uso interno
  - métodos: operaciones que sabe realizar
- Los lenguajes de programación orientados a objetos permiten crear **jerarquías de clases**.
- Si B es una subclase de A, B heredará todas las características de A.

Motivación

Motivación: Ejemplo animales

- Vamos a crear unas clases sin el recurso de la herencia. Estas clases no tienen atributos por ahora:
  - **Perro** con el siguiente comportamiento (métodos públicos)
    - emitir_sonido: imprime por consola “guau!!”
    - lamer_hueso: imprime por consola “lamiendo un hueso!!!”
  - **Gato** con el siguiente comportamiento (métodos públicos)
    - emitir_sonido: imprime por consola “miau!!”
    - Jugar_ovillo: imprime por pantalla “me divierto jugando con un ovillo!”
  - **Oveja** con el siguiente comportamiento (métodos públicos)
    - emitir_sonido: imprime por consola “beee!!”
    - pastar: imprime por pantalla “pastando hierba”
- Creamos un **programa de prueba** que crea una instancia de cada y llama a emitir_sonido()

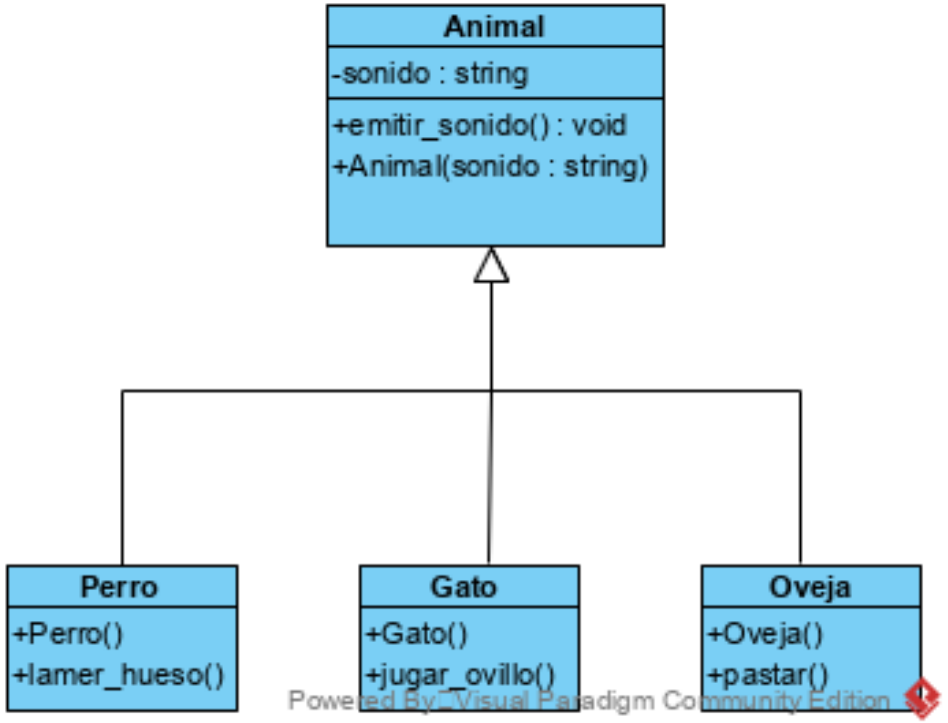
Motivación: Ejemplo animales

- Podemos ver que:
  - Gato, Oveja y Perro comparten un método que es emitir_sonido(), aunque cada uno emite un sonido distinto
  - Cada una de las clases tiene un método específico
  - Existe una clase más general que las abarca ⟶ Animal
  - Implementando la clase Animal vamos a simplificar las otras tres clases
  - Animal:
    - Tendrá un atributo con el sonido que emite
    - Va a tener un constructor recibirá el sonido y lo inicializará
    - Tendrá un método público emitir_sonido
    - Se podrá consultar el sonido por medio de una propiedad

Los hijos lo heredarán

Motivación: diagrama UML

- Animal es la clase padre / clase base
- Gato, Perro y Oveja son las clases hijas ⟶ derivan de Animal
- Usando la notación UML sería:



Herencia simple

Definición

- La **herencia** consiste en que una clase hija (subclase) “extiende” a su clase padre:
  - **Hereda las propiedades**: atributos no privados
  - **Hereda el comportamiento**: métodos no privados
- Esto permite **reutilizar el código de la clase padre** en todas las clases hijas evitando la duplicidad de código
- Una clase hija es una “**correcta subclase**” de una clase padre si y solo si:
  - Cumple la regla de “**es-un**”: un objeto o instancia de la clase hija lo es también de la clase padre
  - Cumple con el **principio de sustitución**: los objetos de la clase hija pueden sustituir a los de la clase padre, cuando se les pida realizar una operación

Herencia de atributos y métodos

- La clase padre Animal se define de la siguiente manera:

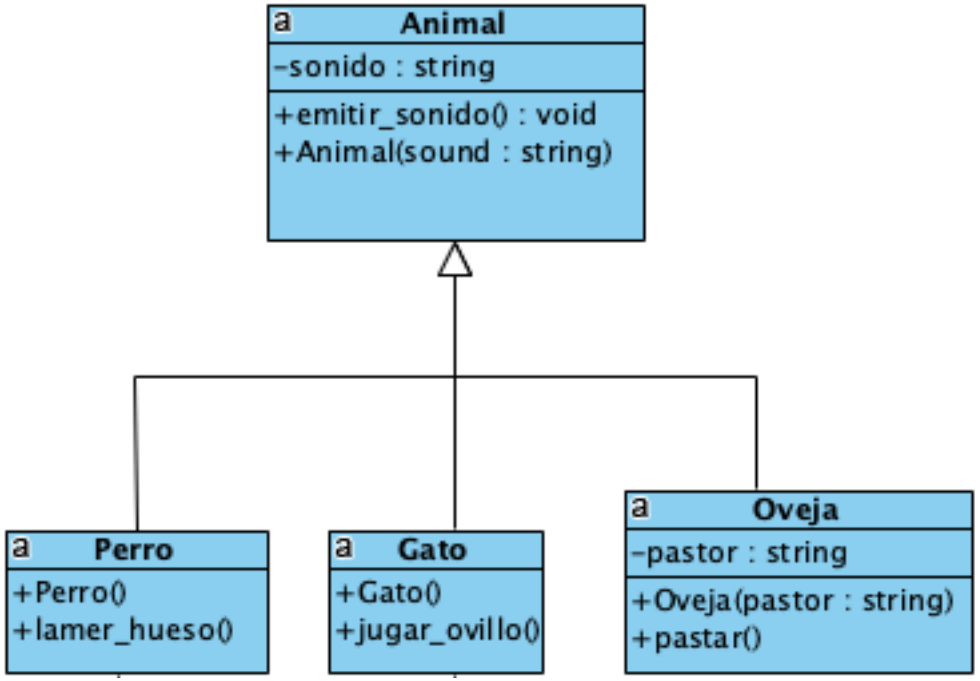
```
class Animal:

  def __init__(self, sound: str):
        self.__sonido: str = sound

    def emitir_sonido(self):
        print(self.__sonido)

    # We have defined sonido as property and getter method is provided
    @property
    def sonido(self) -> str:
        return self.__sonido
```

Atributo privado

Propiedad sólo de lectura

Herencia de atributos y métodos

- Vamos a añadir un nuevo atributo pastor (tipo str) a la clase Oveja.
- Por ejemplo, la clase Oveja posee:
  - todo lo que hereda de la clase padre Animal (sonido y emitir_sonido()) y
  - su propio atributo pastor y sus propios métodos, constructor y pastar().



Constructores de las clases hijas

- Si la clase padre proporciona un constructor:
  - La llamada al constructor del padre debería ser la primera sentencia, pero no es necesario
  - La clase hija puede llamarlo en el constructor de la siguiente forma:

Si B es subclase de A -> class B(A):

```
class Gato(Animal):
    def __init__(self):
        super().__init__("Miau!!")

    def jugar_ovillo(self):
        print("Me divierto jugando con un ovillo!")
```

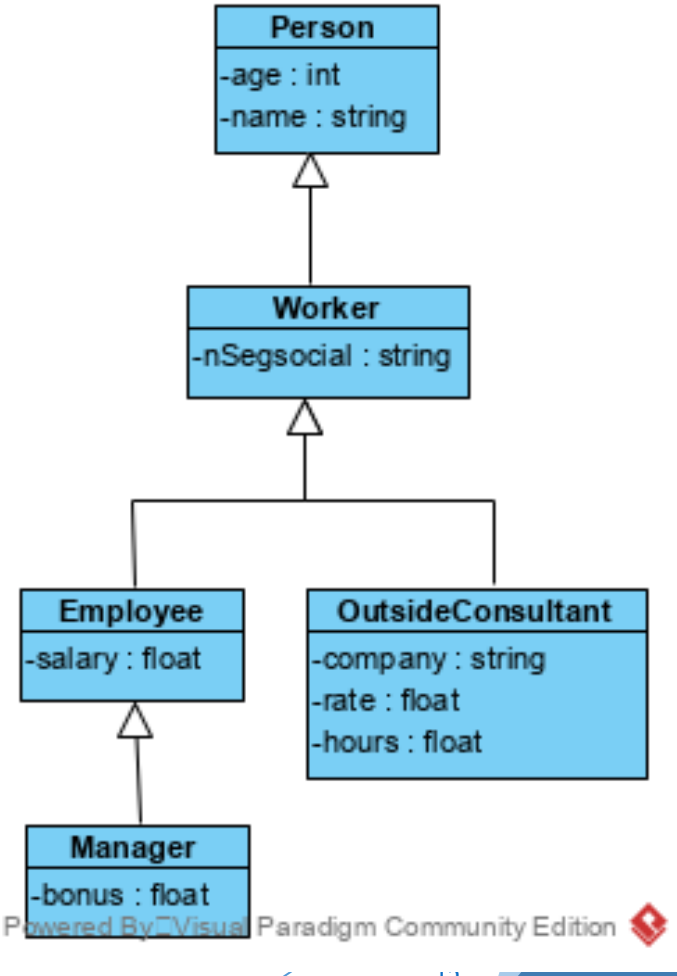
```
class Perro(Animal):
    def __init__(self):
        super().__init__("Guagua!!")

    def lamer_hueso(self):
        print("lamiendo un hueso!!!")
```

```
class Oveja(Animal):
    def __init__(self, pastor: str):
        Animal.__init__(self, "Beee!")
         self.__pastor = pastor

    def pastar(self):
        print("pastando hierba!!")
```

Llamada al constructor del padre

Herencia: Ejercicio Empresa

- Clase **Person** con atributos name (string) y age (int)
  - Constructor que inicialice los atributos con parámetros de entrada y no permite age < 0
- Clase **Worker** que herede de **Person** y tenga un número de la seguridad social (string)
  - Constructor con todos los atributos de la clase con parámetros de entrada
- Clase **Employee** que herede de **Worker.** Tiene un atributo **salary** (float) que representa el salario anual
  - Constructor con todos los atributos de la clase con parámetros de entrada. No se permite un salario < 0
- Clase **Manager** que herede de **Employee.** Tiene un atributo **bonus** (float)
  - Constructor con todos los atributos de la clase con parámetros de entrada. No se permite bonus <= 0
- Clase **OutsideConsultant** que herede de **Worker.** Tiene **rate** (float) y **hours** (float) y **company** (String)
  - Constructor con todos los atributos de la clase con parámetros de entrada. No se permite rate y hours < 0



Referencias a objetos de subclases

```
animal1: Animal = Animal("Cuack")
if isinstance(animal1, Gato): # comprueba el tipo de objeto al que apunta animal1
    animal1.jugar_ovillo()
else:
    print("animal1 no es gato")

gato1: Gato = Gato()
gato2: Animal = Gato()  # upcasting
gato2.jugar_ovillo()  # IDE warning: la clase Animal no tiene ese método
if isinstance(gato2, Gato):
    print("gato2 es un gato")
    gato2.jugar_ovillo() # IDE no indica error, llamada controlada

oveja1: Oveja
oveja2: Animal = Oveja()
# oveja1 = oveja2 # IDE warning: posible asignación entre tipos incompatibles
if isinstance(oveja2, Oveja):
    oveja1 = oveja2  # downcasting controlado
    oveja1.pastar()
```

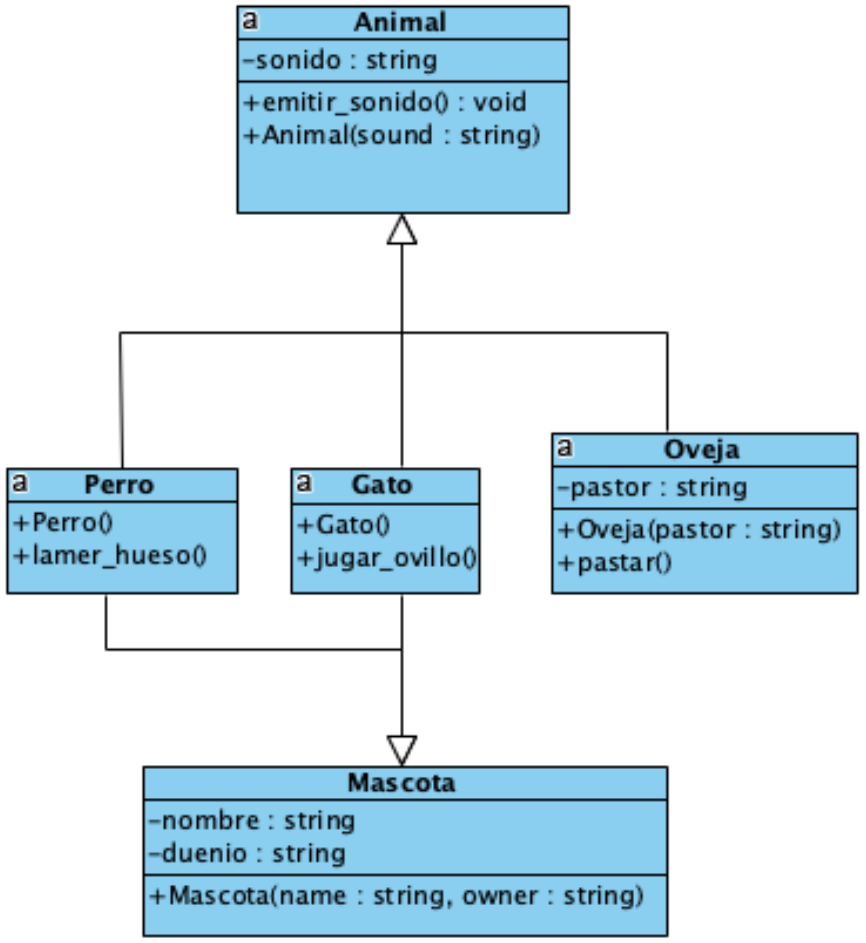
Niveles de acceso

- Los niveles que se proporcionan en Python para atributos y métodos son:
  - **Privado**:
    - El atributo o método debe empezar con la secuencia de caracteres __
    - Sólo es visible dentro de la propia clase
  - **Público**:
    - El atributo o el método no van precedido del carácter _
    - El atributo y el método es accesible desde cualquier parte en el que se use la instancia del objeto
  - **Protegido** (solo tiene sentido con la herencia!!):
    - El atributo o método debe empezar con un carácter _ (sólo uno)
    - El acceso se restringe a las clases hijas
    - Python permite que se acceda a estos atributos desde cualquier sitio dejando al IDE que avise del uso no adecuado de un atributo protegido

Herencia múltiple

Herencia múltiple

- Cuando una clase tiene varios padres tenemos herencia múltiple
- No todos los lenguajes POO soportan la herencia múltiple
- En nuestro ejemplo vamos a definir la clase Mascota
  - Una mascota tiene un nombre
  - Una mascota tiene un dueño
- Vamos a hacer que Gato y Perro hereden también de Mascota



Herencia múltiple

```
class Mascota:
  def __init__(self, name: str,
               owner: str):
        self.__nombre: str = name
        self.__duenio: str = owner

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

  @property
  def duenio(self) -> str:
      return self.__duenio
```

```
class Animal:

    def __init__(self, sonido: str):
        self.__sonido: str = sonido

    def emitir_sonido(self):
        print(self.__sonido)

    # We have defined sound as property
    # and get method is provided
    @property
    def sonido(self) -> str:
        return self.__sonido
```

```
class Gato(Animal, Pet):
    def __init__(self, name: str = "", owner: str = ""):
        Animal.__init__(self, "Miau!!")
        Mascota.__init__(self, name, owner)

    def jugar_ovillo(self):
        print(F'{self.name} has fun playing with wool ball!'')
```

Llamada al constructor de Animal

Llamada al constructor de Pet

Herencia múltiple

```
class Perro(Animal, Pet):

    def __init__(self, name: str = "", owner: str = ""):
        Animal.__init__(self, "Guagua!!")
        Mascota.__init__(self, name, owner)

    def lamer_hueso(self):
        print(f"{self.name} is licking a bone!!!")
```

```
gato2: Animal = Gato("Blacky", "Mary")
mascota1: Pet = Perro("Pluto", "Mickey")
print(f"my name is {mascota1.nombre} and the name of my owner is {mascota1.duenio}")
# To avoid runtime errors, you must ensure that the instance, is a dog instance
# before you call to lamer_hueso.
# Python allows us call to lamer_hueso without further verifications,
# but at risk of provoking a runtime error
if isinstance(mascota1, Perro):
    mascota1.lamer_hueso()
mascota1 = gato2
print(f"my name is {mascota1.name} and the name of my owner is {mascota1.owner}")
if isinstance(mascota1, Gato):
    mascota1.jugar_ovillo()

mascota1.lamer_hueso() #<--- runtime error
```

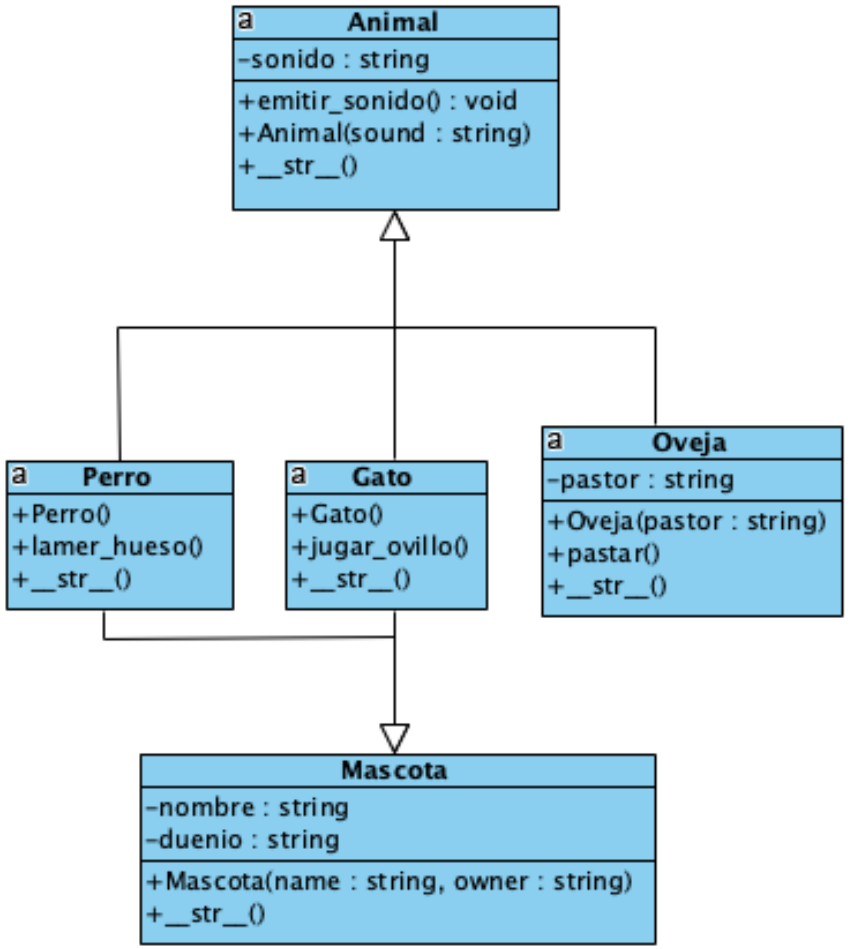
mascota1 es un perro

mascota1 ahora es un gato

Herencia con sobrescritura

Sobrescritura de métodos

- Puede ser que la implementación de un método no sirva para una clase hija:
  - Se sobrescribe el método y hay dos opciones:
    a) Puede prescindirse por completo de la implementación del padre
    b) O puede usarse y complementarse la implementación del padre
- Vamos a definir el método de conversión a string **__str__()** en el ejemplo de los animales con herencia múltiple de forma que:
  - Animal crea el siguiente string: "emito el sonido: <sonido>"
  - Oveja genera el siguiente string: "soy una oveja, mi pastor es <pastor> y emito el sonido: <sonido>"
  - Mascota genera el siguiente string: "nombre <nombre> y dueño <dueño>"
  - Perro genera el siguiente string: "soy un perro con nombre <nombre> y dueño <dueño> y emito el sonido: <sonido>"
  - Gato genera el siguiente string: "soy un gato con nombre <nombre> y dueño <dueño> y emito el sonido: <sonido>"



Sobrescritura de métodos

```
class Mascota:
  …
 def __str__(self) -> str:
       return f"name {self.nombre} y" \
              f" age {self.duenio}"
```

```
class Animal:
…
  def __str__(self) -> str:
       return f"emito el " \
              f"sonido: {self.sonido}"
```

Llamada al str de Pet

```
class Oveja(Animal):
    …
    def __str__(self) -> str:
        return f"soy una oveja, mi
pastor es {self.__pastor} y" \
               f" {super().__str__()}"
```

Llamada al str de Animal

```
class Gato(Animal, Mascota):
    …
 def __str__(self) -> str:
       return f"soy un gato con nombre " \
              f"{Pet.__str__(self)} y {Animal.__str__(self)}"
```

```
class Perro(Animal, Mascota):
    …
 def __str__(self) -> str:
       return f"soy un perro con nombre " \
              f"{Pet.__str__(self)} y {Animal.__str__(self)}"
```

Polimorfismo

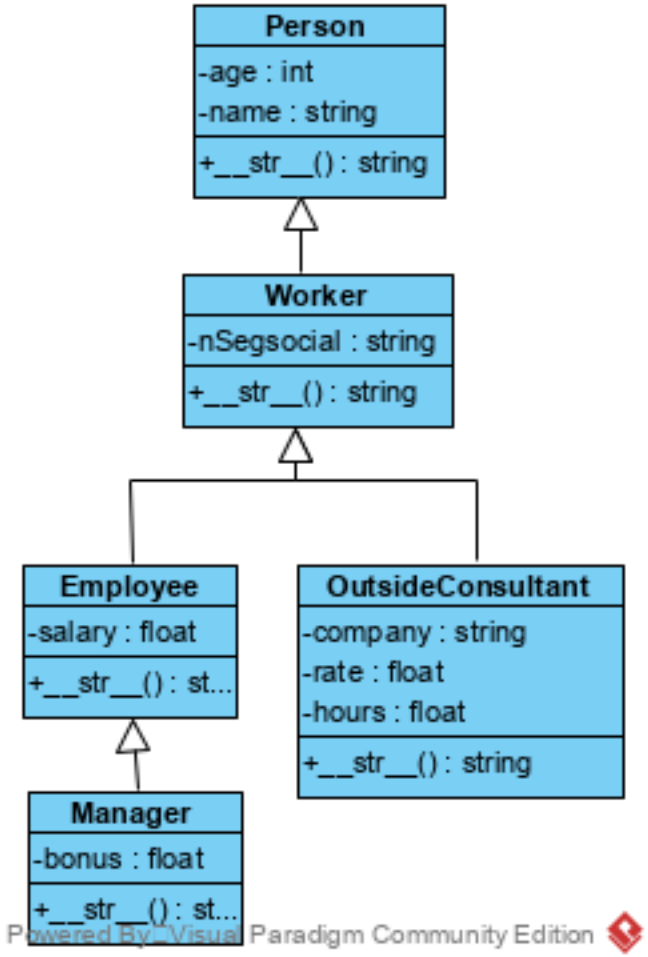
- Queremos un programa que imprima todos los datos de los animales que se tienen en una lista de animales.
- Como la clase Animal es el padre del resto, podemos tener objetos de tipo Oveja, Perro, Gato … en la lista.
- Recorreremos esa lista y haremos uso del __str__() que se ha implementado previamente en cada clase.
- Gracias al polimorfismo se llamará a la implementación adecuada de str() en función de la instancia que haya en cada posición de la lista.
  - Este mecanismo se llama DYNAMIC BINDING o enlazado dinámico

Polimorfismo

- El polimorfismo permite que una misma llamada ejecute distintas sentencias dependiendo de la clase a la que pertenezca el objeto.
  - El código a ejecutar se determina en tiempo de ejecución ⇒ **Enlace dinámico**
- El polimorfismo en POO se da por el uso de la herencia y la sobrescritura de métodos:
  - Distintas implementaciones en la clase hija y en la clase padre
  - Distintas implementaciones en las clases hijas

Herencia: Ejercicio Empresa con sobreescritura

- Dadas las clases del ejercicio de la Empresa:
  - Añadir a cada una de las clases un método que convierta una instancia a un string que devuelve un string con todos los atributos de ese objeto
  - Por ejemplo para Person: “name: <name> age: <nn>”
  - Por ejemplo para Worker: “name: <name> age: <nn> nSegSocial: <nSegSocial>”
  - Al sobrescribir un método se debe reutilizar el código de la clase padre (super).



Herencia: clase object

- Es una clase predefinida en Python en la que se definen los “métodos mágicos”, por ejemplo, el método __str()__ o los métodos para sobrecargar los operadores:

Operator	Method
<	object.__lt__(self, other)
<=	object.__le__(self, other)
==	object.__eq__(self, other)
!=	object.__ne__(self, other)
>=	object.__ge__(self, other)
>	object.__gt__(self, other)

- Todas las clases en Python heredan de forma implícita de la clase **object**.
- Cuando en una clase definimos un método __eq()__ o __str()__ en realidad estamos sobrescribiendo el método que se hereda de la clase **object**.
  - El método __eq__() de object hace lo mismo que el operador **is**.

Herencia: Ejercicio comparando personas

- Partiendo de la clase persona del ejercicio de los trabajadores:
  - Se le añade un atributo *surname* que contendrá el primer apellido
  - Sobrescribir la igualdad.
    - Dos personas son iguales cuando tienen el mismo nombre y apellido.
  - Sobrescribir los operadores de comparación: menor, mayor, menor igual y mayor igual.
    - Para las comparaciones sólo se tendrá en cuenta el nombre y el apellido
  - Escribir un programa de prueba que use el método sort para ordenar una lista de personas.

Clases abstractas

Herencia: motivación de los métodos abstractos

- Crear la clase padre Polígono y las clases hijas Triangulo y Rectángulo.
- Crear un método *get_type(): str* que clasifique el polígono, es decir:
  - Si es un triángulo, debe indicar si es equilátero, isósceles o escaleno.
  - Si es un rectángulo, debe indicar si es un cuadrado o no.
- El método es aplicable a cualquier objeto de tipo Poligono, pero su implementación será diferente para cada tipo.
- El método *get_tipo(): str* lo implementaremos como un método abstracto.

Herencia: métodos abstractos y clases abstractas

- Método abstracto
  - Los métodos abstractos sirven para declarar métodos que son aplicables a una clase padre pero cuya implementación depende de cada clase hija.
  - Se conoce su firma/cabecera (parámetros de entrada y valor retornado).
- Clase abstracta
  - Clase que tiene al menos un método abstracto.
  - No se pueden crear instancias de una clase abstracta.
  - Se usará como padre o superclase de otras clases.
  - Las clases hijas o subclases deben implementar los métodos abstractos o serán también clases abstractas.

Herencia: clases y métodos abstractos en python

- Como se ha podido ver en el código del ejemplo tenemos:
  - Una clase abstracta deriva de la clase *abc.ABC*

```
from abc import ABC, abstractmethod


class Polygon(ABC):
```

  - Los métodos abstractos van precedidos del decorador *@abstractmethod* y el cuerpo del método vacío

```
@abstractmethod
def get_type(self) -> str:
    pass
```

  - El cuerpo del método será definido en las clases hijas respetando la firma del mismo

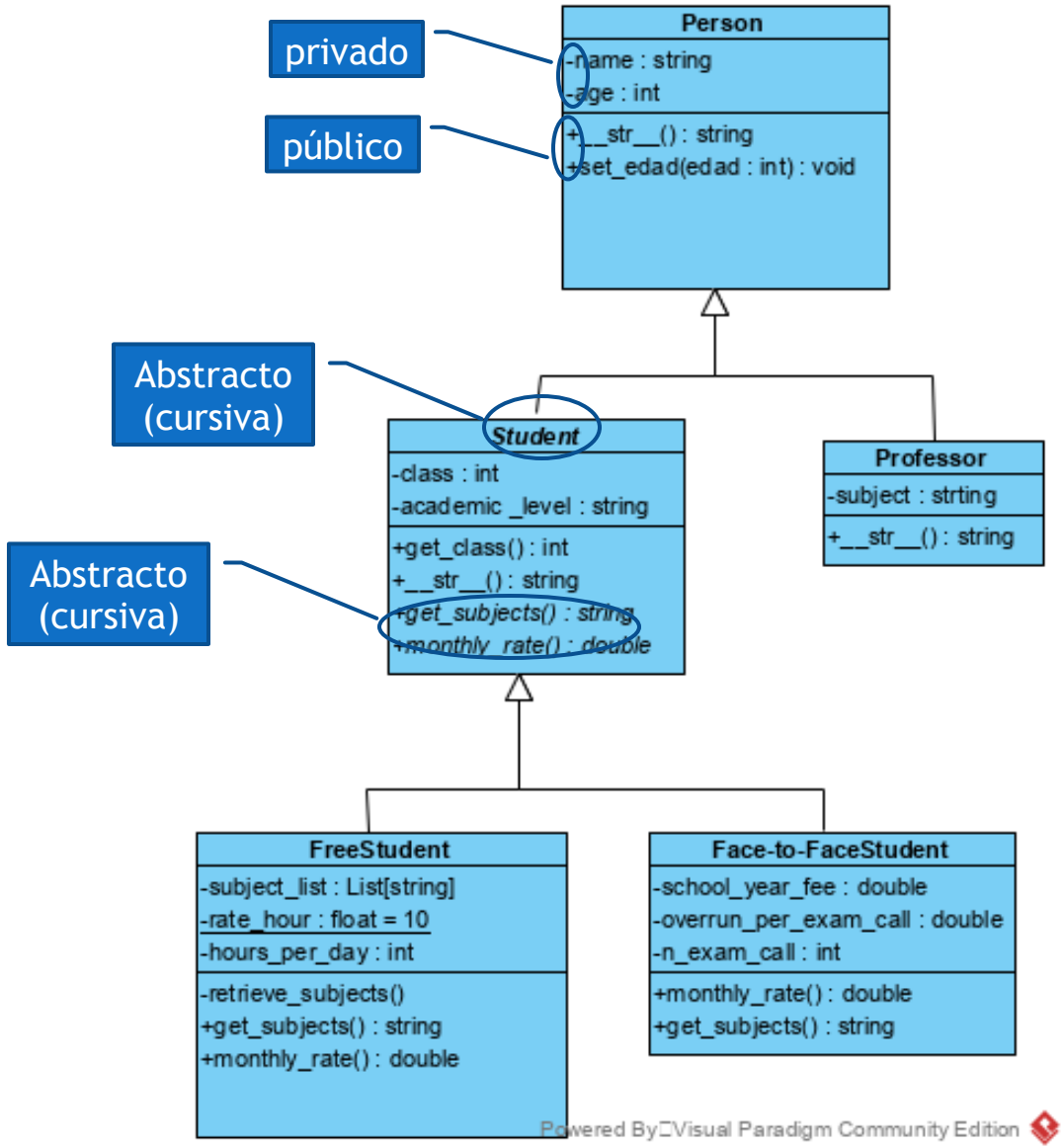
Herencia: ejemplo con alumnos y profesores

- Se pretende modelar parte de la gestión de una institución dedicada a la enseñanza
- En el sistema intervienen
  - Personas que tendrán un nombre y la edad.
    - Se podrá consultar la edad de la persona
  - Se tendrán dos subclases de la clase persona:
    - Profesor: El profesor imparte una asignatura
    - Alumno: Que está matriculado de asignaturas y que tiene un nivel académico. Hay dos tipos:
      - Alumno Libre: se matricula en asignaturas sueltas que no tienen por qué ser del mismo curso.
      - Alumno Presencial: se matricula en todas las asignaturas de un curso.

Herencia: ejemplo con alumnos y profesores

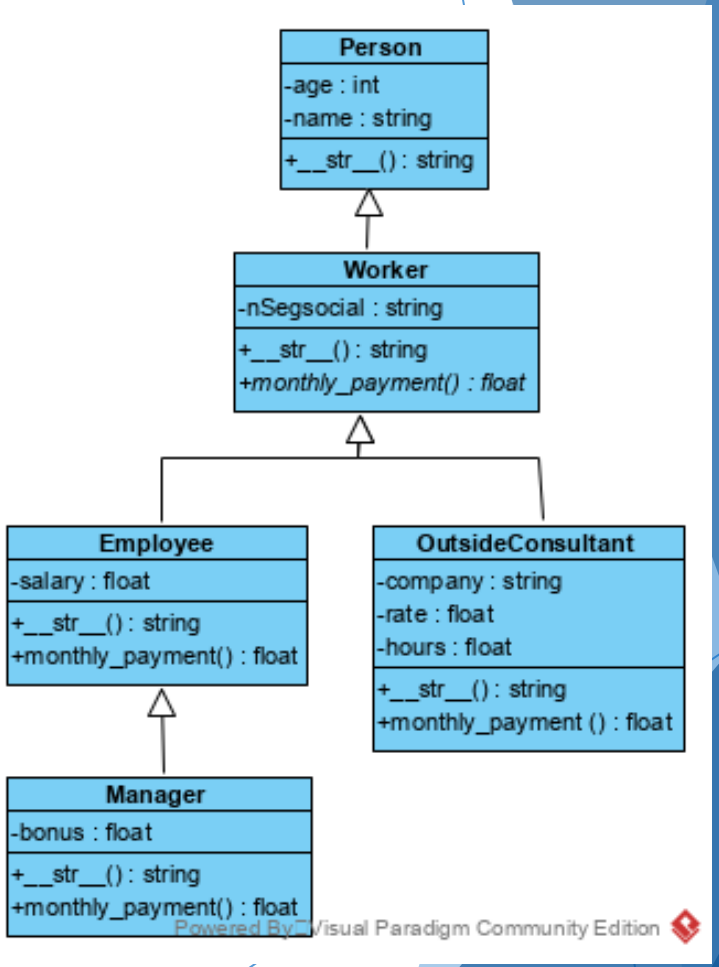
- La clase Alumno debe ofrecer los siguientes métodos:
  - Obtener las asignaturas en las que está matriculado, *get_subjects(): str*
    - El alumno libre tiene una lista de asignaturas en las que está matriculado.
    - El alumno presencial está matriculado de todas las asignaturas de un curso.
  - El pago mensual, *monthly_rate(): float*
    - El alumno libre paga mensualmente por las horas de clase diarias recibidas de las asignaturas en las que está matriculado. Se supone que hay 20 días hábiles por mes.
    - El alumno presencial paga el precio de la matrícula más un precio fijo por convocatoria (*overrun_per_exam_call*) multiplicado por el número de la convocatoria, todo ello dividido por los 12 meses del año.

Herencia: ejemplo con alumnos y profesores



Herencia: Ejercicio Empresa con clases abstractas

- Siguiendo con el ejercicio de la empresa:
  - La clase Worker tendrá un método abstracto *monthly_payment()*.
  - En la clase Employee el salario mensual se calcula dividiendo el salario entre 14.
  - En la clase Manager el sueldo se calcula dividiendo el salario entre 14 y sumando el bonus.
  - En la clase OutSideConsultant el sueldo se calcula a partir de la tarifa por horas (rate) por las horas trabajadas en el periodo facturable (hours).
- Implementar un programa principal que:
  - Que cree una colección de trabajadores que contendrá: 1 directivo, 2 empleados y 1 consultor externo.
  - Escribir una función que retorne el coste mensual de todos los trabajadores de la colección.
    - *calculate_total_payments(workers: List[Worker]) -> float*

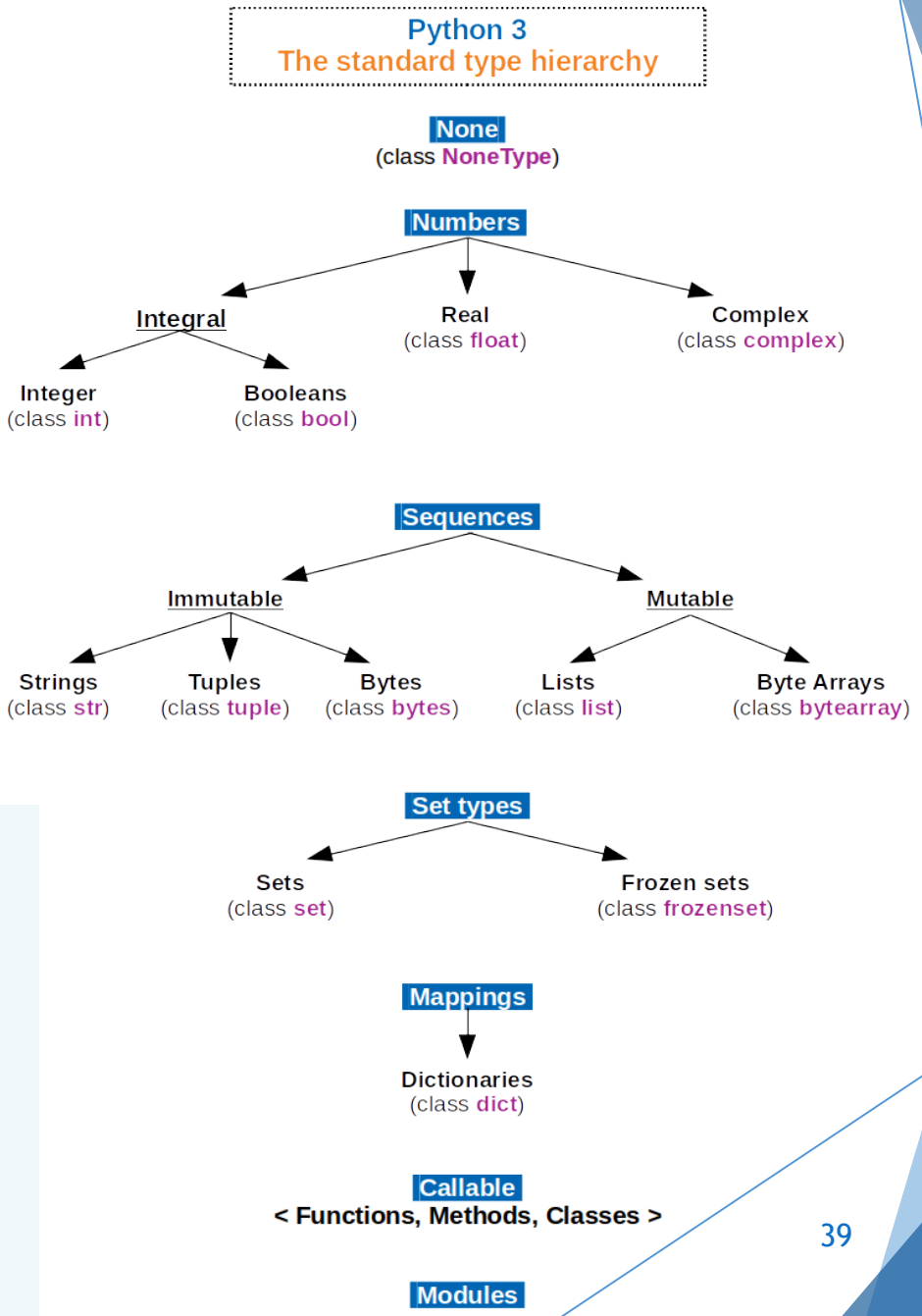


Herencia: Ventajas

- Reutilizar código:
  - Podemos crear nuevas clases hijas a partir del código de una clase padre.
  - Esto es habitual al utilizar una librería.
- El polimorfismo permite escribir un mismo código que sirva para todas las subclases de objetos de una clase padre sin tener que distinguir casos (if … elif).
  - Por ejemplo: el método *calculate_total_payments(workers: List[Worker]) -> float*
- Y si en el futuro añadimos nuevas subclases, el código seguirá siendo válido, ya que las nuevas subclases tendrán que implementar todos los métodos declarados en la clase padre.
  - Por ejemplo: si añadimos una nueva subclase de *Worker*, *FreeLance*, tendrá que incluir una implementación para *monthly_payment()*.

Herencia: jerarquía de tipos predefinidos en Python

- Para la mayoría de los supertipos (con fondo azul y subrayados) existen clases abstractas predefinidas en python.
  - Numbers -> Number
  - Sequences -> Secuence[T]



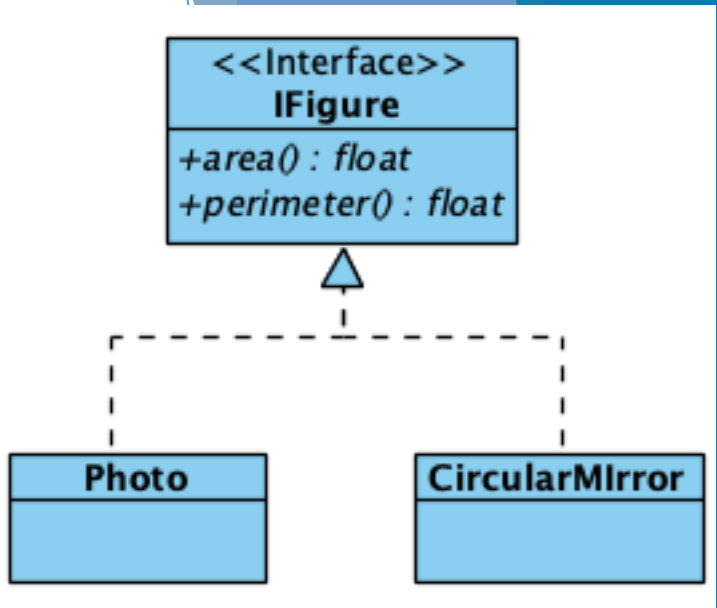

```
def es_capicua(s: Sequence[T]):
    i: int = 0
    ok: bool = True
    medio: int = int(len(s)/2)
    while i < medio and ok:
        ok = s[i] == s[len(s) - i - 1]
        i += 1
    return ok

print(es_capicua("jdksd"))
print(es_capicua([1, 2, 2, 1]))
print(es_capicua(('a', 'b', 'a')))
```

Interfaces

Interfaces: motivación

- Podría suceder que varias clases no necesariamente relacionadas compartan un mismo conjunto de operaciones.
  - La interfaz permite definir ese conjunto de operaciones
  - No implica relación semántica entre la interfaz y la clase que la implementa
  - La interfaz impone ‘un contrato’ que se debe cumplir
  - Si la clase que implementa la interfaz no cumple todo el contrato, la clase será abstracta
- La manera de realizar las operaciones de la interfaz dependerá de cada clase que la implemente.
- Una interface se implementará en Python como una **clase abstracta con todos sus métodos abstractos y sin atributos**.
  - Las clases que implementen la interface A, serán clases hijas de la clase abstracta A.

Interfaces: ejemplo



- Las clases Parcel, Photo, Painting, CircularMirror,... incluyen los métodos area(), perimeter(), etc.
- Vamos a definir una interfaz IFigure con esos dos métodos.

```
class IFigure(ABC):
    @abstractmethod
    def area(self) -> float:
        pass

    @abstractmethod
    def perimeter(self) -> float:
        pass
```

```
class Photo(IFigure):

    def __init__(self, side1: float, side2: float):
        self.__side1 = side1
        self.__side2 = side2

    def area(self) -> float:
        return self.__side1 * self.__side2

    def perimeter(self) -> float:
        return  2 * (self.__side1 + self.__side2)
```

Interfaces en Python

- El paquete *typing* (desde 3.8) incorpora un mecanismo denominado protocolo para definir una interface, pero en este curso no lo vamos a utilizar.
  - No nos va a limitar, porque en esencia es lo mismo que una clase abstracta sin atributos y con todos los métodos abstractos.
- Ejemplos de interfaces predefinidas en Python:
  - Sized: __len__()
  - Container: __contains__() (operador in)

Excepciones

Definición

- Representan situaciones especiales que provocan errores en tiempo de ejecución.
  - Los puede lanzar el intérprete de Python:
    - IndexError se lanza si l = [1, 2] y l[4]
    - TypeError se lanza si print([1, 2] + 4)
  - Los puede lanzar un programa mediante un **raise**, por ejemplo, cuando se viola una PRE.
    - Intentamos crear un worker con age < 0

Tratamiento de las excepciones

- Cuando se lanza una excepción en un programa pueden darse dos casos:
  1. Si el programa no tiene definido código para tratar la excepción, se interrumpe bruscamente la ejecución del programa.
     - En la consola se indica el error y la línea del raise
  2. Se trata la excepción (sentencia **try**)
     a) Si el problema tiene arreglo, se soluciona el problema y se permite que prosiga el programa.
     b) Si no, se termina el programa de una manera “suave”.

Excepciones: Ejemplo 1

Entre el try y el except incluimos el código que puede lanzar la excepción a tratar

```
def pedir_edad() -> int:
    s: str = input("dame una edad: ")
    if not s.isnumeric():
        raise ValueError("La edad dada no es un número natural")
    return int(s)


seguir: bool = True
edad: int = 0
while seguir:
    try:
        edad = pedir_edad()
        seguir = False # solo se ejecuta si la edad es válida
    except ValueError as e:
        print(e, ", prueba otra vez")

print("Naciste en el año", 2021 - edad)
```

Si se lanza una excepción, la ejecución sigue en la rama **except** prevista para esa excepción.

Excepciones

- Es posible definir nuevos tipos de excepciones en un programa.
- Se considera una buena práctica de programación definir excepciones más específicas, que permitan un tratamiento más especializado.
- Todos los tipos de excepciones se definen mediante clases que heredan de la clase **Exception**.

```
class EdadIrrealException (Exception):
    """
    Define the exception that will be raised when
    the age is greater than 200
    """
    pass
```

Excepciones: Ejemplo 2

```
def pedir_edad() -> int:
    s: str = input("dame una edad: ")
    if not s.isnumeric():
        raise ValueError("La edad dada no es un número natural")
    if int(s) > 200:
        raise EdadIrrealException
    return int(s)


seguir: bool = True
edad: int = 0
while seguir:
    try:
        edad = pedir_edad()
        seguir = False # solo se ejecuta si la edad es válida
        print("Naciste en el año", 2021 - edad)
    except ValueError as e:
        print(e, ", prueba otra vez")
    except EdadIrrealException:
        print("Tienes que ser un árbol, mejor lo dejamos")
        seguir = False  # forzamos la salida del bucle

print("FIN")
```