Redefinición de métodos

- Llamamos redefinición a la acción de definir un método con el mismo nombre en distintas clases.
- Gracias a una característica de POO llamada polimorfismo, al redefinir métodos, se ejecuta en cada invocación, al método correcto.
- Ejemplos de uso:

  1) El método __str__ (siempre se invoca el método definido en la clase correspondiente. Si no está definido, busca el más cercano en la jerarquía de objetos de Python. Esto hace que cuando hacemos, *print(cualquier_cosa)* siempre se va responder algo.

  Si el tipo de *cualquier_cosa* tiene definido un __str__ se llamará a ese método, si no, se llamará a uno genérico.

Redefinición de métodos (2)

2) Otro ejemplo:

cuando se recorre una secuencia (lista, tupla, archivo, etc) mediante una misma estructura de código. Esto es posible gracias a la redefinición de un método especial **__iter__**

- Vamos a crear una clase que represente una colección y luego agregar un iterador para poder recorrer sus objetos usando nuestro iterador particular.

Iteradores

```
for elemento in [1, 2, 3]:
    print(elemento)
for elemento in (1, 2, 3):
    print(elemento)
for clave in {'uno':1, 'dos':2}:
    print(clave)
for caracter in "123":
    print(caracter)
for linea in open("miarchivo.txt"):
    print(linea, end='')
```

Iteradores (2)

- ¿Cómo funciona?
  - la sentencia for llama a **__iter()__** en el objeto contenedor.
  - La función devuelve un objeto iterador que define el método **__next__()**
  - Este método accede a los elementos en el contenedor de a uno por vez.
  - Cuando no hay más elementos, **__next__()** levanta una excepción **StopIteration** que le avisa al bucle del for que hay que terminar.

Iteradores (3)

```
>>> s = 'abc'
>>> it = iter(s)
>>> it
<iterator object at 0x00A1DB50>
>>> next(it)
'a'
>>> next(it)
'b'
>>> next(it)
'c'
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
    next(it)
StopIteration
```

Redefiniendo __iter__ en nuestra clase

```
>>> class Reversa:
...     """Iterador para recorrer una secuencia marcha atrás."""
...     def __init__(self, datos):
...         self.datos = datos
...         self.indice = len(datos)
...     def __iter__(self):
...         return self
...     def __next__(self):
...         if self.indice == 0:
...             raise StopIteration
...         self.indice = self.indice - 1
...         return self.datos[self.indice]
...
>>> rev = Reversa('spam')
>>> iter(rev)
<__main__.Reversa object at 0x00A1DB50>
>>> for char in rev:
...     print(char)
...
m
a
p
s
```

Herencia

- La herencia es un mecanismo de la programación orientada a objetos que sirve para crear clases nuevas a partir de clases preexistentes.
- Se toman (heredan) atributos y comportamientos de las clases padres y se los modifica para modelar una nueva situación.
- Además se pueden agregar nuevos atributos y comportamientos correspondientes únicamente a los hijos
- La clase padre se llama **clase base** y la que se construye a partir de ella es una **clase derivada** (o hija).
- Para indicar que una clase hereda de otra se coloca el nombre de la clase de la que se hereda entre paréntesis después del nombre de la clase hija.

Ejemplo

- Persona es la clase base (o madre). No hereda de nadie, que es lo mismo que decir que hereda de object
- Alumno, es la clase derivada de Persona. O sea, Alumno es una Persona, tiene todos sus atributos y métodos, además de los propios

```
class Persona(object):
    def __init__(self, cedula, nombre, apellido):
        self.cedula = cedula
        self.nombre = nombre
        self.apellido = apellido

    def __str__(self):
        return str("persona " +  self.apellido + ", " + self.nombre)


class Alumno(Persona):
    def __init__(self, cedula, nombre, apellido, matricula):
        self.matricula = matricula
        # hay que llamar al constructor de la clase base
        Persona.__init__(self, cedula, nombre, apellido)
```

Ejemplo (2)

Para crear un objeto de la clase Alumno, se pasan en el constructor todos los datos que se necesitan para crear una Persona (ci, nombre, apellido) y se agregan los específicos de Alumno (en este caso, matrícula)

a = Alumno("4234987", "Rodrigo", "Perez", "98765")

print (a) → persona Perez, Rodrigo

Vemos que se heredó el método __str__ de la clase base

Podemos agregar el método __str__ a Alumno para que se imprima también con la matrícula

```
def __str__(self):
    return Persona.__str__(self) + " - matricula: " +  self.matricula
```

Herencia múltiple

- En Python, se permite la herencia múltiple, es decir, una clase puede heredar de varias clases a la vez.
- Para heredar de varias clases basta con enumerar las clases de las que se hereda separándolas por comas:
- Por ejemplo :

```
class Cocodrilo(Terrestre, Acuatico):
  pass
```

Algo sobre decorators

- A veces necesitamos modificar varias funciones o métdos para que se comporten de la misma manera. Por ejemplo, podríamos querer realizar alguna tarea particular antes y después de la ejecución de cada función. Formatear la salida de cierta manera, escribir en un archivo de log, etc.
- Podemos modificar el código de nuestras funciones, pero existe una opción diferente para no modificar ese código y hacer que esa tarea sea reutilizable: definir un **decorador**.
- Con un **decorador** podemos agregar comportamiento adicional a cualquier función, sin necesidad de modificar propiamente el código de la función.

Algo sobre decorators (2)

- Entonces, un decorador es una función, que recibe como parámetro otra función – la función original a decorar- y le agrega el comportamiento deseado.
- El decorador debe ser capaz de recibir cualquier función (porque justamente, queremos que sea de uso genérico).
- Vamos a definir a continuación un decorador (llamado log) para escribir en un archivo cada invocación a la función detallando los datos recibidos en los parámetros.

Ejemplo: decorator con log

```
# nuestro decorador (se llama log) y recibe la función original
def log(original_function):
    def new_function(*args):
        with open("log.txt", "a") as logfile:
            nombre_funcion = original_function.__name__
            logfile.write(f"Funcion {nombre_funcion} llamada con {args}\n")
        return original_function(*args)

    return new_function

# Anotamos nuestra función con el decorador
@log
def my_function(message):
    print(message)

@log
def my_function2(message, name):
    print(message)

my_function("pepe")
my_function2("pepe", "maria")
```

Ejemplo: decorator con log (2)

- Nuestra función decorator (log) modifica la función original pasada como parámetro.
- El uso de una función interna (new_function) permite agregarle la escritura en el log, y luego realizar la invocación a la función original `return original_function(*args)` de modo de mantener el mismo comportamiento que la función original
- El uso de *args permite que el decorador sea generalizable (investigar el posible uso de **kwargs. Parámetros posicionales, args, parámetros con nombre, kwargs)
- Veamos otro ejemplo: un decorator sencillo, que haga algo antes y después de la invocación a la función original

Ejemplo: decorator sencillo

```
def our_decorator(func):
    def function_wrapper(x):
        print("Before calling " + func.__name__)
        func(x)
        print("After calling " + func.__name__)

    return function_wrapper

@our_decorator
def foo(x):
    print("Hi, foo has been called with " + str(x))
foo("Hi")
```

- ¿ Por qué este decorator es menos genérico que el que definimos primero?
- ¿Para qué tipo de funciones sirve?

Decorators con parámetros

- Queremos indicar por parámetro en qué archivo de log escribir:

```
@log("someotherfilename.txt")

def my_function(message):

    print(message)
```

- Es necesario agregar un nivel de anidamiento más:

```
# nuestro decorador (se llama log2) y recibe el parámetro
def log2(log_file="log.txt"):
    def wrap(original_function):
        def wrapped_funcion(*args):
            with open(log_file, "a") as logfile:
                nombre_funcion = original_function.__name__
                logfile.write(f"Funcion {nombre_funcion} llamada con {args}\n")
            return original_function(*args)
        return wrapped_funcion
    return wrap
```

- Cuando se quiere usar el decorator sin parámetros, hay que llamarlo así: @log()

Decoradores para clases

- Se puede decorar funciones usando clases (*https://python-3-patterns-idioms-test.readthedocs.io/en/latest/PythonDecorators.html*
- Python y todos los paquetes que podemos agregar (con pip) suelen tener ya implementados una serie de decoradores que podemos usar.
- En particular, a la hora de definir métodos de clases contamos con *@classmethod* y *@staticmethod*
- Un método estático no lleva como parámetro el objeto implícito. Por lo tanto no tiene acceso a los atributos de la instancia (en cambio, tiene acceso a los atributos estáticos).

```
class Robot:
    __counter = 0
    def __init__(self):
        type(self).__counter += 1
    @staticmethod
    def robotInstances():
        return Robot.__counter
```

```
print(Robot.robotInstances())
x = Robot()
print(x.robotInstances())
y = Robot()
print(x.robotInstances())
print(robot.RobotInstances())
0
1
2
2
```

Métodos de clase y estáticos

- Los métodos de clase no tienen referencia la instancia (igual que los métodos estáticos), en cambio, sí están asociados a la clase.
- El primer parámetro del método es una referencia a la clase (así como *self* hace referencia a la instancia para los métodos de instancia, *cls* hace referencia a la clase para los métodos de clase)

```
class Pet:
    _class_info = "pet animals"

    @classmethod
    def about(cls):
        print("This class is about " + cls._class_info + "!")


class Dog(Pet):
    _class_info = "man's best friends"

class Cat(Pet):
    _class_info = "all kinds of cats"

Pet.about()
Dog.about()
Cat.about()
```

Salida:

```
This class is about pet animals!
This class is about man's best friends!
This class is about all kinds of cats!
```

Decoradores para clases (2)

```
class Person:
    TITLES = ('Dr', 'Mr', 'Mrs', 'Ms')

    def __init__(self, name, surname):
        self.name = name
        self.surname = surname

    def fullname(self): # instance method
        # instance object accessible through self
        return self.name + self.surname


    @classmethod
    def allowed_titles_starting_with(cls, startswith):
        # class accessible through cls
        return [t for t in cls.TITLES if t.startswith(startswith)]

    @staticmethod
    def allowed_titles_ending_with(endswith): # static method
        # no parameter for class
        # we have to use Person directly
        return [t for t in Person.TITLES if t.endswith(endswith)]
```

Decoradores para clases (3)

El decorador @property permite generar al vuelo un atributo, a partir del uso de otros atributos

```
class Person:
    def __init__(self, name, surname):
        self.name = name
        self.surname = surname

    @property
    def fullname(self):
        return "%s %s" % (self.name, self.surname)

jane = Person("Jane", "Smith")
print(jane.fullname) # no brackets!
```

Ejercicios

Ejercicio 1

a) Escribir una clase Personaje que contenga los atributos vida, posición (es un número, representa la posición sobre un eje) y velocidad, y los métodos recibir_ataque, que reduzca la vida según una cantidad recibida y lance una excepción si la vida pasa a ser menor o igual que cero, y mover que reciba una dirección y se mueva en esa dirección la cantidad indicada por velocidad.

b) Escribir una clase Enemigo que herede de Personaje, y agregue el atributo ataque y el método atacar. Este método recibe otro personaje y lo daña según la cantidad indicada por el atributo ataque (le resta vidas).

Las clases Enemigo y Personaje deben estar en diferentes módulos y las instanciaciones y pruebas, en un tercer módulo.