Ayudantía 4

Cosas que reforzar

Múltiples temas (yay :D)

¿Qué veremos?

Funcional

Más estos en el orden de prioridad que ustedes elijan:

- Modelación y concepto de objetos
- Polimorfismo, Duck Typing, Overloading y Overriding
- Estructuras de Datos (AC04 y T02)

Sobre las issues

¡Busquen siempre primero!

– – –

- Recuerden tips para buscar en Google (ver Ayudantía 2):

  python3 + clase + librería + problema (en inglés)

- En la búsqueda de un error, reemplazar nombres de clases/variables particulares de su código por *
- ¡La issues son para preguntas de última instancia! No siempre tendrán a un equipo de ayudantes deseoso por responder.

Funcional

¿Qué es programación funcional? ¿Por qué funcional?

Un enfoque, una forma de pensar… ¡En que es natural que funciones sean el argumento recibido por otras funciones!

filter

generadores

iteradores

zip

enumerate

Estructuras por comprensión

lambda

reversed

reduce

funciones generadoras

map

iterables

Conceptos clave: iterable vs. iterador

- Un **iterable** es un objeto que tiene definido el método `__iter__`. Se puede iterar sobre estos de la forma:

```
for elemento in iterable:
```

- Un **iterador** es un objeto retornado por `iter`. Los iteradores son aquellos que iteran sobre un iterable. Se puede acceder al siguiente elemento vía el built-in `next`:

```
>>> iterable = ['a', 'b', 'c', 'd']
>>> iterador = iter(iterable)
>>> next(iterador)
'a'
```

¡Se pone rara la cosa!

- Típicamente los iteradores también tienen definido el método `__iter__`, que los retorna a ellos mismos.
- Entonces, los iteradores son, a su vez, iterables, y quien itera sobre un iterador... ¡es el mismo iterador!

```
class Iterador:

    def __iter__(self):

        return self

    def __next__(self):

        return random.random()
```

Conceptos clave: generador vs. función generadora

- Un **generador** es un iterador que permite iterar sobre una secuencia de datos **sin almacenarlos en una estructura.**

```
generador = (line for line in open('archivo.txt', 'r'))
```

- Una **función generadora** es una función que retorna un generador. Cuando se llama a `next` del generador resultante, se ejecuta el código en la función hasta que se encuentre un `yield`

Ejemplo de uso

```
>>> def funcion_generadora():
        print('Me pidieron el primer elemento!')
        while True:
            yield random.random()
>>> generador = funcion_generadora()
>>> next(generador)
Me pidieron el primer elemento!
0.24770746389601173
>>> for i in generador:
        print(i)  # qué hará esto?
```

Funciones generadoras y método send

```
>>> def promedio_movil():
        total_acumulado = float((yield))
        cantidad_numeros = 1
        while True:
            nuevo = yield total_acumulado / cantidad_numeros
            cantidad_numeros += 1
            total_acumulado += nuevo
>>> pm = promedio_movil(); next(pm);
>>> for i in range(10): pm.send(i)
```

lambda, map y filter

```
>>> suma2 = lambda x, y: x + y    # creada “on the fly”
>>> iterador = map(suma2, [‘a’, ‘b’, ‘c’], (‘0’, ‘1’, ‘2’))
>>> next(iterador)
‘a0’
>>> filtrado = filter(lambda s: s[0] != ‘c’, iterador)
>>> next(filtrado)
‘b1’
>>> next(filtrado)      # qué mostrará esto?
```

¿Y si incorporamos reduce?

```
>>> for _ in map(
        print,
        filter(
            lambda y: y % 2 == 0,
            map(
                lambda x: reduce(
                    lambda x, y: y-x,
                    range(x)),
                range(1, 10)
            )
        )
    ): pass
```

Programación Orientada a Objetos (OOP)

¿Cuáles de los siguientes son objetos?

`Integer`

`Float`

`Function`

`String`

`List`

`Dictionary`

`Class`

`Tuple`

En Python (casi)
TODO es un objeto.

Por ejemplo

```
def factorial(n):
    if n <= 0:
        return 1
    return n*factorial(n-1)
```

```
funcion = factorial

resultado = funcion(5)
```

```
class Gato(Animal):
    def __init__(self, edad):
        self.edad = edad
    def maullar(self):
        print(“Miau”)
```

```
clase = Gato

un_gato_viejo = clase(15)
```

Algunos objetos tienen un método __call__

Dichos objetos se dicen **callable** o “llamables”. Ejemplos:

```
>>> objeto = Clase()
```

Retorna el resultado de Clase.__call__() y lo asigna a objeto. Cuando se instancia un objeto de una clase, ¡lo que en realidad se hace es llamar al método call de la clase!

```
>>> objeto()
```

Si se definió el método __call__ para las instancias de la clase, se ejecuta así.

```
>>> funcion()
```

¡__call__ ya está definido en las instancias de funcion!

¡MALAS PRÁCTICAS!

```
---

if 'Triatleta' in str(type(self)):

    ...


def agregar_nodo(self, valor, prioridad):

    ...
```

Ventaja de que todo sea un objeto

```
if button == “login”:
    login()
elif button == “menu”:
    display_options()
elif button == “logout”:
    logout()
...
```

Ventaja de que todo sea un objeto: alternativa a cases

```
options = {“login”: login,
           “menu”: display_options,
           “logout”: logout}
options[button]()
```

Estructuras de datos

¿Qué es una EDD?

Forma de organizar y almacenar datos para poder trabajarlos **eficientemente.**

Algunas estructuras de datos de Python:

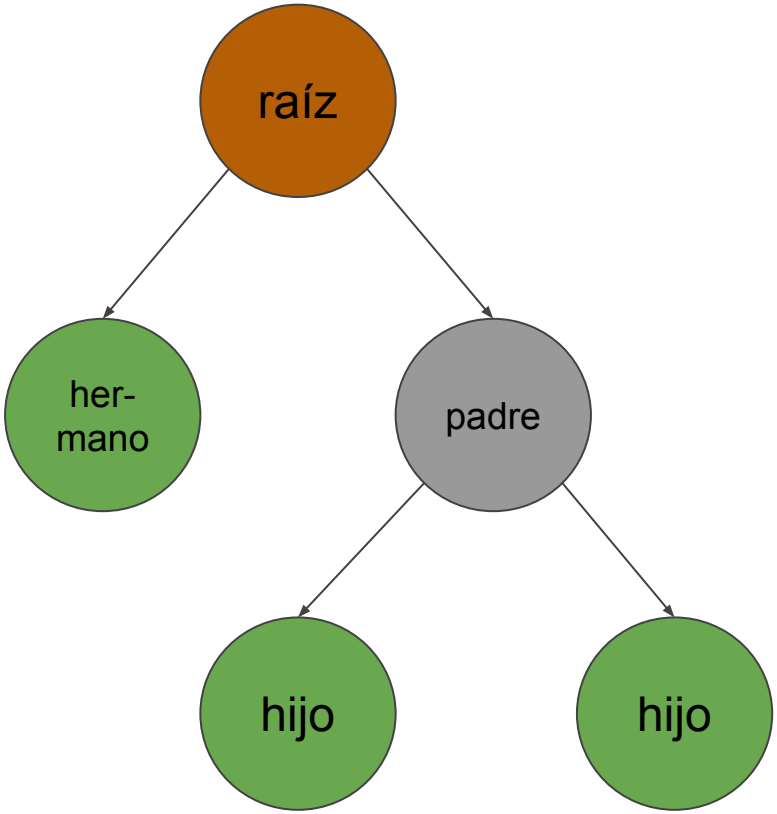
- Listas (stacks/pilas)
- Colas (deque)
- Conjuntos (sets)
- Diccionarios
- Tuplas

¡Prohibidas en la Tarea 2!

Grafos (Revisar Ayudantía 3)

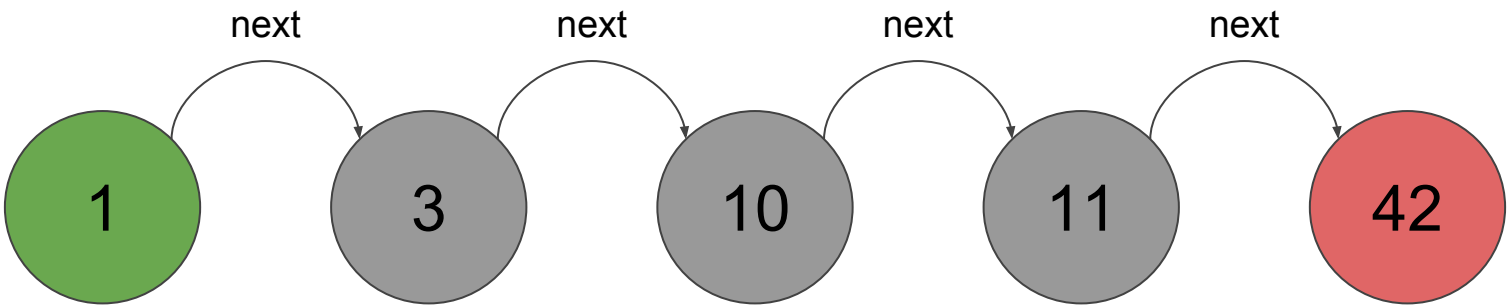
- **G(V,E)**: Par ordenado de conjuntos de Nodos (Vértices) y Arcos (Aristas).
- Existen muchos algoritmos muy estudiados para trabajar sobre ellos:
  - Búsqueda en amplitud
  - Búsqueda en profundidad
  - Dijkstra (ruta mínima)
  - Ford-Fulkerson (flujo máximo)
- Un **árbol** es un grafo particular que no tiene ciclos.

Árbol binario



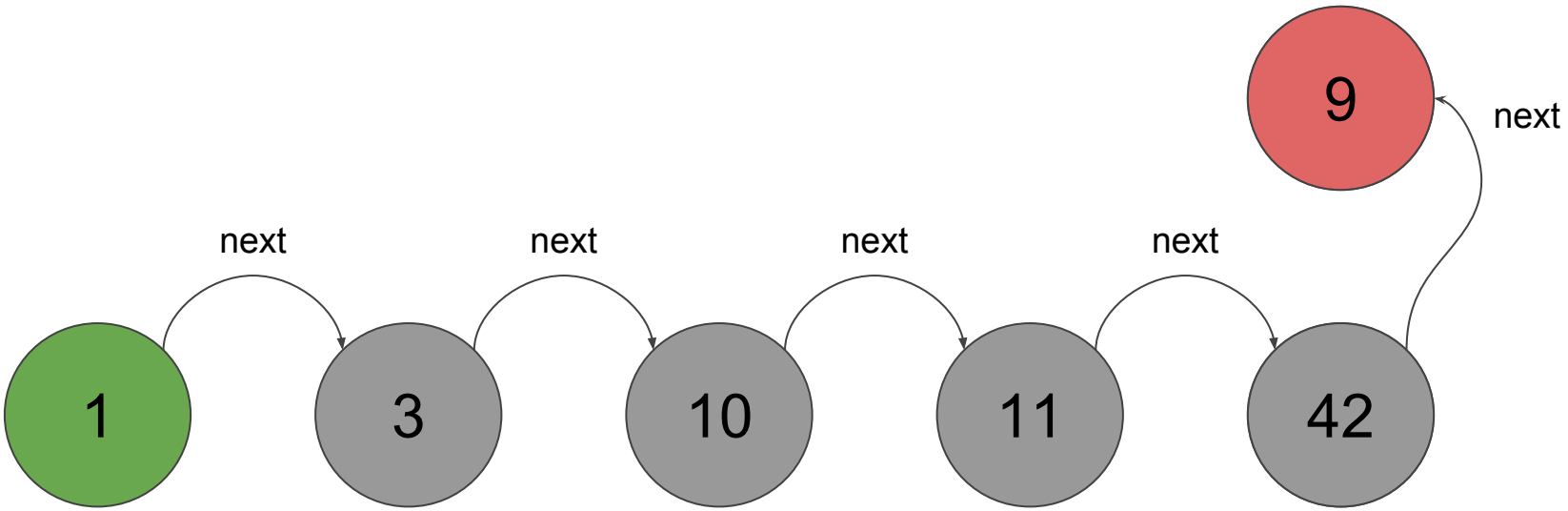
Lista Ligada

```
class Nodo:

    def __init__(self, valor):

        self.valor = valor

        self.next = None
```

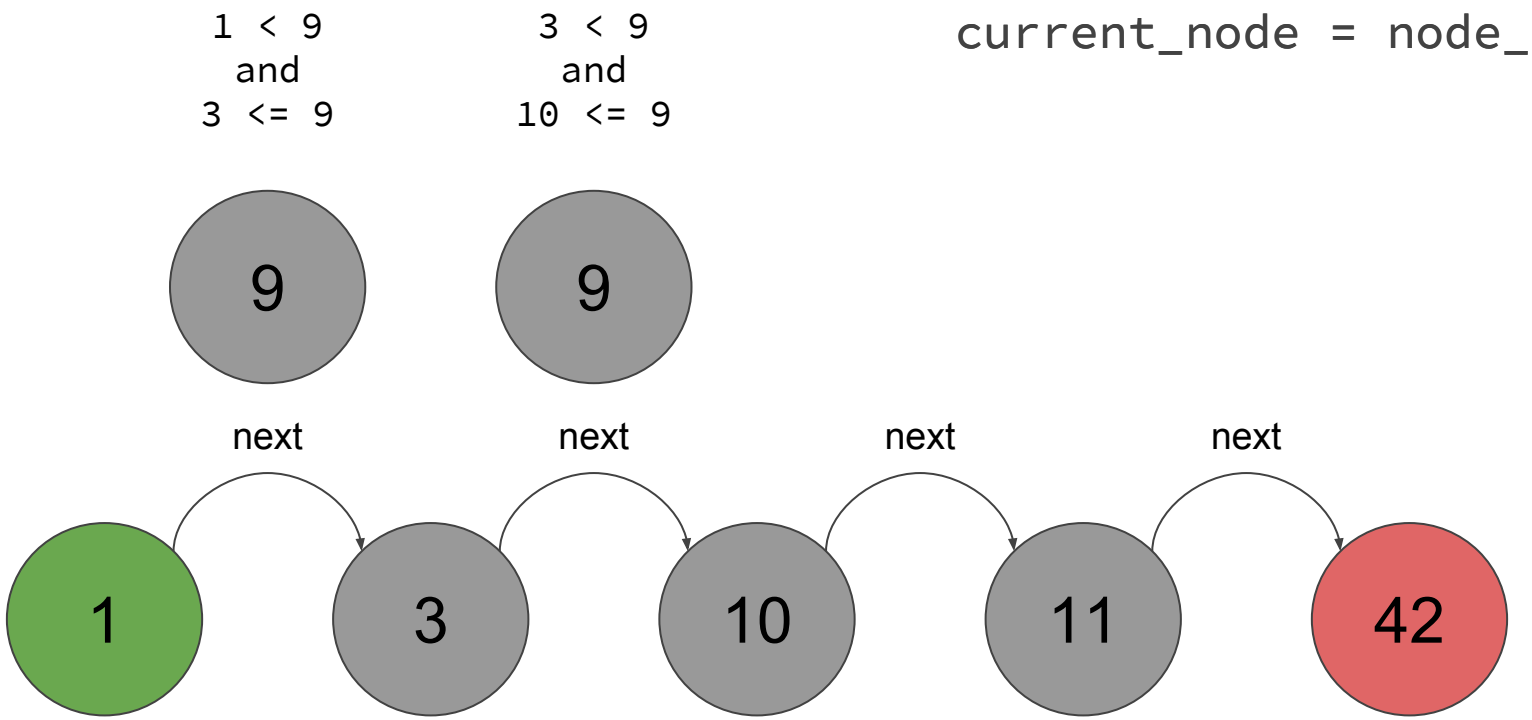


Lista Ligada: cómo agregar un nodo

```
self.tail = new_node
```

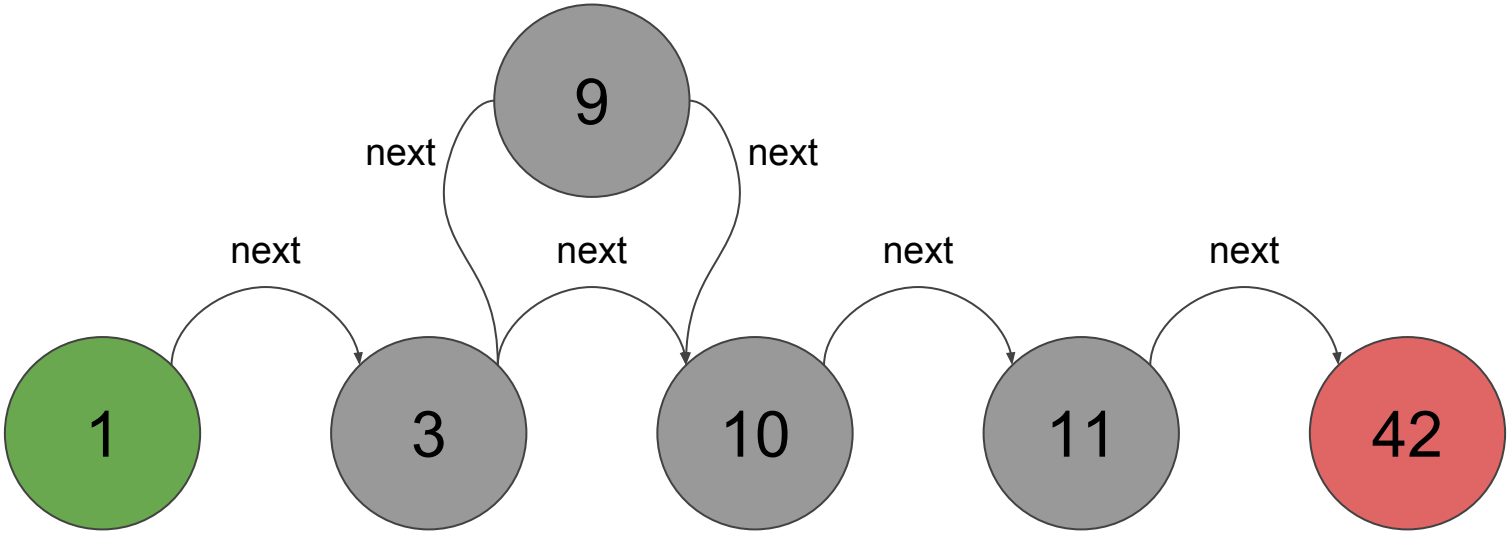


Lista Ligada: cómo insertar un nodo



Una forma es recorrer la lista manteniendo el orden

Lista Ligada: cómo insertar un nodo




```
aux = node_3.next
```

Duck Typing, Polimorfismo, Overloading y Overriding

Python es especial

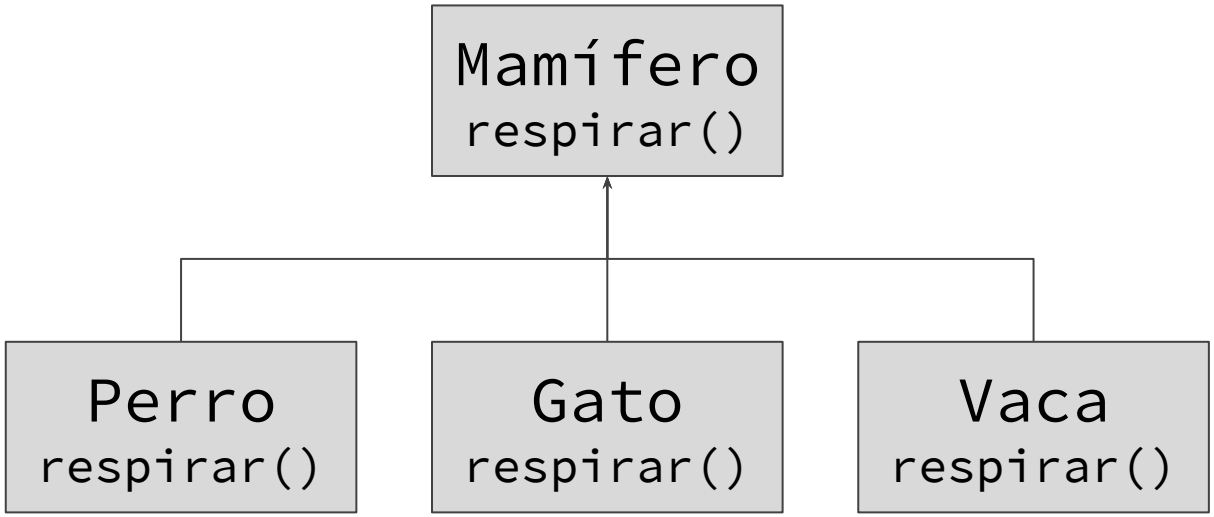
- Dinámicamente tipado

```
>>> a = 5

>>> a = “Hola Mundo!”
```

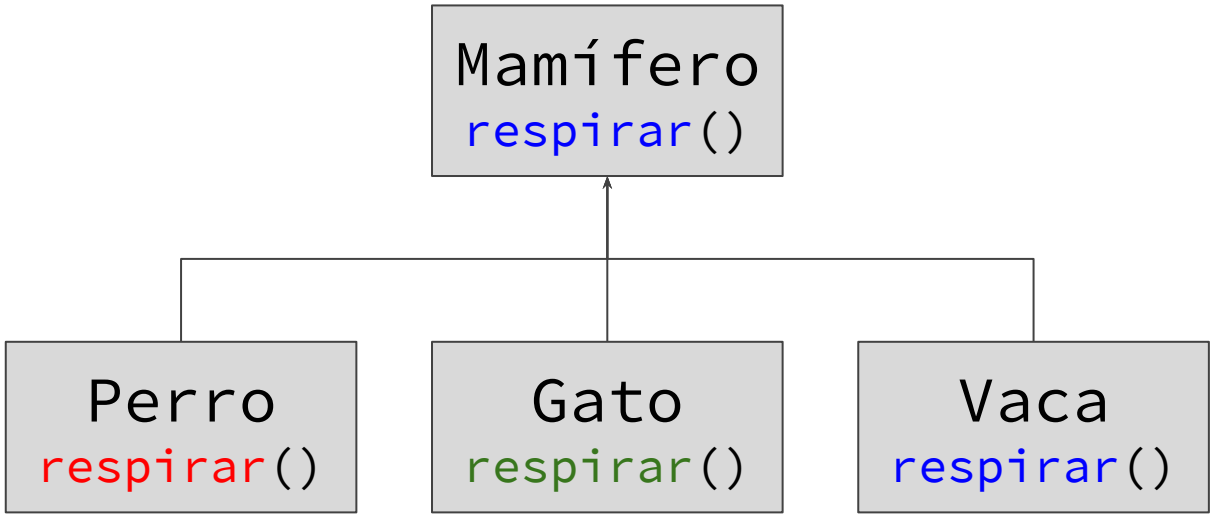
- Débilmente tipado: revisa los tipos durante la interpretación

```
def funcion(argumento):

    print(argumento.__class__.__name__)
```

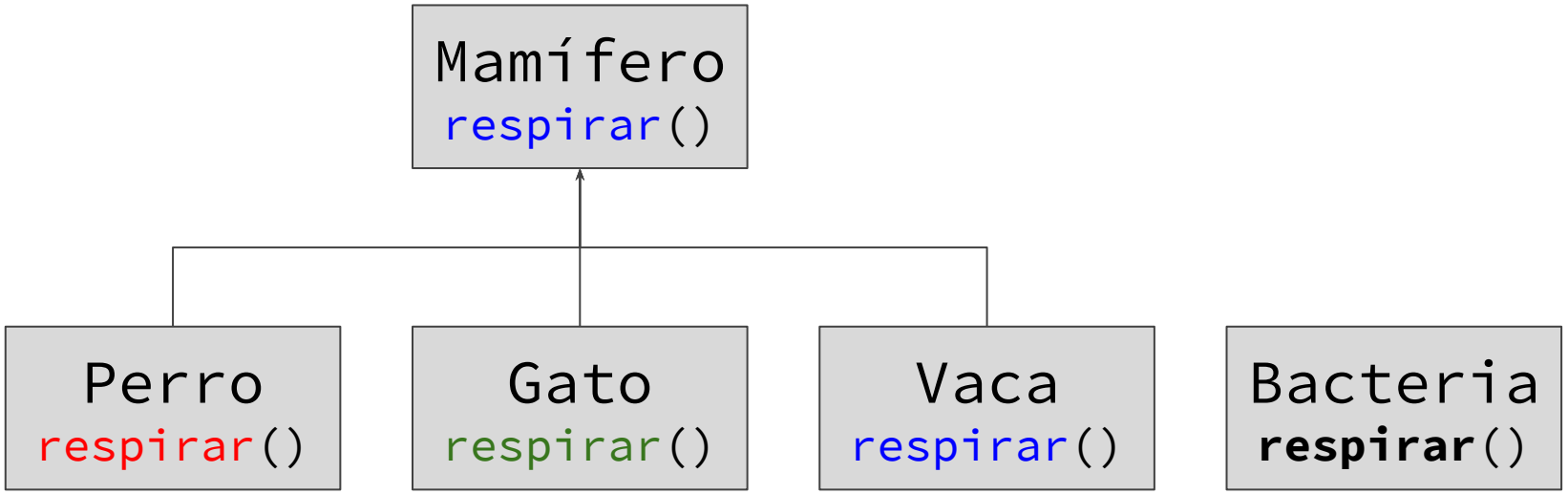
Polimorfismo



Overriding


Mamífero
respirar()
Perro
respirar()
Gato
respirar()
Vaca
respirar()

Duck Typing


Mamífero
respirar()
Perro
respirar()
Gato
respirar()
Vaca
respirar()
Bacteria
respirar()

Overloading

Mismo método varias veces, pero que se diferencian por:

- tipos de argumentos
- cantidad de argumentos

En Python no existe, pero se simula así:

```
def funcion(argumento, i=0, j=None, k=False):
    pass
```