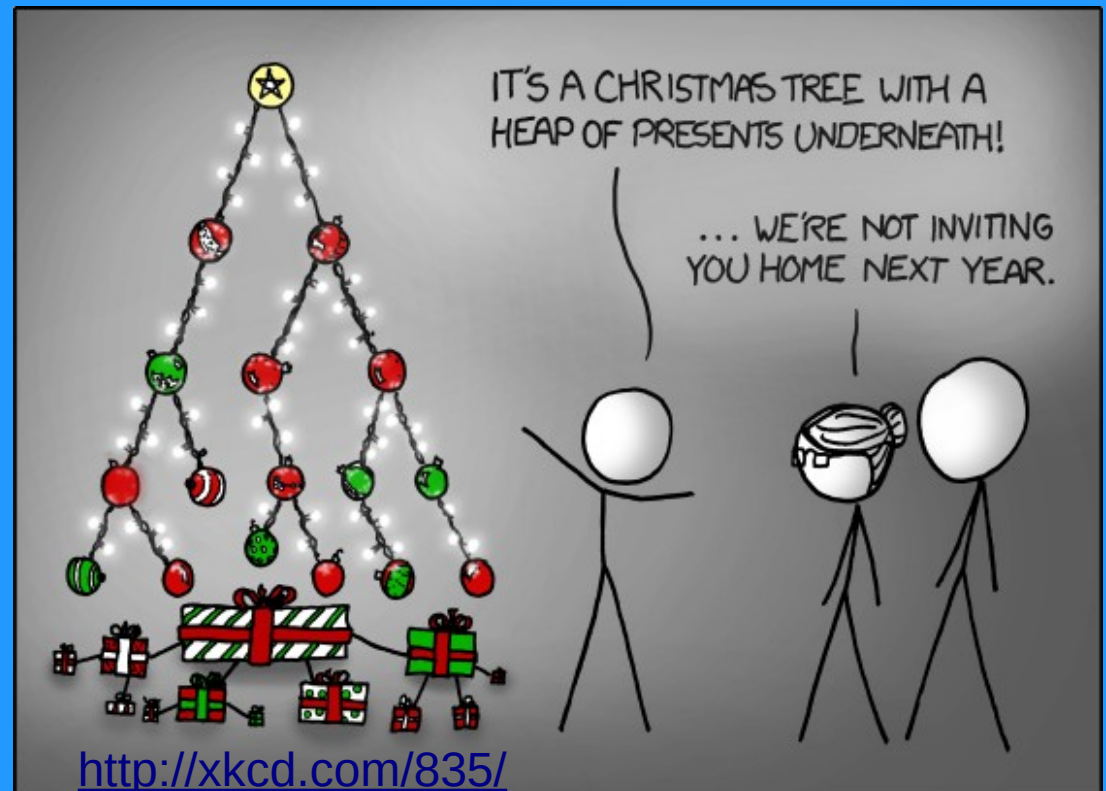


# Estructuras de Datos

## 2020



**Al especificar la lista  
doble...**

# Especificación. Lista Doble

## Otras Operaciones

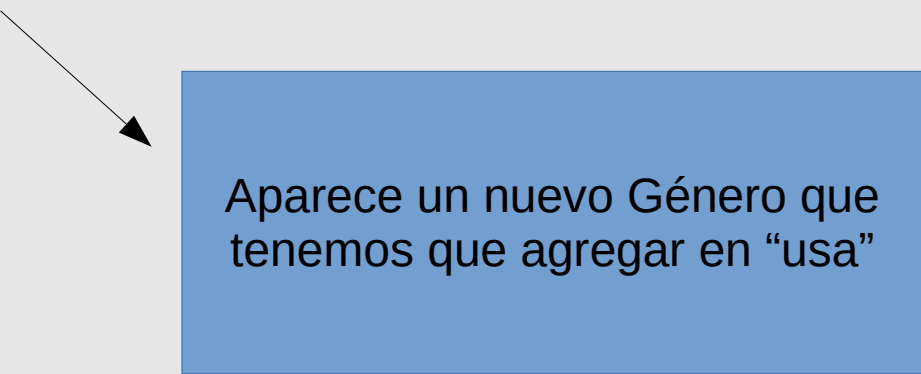
`insertar(ListaDoble<a>, coordenada<ListaDoble<a>>, a )→None`

`{Pre: la coordenada es válida}`

`{Pos: la Lista Doble no esta vacía}`

`borrar(ListaDoble<a>, coordenada<listaDoble<a>>)→  
Coordenada<listaDoble<a>>`

`{Pre: la coordenada es válida}`



Aparece un nuevo Género que  
tenemos que agregar en "usa"

# Especificación. Lista Doble

**TAD** ListaDoble <a>

**Igualdad Observacional**

Si **a** y **b** son dos listas dobles

**a** es igual a **b** si se cumple que: Las longitudes de **a** y **b** son iguales **Y** cada elemento en **a** es igual al correspondiente elemento en **b**.

**Usa**

Natural, Bool, Secuencia<a>, None, **coordenada<ListaDoble<a>>**

**Parámetro Formal** a

**Géneros** ListaDoble<a>

# Especificación. Lista Doble

## Otras Operaciones

**inicio(ListaDoble<a>) → coordenada<ListaDoble<a>>**

**fin(ListaDoble<a>) → coordenada<ListaDoble<a>>**

El ¿por qué?

Lo vemos ahora...

# **Iteradores y Coordenadas o Posiciones**

# TAD. Coordenada

En forma genérica, una **Coordenada** es un **objeto** que permite **hacer referencia** a cada uno de los **elementos** almacenados dentro un **contenedor** o, en forma especial, **indicar** que no está haciendo referencia **a ninguno de ellos**.

**Por ejemplo todos los contenedores** de C++ pueden devolver iteradores mediante **begin()** y **end()**.

A la Coordenada C++ la llama Iterador.

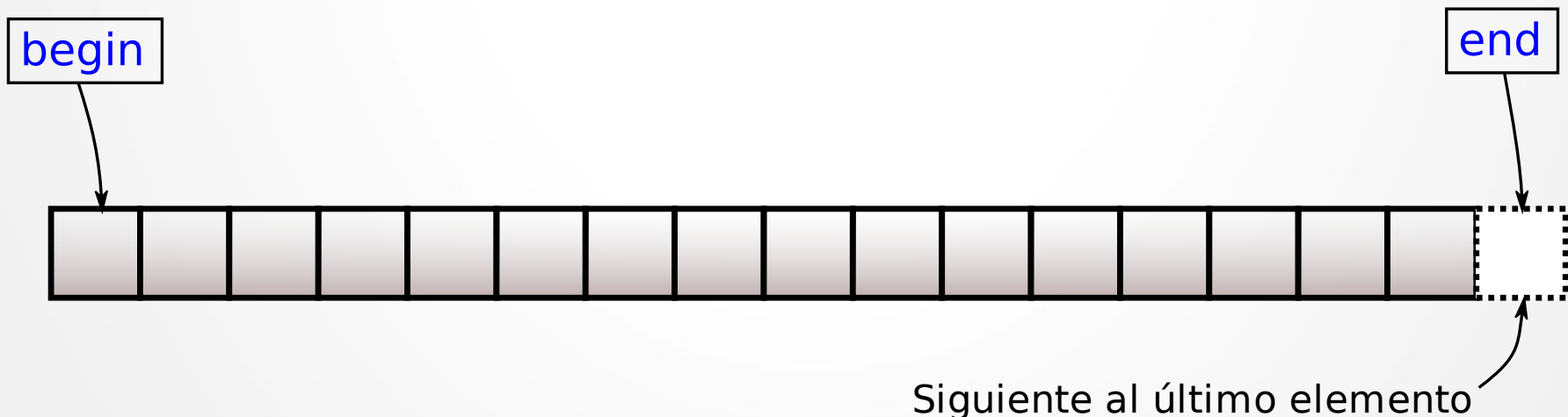
Cuidado: No es el mismo concepto que el Iterador de Python.

# TAD. Coordenada

En C++:

`begin()` devuelve un iterador que hace referencia al primer elemento del contenedor.

`end()` devuelve un iterador que hace referencia al siguiente del último elemento del contenedor. Éste es una coordenada **especial**, en el sentido que **no hace referencia** realmente **a un elemento**.





# TAD. Coordenadas

Toda coordenada debería soportar **dos operaciones básicas**: **Obtener el elemento** al que hace referencia y **avanzar al próximo elemento** del contenedor.

Para obtener el elemento al que hace referencia una coordenada vamos a necesitar una operación: nosotros le vamos a llamar **value()**.

A esto se lo llama “**desreferenciar**” a la coordenada. **No vamos a permitir desreferenciar** a la coordenada devuelta por `end()`.

En C++ anteponemos `*` para desreferenciar un iterador

# TAD. Coordenadas

Para avanzar al próximo elemento, se debe incrementar a la coordenada. En C++ usamos el **operador ++**.

Vamos a especificar ahora las operaciones: **advance()** y **next()**

En el caso de `avance()` vamos a pedir que avance y pierdo la coordenada original.

En el caso de `next()` obtengo una nueva coordenada que hace referencia a la siguiente posición pero no pierdo la coordenada original.

# TAD. Coordenada

Como estamos especificando una lista doble su **coordenada debería ser bidireccional** que además puede **retroceder** al elemento anterior (en C++ usamos **operador --**).

En este caso vamos a especificar las operaciones: **retreat()** y **prev()**

Dos coordenadas también deben poder **compararse** por igualdad (**operador ==**)

# Coordenadas. Se acuerdan...

```
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    vector<int> v = { 4, 8, 15, 16, 23, 42};

    cout << "Usando índices:" << endl;
    for(size_t i = 0; i < v.size(); ++i)
        cout << v[i] << endl;

    cout << "Usando iteradores:" << endl;
    for(auto p = begin(v); p != end(v); ++p)
        cout << *p << endl;

    return 0;
}
```

¿Cuál era el **tipo** del iterador p?

No nos importaba, gracias a **auto**.  
Pero si quisiéramos saberlo, es  
**vector<int>::iterator**.  
Iterator es una clase definida dentro  
de la clase vector

# TAD. Coordenada

Se acuerdan que `begin()` y `end()` eran “operaciones del contenedor”.

En C++ hacíamos **`v.begin()`** y **`v.end()`**

Se los “pedimos”  
al vector..

Entonces en nuestro caso vamos a especificar en “otras operaciones” de nuestra lista doble estas operaciones:

# Especificación. Lista Doble

## Otras Operaciones

**inicio(ListaDoble<a>)→ coordenada<ListaDoble<a>>**

**fin(ListaDoble<a>)→ coordenada<ListaDoble<a>>**

**Especificamos la  
Coordenada**

# Especificación. “Molde General”

TAD Nombre TAD

Igualdad Observacional

Usa

Parámetro Formal

Géneros

observadores básicos

Generadores

otras operaciones

Axiomas

Exporta



# Especificación. Coordenada

**TAD** Coordenada  $\langle b \langle a \rangle \rangle$

**Igualdad Observacional**

Si **a** y **b** son dos coordenadas

**a** es igual a **b** si se cumple que: hacen referencia a la misma posición dentro del mismo contenedor

**Usa**

**Parámetro Formal**

**b, a**

**Géneros**

Coordenada  $\langle b \langle a \rangle \rangle$

# Especificación. Coordenada

## observadores básicos

`valor(Coordenada<b<a>>)` → a

{Pre: La coordenada es válida}

## Generadores

`a_partir_de(Coordenada <b<a>>)` → Coordenada <b<a>>

# Especificación. Coordenada

## Otras Operaciones

**avanzar(Coordenada<b<a>>) → Coordenada<b<a>>**

**siguiente(Coordenada<b<a>>) → Coordenada<b<a>>**

**retroceder(Coordenada<b<a>>) → Coordenada<b<a>>**

**previo(Coordenada<b<a>>) → Coordenada<b<a>>**

# Especificación. Coordenada

## Axiomas

`a_partir_de(Coordenada <b<a>> c)`: crea una coordenada a partir de la coordenada c

`valor(Coordenada<b<a>> c)`: retorna el valor al que hace referencia la coordenada c

`siguiente(Coordenada<b<a>> c)`: retorna una nueva coordenada que hace referencia a la posición siguiente de c.

`previo(Coordenada<b<a>> c)`: retorna una nueva coordenada que hace referencia a la posición anterior de c.

# Especificación. Coordenada

## Axiomas

`avanzar(Coordenada<b<a>> c)` avanza a la siguiente a c

`retroceder(Coordenada<b<a>> c)` retrocede a la anterior a c

## Exporta

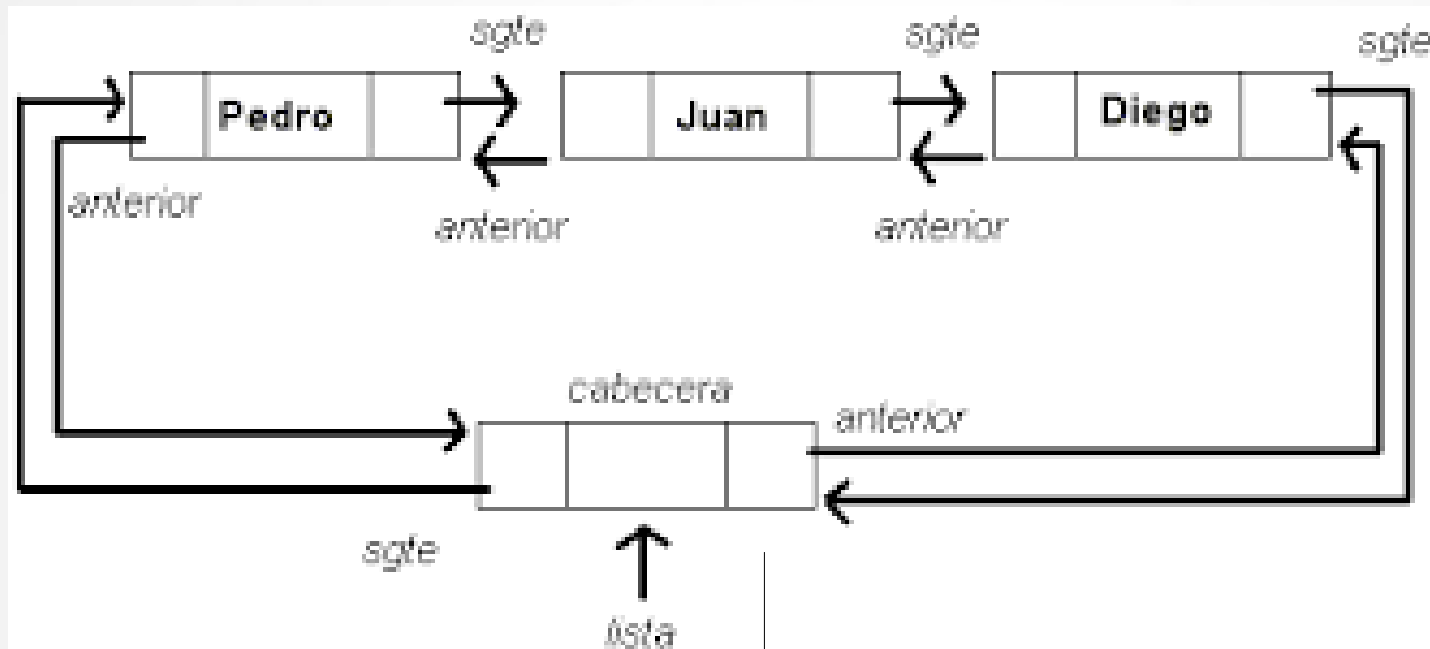
`Coordenada<b<a>>, a_partir_de, siguiente, previo, valor, avanzar, retroceder`

# **Implementación del TAD**

## **Lista Doble**

# **Estructura Interna que define a la Lista Doble**

# TAD Lista Doble



“Nodo escondido” que no  
almacena un valor...



# TAD Lista Doble

```
class DoublyLinkedList():  
    ...  
    @dataclass  
    class _Node:  
        value: Any  
        prev: '_Node' = None  
        next: '_Node' = None  
  
    @dataclass  
    class _Head:  
        prev: '_Node' = None  
        next: '_Node' = None  
  
    __slots__ = ['_head']
```

→ “Nodo escondido” que no  
almacena un valor...

# **Generadores...**

# Especificación. Lista Doble

## Generadores

`vacía() → ListaDoble<a>`

`{Post: La lista doble retornada esta vacía}`

`a_partir_de(Secuencia<a>)→ ListaDoble<a>`

`{Post: La lista doble contiene los elementos de la secuencia recibida con el mismo orden que tenían en la secuencia}`

# Implementación. Lista Doble

```
__slots__ = ['_head']
```

```
def __init__(self, iterable=None):  
    self._head = DoublyLinkedList._Head()  
    self._head.prev = self._head.next = self._head  
    if iterable is not None:  
        for value in iterable:  
            self.append_back(value)
```

**Implementamos las  
operaciones..**

# Especificación. Lista Doble

## observadores básicos

`tamaño(ListaDoble<a>) → Natural`

`es_vacia(ListaDoble<a>) → Bool`

`primero(ListaDoble<a>) → a`

`{Pre: la cola tiene al menos un elemento}`

`último(ListaDoble<a>) → a`

`{Pre: la cola tiene al menos un elemento}`

# Implementación. tamaño?

```
def __len__(self):  
    n = 0  
    nodo = self._head.next  
    while nodo is not self._head:  
        n += 1  
        nodo = nodo.next  
    return n
```

# Implementación. Vacía?, primero y último

```
def is_empty(self):  
    return self._head.next is self._head  
  
@property  
def front(self):  
    assert not self.is_empty(), 'lista vacía'  
    return self._head.next.value  
  
@property  
def back(self):  
    assert not self.is_empty(), 'Lista vacía'  
    return self._head.prev.value
```



# Especificación. Lista Doble

## Otras Operaciones

`agregar_frente(ListaDoble<a>, a) → None`

`{Post: la Lista Doble tiene al menos un elemento}`

`agregar_final(ListaDoble<a>, a) → None`

`{Post: la Lista Doble tiene al menos un elemento}`

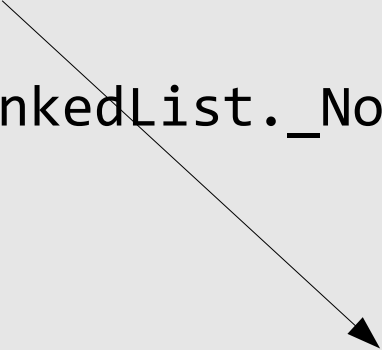
# Implementación

```
def append_front(self, value):  
    self.insert(self.begin(), value)
```

```
def append_back(self, value):  
    self.insert(self.end(), value)
```

# Implementación. Copy

```
def copy(self):  
    new_list = DoublyLinkedList()  
    act = new_list._head  
    for value in self:  
        node = DoublyLinkedList._Node(value)  
        node.prev = act  
        act.next = node  
        act = node  
    new_list._head.prev = act  
    act.next = new_list._head  
    return new_list
```



Puedo iterarla porque  
luego  
Vamos a implementar  
Los métodos  
`__iter__()` y  
`__next__()`

# Implementación. Clear, begin y end

```
def clear(self):  
    self._head.prev = self._head.next = self._head  
  
def begin(self):  
    return DoublyLinkedList._Coordinate(self._head.next)  
  
def end(self):  
    return DoublyLinkedList._Coordinate(self._head)
```

# Implementación. Comparación

```
def __eq__(self, other):  
    p = self.begin()  
    q = other.begin()  
    while p != self.end() and q != other.end():  
        if p.value != q.value:  
            return False  
        p.advance()  
        q.advance()  
    return p == self.end() and q == other.end()
```

# Implementación. Representación

```
def __repr__(self):  
    return 'DoublyLinkedList([' + ', '.join(repr(v)  
for v in self) + '])'
```

# Especificación. Lista Doble

## Otras Operaciones

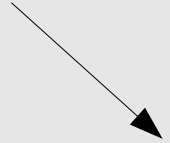
`insertar(ListaDoble<a>, coordenada<b<a>, a )→coordenada<b<a>`

`{Pre: la coordenada es válida}`

`{Pos: la Lista Doble no esta vacía}`

`borrar(ListaDoble<a>, coordenada<b<a> )→coordenada<b<a>`

`{Pre: la coordenada es válida}`



Vamos a implementar  
Insertar y borrar  
luego  
de ver la Implementación de  
Coordenada..ya que están relacionadas!

**Implementamos  
Coordenada**



# Implementación. Coordenada

```
class _Coordinate():
    __slots__ = ['_node']
    def __init__(self, coordinate_or_node):
        If isinstance(coordinate_or_node, DoublyLinkedList._Coordinate):
            self._node = coordinate_or_node._node
        else:
            self._node = coordinate_or_node
```

# Implementación. Coordenada

```
@property
def value(self):
    return self._node.value

@value.setter
def value(self, value):
    self._node.value = value
```

# Implementación. Coordenada

```
def advance(self):  
    self._node = self._node.next  
    return self  
  
def next(self):  
    Return DoublyLinkedList._Coordinate(self._node).advance()
```

# Implementación. Coordenada

```
def retreat(self):  
    self._node = self._node.prev  
    return self
```

```
def prev(self):  
    return DoublyLinkedList._Coordinate(self._node).retreat()
```

# Implementación. Coordenada

```
def __eq__(self, other):  
    return self._node is other._node
```

Dos coordenadas que apuntan al mismo  
Objeto...

Si usara == con la estructura interna  
Circular que manejamos  
“Nunca terminaría”

# Coordenada o Iteradores

## Conclusiones:

- Los **iteradores** fueron **inspirados en los punteros**, imitando su sintaxis y uso.
- El tipo de iterador **depende de la estructura de datos sobre la que itera** y, a su vez, ésta **limita las capacidades del iterador** (por ejemplo, no es buena idea un iterador de acceso aleatorio para una lista enlazada simple).
- Los iteradores permiten **independizar a un algoritmo de la estructura de datos** sobre la cuál opera (por ejemplo, es el mismo algoritmo el usado para buscar un valor en un arreglo desordenado o en una lista enlazada simple).

# Coordenadas o Iteradores

```
def find(first, last, value):  
    while first != last:  
        if first.value == value:  
            return first  
        first = first.next()  
    return last
```

De esta forma desacoplamos  
Nuestro algoritmo de la  
Estructura de Datos :-)

Find() así implementado  
“funciona” para cualquier tipo  
Que soporte coordenadas

**Implementamos ahora  
insertar y borrar en una  
posición en nuestra Lista  
Doble**



# Implementación. Insertar

```
def insert(self, coord, value):  
    current = coord._node  
    new_node = DoublyLinkedList._Node(value)  
    new_node.prev = current.prev  
    new_node.next = current  
    new_node.prev.next = new_node  
    new_node.next.prev = new_node  
    #return DoublyLinkedList._Coordinate(current)
```

# Implementación. Borrar

```
def erase(self, coord):  
    current = coord._node  
    current.prev.next = current.next  
    current.next.prev = current.prev  
    return coord.next()
```

**Y si queremos iterar  
nuestra lista doble...  
¿Qué debemos  
implementar para que un  
objeto pueda iterar?**

# **Iteraciones, Iterables e Iteradores**

# Iteraciones

La misma estructura **for/in** se puede usar para iterar sobre los valores numéricos generados por **range**, sobre los caracteres de una cadena de caracteres y sobre los elementos de una tupla.

```
>>> for x in range(10)
...     print(x)

>>> for x in 'Una cadena':
...     print(x)

>>> for x in (1, 2, 3):
...     print(x)
```

¿Cómo hace **for/in** para lograrlo?

¿Y se definen otros contenedores nuevos?

.

# Iterador

Un iterador es un objeto que hace referencia a elementos de una secuencia. Esos elementos pueden ser calculados (como en el caso de `range()`) o tomados de un contenedor (como en el caso de las cadenas de caracteres o las tuplas).

Es posible obtener un iterador sobre un objeto usando `iter()`.

```
>>> iter(range(10))
<range_iterator object at 0xb5fc3e60>

>>> iter('una cadena')
<str_iterator object at 0xb552a60c>

>>> iter((1, 2, 3))
<tuple_iterator object at 0xb552a5cc>
```

# Iterador

Es posible obtener uno a uno los elementos del objeto sobre el que se está iterando usando `next()` hasta que indique el fin de la iteración con un error de tipo **StopIteration**.

```
>>> i = iter(range(4))
>>> next(i)
0
>>> next(i)
1
>>> next(i)
2
>>> next(i)
3
>>> next(i)
StopIteration
```

# Iterable

Se podrá iterar con `for/in` a cualquier objeto que `iter()` pueda obtener un iterador. A dicho objeto se lo denominará **iterable**.

No todo objeto es iterable (por ejemplo: `int`, `float`, etc).

En Python existen muchos objetos iterables. Todos se comportan de la misma manera.

```
>>> iterador = iter(iterable)
>>> next(iterador) → primer elemento
>>> next(iterador) → segundo elemento
>>> :
>>> next(iterador) → último elemento
>>> next(iterador) → StopIteration
```



# Lista Doble Iterable

Para poder iterar sobre nuestra lista doble vamos a implementar entonces:

**la clase `_Iterator` con los métodos `iter()` y `next`**

**Y el método `iter()` en la lista doble**

# Implementación. En la Clase Iterator

```
class _Iterator():  
    __slots__ = ['_node', '_end']  
  
    def __init__(self, head, end):  
        self._node = head  
        self._end = end
```

# Implementación. En la Clase Iterator

```
def __iter__(self):  
    return self  
  
def __next__(self):  
    if self._node is self._end:  
        raise StopIteration  
    value = self._node.value  
    self._node = self._node.next  
    return value
```

# Implementación. En la Lista Doble

```
def __iter__(self):  
    return DoublyLinkedList._Iterator(self._head.next, self._head)
```