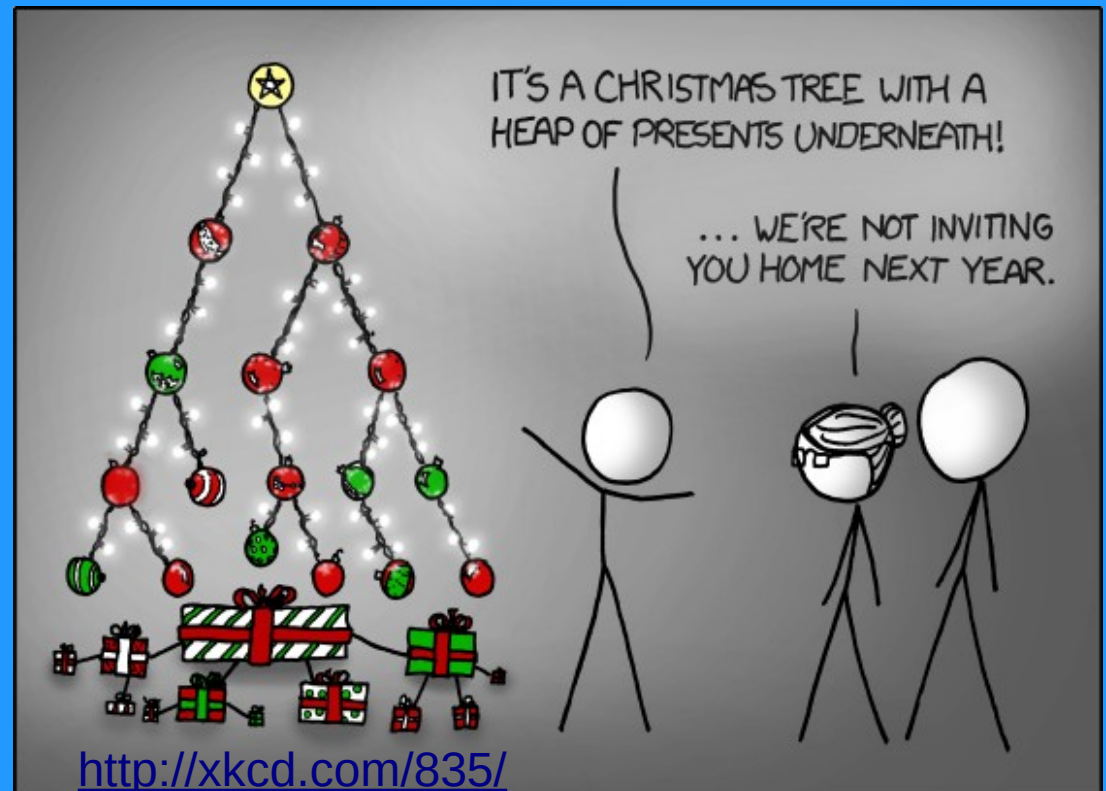


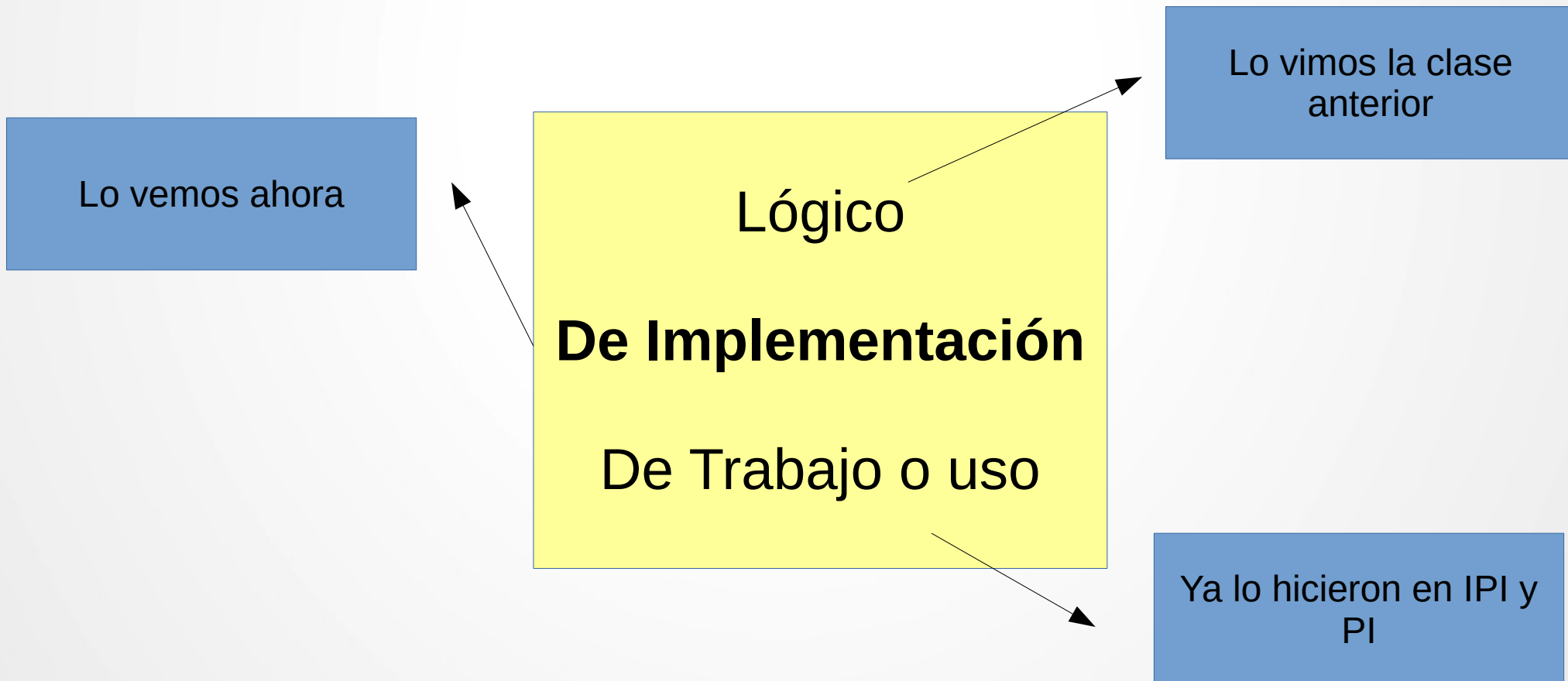
Estructuras de Datos

2020



Niveles de Trabajo. Recordemos...

Vimos que podemos diferenciar tres niveles de trabajo:



TAD. Especificación e Implementación

La especificación lógica de un TAD es un documento en el que se plasma la abstracción realizada al diseñar una estructura de datos.

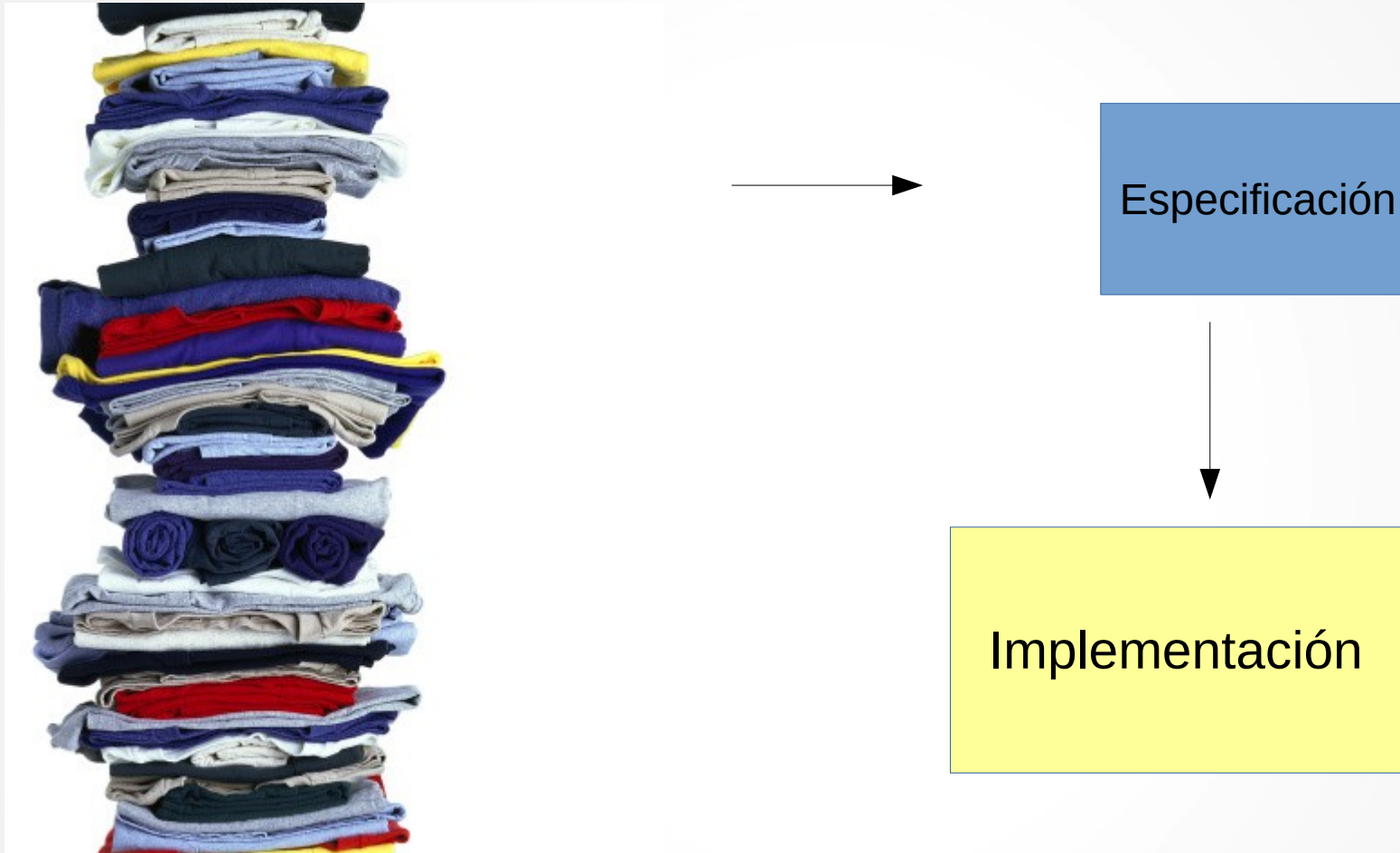
Dicho documento pasará a ser el **plano mediante el cual se construirá (implementará) la estructura de datos** y en el que se definirá claramente las **reglas** en las que podrá usarse (aplicarse) el TAD.

En el TP 1 diseñamos las Estructuras
para obtener ese plano
En el TP 2 vamos a construir esas estructuras
implementándolas en un Lenguaje de Programación.

TAD.

Implementación

TAD. Implementación



Vamos a escribir la implementación de una Pila para introducir los detalles de Implementación en Python...

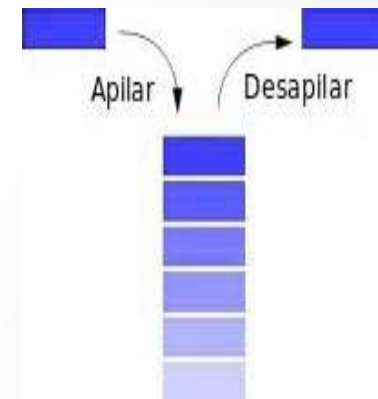
Recordemos la Especificación del TAD Pila Dinámica

Especificación. Pila Dinámica

Una pila representa una estructura lineal de datos en la que se puede agregar o quitar elementos únicamente por uno de los dos extremos.

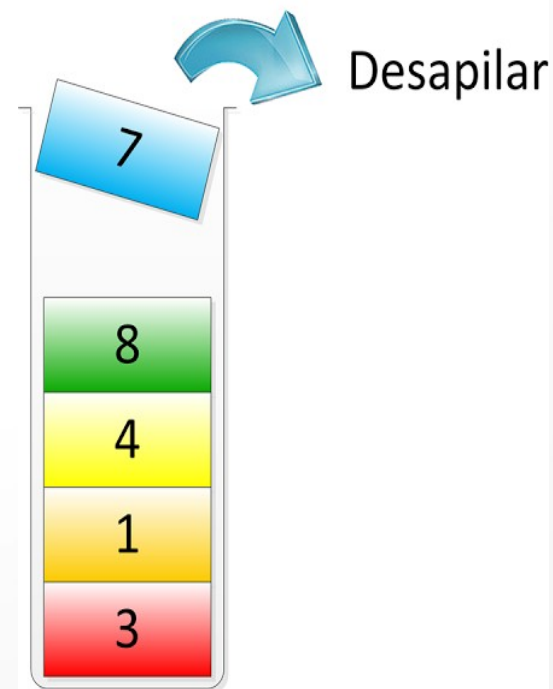
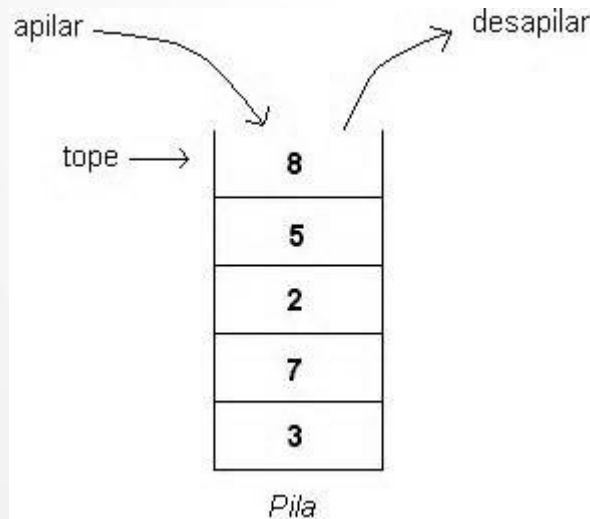
Los elementos de una pila se eliminan en orden inverso al que se insertaron; es decir, el último elemento que se mete en la pila es el primero que se saca.

Suele denominarse **lifo** (last in, first out)



Especificación. Pila Dinámica

Una pila es una estructura de datos lineal, como un arreglo; y se puede definir formalmente como una colección de datos a los cuales se puede acceder mediante un extremo, que se conoce generalmente como **tope**.



Especificación. “Molde General”

TAD Nombre TAD

Igualdad Observacional

Usa

Parámetro Formal

Géneros

observadores básicos

Generadores

otras operaciones

Axiomas

Exporta

Especificación. Pila Dinámica

TAD Pila <a>

Igualdad Observacional

Si **a** y **b** son dos pilas

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

Parámetro Formal

a

Géneros

Pila<a>

Especificación. Pila Dinámica

Generadores

`vacía() → Pila<a>`

`{Post: La pila retornada esta vacía}`

`a_partir_de(Secuencia<a>) → Pila<a>`

`{Post: La pila contiene apilados los elementos de la secuencia recibida}`

Especificación. Pila Dinámica

observadores básicos

`tamaño(Pila<a>) → Natural`

`es_vacía(Pila<a>) → Bool`

`tope(Pila<a>) → a`

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

Especificación. Pila Dinámica

Otras Operaciones

apilar(Pila<a>, a) → None

{Pos: la pila no esta vacía}

desapilar(Pila<a>) → a

{Pre: la pila tiene al menos un elemento}

{Pos: la pila perdió el tope que tenía antes de desapilar}

Especificación. Pila Dinámica

Axiomas

vacía(): Crea una pila (sin elementos)

a_partir_de(Secuencia<a> s): crea una pila que contiene apilados los elementos de la secuencia **s** siendo el último elemento de la secuencia el que quede en el tope de la pila.

apilar(Pila<a> p, a elem): apila en el tope de **p** el elemento **elem**.

desapilar(Pila<a> p): quita el elemento que se encuentra en el tope de **p**.

tamaño(Pila<a> p): Retorna/devuelve la cantidad de elementos de la pila **p**.

Especificación. Pila Dinámica

Axiomas

es_vacíá(Pila<a> p): Retorna/devuelve verdadero si la pila **p** esta vacía y falso en caso contrario.

tope(Pila<a> p): retorna/devuelve el elemento que se encuentra en el tope de la pila **p**.

Exporta

Pila<a>, vacía, a_partir_de, es_vacíá, tope, tamaño, apilar, desapilar

Implementación del TAD Pila usando Python

Implementación. Software



Distribución de Python llamada Anaconda.

Se descarga para la plataforma correspondiente desde:

<https://www.continuum.io/downloads>

Incluye la versión más reciente de Python 3 (actualmente la 3.8) junto con la versión más reciente de Ipython/Jupyter.

Relación entre Tipos y Clases

Variables. Objetos. Clases. Tipos.

En Python definimos una **variable** a partir de su **asignación**

Variable = una- expresión

La Variable es usada como nombre para un **objeto**.

Objetos y Tipos.

En Python se dice que “todo es un objeto”.

Entonces vamos a definir en este curso:

Objeto: Todo valor que se puede manipular en Python. Todo objeto tiene un **tipo**.

Es por esto que Python es **fuertemente tipado**.

Su **tipado es dinámico** (ya que los tipos de las variables se determinan en tiempo de ejecución)

Clases y tipos

Tipo: Define los valores que pueden ser representados en un objeto de ese tipo, como se almacenará en memoria y que operaciones podrán realizarse sobre él.

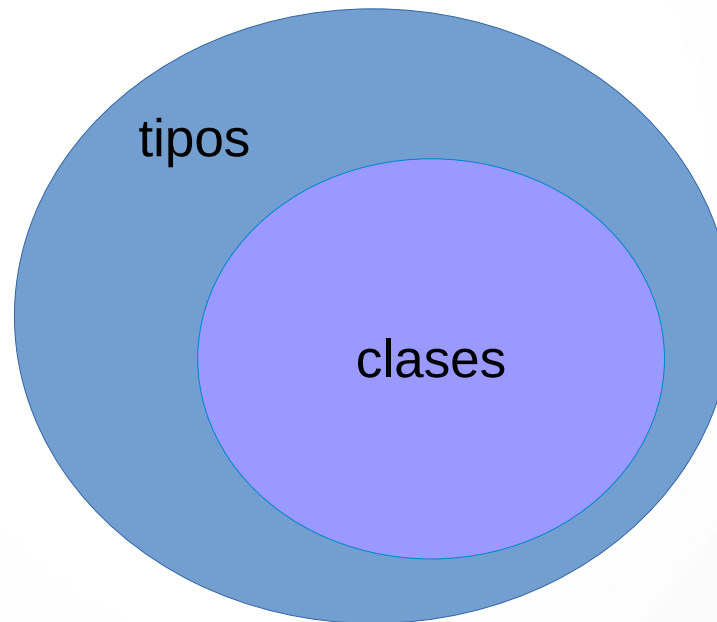
Clase: Es una forma de concretar o implementar un tipo.

```
>>> n = 8  
  
>>> type(n)  
<class 'int'>
```

La variable que definimos se llama `n`, `8` es un objeto que representa un número entero e `int` es la clase que implementa al tipo “número entero”.

Clases y Tipos

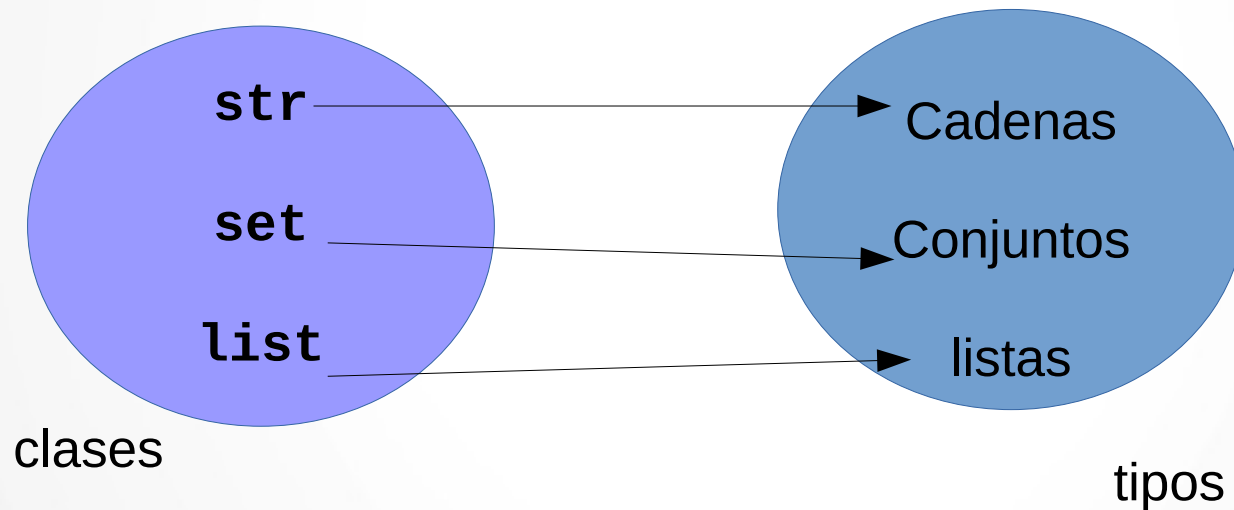
En general toda clase define a un tipo. Pero todo tipo NO necesariamente tiene una clase que lo defina:



En otros lenguajes hay tipos que llamamos primitivos...
Ejemplo: en el lenguaje C++ al tipo float no hay clase que lo defina

Clases y Tipos en Python

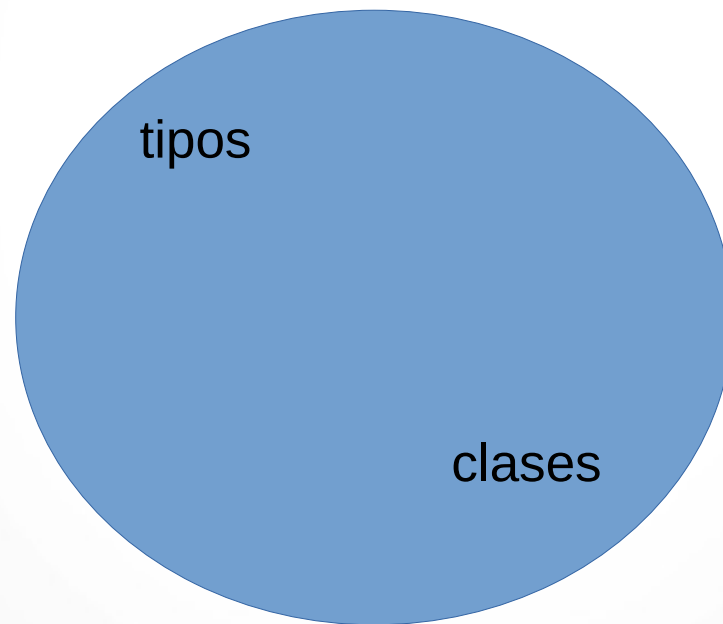
En Python la relación es uno a uno. Toda clase define a un tipo y todo tipo tiene una clase que lo define:



Presentamos tres ejemplos en el diagrama anterior

Clases y tipos en Python

Entonces en Python ambos conjuntos son iguales:



TAD. Clases

Vamos a implementar nuestro tipo Pila usando una clase.

Para definir una clase en Python usamos la palabra clave **class**

TAD. Clases

Cada vez que se cree un valor a partir de una clase vamos a decir que el valor creado es un **OBJETO** de dicha clase.

A un objeto de una clase se lo denomina una **INSTANCIA** de la clase y al acto de crearlo se lo llama **INSTANCIAR** un objeto.

Encapsulamiento

TAD. Encapsulamiento

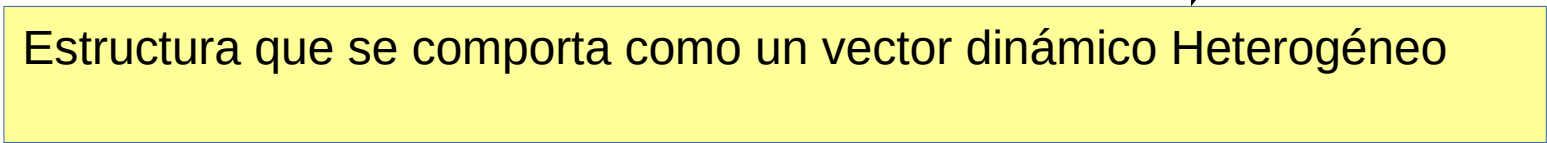
Vamos a tener una implementación de la Pila donde los datos y el código que los manipula están **ENCAPSULADOS** en una única entidad: la Clase.

**Estructura Interna
que define a la Pila**

TAD Pila dinámica

Comencemos con un ejemplo de como implementar una Pila o Stack usando Python que crezca y decrezca en forma dinámica.

En esta implementación vamos a **usar una lista de Python** para guardar los elementos/apilados.



Estructura que se comporta como un vector dinámico Heterogéneo

TAD Pila dinámica

Indicamos todos los atributos que forman la estructura interna:

```
class Stack()  
    __slots__ = ["_values"]  
    ...
```

→ `_values` es el nombre de la lista que va a contener los elementos de la pila

Generadores del TAD

Especificación. Pila Dinámica

Generadores

`vacía() → Pila<a>`

`{Post: La pila retornada esta vacía}`

`a_partir_de(Secuencia<a>) → Pila<a>`

`{Post: La pila contiene apilados los elementos de la secuencia recibida}`

Especificación. Pila Dinámica

Axiomas

vacía(): Crea una pila (sin elementos)

a_partir_de(Secuencia<a> s): crea una pila que contiene apilados los elementos de la secuencia **s** siendo el último elemento de la secuencia el que quede en el tope de la pila.

TAD Pila dinámica

Cuando creamos un objeto de nuestro tipo, éste debería estar listo para ser usado.

Luego de crear un objeto de una clase dada, Python invoca al método `__init__` de esa clase para que lo inicialice.

Todo lo anterior es **transparente al usuario**.

Cuando creamos una variable `int`, `float` o `tuple`, no tenemos que preocuparnos por llamar a `__init__` antes de poder usarlos.

Implementación. Generadores

Lista donde podemos
mencionar todos los atributos
(Est. Int.)
No es obligatorio su uso

Siempre me “pasan a mí”
cuando invocan un
método de la clase

```
class Stack():
```

```
    __slots__ = ["_values"]
```

```
    def __init__(self, iterable=None):
```

```
        self._values = []
```

```
        if iterable is not None:
```

```
            for value in iterable:
```

```
                self.push(value)
```

Si en la creación
del objeto no se
da un parámetro,
iterable es None.

Los atributos los referenciamos
a partir del parámetro self.
self._values es un atributo:
Una lista que contiene todos los
elementos de la pila.

Similar al for basado
en rangos de C++.

TAD Pila dinámica

Ahora podemos crear una pila usando el nombre de la Clase de dos formas:

```
s = Stack()  
s1 = Stack('abc')  
s2 = Stack([1,2,3,4,5])
```

Obtenemos una pila vacía o a partir de una secuencia tal cual lo habíamos especificado

Luego de crear al objeto, Python invoca automáticamente al método `__init__`.

Ocultamiento

TAD. Ocultamiento

Debemos restringir que cosas pueden ser accedidas desde “afuera” de la clase.

Python no nos permite usar modificadores de acceso como private y public. Pero podemos indicar con un “_” delante del nombre que eso es un detalle de Implementación privado y no debe usarse.

TAD. Ocultamiento

Un mecanismo que permite tener cierto control sobre el acceso al estado interno (que además resulta conveniente para implementar observadores) es utilizar **property** para definir propiedades.

TAD. Ocultamiento

Una propiedad simula ser un atributo normal de una clase pero en realidad tanto para acceder a su valor así como para cambiarlo o borrarlo se invocan a métodos de la clase.

Para acceder al valor, se usa un método denominado *getter*, para cambiarlo se usa otro conocido como *setter* y, finalmente, para borrarlo (usando la sentencia **del**), se usa uno más llamado *deleter*.

TAD. Ocultamiento

Ejemplo sin usar **property**:

```
class Inocente():  
    def __init__(self):  
        self.datos = []
```

```
>>> x = Inocente()  
>>> x.datos  
[]  
>>> x.datos.append("Hola")  
>>> x.datos  
['Hola']
```

TAD. Ocultamiento

Ejemplo sin usar **property**:

```
class Inocente():  
    def __init__(self):  
        self.datos = []
```

```
>>> x.datos = 1
```

```
>>> x.datos
```

```
1
```

```
>>> del x.datos
```

```
>>> x.datos
```

```
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
```

```
AttributeError: 'Inocente' object has no attribute 'datos'
```

TAD. Ocultamiento

Ejemplo usando **property**:

```
class Protegida():  
    def __init__(self):  
        self._datos = []  
  
    @property  
    def datos(self):  
        return self._datos.copy()
```

TAD. Ocultamiento

Ejemplo usando **property**:

```
>>> x = Protegida()  
>>> x.datos  
[]  
>>> x.datos.append("Hola")  
>>> x.datos  
[]
```

TAD. Ocultamiento

Ejemplo usando **property**:

```
>>> x.datos = 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute

>>> x.datos
[]

>>> del x.datos
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't delete attribute

>>> x.datos
[]
```

Ahora usando
property
obtenemos un
error al querer
asignar o borrar

TAD. Ocultamiento

Ejemplo usando **property** con *setter*:

```
class Paranoica():  
    def __init__(self):  
        self._datos = []  
  
    @property  
    def datos(self):  
        return self._datos.copy()  
  
    @datos.setter  
    def datos(self, nuevos):  
        self._datos = list(nuevos)
```

TAD. Ocultamiento

Ejemplo usando **property**:

```
>>> x = Paranoica()
>>> x.datos
[]
>>> lista = [1, 2, 3]
>>> x.datos = lista
>>> x.datos
[1, 2, 3]
>>> lista.append(4)
>>> lista
[1, 2, 3, 4]
>>> x.datos
[1, 2, 3]
```


Manejo de la memoria

TAD. Destructor?

Cuando un objeto reserva memoria en forma dinámica, éste deberá asegurarse de liberar dicha memoria antes de que sea destruido. Si no lo hace, esa memoria reservada no podrá ser utilizada durante el resto de la ejecución del programa.

En algunos lenguajes tenemos que definir en la clase un método especial que sea invocado automáticamente justo antes de que un objeto de dicha clase sea destruido, asegurándose de que ese objeto libere correctamente todo recurso que tenga en uso. A este método lo llamaremos un **DESTRUCTOR**.

TAD. Destructor?

En Python no es necesario definir un destructor porque cuenta con un Recolector de Basura.

Todo objeto que no posee referencias a él es automáticamente liberado de la memoria.

Entonces, no necesitamos implementar un Destructor.

**Implementamos las
operaciones...**

Especificación. Pila Dinámica

observadores básicos

`tamaño(Pila<a>) → Natural`

`es_vacía(Pila<a>) → Bool`

`tope(Pila<a>) → a`

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

Especificación. Pila Dinámica

Axiomas

tamaño(Pila<a> p): Retorna/devuelve la cantidad de elementos de la pila **p**.

es_vacía(Pila<a> p): Retorna/devuelve verdadero si la pila **p** esta vacía y falso en caso contrario.

tope(Pila<a> p): retorna/devuelve el elemento que se encuentra en el tope de la pila **p**.

Implementación. Tamaño y es_vacíá?

```
class Stack():  
    ...  
    def __len__(self):  
        return len(self._values)  
  
    def empty(self):  
        return len(self._values) == 0
```

Implementación. tope

```
class Stack():  
    ...
```

Assert: Si la condición no se cumple → produce una excepción y muestra lo que le indicamos entre comillas

```
# Pre: la pila tiene al menos un elemento
```

```
    @property  
    def top(self):  
        assert not self.empty(), 'Sin elementos'  
        return self._values[-1]
```


Especificación. Pila Dinámica

Otras Operaciones

apilar(Pila<a>, a) → None

{Pos: la pila no esta vacía}

desapilar(Pila<a>) → Pila<a>

{Pre: la pila tiene al menos un elemento}

{Pos: la pila perdió el tope que tenía antes de desapilar}

Especificación. Pila Dinámica

Axiomas

apilar(Pila<a> p, a elem): apila en el tope de **p** el elemento **elem**.

desapilar(Pila<a> p): quita el elemento que se encuentra en el tope de **p**.

Implementación. Apilar y desapilar

```
class Stack():  
    ...  
    def push(self, value):  
        self._values.append(value)  
  
    def pop(self):  
        assert not self.empty(), 'Sin elementos'  
        return self._values.pop()
```

Implementación. Clear

```
class Stack():  
    ...  
    def clear(self):  
        self._values.clear()
```

Los contenedores suelen soportar una operación que permite vaciarlos rápida y eficientemente.

En el caso de la pila, esto evitaría tener que vaciarla desapilando elementos mientras que la pila no esté vacía.

Implementación. Copy y Clear

```
class Stack():  
  
    ...  
  
    def copy(self):  
        new_stack = Stack()  
        new_stack._values = self._values.copy()  
        return new_stack
```

Dado que no es posible cambiar la semántica de la asignación en Python, suele resultar conveniente agregar una operación que retorne una copia del TAD implementado ya que asignar un objeto a dos variables distintas no crea copias del objeto sino que ambas lo referencian.

TAD Pila dinámica

¿Y si queremos comparar dos pilas?

Deberíamos implementar el “==” teniendo en cuenta lo especificado en “Igualdad Observacional”

Redefinimos operadores..

Especificación. Pila Dinámica

TAD Pila <a>

Igualdad Observacional

Si *a* y *b* son dos pilas

***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*.**


Implementación. Comparación

```
class Stack():
```

```
...
```

```
def __eq__(self, other):  
    return self._values == other._values
```

En este caso como solo tengo una lista como atributo y sabe "compararse" simplemente uso el '==' de listas



Implementación. Representación

```
class Stack():
```

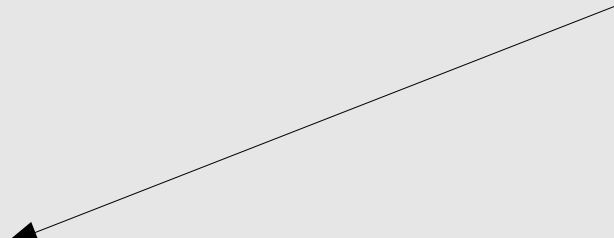
```
...
```

```
def __repr__(self):
```

```
    return ('Stack([' +
```

```
        ', '.join(repr(x) for x in self._values) +  
        '])' )
```

`__repr__` se invoca automáticamente cada vez que alguien pida que “me muestre”



**Usando la pila
implementada...**

Implementación. Usamos la pila

```
import pila
```



```
p = pila.Stack('Hola')
```

```
tope = p.top
```

```
print(tope)
```

En la misma carpeta debe existir el archivo pila.py con el código

Implementación. Usamos la pila

```
from pila import Stack
```

```
p = Stack('Hola')
```

```
tope = p.top
```

```
print(tope)
```

Podemos agregar todos los nombres que necesitemos separadas por comas. Luego no necesitamos mencionar el nombre del módulo.

TAD. Implementación

En las siguientes clases:

- Vamos a implementar la pila usando nodos enlazados
- Vamos a implementar una lista (al estilo de las vistas en PI)
- Vamos a ver Iteradores y Coordenadas o Posiciones