Clase 2

TAD Simple: Implementación

TAD Compuesto: Especificación y aplicación

Pila y Cola

Desarrollo de la clase 2

TAD Simple: Implementación

TAD compuesto

Ejemplo de Especificación de TAD compuesto

Ejemplo de aplicación utilizando TAD compuesto

TAD Pila

TAD Cola

Repaso Sintaxis Python

Revisar solución ejercicio de teoría

Revisar soluciones de TP 1

Ejemplo Implementación TAD Simple

```
# Tad Libro

#libro=["","","",0]

#def crearLib():
    #Crea un libro vacio
    libro=["", "", "",0]
    return libro

#def cargarLib(libro,n,e,a,p):
    #Carga los datos de un libro:
    # Nom, Edit, Autor, Precio
    libro[0]=n
    libro[1]=e
    libro[2]=a
    libro[3]=p
```

```
#def verNom(libro):
    #Retorna el nombre de un libro
    return libro[0]

#def verEdit(libro):
    #Retorna la editorial de un libro
    return libro[1]

#def verAutor(libro):
    #Retorna el autor de un libro
    return libro[2]

#def verPre(libro):
    #Retorna el autor de un libro
    return libro[3]
```

Ejemplo Implementación TAD Simple

```
#def modNom(libro,n):
    #Modifica el nombre de un libro
    libro[0]=n

#def modEdit(libro,e):
    #Modifica la editorial de un libro
    libro[1]=e

#def modAutor(libro,a):
    #Modifica el autor de un libro
    libro[2]=a

#def modPre(libro,p):
    #Modifica el autor de un libro
    libro[3]=p
```

```
#def asignarLib(libro1,libro2):
    #Asigna datos de un libro en otro
    modNom(libro1,verNom(libro2))
    modEdit(libro1,verEdit(libro2))
    modAutor(libro1,verAutor(libro2))
    modPre(libro1,verPre(libro2))
```

Operaciones TAD Simple y Compuesto

TAD Simple
Ejemplos: libro, auto, alumno, persona, etc
Cada uno de ellos tendrán sus datos en la estructura interna
Las operaciones de los tads simples por lo general van a ser las mismas:

crear()
cargar()
modifcar() #para cada dato
Ver() #para cada dato
Copiar()

TAD Compuesto
Ejemplos: librería, concesionaria, comisión, empresa, etc
Son Tad de tipo colección de datos
Las operaciones de los tads compuestos por lo general van a ser las mismas:

crear()
agregar()
eliminar()
cantidad()
existe()
recuperar()

Ejemplo Especificación TAD Compuesto

```
# Tad Libreria

#def crearlibreria():
    #Crea una libreria vacia

#def agregarLibro(libreria,l):
    #Agrega un libro a la libreria

#def eliminarLibro(libreria,l):
    #Elimina un libro de la libreria

#def recuperarLibro(libreria,i):
    #Retorna el libro de la posicion iesima

#def tamanio(libreria):
    #Retorna la cantidad de libros de la libreria
```

Ejemplo Aplicación utilizando TAD Compuesto

- Cargar 4 libros a una librería
- Imprimir los datos de todos los libros de la libreria

```
import TadLibro, TadLibreria

from TadLibro import *
from TadLibreria import *

#Crea la libreria
lib=crearLibreria()

#Cargar la Libreria
for i in range(1,4):
    #Crea y carga los datos del libro1
    l=crearLib()
    n=input("Ingrese un nombre")
    e=input("Ingrese un editorial")
    a=input("Ingrese un autor")
    p=float(input("Ingrese un precio"))
    cargarLib(l,n,e,a,p)
    agregarLibro(Lib,l)
```

```
print (lib)
#Imprime los datos de la libreria
print ("Imprime los datos de la libreria")
for i in range(0,len(lib)):
    a=recuperarLibro(lib,i)
    print ("Imprime los datos de un libro")
    print (verNom(l))
    print (verEdit(l))
    print (verAutor(l))
    print (verPre(l))
```

Ejemplo Aplicación utilizando TAD Compuesto

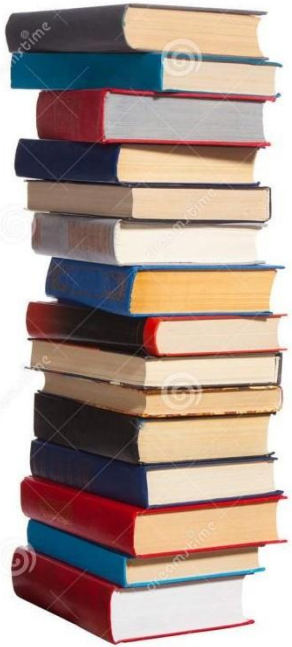
- Recuperar e imprimir el segundo libro
- Eliminar el libro recuperado e imprimir los datos de todos los libros de la libreria

```
#Recupera e imprime el segundo libro
print ("Imprime datos del libro recup")

l=recuperarLibro(lib,1)

print ("Imprime los datos de un libro")
print (verNom(l))
print (verEdit(l))
print (verAutor(l))
print (verPre(l))

print ("Elimina el libro recuperado")
#Elimina el libro recuperado

eliminarLibro(lib,l)
```

```
#Imprime libros que quedan en la libreria
print ("Imprime datos de libros de libreria")

for i in range(0,len(lib)):
    a=recuperarLibro(lib,i)
    print ("Imprime los datos de un libro")
    print (verNom(l))
    print (verEdit(l))
    print (verAutor(l))
    print (verPre(l))
```

TAD Pila

- ¿Qué es una Pila?

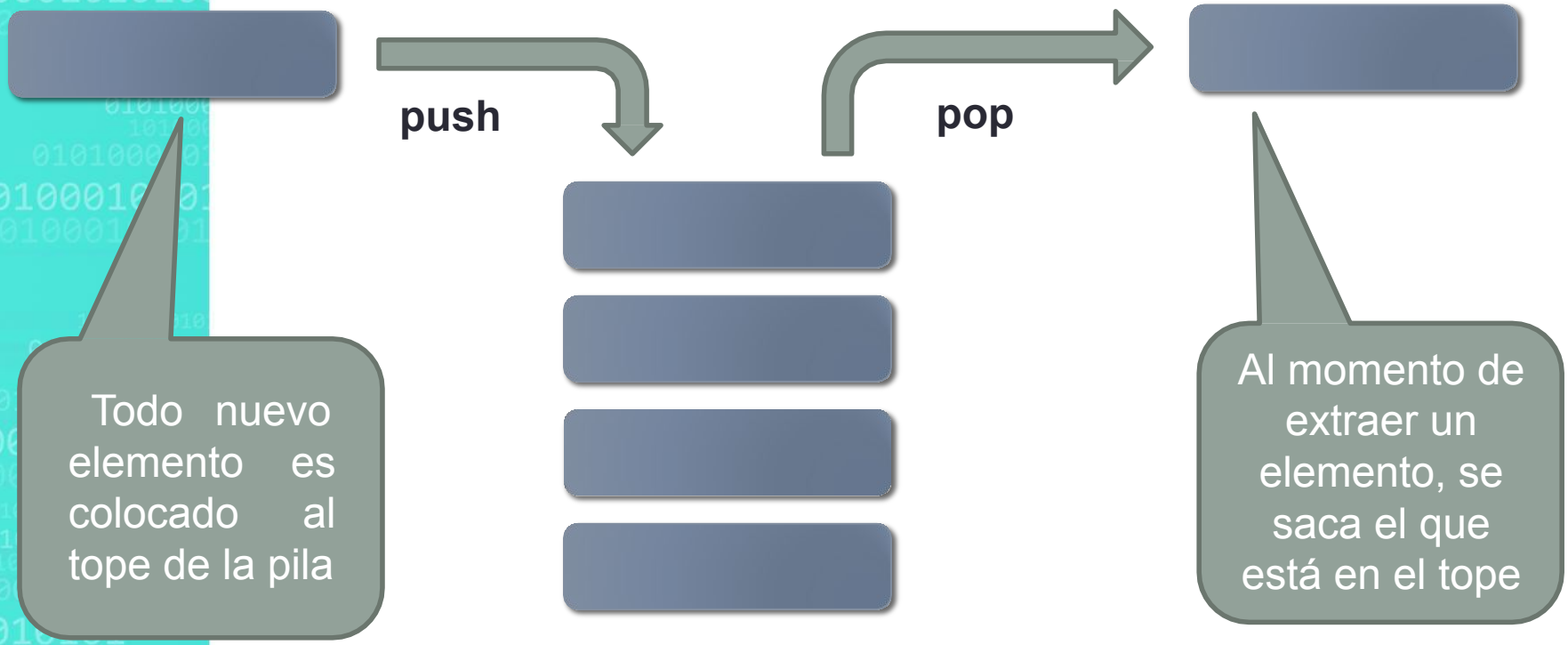


TAD Pila

- Una pila es una colección, lo que significa que es una estructura de datos que contiene múltiples elementos.
- Los elementos que se almacenan en una pila pueden ser de cualquier tipo. Por esta cualidad, a la pila se la conoce como tipo de dato genérico.
- La pila implementa una estructura “último en entrar primero en salir” (LIFO - “last in, first out” en inglés), porque el elemento añadido en último lugar es el primero que se extrae.

TAD Pila

- Las operaciones básicas que se pueden realizar con una pila son apilar (push) y desapilar (pop).



Ejemplo Especificación TAD Pila

```
def crearPila():
    #Crea una pila vacia

def esVacia(pila):
    #Retorna Verdadero si la pila no tiene elementos

def apilar(pila,elem):
    #Agrega un elemento al final de la pila

def desapilar(pila):
    #Retorna y elimina el ultimo elemento de la pila

def tamanio(pila):
    #Retorna la cantidad de elementos de la pila

def copiarPila(pila1,pila2):
    #Copia los datos de una pila a otra
```

```
def crearExpe():
    #Crea un expediente vacio

def cargarExpe(expediente,n,t):
    #Carga los datos de un expediente

def verNum(expediente):
    #Retorna el numero de un expediente

def verTipo(expediente):
    #Retorna el tipo de un expediente

def modiNum(expediente,n):
    #Modifica el numero de expediente

def modiTipo(expediente,t):
    #Modifica el tipo de un expediente

def asignarExpe(expediente1,expediente2):
    #Asigna los datos de un expediente en otro
```

Ejemplo Aplicación utilizando TAD Pila

```
import TadExpediente, TadPila

from TadExpediente import *
from TadPila import *


p=crearPila()
paux=crearPila()
#Carga la Pila original
print ("Carga datos en la pila")
for i in range (1,5):
    e=crearExpe()
    n=input("Ingrese un numero")
    t=raw_input("Ingrese un tipo de expediente")
    cargarExpe(e,n,t)
    apilar(p,e)
```

```
#Imprime la Pila original
print ("Imprime los datos de la pila")
for i in range (0, tamanio(p)):
    elem=desapilar(p)
    apilar(paux,elem)
    print elem

#pasamos los datos de la pila auxiliar a la original
copiarPila(p,paux)
print ("pila original y auxiliar despues del copiarpila")
```

TAD Cola

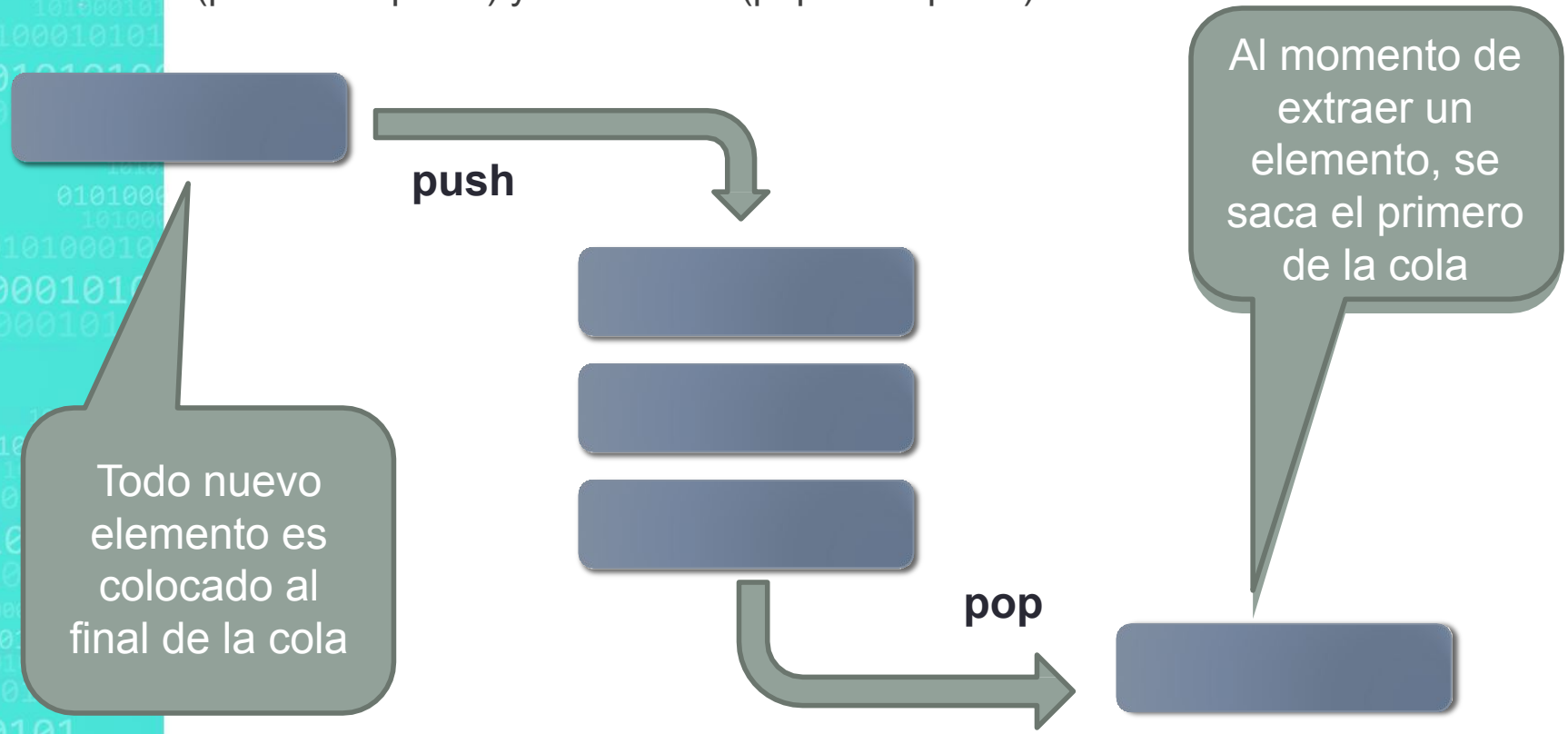
- ¿Qué es una Cola?

TAD Cola

- Una cola es una colección, lo que significa que es una estructura de datos que contiene múltiples elementos.
- Los elementos que se almacenan en una cola pueden ser de cualquier tipo. Por esta cualidad, a la cola se la conoce como tipo de dato genérico.
- La cola implementa una estructura “primero en entrar primero en salir” (FIFO - “first in, first out” en inglés), porque el elemento añadido en primer lugar es el primero que se extrae.

TAD Cola

- Las operaciones básicas que se pueden realizar con una cola son encolar (push o enqueue) y desencolar (pop o dequeue).



PYTHON

Repaso Sintaxis del Lenguaje

Operadores

1-LOGICOS

Operador	Descripción	Ejemplo
and	¿se cumple a y b?	`>>> True and False` `False`
or	¿se cumple a o b?	`>>> True or False` `True`
not	No al valor	`>>> not True` `False`

2-RELACIONALES

Operador	Descripción	Ejemplo
==	¿son iguales a y b?	`>>> 5 == 3` `False`
!=	¿son distintos a y b?	`>>> 5 != 3` `True`
<	¿es a menor que b?	`>>> 5 < 3` `False`
>	¿es a mayor que b?	`>>> 5 > 3` `True`
<=	¿es a menor o igual que b?	`>>> 5 <= 5` `True`
>=	¿es a mayor o igual que b?	`>>> 5 >= 3` `True`

ESTRUCTURA DE CONTROL IF

Una sentencia condicional consiste básicamente en:

- una prueba que evalúa verdadero o falso
- un bloque de código que se ejecuta si la prueba es verdadero
- un bloque opcional de código si la prueba es falsa

1-Sentencia condicional simple

```
#El bloque de condiciones puede estar formada por una o mas
condiciones, separadas por un operador logico
if bloqueDeCondiciones:
  #Bloque de codigo True
else:
  #Bloque de codigo False
```

2- Sentencia condicional anidada

```
#Un if anidado es cuando el bloque verdadeo o falso contiene otra
sentencia condicional
if bloqueDeCodiciones1:
  if bloqueDeCondiciones2:
    #Bloque de codigo True
else:
  #Bloque de codigo False
```

3- Sentencia condicional compuesta

```
#Un if compuesto es cuando queremos encadenar varias condiciones.
Es una forma de evitar muchos if anidados

if bloqueDeCondiciones1:
  #Bloque de codigo True
elif bloqueDeCondiciones2:
  #Bloque de codigo True
else:
  #Bloque de codigo False
```

ESTRUCTURA DE REPETICION CONDICIONAL WHILE

El WHILE repite la secuencia de acciones muchas veces hasta que alguna condición se evalúa como False . La condición se da antes del cuerpo del bucle y se comprueba antes de cada ejecución del cuerpo del bucle.

1-While

```
#El bloque de condiciones puede estar formada por una o mas
condiciones, separadas por un operador logico
while bloqueDeCondiciones:
  #Bloque de codigo
```

2-While else

```
#Se puede escribir una instrucción else después de un cuerpo de
bucle que se ejecuta UNA SOLA VEZ después del final del bucle
while
  #Bloque de codigo
else:
  #Bloque de codigo
```

ESTRUCTURA DE REPETICIÓN INCONDICIONAL FOR

En Python, la estructura FOR declara una variable local al bloque de iteración (fuera de él ya no existe) que por cada “pasada” adquiere el valor de c/elemento de la colección que esté iterando.

1. Repetición tradicional (n iteraciones)

```
1   #La función range() genera un arreglo interno de n elementos (1-n)
2   # range(5) = [1, 2, 3 ,4 ,5]
3
4   for indice in range(n):   #siendo n la cantidad de iteraciones
5       #bloque de código
```

2. Iteración de listas/tuplas/conjuntos/strings

2.1. Iteración de sobre cada elemento de la colección.

```
10  #En cada iteración, la variable local "e"
11  # toma el valor de cada elemento de la lista
12
13  for e in lista:
14      #bloque de código
```

2.2. Iteración de sobre cada elemento de la colección, con su índice.

```
19  #En cada iteración, la variable local "indice" obtiene
20  # el INDICE del elemento de la lista, cuyo VALOR toma "e"
21
22  for indice,e in enumerate(lista):
23      #bloque de código
```

ESTRUCTURA DE REPETICIÓN INCONDICIONAL FOR

En Python, la estructura FOR declara una variable local al bloque de iteración (fuera de él ya no existe) que por cada “pasada” adquiere el valor de c/elemento de la colección que esté iterando.

3. Iteración de diccionarios

3.1. Iteración de sobre cada elemento de la colección.

```
1   # diccionario = {"clave1":"valor1", "clave2":"valor2"}
2
3   #En cada iteración, la variable local "v" toma el VALOR
4   # de cada elemento del diccionario
5
6   for v in diccionario:
7       #bloque de código
```

3.2. Iteración de sobre cada elemento de la colección, con su clave.

```
11   #En cada iteración, la variable "clave" obtendra la CLAVE,
12   # y "valor" su VALOR respectivo
13
14  for clave,valor in diccionario.items():
15      #bloque de código
```

FUNCIONES

Una función es un bloque de código con un nombre asociado, que recibe cero o más argumentos como entrada, sigue una secuencia de sentencias, la cuales ejecuta una operación deseada y devuelve un valor y/o realiza una tarea, este bloque puede ser llamados cuando se necesite.

1-Definicion de una Funcion

```
#El nombre identifica la funcion
#La lista de parametros son los parametros que la funcion puede recibir
def nombreDeLaFuncion(listaDeParametros):
  """Documentacion de funcionalidades"""
  #Bloque de codigo
  return expresion

#Invocacion de la funcion
nombreDeLaFuncion(ListadeParametros)
```

2- Invocacion de funcion

```
#Teniendo una funcion definida por:
def suma(a,b):
  """Suma los parametros a y b"""
  return a+b

#La invocacion la puedo hacer de 2 formas distintas:
#Pasando los parametros en orden
suma(15,10)
#Pasando los parametros por nombre
suma(b=10,a=15)
```

3- Definicion de function con parametros inicializados

```
#Al momento de definir la funcion, puedo determinar valores iniciales para
la ejecucion

def suma(a=10,b=15):
  """
  Esta funcion siempre devolvera el valor de la suma entre 10 y 15, sin
  importar los valores que se pasen por parametro
  """
  return a+b

#Invocacion de la funcion
suma(30,15)
```