

Tipos de datos abstractos (TAD) compuestos



SINTAXIS Y SEMÁNTICA DEL
LENGUAJE



Tipos de datos abstractos (TAD) compuestos

- Temario:
- Diseño de TADs compuestos en Python
- TADs en ADA
 - Uso de packages
 - TADs genéricos



TADs Compuestos en Python

Un TAD compuesto es aquel TAD que en su estructura interna utiliza uno o más TADs ya existentes.

De esta forma el TAD puede ser doble, triple, cuádruple, etc., dependiendo de cuántos TADs intervengan en su diseño.





TADs Compuestos en Python

Veamos como ejemplo el diseño del TAD Curso, que contiene la información de todos los alumnos que forman parte del mismo. Esto significa que vamos a construir el nuevo TAD usando el TAD alumno ya diseñado.

Vamos a llevar a cabo los 3 pasos del diseño y a desarrollar un programa de aplicación.



TAD Curso de alumnos

Especificación

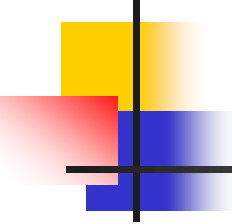
El TAD Curso sirve para almacenar una colección de alumnos

Sus operaciones elementales son:

- Crear el curso vacío
- Agregar un alumno
- Eliminar un alumno
- Consultar la cantidad total de alumnos del curso
- Ver los datos de un alumno determinado
- Consultar si existe un alumno dado en el curso

Observación: acá las operaciones básicas no son las mismas que para alumno Por qué???

• • •



TAD Curso de alumnos

Especificación

#El TAD Curso sirve para almacenar una colección de alumnos

crearCurso():

#Crea un curso vacío

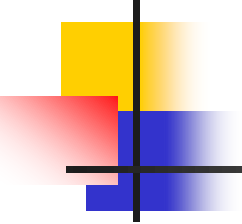
agregarAlu(cur,a):

#Agrega al alumno a al curso

eliminarAlu(cur,a):

#Elimina al alumno a del curso





TAD Curso de alumnos. Especificación

existeAlu(cur, a):

#Retorna True o False si el alumno a pertenece al curso

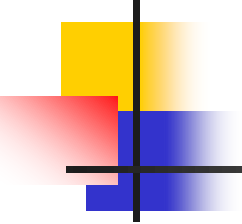
tamano(cur):

#Retorna la cantidad de alumnos del curso

recuperarAlu(cur,i):

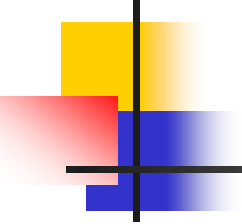
#Retorna el alumno de la posición iésima





TAD Curso de alumnos. Aplicación.

Utilizando la interfaz de usuario recién definida, vamos a desarrollar un programa de aplicación que crea un curso, le carga alumnos y luego imprime el listado de alumnos con promedio mayor a 8, recupera los datos del 2do alumno y lo elimina.



TAD Curso de alumnos. Aplicación.

```
from TadAlu import *
```

→ suponemos creados y guardados en el ambiente Python el módulo correspondiente a cada TAD. Así de cada módulo se importan todas las operaciones

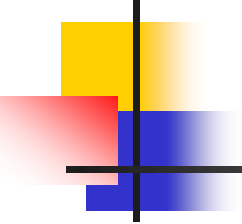
```
from TadCurso import *
```

```
c=crearCurso()
```

```
#Crea el curso vacío
```

TAD Curso de alumnos. Aplicación.

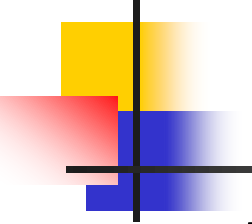
```
#Cargar el curso con 3 alumnos
for i in range(1,4):
    a=crearAlu()
    n=input("Ingrese un nombre ")
    l=input("Ingrese un legajo ")
    p=float(input("Ingrese un promedio "))
    cargarAlu(a,n,l,p)
    agregarAlu(c,a)
```



TAD Curso de alumnos. Aplicación.

```
#Imprime listado de alumnos
print ("Imprime los alumnos con promedio
mayor que 8 ")
tam= tamaño(c )
for i in range(0,tam):
    a=recuperarAlu(c,i)
    if (verProm(a) > 8):
        print ("Nombre: ",verNom(a))
        print ("Legajo : ", verLega(a))
        print ("Promedio: ", verProm(a))
        print ("_____"*3)
```

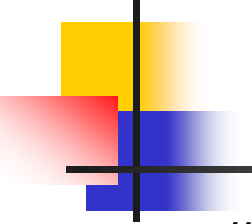
TAD Curso de alumnos. Aplicación.



```
#Recupera e imprime los datos del segundo alumno
print ("Imprime los datos del 2do alumno")
a=recuperarAlu(c,1)
print ("Imprime los datos de un alumno")
print ("Nombre: ",verNom(a))
print ("Legajo : ", verLega(a))
print ("Promedio: ", verProm(a))
print ("_____"*3)
```

```
#Elimina el alumno recuperado
print ("Elimina el alumno recuperado")
eliminarAlu(c,a)
```

TAD Curso de alumnos. Aplicación.



```
#Imprime los alumnos que quedan en el curso
print "Imprime los datos de los alumnos del curso"
tam=tamano( c)
for i in range(0, tam):
    a=recuperarAlu(c,i)
    print ("Imprime los datos de un alumno")
    print ("Nombre: ",verNom(a))
    print ("Legajo :", verLega(a))
    print ("Promedio", verProm(a))
    print ("_____"*3)
```



TAD Curso de alumnos

Diseño de la estructura interna

Si volvemos al diseño:

Vamos a elegir como estructura interna el tipo de datos lista.

Así entonces se define:

```
curso=[ ]
```

La lista permitirá agregar tantos alumnos como desee cargar el usuario



TAD Curso de alumnos. Implementación.

TAD Curso

```
def crearCurso():  
    #Crea un curso vacio  
    curso=[]  
    return curso
```

```
def agregarAlu(curso,a):  
    #Agrega un alumno al curso  
    curso.append(a)
```



TAD Curso de alumnos. Implementación.

```
def eliminarAlu(curso,a):  
    #Elimina un alumno del curso  
    curso.remove(a)
```

```
def recuperarAlu(curso,i):  
    #Retorna el alumno de la posicion iesima  
    return curso[i]
```

```
def tamaño(curso):  
    #Retorna la cantidad de alumnos del curso  
    return len(curso)
```


TAD Curso de alumnos. Implementación.



```
def existeAlu(curso, a):  
    #Retorna True o False si el alumno a pertenece al curso  
    return a in curso
```

Toda esta implementación se copia en un módulo Python y se guarda (save) como `TadCurso.py`, de forma tal que después pueda ser incluido en cualquier programa de aplicación Python.



TADs en ADA

Uso de paquetes (packages)

Tads genéricos.



Lenguaje ADA

Este lenguaje de programación tiene herramientas para implementar tipos de datos abstractos y proveer tanto ocultamiento como encapsulamiento de forma eficiente, a través de los packages.

The logo consists of a yellow square, a red square, and a blue square arranged in a 2x2 grid, with a black crosshair overlaid. The word "ADA" is written in blue to the right of the graphic.

ADA

Todo programa en ADA tiene 4 partes:

- Especificación de contexto (uso de librerías ej. `with ada_io`);
- Especificación del programa (encabezado)
- Parte declarativa
- Cuerpo



Ejemplo

1)

with ada_io;

use ada_io;

procedure Doble is

 x, y: integer;

begin

 get(x);

 y:=x+2;

 put(y);

end Doble;

} Especificación del contexto

→ Encabezado

→ Parte declarativa

} cuerpo



Ejemplo

2)

use ada_io;  Especificación del contexto

function pepe (n:integer) return integer is  Encabezado

z: integer;  Parte declarativa

begin

z:= 2 + n

return z

end pepe;



cuerpo



Parámetros en ADA

- Por defecto son por copia → in
- Pueden ser por resultado → out
- Por valor resultado → in out, si el parámetro es un tipo primitivo
- O por referencia → in out, si no es un tipo primitivo



Parámetros en ADA

- Pasaje por resultado:

en el momento de la llamada el parámetro actual se liga al formal pero no le pasa valor inicial. Cuando el subprograma finaliza le pasa el valor final del parámetro formal al actual modificándolo de forma permanente.

```
n=3;  
m=4;  
modi(n,m);  
put(n,m);
```

→ 3 8
m vale 8 de ahora en adelante

```
procedure modi(y: integer, x: out integer) is  
begin  
  x=2;  
  y=y*2;  
  x= x + y; → retorna este último valor  
end;
```




Parámetros en ADA

- Pasaje por valor resultado:

en el momento de la llamada el parámetro actual se liga al formal y se copia su valor.

El parámetro formal se trabaja como variable local al subprograma pero cuando éste finaliza su ejecución se pasa el valor final del parámetro formal al actual, que queda modificado.

```
n=3;  
m=4;  
modi(n,m);  
put(n,m);
```

→ 3 17

```
procedure modi(y: integer, x: in out integer) is  
begin  
  y=y+10;  
  x= x + y; → valor final de retorno  
end;
```



TAD's en ADA

- Encapsula mediante paquetes.
- Los paquetes se definen en archivos fuentes separados de la aplicación que los usa.
- El paquete tiene dos partes:
 - Especificación (pública) → nombre + protocolo + estruc. interna privada.
 - Cuerpo o body (privada) → contiene la implementación de las operaciones



TAD's en ADA

- La parte visible o pública del TAD es leída por el compilador cuando se compila la aplicación que lo usa
- La parte privada NO.
- Para usar un TAD debo incluirlo en la aplicación con `with` (calificado) o `use`.



Ejemplo TAD Persona

```
package Tpersona is
```

```
  type cadena is string(1..30);
```

```
  type persona is private;
```

```
  function vernom(p:persona) return cadena;
```

```
  function veredad(p:persona) return integer;
```

```
  procedure modedad(p: in out persona; nue:integer);
```

```
  procedure modnom(p: in out persona; otro: cadena);
```

```
  private
```

```
  type persona is record
```

```
    nom: cadena;
```

```
    edad: integer;
```

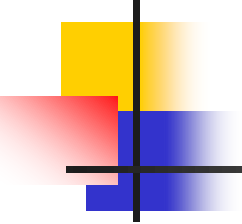
```
  endrecord;
```

```
end Tpersona;
```

**Este archivo es la
especificación**

**Esto se compila y genera un
archivo Tpersona.ads**

Este archivo es el body o cuerpo



```
package body Tpersona is
function vernom (p:persona) return cadena is
begin
    return p.nom;
end;
function veredad(p:persona) return integer is
begin
    return p.edad;
end;
procedure modedad(p: in out persona; nue: integer) is
begin
    p.edad:= nue
end;
.....
end Tpersona;  Se compila y genera un arch  Tpersona.adb
```



Ejemplo de uso del TAD

```
use Tpersona;
procedure Main is
  p1, p2:persona;
  e: integer;
begin
  put("ingrese edad");
  get( e );
  modedad(p1,e);
  ----
  e:=veredad(p1);
  put( e);
end;
```

zona de inclusión de TADs y librerías
cabecera del prog.aplicación
zona de declaraciones

cuerpo



Características particulares:

- En la parte pública de la especificación se pueden declarar tipos públicos, privados y limitados privados
 - A los datos públicos se los accede desde cualquier lugar.
 - A los datos privados solo los accedo por operaciones del TAD pero admiten asignación directa, comparación y desigualdad
 - Si el dato es limitado-privado solo se accede por operaciones del TAD.



Ejemplo TAD Persona

package Tpersona is

type cadena is string(1..30);

cadena es público

type persona is private;

persona es privado

function vernom(p:persona) return cadena;

function veredad(p:persona) return integer;

procedure modedad(p: in out persona; nue:integer);

procedure modnom(p: in out persona; otro: cadena);

private →

esto limita el acceso a la est.interna

type persona is record

solo se puede manipular via operaciones

nom: cadena;

edad: integer;

endrecord;

end Tpersona;



TAD's genéricos

- Permite definir un parámetro o elemento genérico que se instancia en ejecución con un tipo de dato específico
- Admite uno o varios parámetros genéricos



TAD's genéricos

- Como ejemplo veamos el TAD pila genérica. En el programa de aplicación vamos a crear distintas pilas pasandole como parámetro un tipo de elemento diferente en cada caso (enteros, reales, libros, personas, etc.)



generic

type elemento is private;

package Tpila is

type pila is private;

function vacia(p:pila) return boolean;

procedure crear(p:in out pila);

procedure apilar(p: in out pila; e:elemento);

procedure desapilar(p: in out pila; e: in out elemento);

private

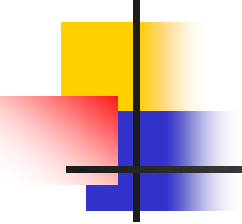
type pila is record

datos: array[1..30] of elemento;

tope: integer;

endrecord;

end Tpila;



```
package body Tpila is
  function vacia(p:pila) return boolean is
  begin
    if p.tope=0 then return true
      else return false
    end;
  .....
end Tpila;
```



Programa de aplicación

declare

package PilaEnt is new Tpila(integer);

package PilaReal is new Tpila(float);

.....

.....

use PilaEnt;

procedure Main is

 p1: pila;

.....

Acá se definen dos tipos de pilas distintos, una de elementos enteros y otra de reales.

Luego se crea una pila p1 de enteros (lo habilita el use)

