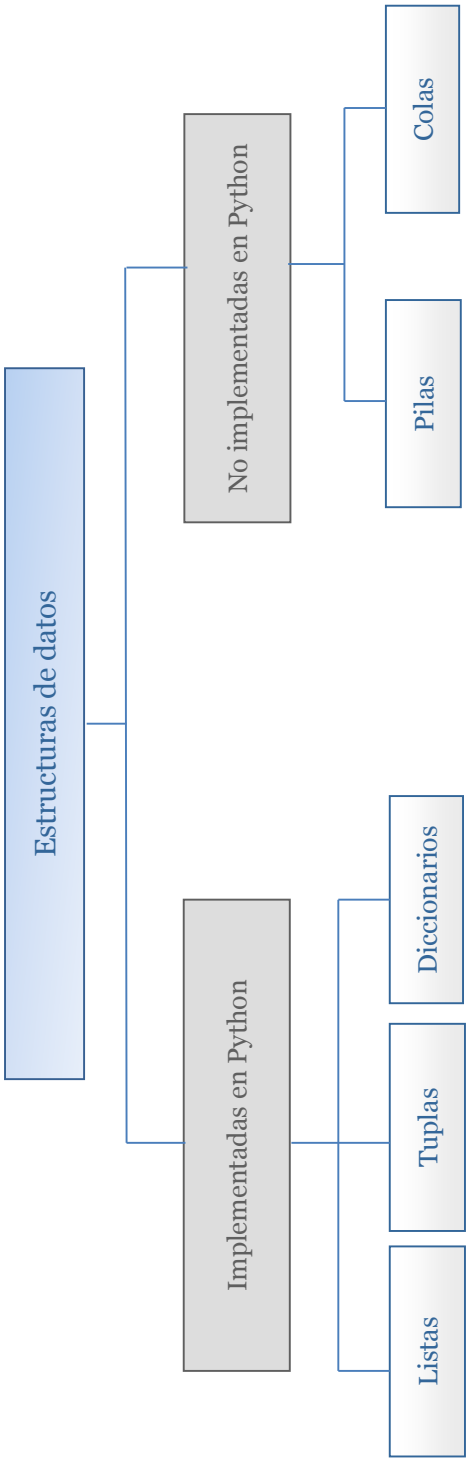
Estructuras de datos

TEMA 8

Esquema


Estructuras de datos
No implementadas en Python
Colas
Pilas
Implementadas en Python
Diccionarios
Tuplas
Listas

Ideas clave

8.1. ¿Cómo estudiar este tema?

Para estudiar este tema deberás comprender las **Ideas clave** expuestas en este documento, que se complementan con lecturas y otros recursos para que puedas ampliar los conocimientos sobre el mismo.

En este tema se estudian las principales estructuras de datos y su implementación en Python. Por tanto, los conceptos fundamentales de este tema son:

- **Listas**: definición, sintaxis y operaciones con Python.
- **Tuplas**: definición, sintaxis y operaciones con Python.
- **Diccionarios**: definición, sintaxis y operaciones con Python.
- **Pilas**: definición, operaciones y métodos.
- **Colas**: definición, operaciones y métodos.

8.2. Introducción

En este capítulo se estudian estructuras de datos, de las cuales solo se han estudiado previamente las listas. La razón es que las listas son estructuras muy importantes en casi todos los lenguajes de programación.

Sin embargo, las tuplas y diccionarios, que están implementadas en Python, no son tan comunes en otros lenguajes.

Las pilas y colas, que son estructuras muy utilizadas en programación no suelen venir implementadas. Sin embargo, Python dispone de métodos para que trabajar con ellas sea muy sencillo e intuitivo y no haya que implementar toda la estructura como se hace, por ejemplo en C.

Los arrays, que son estructuras que es muy habitual encontrar implementadas, no lo están en Python. Si se desea trabajar con ellos, debe instalarse en paquete Numpy. Como el uso de este paquete tiene interés fundamentalmente para programadores con

perfil técnico (que desean realizar operaciones con matrices, obtener estadísticas, etc.) no se va a dar en este curso.

8.3. Listas

Como ya se vio en el Tema 3, una **lista** es un **conjunto ordenado de elementos** que tiene una **estructura muy flexible**: permite la eliminación o la inserción de nuevos elementos de forma muy sencilla. Para definir una lista puede utilizarse la función `list()` o los corchetes `[]`. Como en casos anteriores, la función `list()` también se puede usar para transformar otro tipo de dato en lista, teniendo en cuenta que solo admite un argumento.

```
>>>> l = ["Esto", "es", 1, "lista"]
```

`list()` también transforma otros tipos de datos a lista

```
>>> list("Esto")
['E', 's', 't', 'o']
```

Pero si se quiere una cadena como elemento la sintaxis es

```
>>> list(("Esto", ))
['Esto']
```

Que solo admita un argumento implica

```
>>> a = "Esto"
>>> b="Aquello"
>>> list(a, b)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: list() takes at most 1 argument (2 given)
```

Se pueden anidar listas en listas en tantos niveles de profundidad como se quiera

```
>>>> anidadas = ["3", "-42", ["casa", [3, "arbol"], "AaSs"]]
>>>> anidadas[2][1][1][3]
'o'
```

ya que `anidadas[2]` devuelve el tercer elemento de la lista, en este caso `["casa", [3, "arbol"], "AaSs"]`. Como el siguiente índice es `[1]`, se toma el segundo elemento, es decir, la lista `[3, "arbol"]`. De ahí se vuelve a tomar el segundo elemento, `"arbol"`. Y con `[3]` devuelve el cuarto elemento de la cadena, `'o'`.

La manera de recorrer una lista y de extraer elementos de ella es completamente análoga a la ya explicada para las cadenas y a la que se verá a continuación en la tuplas. Del mismo modo, soportan los operadores de pertenencia `in` y `not in`, la concatenación con `+` y la repetición con `*`, así como sus respectivas asignaciones aumentadas y la función `len()` para saber la longitud de la lista.

Operaciones con listas

Sea L una lista, pueden utilizarse varios métodos con ella (Tabla 5.8)

Sintaxis	Descripción
`L.append(x)`	Añade el elemento `x` al final de `L`
`L.count(x)`	Devuelve la cantidad de veces que está `x` en `L`
`L.extend(a)`	Es equivalente a la asignación aumentada `L` +=a
`L.index(x, inicio, fin)`	Devuelve la posición de `x` en la sublista `L` [inicio:fin]
`L.insert(i, x)`	Inserta `x` en la posición i de `L`
`L.pop(i)`	Elimina y devuelve el elemento `L`[i]. Si se pasa sin argumentos, elimina el último
`L.remove(x)`	Elimina el elemento `x`
`L.sort()`	Ordena `L`

Tabla 5.8: Métodos de lista

Por ejemplo, si

```
>>>> L = [12, 356, -23, 5, -0.6]
>>>> L.append(8)
[12, 356, -23, 5, -0.6, 8]

>>> L.index(356, 1, 3)
1

>>> L.pop(5)
8
>>> L
[12, 356, -23, 5, -0.6]

>>> L.sort()
>>> L
[-23, -0.6, 5, 12, 356]
```

Listas por compresión

Si la lista no es muy grande no hay ningún problema en asignar los valores manualmente, como se ha hecho en los ejemplos anteriores, pero para listas grandes, esto puede ser muy poco operativo. Veamos distintos modos de crear listas mediante la programación.

Si se quiere hacer una lista de números enteros (por ejemplo 30), la forma más sencilla es:

```
>>> list(range(30))
```

Para otro tipo de listas pueden emplearse estructuras selectivas y repetitivas, descritas en el Tema 4. Aunque la implementación de estas estructuras en Python se explica en el apartado 5.5, vamos a adelantar algo de la sintaxis para poder definir listas por compresión.

Si, por ejemplo se quiere definir una lista que contenga todos los años bisiestos desde 1800 hasta ahora, se puede definir el bucle:

```
bisiesto = []
for año in range(1800, 2014):
    if (año % 4 == 0 and año % 100 !=0) or (año % 400 == 0):
        bisiesto.append(año)
```

Recordemos que los años bisiestos son todos aquellos que son múltiplos de cuatro, excepto los múltiplos de 100 que no son múltiplos de 400. Por eso, la primera condición de la sentencia `if` está puesta para incluir a todos los múltiplos de 4 que no son múltiplos de 100 y la segunda, para incluir a los múltiplos de 400.

Lo que hace el bucle es recorrer todos los números de 1800 a 2014 (sentencia `for`) y, cada vez que una de las condiciones que se evalúan sea cierta, añadir al final de bisiesto el año correspondiente.

Otra manera de expresar lo mismo de manera más comprimida es

```
bisiesto = [y for y in range(1800, 2014)
    if (y % 4 == 0 and y % 100 != 0) or (y % 400 == 0)]
```

Que sigue la estructura general

```
[<expresión> for <elemento> in <bucle> si <condición>]
```

Por ejemplo, supongamos que una empresa que fabrica camisas quiere generar una lista con todas las etiquetas posibles. Hay camisas para hombre y para mujer, de tallas "`S`", "`M`", "`L`" o "`XL`" y en los colores "`Blanco`", "`Azul`" y "`Gris`"

```
etiquetas = []
for sexo in "HM":         #Hombre, Mujer
    for talla in "SMLX":  #Pequeña, Mediana, Grande, Extra Grande
        for color in "BAG":     #Blanco, Azul, Gris
            etiquetas.append(sexo + talla + color)
```

O, equivalentemente,

```
etiquetas = [ s + t + c for s in "HM" for t in "SMLX" for c in "BAG"]
```

Supongamos que se quiere eliminar la etiqueta de color gris para hombre, en todas las tallas. Simplemente habrá que añadir a la sentencia anterior

```
etiquetas = [ s + t + c for s in "HM" for t in "SMLX" for c in "BAG"
    if not (s == "H" and c == "G")]
```

Valores por defecto

Cuando se defina una función que tenga una lista como argumento y se quiera definir un valor por defecto, hay que tener cuidado. Esto es válido en general para las estructuras de datos mutables (como por ejemplo, las listas), ya que se puede cometer un error bastante difícil de encontrar. Supongamos que se quiere crear una función con una lista como argumento, que tenga el valor de lista vacía `[]` por defecto. En principio, se podría pensar en un código como

```
def funcion(lista = []):
    <acciones>
return(lista)
```

El problema es que los valores por defecto se crean en el momento en que se crea la función, esto es, cuando se ejecuta `def funcion(lista = [])`. Por tanto, no se creará una nueva lista vacía cada vez que se ejecute la función sin parámetros, sino que se tomará cada vez esa primera lista que se creó con la función. Para evitar esta situación un código correcto sería

```
def funcion(lista = None):
    if lista == None
        lista = []
    <acciones>
return(lista)
```

De esta manera, se creará una nueva lista cada vez que se invoque a la función. Esta metodología debería aplicarse siempre que se quieran crear valores por defecto vacíos en objetos como listas, tuplas o diccionarios. En datos inmutables como cadenas, es irrelevante. Otra forma, ligeramente más abreviada de implementar este código es

```
def funcion(lista = None):
    lista = [] if lista == None else lista
<acciones>
return(lista)
```

8.4. Tuplas

Las tuplas son secuencias de **datos ordenados e inmutables**. Se pueden definir utilizando paréntesis y escribiendo los elementos de la tupla entre comillas

```
>>>> t = ("Ejemplo", "de", 1, "tupla")
```

Aunque también se puede prescindir de los paréntesis

```
>>> t = "Ejemplo", "de", 1, "tupla"
```

Al ser una secuencia podemos preguntar por su longitud con el operador `len()`

```
>>>> len(t)
4
```

Y al ser ordenado se puede acceder a sus elementos con el operador `[]`, como se hacía en las cadenas

```
>>>> t[3]
'tupla'
```

Como puede observarse, no es necesario que todos los elementos de la tupla sean del mismo tipo. Para saber el tipo de dato se utiliza la función `type()`

```
>>>> type(t)
<class 'tuple'>

>>>> type(t[1]), type(t[2])
(<class 'str'>, <class 'int'>)
```

También se pueden crear tuplas mediante la utilización de `tuple()`. Hay que tener en cuenta que no acepta más de un argumento

```
>>>> t = tuple()
>>>> t
()
>>>> len(t)
0

>>>> tuple("cadena")
('c', 'a', 'd', 'e', 'n', 'a')
```

Si se quiere crear la tupla `‘cadena’`, la sintaxis es

```
>>>> ("cadena",)
(‘cadena’)

>>>> l = ["Lista", "que", "se", “convierte", "en", "tupla"]
>>>> tuple(l)

>>>> b = "otra cadena"
>>>> tuple(a, b)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: tuple() takes at most 1 argument (2 given)
```

Que las tuplas sean inmutables implica que no se puede modificar ninguno de sus elementos:

```
>>>> t[3] = "casa"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Por tanto, la principal diferencia entre listas y tuplas es que las tuplas son inmutables, las listas se pueden modificar. Esto implica que la elección entre una lista y una tupla dependerá de lo que se quiera hacer: si se quieren almacenar datos que se sabe que pueden cambiar, por lo general será mejor una lista, ya que la tupla no puede modificarse, sino volverse a crear.

Pero esta diferencia es más profunda de lo que parece: **en las tuplas, la posición que ocupa cada elemento, es significativa**, precisamente porque es inmutable. En otras palabras, la posición tiene **valor semántico** ya si en una tupla está almacenado un año en la primera posición y un apellido en la cuarta, sabemos que siempre será así. Sin embargo, en una lista, no es que no preocupe por la posición de los elementos (de hecho, es un aspecto fundamental para poder operar con listas), sino que cada valor almacenado en ella es estructuralmente equivalente.

Operaciones con tuplas

Sea la tupla

```
>>>> t = ("Encantado", ",", "señor", "Hueso")
```

t[-4]	t[-3]	t[-2]	t[-1]
'Encantado'	**','**	**'señor'**	**'Hueso'**
t[0]	t[1]	t[2]	t[3]

Figura 5.2: Posiciones de una tupla

Los índices en una tupla funcionan igual que en una cadena y, tal como ocurre también con ellas, pueden utilizarse los operadores `+` (concatenación) y `*` (repetición), así como sus respectivas asignaciones aumentadas, `+=` y `*=`. Puede resultar raro que, tanto para las cadenas como para las tuplas, que son inmutables, estén definidas las asignaciones aumentadas. En realidad, en ambos casos, Python crea una tupla nueva.

Para localizar elementos de la tupla se emplea también `[]`

```
>>>> color = ("Rojo", "Azul", "Verde", "Morado")
>>>> color[3]
'Morado'
>>>> color[-3:]
('Azul', 'Verde', 'Morado')
```

Las tuplas ofrecen dos métodos, `.count()` y `.index()`, por ejemplo

```
>>>> color.count("Rojo")
1
```

Devuelve el número de veces que aparece la cadena "`Rojo`" en la tupla `color`. Nótese que

```
>>>> color.count("r")
0
```

Ya que, aunque `r` forma parte de varias cadenas de la tupla `color`, `r` no es un elemento de `color`.

```
>>> color.index("Rojo")
0
```

Ya que este método devuelve la posición de la cadena "`Rojo`" en la tupla `color`.

También se pueden emplear los operadores de pertenencia `in` y `not in`:

```
>>> "Rojo" in color
True
>>> "Rojo" not in color
False
```

Si se quiere crear una tupla nueva con el color "`Amarillo`" en la segunda posición

```
>>>> color[:2] + ("Amarillo",) + color[2:]
('Rojo', 'Azul', 'Amarillo', 'Verde', 'Morado')
```

Ya que la expresión

```
>>>> color[:2], "Amarillo", color[2:]
(('Rojo', 'Azul'), 'Amarillo', ('Verde', 'Morado'))
```

Produce en realidad una tupla de tres elementos que contiene dos tuplas de dos elementos.

Como ocurre con las listas, se pueden anidar tuplas en tuplas en tantos niveles de profundidad como se quiera

```
>>>> anidadas = 45.3, "kg", ("A", (0, 1), "entero")
>>>> anidadas[3][2][4]
'r'
```

8.5. Diccionarios

Un **diccionario** es una **colección desordenada de pares de valores y claves**. La información que interesa guardar son los valores y las claves se emplean para poder encontrar los valores (recordemos que es un conjunto desordenado).

En la Figura 5.3 pueden verse cómo los valores "alpha", "omega" y "gamma" están asociados a las claves "a", "o" y "g", respectivamente

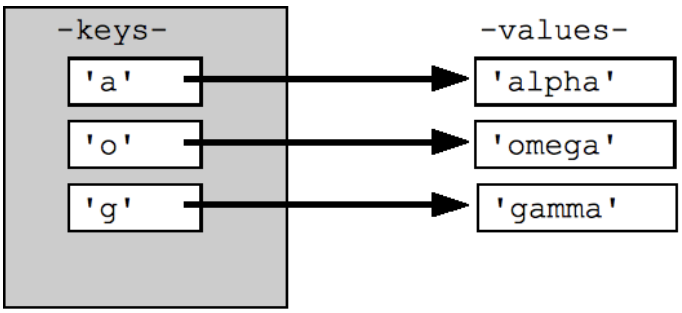


Figura 5.3: Representación de valores y claves en un diccionario. Fuente: http://cs.stanford.edu/

Los diccionarios son estructuras de datos que pueden resultar muy útiles. En una lista, para localizar un elemento, se utiliza el hecho de que es una estructura ordenada y se busca a través de su posición. Un diccionario podría hacer lo mismo, si las claves que se asignan a cada valor del conjunto fueran números. Por ejemplo, si al valor "omega" se le asigna la clave 0, si al valor "alpha" se le asigna la clave 1 y al valor "gamma" se le asigna la clave 2, el diccionario sería como una lista. Pero al poder emplear como clave, además de números, cadenas o tuplas, la estructura es mucho más flexible.

El diccionario, como estructura de datos, es **mutable**, es decir, pueden modificarse los valores que contiene. Lo que no es mutable es la clave de cada valor, por eso solo pueden ser números, cadenas o tuplas.

Para definir un diccionario pueden utilizarse las llaves {} o la función dict(). Las siguientes sentencias producen el mismo diccionario

```
d1 = {"Año": 1998, "Apellido": "Fernández", "Curso": 4}
d2 = ({"Año": 1998, "Apellido": "Fernández", "Curso": 4})
d3 = dict(Año = 1948, Apellido = "Fernández", Curso = 4)
d4 = dict(zip(("Año", "Apellido", "Curso"), (1998, "Fernández", 4)))
d5 = dict([("Año", 1998), ("Apellido", "Fernández"), ("Curso", 4)])
```

Como puede verse, d4 y d5 crean un diccionario a partir de tuplas y listas, respectivamente.

Operaciones con diccionarios

Algunas de las operaciones que pueden realizarse con diccionarios se muestran en la Tabla 5.9 para un diccionario `d`.

Sintaxis	Descripción
`d.clear()`	Elimina todos los elementos de `d`
`d.get(k)`	Devuelve el valor asociado a la clave `k`
`d.items()`	Devuelve todas las parejas (clave, valor) de `d`
`d.values()`	Devuelve todos los valores de `d`

Tabla 5.9: Algunas operaciones con diccionarios

Por ejemplo,

```
>>> d1 = {"Año": 1998, "Apellido": "Fernández", "Curso": 4}
>>> d1.get("Año")
1998

>>> d1.items()
[('A\xc3\xb1o', 1998), ('Curso', 4), ('Apellido', 'Fern\xc3\xa1ndez')]
```

Nótese que los caracteres que no pertenecen al inglés (ñ y á) se muestran con la representación Unicode. En general, es preferible evitar este tipo de caracteres, en la medida de lo posible.

```
>>> d1.values()
[1998, 4, 'Fern\xc3\xa1ndez']
```

Como ocurre con las listas, existen diccionarios por compresión que evitan tener que introducir manualmente todos los elementos del mismo.

8.6. Pilas

Las pilas y colas son un tipo particular de estructura lineal (lista, secuencia) que, debido a sus numerosas aplicaciones en programación, se estudian por separado.

Una **pila** es una lista de elementos que se caracteriza porque las operaciones de inserción y eliminación se realizan por un extremo de la lista, que recibe el nombre de cima (Figura 8.1).

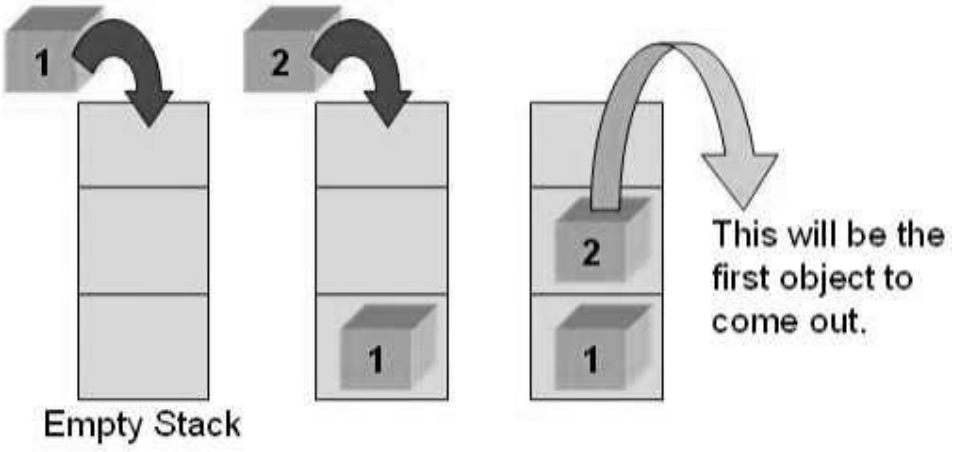


Figura 8.1: Representación de una pila. Fuente: http://www.puntopeek.com/

Supongamos que tenemos una pila `P` que contiene los elementos `a`, `b`, `c`, `d`, `e`. Lo representaremos como `P = [a, b, c, d, e]`. Se dice que `a`, que es el elemento más inaccesible de la pila, está en el **fondo** de la misma y que `e`, que es el más accesible, está en la **cima**.

Si se quisiera añadir un nuevo elemento a la pila, `f`, se haría a continuación de `e`. Si, posteriormente, se quisiera eliminar un elemento de la pila, se debería eliminar `f`. Es decir, el último elemento en entrar es el primero en salir. Esta estructura también se conoce por el acrónimo **LIFO** (del inglés, Last In First Out).

Un ejemplo cotidiano de pila es un montón de platos apilados. Si se quiere añadir uno nuevo, debe hacerse sobre el último plato de la pila (cima) y el primer plato que se extrae es el que está en la cima, es decir, el último que se añadió.

Otro ejemplo que afina más es una caja llena de libros. Los libros se apilan exactamente igual que los platos, así que también mantienen una estructura LIFO. Pero además, dado que no se pueden atravesar las paredes de la caja, es completamente imposible pensar en otra manera de añadir o eliminar objetos y, por el mismo motivo, tampoco podemos contar cuántos libros hay en la caja. La única posibilidad sería contarlos mientras los vamos extrayendo hasta dejar la caja vacía.

Por último, hay otro ejemplo que se vio de refilón en el Tema 6, cuando se hablaba de las llamadas que hace el programa principal a las distintas funciones. Si no hay ejecución concurrente dentro del programa, solo se puede estar ejecutando una función. Por eso, cuando se ejecuta un programa que llama a una función, el programa no se sigue ejecutando hasta que la función no haya terminado y devuelva en control al programa principal.

En general, las pilas se emplean cuando se quiere recuperar el último dato o información que se haya generado.

Las operaciones asociadas a las pilas son:

- Crear la pila. Crea una pila vacía.
- Añadir elementos a la pila (apilar). Añade un elemento sobre la cima.
- Eliminar elementos de la pila (desapilar). Elimina el elemento que está en la pila.
- Comprobar si la pila está vacía. Esta operación es útil hacerla, por ejemplo, cuando se quieren eliminar elementos de la pila, para evitar que se produzca un error.
- Consultar el último. Consultar el elemento de la cima.

Con todas estas operaciones debidamente implementadas, puede trabajarse con una pila.

Veamos ahora cómo trabajar con pilas en Python. En realidad es muy sencillo, ya que las listas incorporar los métodos predefinidos `append` y `pop` que hacen que las operaciones de extracción e inserción sean triviales. Aunque las pilas se van a definir sobre listas (por lo que, teóricamente, será posible acceder a cualquier elemento de la misma), vamos a considerar que solo se pueden realizar las operaciones definidas para pilas.

Lo que vamos a hace es definir la clase pila, en la que las operaciones serán los métodos. Una manera de hacerlo es:

```
class Pila():
    def __init__(self):
        self.items = []                #Crea un pila vacia

    def vacia(self):                   #Comprueba si la pila esta vacia
        if self.items == []:
            return True
        else:
            return False

    def apilar(self, valor):           #Añade un elemento a la pila
        self.items.append(valor)

    def desapilar(self):        #Elimina un elemento de la pila
        if self.vacia():
            return None
        else:
            return self.items.pop()


    def ultimo(self):                  #Devuelve el último elemento
        if self.vacia():
            return None
        else:
            return self.items[-1]
```

Comprobamos que funciona según lo esperado:

```
>>> p = Pila()
>>>
>>> p.ultimo()
>>> p.vacia()
True
>>> p.desapilar()
>>> p.apilar(3)
>>> p.apilar(4)
>>> p.apilar(1)
>>> p.ultimo()
1
>>> p.vacia()
False
>>> p.desapilar()
1
>>> p.ultimo()
4
```

Como puede comprobarse, cuando se crea la pila, como se crea vacía, los métodos `ultimo` y `desapilar` no devuelven nada. Se han ido apilando elementos hasta obtener la pila `p = (3, 4, 1)`. Como `1` ha sido el último elemento en entrar es el primero en salir.

8.7. Colas

Las **colas** son estructuras de datos lineales en las que las estructuras de entrada y salida de los datos se realizan por extremos opuestos (Figura 8.2)

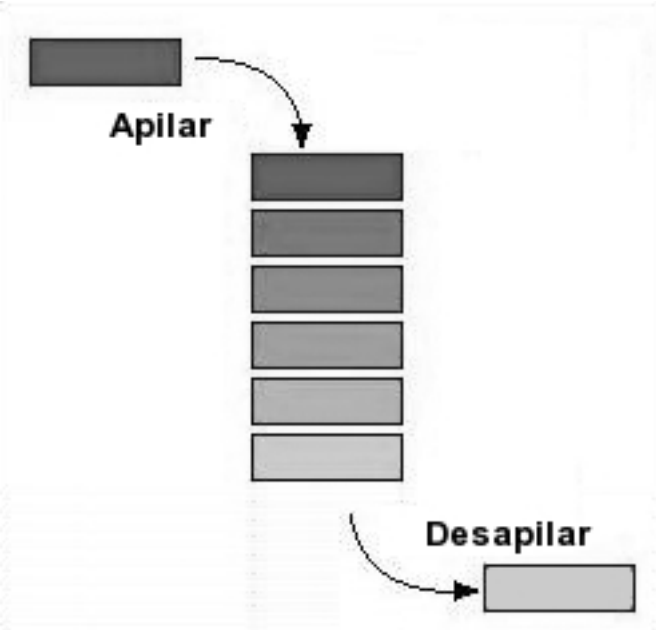


Figura 8.2: Representación de una cola.
Fuente: http://estructuradedatos10111248.blogspot.com.es/

Supongamos que tenemos una cola `Q` que contiene los elementos `a`, `b`, `c`, `d`, `e`. Lo representaremos como `Q = (a, b, c, d, e)`. Se dice que `a`, que es el elemento más accesible de la pila, está en el **frente** de la misma y que `e`, que es el más inaccesible, está en el **final**.

Si se quisiera añadir un nuevo elemento a la pila, `f`, se haría a continuación de `e`. Si, posteriormente, se quisiera eliminar un elemento de la pila, se debería eliminar `a`. Es decir, el primer elemento en entrar es el primero en salir. Esta estructura también se conoce por el acrónimo **FIFO** (del inglés, First In First Out).

La Figura 8.3 representa diferencia entre una pila y una cola.

Un ejemplo cotidiano sería la cola que se forma, por ejemplo, para comprar entradas para el cine. El primero que llega es el primero en ser atendido.

Otro ejemplo, las colas de impresión. El ordenador controla el envío de documentos a una impresora con una cola: el primero que entra es el primero que sale y no envía un nuevo documento hasta que no haya finalizado la impresión del anterior.

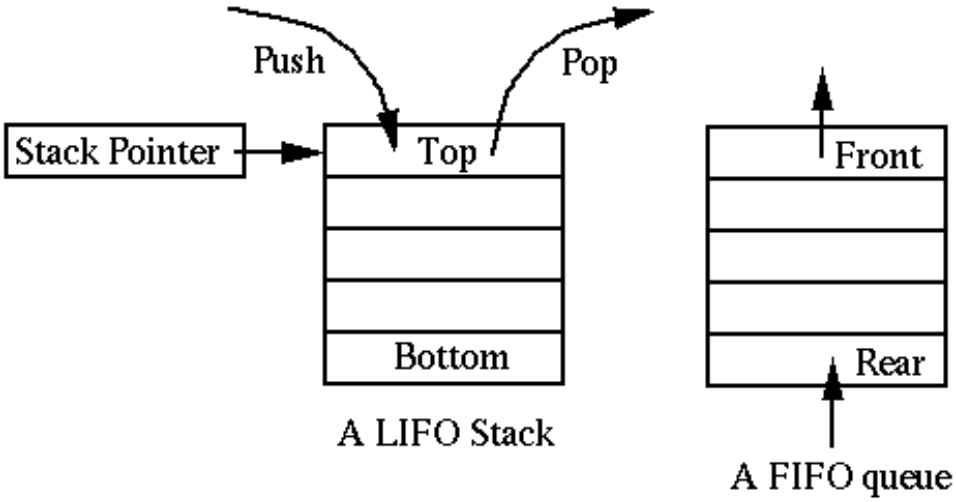


Figura 8.3: Comparación entre las estructuras de pila y cola.
Fuente: http://www.programmingspark.com/

Análogamente, las operaciones asociadas a las colas son:

- Crear la cola. Crea una cola vacía
- Añadir elementos a la cola. Añade un elemento al final de la cola
- Eliminar elementos de la pila. Elimina el elemento que está en el frente de la cola
- Comprobar si la cola está vacía. Esta operación es útil hacerla, por ejemplo, cuando se quieren eliminar elementos de la cola, para evitar que se produzca un error
- Consultar el primer elemento. Consulta el elemento del frente de la cola.

En una primera solución, todo se puede realizar de forma completamente análoga a como se ha procedido con las pilas. Así se puede definir la clase:

```
class Cola():
    def __init__(self):
        self.items = []
    def vacia(self):
        if self.items == []:
            return True
        else:
            return False
    def introducir(self, valor):
        self.items.append(valor)
    def eliminar(self):
        if self.vacia():
            return None
        else:
            return self.items.pop(0)
    def primero(self):
        if self.vacia():
            return None
        else:
            return self.items[0]
```

Nótese que lo único que ha cambiado es que se extrae el primer elemento, que también es el elemento sobre el que se realiza la consulta.

Si no se quieren hacer muchas operaciones sobre la cola, esta implementación funciona sin problema, como puede comprobar el alumno. El problema es que la operación de eliminar el primer elemento de una lista (`pop(0)`) no está optimizada en Python, por lo que se recomienda utilizar la clase `deque`, del módulo `collections`, en concreto el método `popleft()`, que elimina el primer elemento. Así, el código resulta

```
from collections import deque

class Cola():
    def __init__(self):         #Se crea un elemento de la
        self.items = deque()           #clase deque vacio (cola vacia)
    def vacia(self):
        if self.items == deque([]):    #Ojo,   hay   que   cambiar   la
condición
            return True
        else:
            return False
    def introducir(self, valor):
        self.items.append(valor)
    def eliminar(self):
        if self.vacia():
            return None
        else:
            return self.items.popleft()      #Eliminamos    de    forma
óptima
    def primero(self):
        if self.vacia():
            return None
        else:
            return self.items[0]
```

Lo + recomendado

No dejes de leer…

Estructura tipo *range*

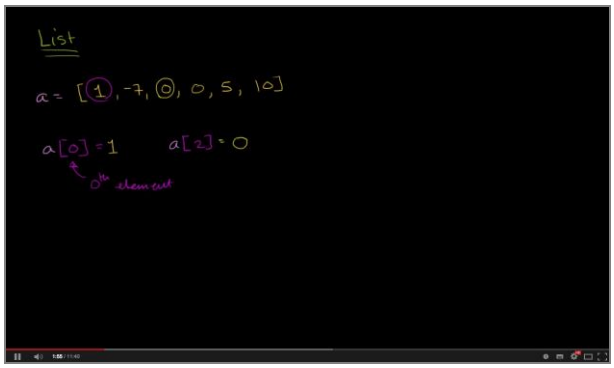
Documento de la biblioteca de Python sobre estructuras tipo *range*.

Accede al documento a través del aula virtual o desde la siguiente dirección web:
https://docs.python.org/dev/library/stdtypes.html#range

No dejes de ver…

Python lists

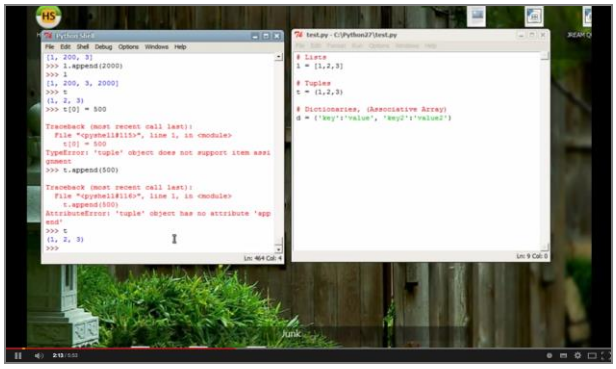
En el siguiente enlace se explican la sintaxis y operaciones básicas con listas en Python.



Accede al vídeo a través del aula virtual o desde la siguiente dirección web:
https://www.youtube.com/watch?v=zEyEC34MY1A

Python tuples and dictionaries

En el siguiente enlace se explican la sintaxis y operaciones básicas con tuplas y diccionarios en Python.



Accede al vídeo a través del aula virtual o desde la siguiente dirección web:
https://www.youtube.com/watch?v=qWY9_GZm4f4

+ Información

A fondo

Data structures

Descripción de y uso de las estructuras de datos Python.

Accede al documento a través del aula virtual o desde la siguiente dirección web:
https://docs.python.org/2/tutorial/datastructures.html

Dictionaries, Oh Lovely Dictionaries

Zed, A. (s. f.) Learn Python The Hard Way. Recuperado de http://learnpythonthehardway.org/book/index.html

Descripción del uso y ventajas de los diccionarios en Python.

Accede al documento a través del aula virtual o desde la siguiente dirección web:
http://learnpythonthehardway.org/book/ex39.html

Conjuntos en Python

Conti, J, (2 d noviembre de 2009). Conjuntos en Python. [Entrada en un blog]. Recuperado de http://www.juanjoconti.com.ar/2009/11/02/conjuntos-en-python/

Descripción del uso y ventajas de los conjuntos en Python.

Accede al documento a través del aula virtual o desde la siguiente dirección web:
http://www.juanjoconti.com.ar/2009/11/02/conjuntos-en-python/

Tuplas en Python

Descripción del uso y ventajas de las tuplas en Python.

Accede al documento a través del aula virtual o desde la siguiente dirección web:
http://www.dotnetperls.com/tuple-python

Bibliografía

Joyanes, L. (2008). *Fundamentos de la programación. Algoritmos y Estructura de Datos (4ª Ed)*. Madrid: McGraw-Hill.

Summerfield, M. (2009). *Programación en Python 3*. Madrid: Pearson Educación.

Valls, J. M. & Camacho, C. (2004). *Programación estructurada y algoritmos en PASCAL*. Madrid: Pearson Educación.

Actividades

Trabajo: Método para pilas y colas

Metodología

- Escribir un nuevo método para pilas y colas que vaya extrayendo elementos hasta encontrar un elemento dado, que se introduce como parámetro.

Criterios de evaluación

- Se evaluará el diseño y la claridad del programa así como el parecido de la figura generada con el modelo.

Extensión máxima: 7 páginas (1 página de portada, 1 página de índice, 2 páginas para escribir el código del método, su explicación y algunos ejemplos, como máximo), fuente Georgia 11 e interlineado 1,5.

Test

1. Un tipo de dato inmutable y ordenado es:

A. Diccionario.

B. Lista.

C. Tupla.

2. Sea la lista `l = [("lista", "con"), "varias", ("anidaciones", 1, 34, [3, 35])]`. Para obtener `35` hay que ejecutar:

A. l[2][3][1]

B. l[3][4][2]

C. l[2][4][3]

3. Si `a = {}, a` es:

A. Un diccionario.

B. Una lista.

C. Una tupla.

4. Si `l` es una lista, para saber el número de elementos que contiene debe usarse el método,

A. `.count()`

B. `.extend()`

C. `.len()`

5. Para definir una tupla que contenga la cadena `"prueba"` debe ejecutarse:

A. `tuple("prueba")`

B. `("prueba",)`

C. `"prueba"`

6. Una colección desordenada de pares de valores y claves es:

A. Una pila.

B. Un diccionario.

C. Una lista.

7. Una pila tiene una estructura:

A. FIFO.

B. LIFO.

C. A y B son correctas.

8. FIFO es la estructura de

A. Una cola.

B. Una pila.

C. Una lista.

9. En una pila, la cima es:

A. El elemento más accesible.

B. El último elemento que se ha introducido.

C. A y B son correctas.

10. La operación que extrae elementos en pilas y colas

A. Puede extraer elementos de dos en dos.

B. Extrae los elementos de la misma forma en los dos casos.

C. Debe comprobar antes que la estructura no está vacía.