# PYTHON : LE METACLASSI

Una METACLASSE è una classe le cui istanze sono anch'esse CLASSI.

Così come una classe "semplice" definisce l'identità delle istanze della classe, una Metaclasse definisce l'identità delle CLASSI e delle loro ISTANZE.

Una differenza è che le METACLASSI, quando vengono definite, ereditano da "type":

```
class ExampleMeta (type):
    def ...
    ...
```

## TIPI DI DATO ASTRATTO : ABSTRACT BASE CLASSES (ABCs)

Il Tipo di Dato Astratto (ADT) è un insieme di oggetti con un insieme di operazioni definite su di esso. Le operazioni sono DEFINITE, NON necessariamente IMPLEMENTATE!

In Python, ci sono le ABSTRACT BASE CLASSES (ABCs):

- sono classi astratte per natura;
- sono definiti dei metodi in queste classi;
- se una nuova classe estende la ABC, si DEVE IMPLEMENTARE i metodi definiti nella ABC.

Le CLASSI ASTRATTE sono classi che contengono uno o più METODI ASTRATTI.

Un METODO ASTRATTO è un metodo dichiarato, MA NON implementato.

Esempio:

```
import abc                                  <- importo il modulo per definire un'ABC

class Uccello (abc.ABC):                    <- abc.ABC è una classe che ha ABCMeta come METACLASSE: ereditiamo da questa
    @abc.abstractmethod                     <- decorator "abstractmethod" per definire un metodo astratto
    def vola (self):
        pass                                <- istanze dichiarate, ma non implementate
```

# 4. Strutture Dati di Base

## • Array Dinamico:

È un tipo di dato ASTRATTO che può CONTENERE oggetti di qualsiasi tipo; si basa sull'ABC Collection.

L'unica operazione esplicita che fornisce è: append()

Internamente conserva riferimenti agli oggetti passati all'interno di una lista.

```
import ctypes
from collections.abc import Collection

class DynamicArray (Collection):          # eredita da Collection, quindi bisogna implementare i metodi di Collection

    def __init__ (self):
        self.n = 0                        # Count of elements in DynamicArray
        self.capacity = 1                 # Di Default
        self.A = self._make_array (self.capacity)
        self.count = 0

    def append (self, elem):
        if (self.n == self.capacity):
            self._resize (2 * self.capacity)      # raddoppio dimensione array
        self.A[self.n] = elem
        self.n += 1

    def _resize (self, new_cap):
        B = self._make_array (new_cap)            # Nuovo array più grande
        for k in range (self.n):                  # Copia elementi preesistenti
            B[k] = self.A[k]
        self.A = B                                # Assegnazione nuovo array
        self.capacity = new_cap

    def _make_array (self, new_cap):
        return (new_cap * ctypes.py_object)()     # Costruisce array di capacità new_cap.
```

## * Analisi Costo: Array Dinamico

Assumiamo che il costo di append() sia 1, ma il costo per fare _resize() da dimensione $k$ e $2k$ sia $k$.

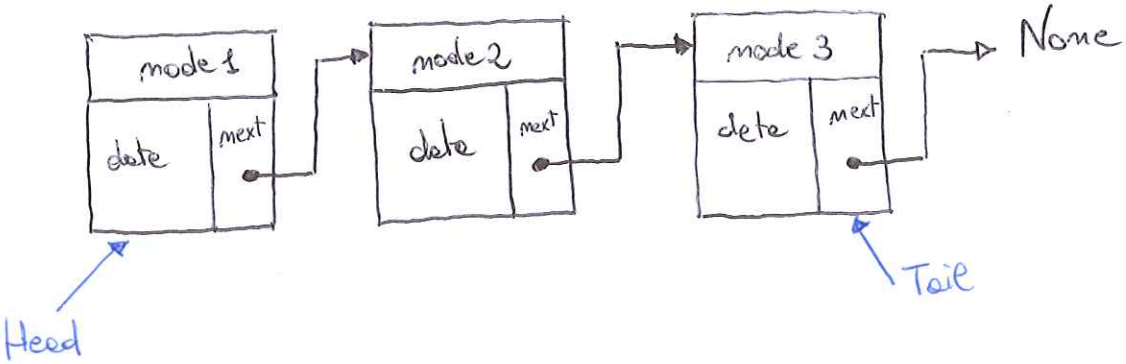
**Analisi Ammortizzata** (Metodo degli Accantonamenti):

Associamo ad ogni append() un costo di 3 → +2 di credito.
In questo modo, si ha sempre credito sufficiente quando bisogna fare _resize() di $k$, poiché $k = 2^i$, $i \geq 0$.

⇒ Per $n$ invocazioni di append(), il costo può essere pagato con un credito di $3n$.

⇒ $T(n) = O'(n)$ → COSTO AMMORTIZZATO singola operazione: $\boxed{\frac{T(n)}{n} = O'(1)}$

## Lista Collegata




ListNode eredita da Container di ABC Collection (metodi __contains__, __str__):

```
from collections.abc import Container

class ListNode(Container):
    def __init__(self, data):
        self._data = data          # data
        self.next = None           # puntatore next
```

Le Liste Collegate deve supportare le seguenti operazioni:

- Inserimento in testa / coda / indicizzato;
- Rimozione in testa / coda / elemento i-esimo;
- Ricerca di un valore nella Lista.

Le Liste eredita da MUTABLE SEQUENCE ( getitem , setitem , delitem , len , str ):

```
from collections.abc import MutableSequence
from ListNode import ListNode

class SinglyLinkedList (MutableSequence):
    def __init__ (self):
        self.head = None
        self.tail = None

    def insert (self, index, element):
        count = 0
        current = self.head
        prev = None
        node = ListNode (element)

        while current is not None and count < index:
            prev = current
            current = current.next
            count += 1

        if count < index - 1:
            raise IndexError ("Index" + str (index) + "does not fit in list")

        if prev is None:                    # inserimento in testa
            self.head = node
        else:
            prev.next = node

        if current is None:                 # Liste ruote
            self.tail.next = node
            self.tail = node
        else:
            node.next = current
```

# Analisi dei Costi

Cerca elemento k-esimo: $O(k)$

Ricerca di un elemento: $O(n)$, se l'elemento non è presente

Inserimento in posizione k-esima: $O(k)$

Inserimento in teste / code: $O(1)$

Rimozione elemento k-esimo: $O(k)$

# Lista Doppiamente Collegata

Risolve facilmente il problema dell'ATTRAVERSAMENTO AL CONTRARIO e l'utilizzo di nodi sentinelle permette di risolvere casi particolari nell'inserimento / eliminazione dei nodi in teste / code.

⇒ Inserimento ed Eliminazione in $O(1)$.

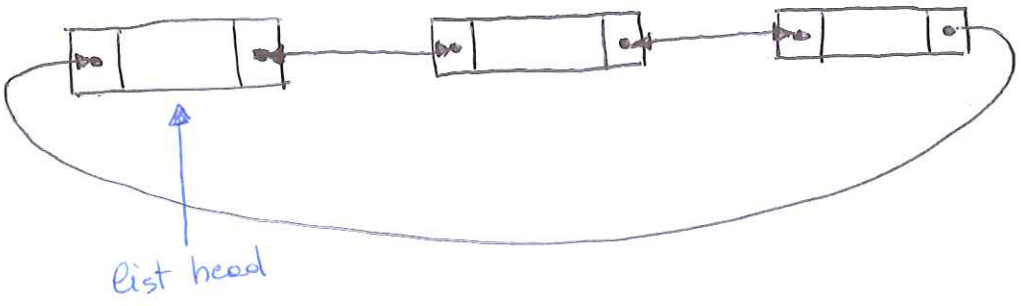
```
class ListNode (Container):
    def __init__ (self, data):
        self._data = data
        self.prev = None
        self.next = None

class DoubleyLinkedList (MutableSequence):
    def __init__ (self):
        self.head = ListNode (None)
        self.tail = ListNode (None)
        self.head.next = self.tail
        self.tail.prev = self.head
        self.size = 0

    def insert_between (self, node, predecessor, successor):
        node.next = successor
        node.prev = predecessor
        predecessor.next = node
        successor.prev = node
        self.size += 1
```

# Liste Circolari

E' possibile navigare in avanti o all'indietro a partire da un nodo qualsiasi.




Bisogna mantenere un riferimento ad un elemento delle liste per poterla navigare.

# Deque

E' una "DOUBLE-ENDED QUEUE", cioé una lista "testa-coda". E' una lista doppiamente collegata dove si possono inserire/rimuovere elementi soli in TESTA o in CODA.

Si può implementare mediante array dinamico (costo ammortizzato $O(1)$) o mediante liste doppiamente collegate ($O(1)$).

# Pile (Stack)

Gli STACK sono di tipo LIFO (Last In, First Out). Ha 2 operazioni:

- push(): inserimento
- pop(): estrazione

Si opera solo sulla TESTA (cima) dello STACK.

- Implementazione con ARRAY DINAMICO:

Se si usa un array dinamico, lo STACK ha dimensione MASSIMA prefissata.

```
structure stack:
    maxsize: integer
    top: integer
    items: array of item
```

```
INITIALIZE (S, size):
    S.items ← new empty array di dim. items
    S.maxsize ← size
    S.top ← 0
```

```
PUSH (S, el):
  if S.top = S.maxsize :
    return false
  else :
    S.items [S.top] ← el
    S.top ← S.top + 1
    return true
```

```
POP (S) :
  if S.top = 0 :
    return ⊥
  else :
    S.top ← S.top - 1
    return S.items [S.top]
```

- Implementazione tramite LISTA:

NON c'è bisogno di prefissare una DIMENSIONE massima. C'è solo inserimento in testa (e rimozione in testa). COSTO DI OGNI OPERAZIONE: $\Theta(1)$.

```
structure stack:
  head: Node
  size:
```

```
INITIALIZE (S, size):
  S.head ← ⊥
  S.size ← 0
```

```
PUSH (S, el):
  newhead ← newNode
  newhead.data ← el
  newhead.next ← S.head
  S.head ← newhead
  S.size ← S.size + 1
```

```
POP (S):
  if S.head = ⊥ :
    return ⊥
  x ← S.head.data
  S.head ← S.head.next
  S.size ← S.size - 1
  return x
```

## CODE (QUEUE)

Sono strutture di tipo FIFO (First In First Out).

Ci sono 2 tipi di operazioni:

- enqueue(): inserimento IN TESTA
- dequeue(): rimozione IN CODA

- Implementata tramite LISTA (deque): COSTO → $\Theta(1)$ per ogni operazione!

- Implementazione tramite ARRAY CIRCOLARE:

Vettore di DIMENSIONE PREFISSATA, è una LISTA CIRCOLARE.

Si implementano dei CURSORI:

- posizione della prossima lettura (READ POINTER)
- posizione della prossima scrittura (WRITE POINTER)

Poiché i cursori crescono sempre, si devono riportare al valore "fisico" della cella del vettore (mod N, dove N è dimensione dell'array):

```
DEQUEUE (el):
    ret ← buffer[read]
    read ← (read + 1) % size
    return ret
```

```
ENQUEUE (el):
    buffer[write] ← el
    write ← (write + 1) % size
```

La Coda è VUOTA : se READ ≡ WRITE

è PIENA : se READ ≡ (WRITE + 1) % SIZE (si spreca uno slot)

# CODE DI PRIORITA'

Ad ogni elemento è associata una PRIORITA'; gli elementi ad alta priorità vengono estratti prima delle code rispetto a quelli a basse priorità.

- La priorità MASSIMA può essere il valore più grande o più piccolo.

Tipicamente NON si può inserire un elemento a priorità minore dell'ultimo estratto.

Supporta le seguente operazioni:

- ENQUEUE (prio, el): inserisce l'elemento el con priorità prio
- getMin (): restituisce l'elemento a PRIORITÀ MASSIMA attualmente presente nelle code.

# CALENDAR QUEUE

CALENDARIO DA SCRIVANIA: si possono inserire gli appuntamenti per ciascun giorno e vengono messi in ordine di tempo.

! Si utilizza 1 solo FOGLIO per tutti i mesi ed in 1 GIORNO ci possono essere appuntamenti di MESI DIFFERENTI.

- L'ASSE TEMPORALE viene diviso in BUCKET, ognuno con una certa GRANDEZZA $W$.

  C'è la nozione di "ULTIMA PRIORITA' ESTRATTA", cioè TEMPO CORRENTE.

  Se viene chiesto di inserire un elemento con priorità $p >$ tempo corrente, viene inserito nel BUCKET: $\left\lfloor \frac{p}{W} \right\rfloor \bmod n$, dove $n = \#$ buckets

  * (n) e (W) devono essere scelti in modo tale da MINIMIZZARE il numero di elementi in ciascun BUCKET.

    ⇒ RIDIMENSIONAMENTO: si RADDOPPIA / DIMEZZA $n$ (numero di bucket) se il numero di elementi PER BUCKET cresce / decresce troppo.

- PROBLEMA DEL RIDIMENSIONAMENTO:

  La grandezza dei bucket $W$ viene ricalcolata considerando la SEPARAZIONE MEDIA TRA GLI EVENTI. Per evitare problemi, si escludono dal calcolo gli eventi con separazione troppo grande.

  ⇒ $\boxed{\text{new } W = 3 \cdot \overline{\text{separazione}}}$

- ANALISI AMMORTIZZATA:

  Costo singola operazione: $o_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$

  Costo TOTALE: $A = \sum_{i=1}^{m} o_i = \boxed{C + \Phi(D_m) - \Phi(D_0)}$

  Se $\Phi(D_m) - \Phi(D_0) \geq 0$ ⇒ Costo della Struttura è: $\boxed{O(A)}$

Si prende $\Phi(D_0) = 0$ ; $\Delta\Phi(D_i) = +2$ per la enqueue / dequeue

$\Delta\Phi(D_i) = +2 - m$ se c'è bisogno di fare RESIZE

Il costo EFFETTIVO $C_i$ per enqueue / dequeue è 1.