

# Programmazione Avanzata e parallela

Lezione 20 bis

# Python e Oggetti

## Programmazione orientata agli oggetti (OOP)

- Pubblico e privato (o la loro mancanza)
- Metodi di classe
- Dunder methods

# Accesso

## Pubblico e privato

- Alcuni degli attributi e dei metodi potrebbero essere specifici dell'implementazione e vorremmo non renderli visibili all'esterno:
- E.g., se usiamo una lista per implementare una coda vorremmo evitare che dall'esterno si acceda direttamente alla lista
- Alcuni linguaggi di programmazione forzano il controllo degli accessi con i modificatori **public**, **protected** e **private**
- **Public**: accessibile a tutti
- **Protected**: accessibili alla classe stessa e alle sottoclassi
- **Private**: accessibili solo dalla classe stessa

# Accesso

## Pubblico e privato

- L'utilizzo di metodi e attributi privati ci permette di cambiare l'implementazione senza dover cambiare chi usa la classe...
- ...perché non può accedere ai dettagli dell'implementazione
- Però Python non supporta i modificatori “public”, “protected” e “private”
- Si utilizzano una serie di convenzioni per segnalare che alcuni attributi e metodi non sono pubblici

# Accesso

## Pubblico e privato

- *Metodi/attributi pubblici*  
Nessun cambiamento
- *Metodi/attributi protetti*  
Il nome del metodo/attributo inizia con “\_”:  
**def \_some\_stuff(self):**  
Possiamo comunque accedere senza problemi
- *Metodi/attributi privati*  
Il nome del metodo/attributo inizia con “\_\_” (doppio underscore):  
**def \_\_this\_is\_private(self):**  
In questo caso viene effettuato del name mangling:  
il nome con cui il metodo/attributo viene visto all'esterno è  
\_nomeclasse\_\_nomemetodo

# Accesso

## Pubblico e privato

- Se deve essere accessibile all'esterno: pubblico
- Se è un dettaglio implementativo protetto o privato (dipende come vogliamo che sia l'accesso da parte di classi che estendono quella attuale — uno dei prossimi argomenti)
- Se rendete qualcosa pubblico verrà utilizzato e sarà difficile cambiarlo dopo senza “rompere” del codice che usa la classe
- Se però rendete troppe funzionalità private potreste rendere la classe difficile da utilizzare (e.g., una coda senza “isempty”)

# Metodi di classe

## Chiamare metodi della classe

- Un metodo di classe è un metodo che è della classe e non dell'oggetto:
- Metodi normali:  
**q = Queue()**  
**q.enqueue(3)**
- Metodo di classe:  
**Queue.do\_stuff()**
- Perché dovrebbero servirci?

# Metodi di classe

## Chiamare metodi della classe

- I metodi di classe possono essere chiamati prima che sia creato un oggetto...
- ...quindi possiamo usarli per fornire metodi alternativi di costruzione di oggetti:
  - Costruttore “standard”
  - Costruttore copiando da altro oggetto o struttura dati differente
- Evitare di creare una funzione globale per lavorare solo su oggetti di un tipo specifico




# Metodi di classe

## Chiamare metodi della classe

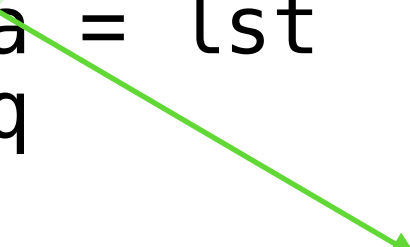
- Un metodo di classe è creato precedendo la definizione con **@classmethod** (il primo argomento per convenzione si chiamerà “cls” e non “self”)

```
@classmethod
def from_list(cls, lst):
    q = cls()
    q.__data = lst
    return q
```

Qui verrà passata  
la classe, non l'oggetto



Invochiamo il  
costruttore “standard”



# Dunder methods

## Metodi speciali

- Alcuni metodi hanno un significato speciale:
  - `__init__` per inizializzare un oggetto
  - `__str__` per avere una rappresentazione in stringa dell'oggetto (per funzioni come `print`)
- Esistono molti altri metodi di questo tipo che ci permettono, per esempio, di definire:
  - Il comportamento con le normali operazioni (somma, prodotto, divisione, etc.
  - Accesso con indici

# Dunder methods

## Confronto tra oggetti

Metodo	Utilizzo
<code>__lt__</code>	<code>obj &lt; ...</code>
<code>__le__</code>	<code>obj &lt;= ...</code>
<code>__eq__</code>	<code>obj == ...</code>
<code>__ne__</code>	<code>obj != ...</code>
<code>__gt__</code>	<code>obj &gt; ...</code>
<code>__ge__</code>	<code>obj &gt;= ...</code>

Tutti questi metodi prendono due argomenti (self e un altro oggetto) e ritornano un valore Booleano

# Dunder methods

## Accesso con chiavi

Metodo	Utilizzo	Argomenti
<code>__getitem__</code>	Lettura di <code>obj[key]</code>	<code>(self, key)</code>
<code>__setitem__</code>	<code>obj[key] = ...</code>	<code>(self, key, value)</code>
<code>__delitem__</code>	<code>del obj[key]</code>	<code>(self, key)</code>

Questi metodi permettono di accedere a dei valori usando una chiave, quindi potremmo utilizzarli per implementare dizionari, matrici sparse, etc

# Dunder methods

## Operazioni matematiche

Metodo	Utilizzo	Argomenti
<code>__add__</code>	<code>obj + ...</code>	<code>(self, other)</code>
<code>__radd__</code>	<code>... + obj</code>	<code>(self, other)</code>
<code>__iadd__</code>	<code>obj += ...</code>	<code>(self, other)</code>
<code>__sub__</code>	<code>obj - ...</code>	<code>(self, other)</code>
<code>__mul__</code>	<code>obj * ...</code>	<code>(self, other)</code>

Se volessimo implementare operazioni di somma tra vettori si basterebbe definire i metodi corrispondenti (ci sono anche metodi per divisione, modulo, elevamento a potenza, etc)

# Dunder methods

## Altri metodi utili

Metodo	Utilizzo	Argomenti
<code>__len__</code>	<code>len(obj)</code>	<code>(self)</code>
<code>__contains__</code>	<code>... in obj</code>	<code>(self, other)</code>
<code>__call__</code>	<code>obj(args)</code>	<code>(self, ...)</code>

Di particolare interesse è il metodo `__call__` che permette di far comportare l'oggetto come se fosse una funzione, facendo cose come:

```
f = NeuralNetwork(parametri)
y = f(x)
```

# Esempio coi dunder methods

## Numeri Duali

- Un numero duale è un numero nella forma  $a + b\varepsilon$
- La somma di numeri duali è
$$(a + b\varepsilon) + (c + d\varepsilon) = (a + c) + (b + d)\varepsilon$$
- Il prodotto di numeri duali è
$$(a + b\varepsilon) \times (c + d\varepsilon) = ac + (ad + bc)\varepsilon$$
- Se applichiamo una funzione  $f$  a un numero duale  $a + b\varepsilon$  vale che  $f(a + b\varepsilon) = f(a) + bf'(a)\varepsilon$
- Ovvero se iniziamo con  $a + 1\varepsilon$  e applichiamo  $f$  otteniamo  $f(a) + f'(a)\varepsilon$ : abbiamo la derivata di  $f$  nel punto

# Esempio coi dunder methods

## Numeri Duali

- Possiamo definire una classe di numeri duali
- Definiamo somma e prodotto
  - Sia tra duali che tra un duale e un numero “normale”
    - Questo è fattibile con `isinstance(oggetto, classe)` per testare il tipo dell'argomento che dobbiamo gestire
- Così facendo abbiamo una classe in grado di calcolarci in automatico la derivata in ogni punto...
- ...ovvero abbiamo la base per un sistema di differenziazione automatica