Corso ITS: *ARTIFICIAL INTELLIGENCE SPECIALIST*

Modulo: **Programmazione ad oggetti in Python e librerie esterne**

Docente: *Andrea Ribuoli*

Giovedì **20 Febbraio** 2025

09:00 - 14:00

Analisi esercizio creazione classe *Message*

message.py

Realizzate una classe, *Message*, che rappresenti un modello per messaggi di posta elettronica. Un messaggio ha un destinatario, un mittente e un testo. Progettate i metodi seguenti:

- un costruttore che riceve il mittente e il destinatario;
- il metodo *append* che aggiunge una riga in fondo al testo del messaggio;
- il metodo *toString* che trasforma il messaggio in un'unica, lunga stringa.

Scrivete un programma che usi questa classe per creare un messaggio e visualizzarlo.

- l'esercizio è derivato dal *P9.24* a pagina 605 del testo adottato
- prestando molta attenzione alla consegna si nota che:
- - il costruttore non è richiesto acquisire il corpo del messaggio
- - tuttavia è bene inizializzare quest'ultimo nel costruttore
- - il metodo *toString* ci introdurrrà al concetto dei **metodi speciali** in Python

In [1]:
```
class Message :
    def __init__(self, mittente, destinatario) :
        pass # da implementare
```

```
    def append(self, nuovaRiga) :
        pass # da implementare
    def toString(self) :
        pass # da implementare
```

- così abbiamo solo definito la cosiddetta **interfaccia Pubblica**
- è importante capire che questa è direttamente derivabile da un diagramma **UML**
- *UML* sta per **Unified Modeling Language**
- è la notazione standard per la rappresentazione grafica dei progetti a oggetti

- nella implementazione creiamo le variabili di istanza dando a ciascuna un nome
- è una pura convenzione (ma **importante** da rispettare): inizino dunque con `_`
- accumuleremo le righe del corpo del messaggio in una lista
- l'esercizio può anche essere svolto utilizzando una semplice stringa

In [2]:
```
class Message :
    def __init__(self, mittente, destinatario) :
        self._sender = mittente
        self._recipient = destinatario
        self._messageBody = []
```

- per enfatizzare la differenza tra ciò che è visibile all'esterno
- e ciò che è interno alla implementazione
- usiamo nomi in inglese per le *variabili di istanza*

In [3]:
```
class Message :
    def __init__(self, mittente, destinatario) :
        self._sender = mittente
        self._recipient = destinatario
        self._messageBody = []
    def append(self, nuovaRiga) :
        self._messageBody.append(nuovaRiga)
    def toString(self) :
        return (   "Mittente . . . . . : " + self._sender +
                 "\nDestinatario . . . : " + self._recipient +
                 "\nMessaggio :\n  " + "\n  ".join(self._messageBody) )
```

- si noti l'uso del metodo `join` dell'oggetto stringa
- a questo punto possiamo scrivere un esempio di utilizzo della classe appena definita

In [4]:
```
m = Message("andrea.ribuoli@yahoo.com", "aicoordinamento@itsturismomarche.it
m.append("Ciao Riccardo,")
m.append("  ho iniziato le lezioni sulla OOP in Python.")
m.append("Mi sembra gli allievi seguano con attenzione.")
m.append("")
m.append("Buon lavoro e a presto,")
m.append("  Andrea")
print(m.toString())
```

```
Mittente . . . . . : andrea.ribuoli@yahoo.com
Destinatario . . . : aicoordinamento@itsturismomarche.it
Messaggio :
  Ciao Riccardo,
    ho iniziato le lezioni sulla OOP in Python.
  Mi sembra gli allievi seguano con attenzione.

  Buon lavoro e a presto,
    Andrea
```

- da alcune domande in classe è emersa l'esigenza di un chiarimento
- quando si implementa una classe in Python, oltre al costruttore, esistono vari **metodi speciali**
- per un elenco più esteso si veda la **Tabella 1** di pagina 587
- uno di questi è `__str__` che (ove implementato) verrà usato per una stampa aggraziata dell'oggetto
- se avessimo utilizzato tale nome al posto di *toString* avremmo potuto invocare `print(m)`

In [5]:
```
print(m)
```

```
<__main__.Message object at 0x105964ed0>
```

In [6]:
```
class Message :
    def __init__(self, mittente, destinatario) :
        self._sender = mittente
        self._recipient = destinatario
        self._messageBody = []
    def append(self, nuovaRiga) :
        self._messageBody.append(nuovaRiga)
    def __str__(self) :
        return (   "Mittente . . . . . : " + self._sender +
                 "\nDestinatario . . . : " + self._recipient +
                 "\nMessaggio :\n  " + "\n  ".join(self._messageBody) )
```

In [7]:
```
m = Message("andrea.ribuoli@yahoo.com", "aicoordinamento@itsturismomarche.it
m.append("Ciao Riccardo,")
m.append("  ho iniziato le lezioni sulla OOP in Python.")
m.append("Mi sembra gli allievi seguano con attenzione.")
m.append("")
m.append("Buon lavoro e a presto,")
m.append("  Andrea")
print(m)
```

```
Mittente . . . . . : andrea.ribuoli@yahoo.com
Destinatario . . . : aicoordinamento@itsturismomarche.it
Messaggio :
  Ciao Riccardo,
    ho iniziato le lezioni sulla OOP in Python.
  Mi sembra gli allievi seguano con attenzione.

  Buon lavoro e a presto,
    Andrea
```

- un altro aspetto emerso in classe è quello delle **variabili di classe**
- analogamente parleremo anche dei **metodi di classe**
- come monito nell'utilizzo anche per le variabili di classe è opportuno denominarle partendo con `_`
- preferendo restituire informazioni su di esse mediante metodi di classe
- quest'ultimi **non ricevono** il parametro `self`

In [8]:
```python
class Message :
    _counter = 0
    def getCounter() :
        return Message._counter
    def resetCounter() :
        Message._counter = 0
    def __init__(self, mittente, destinatario) :
        self._sender = mittente
        self._recipient = destinatario
        self._messageBody = []
        Message._counter += 1
    def append(self, nuovaRiga) :
        self._messageBody.append(nuovaRiga)
    def __str__(self) :
        return (   "Mittente . . . . . : " + self._sender +
                 "\nDestinatario . . . : " + self._recipient +
                 "\nMessaggio :\n  " + "\n  ".join(self._messageBody) )
```

In [9]:
```python
print(Message.getCounter())
m = Message("andrea.ribuoli@yahoo.com", "aicoordinamento@itsturismomarche.it
m.append("Ciao Riccardo,")
m.append("  ho iniziato le lezioni sulla OOP in Python.")
m.append("Mi sembra gli allievi seguano con attenzione.")
m.append("")
m.append("Buon lavoro e a presto,")
m.append("  Andrea")
print(Message.getCounter())
```

```
0
1
```

- un altro apetto emerso è nato da una considerazione
- così come posso inizializzare una nuova lista passando al costruttore una istanza di lista
- e questa ne diventa una copia **profonda** (completamente indipendente)

- posso adattare il mio costruttore per la classe Message in tal senso?
- sì ma dobbiamo rilavorare la nostra interfaccia publlica e la conseguente implementazione
- preferendo omettere *mittente* e *destinatario* dal costruttore e introducendo la variabile *copyFrom*
- necessiteremo di due metodi (prima assenti): *impostaMittente*, *impostaDestinatario*
- ripartiamo dal'inizio:

In [10]:
```
class Message :
    def __init__(self, copyFrom = None) :
        pass # da implementare
    def impostaMittente(self, mittente) :
        pass # da implementare
    def impostaDestinatario(self, destinatario) :
        pass # da implementare
    def append(self, nuovaRiga) :
        pass # da implementare
    def toString(self) :
        pass # da implementare
```

- se omesso, il parametro **copyFrom** varrà **None**
- si tratta di un *valore speciale* che significa *niente*
- con il valore *None* non si può fare alcuna elaborazione
- *None* può essere assegnato ad una variabile che a quel punto non farà riferimento a nulla

In [11]:
```
class Message :
    _counter = 0
    def getCounter() :
        return Message._counter
    def resetCounter() :
        Message._counter = 0
    def __init__(self, copyFrom = None) :
        if copyFrom != None :
            self._sender      = copyFrom._sender
            self._recipient   = copyFrom._recipient
            self._messageBody = list(copyFrom._messageBody)
        else :
            self._sender = ""
            self._recipient = ""
            self._messageBody = []
        Message._counter += 1
    def impostaMittente(self, mittente) :
        self._sender = mittente
    def impostaDestinatario(self, destinatario) :
        self._recipient = destinatario
    def append(self, nuovaRiga) :
        self._messageBody.append(nuovaRiga)
    def __str__(self) :
        return (   "Mittente . . . . . : " + self._sender +
                 "\nDestinatario . . . : " + self._recipient +
```

```
                "\nMessaggio :\n  " + "\n  ".join(self._messageBody) )
```

In [12]:
```
m1 = Message()
m1.impostaMittente("andrea.ribuoli@yahoo.com")
m1.impostaDestinatario("aicoordinamento@itsturismomarche.it")
m1.append("Ciao Riccardo,")
m1.append("  Le domande degli allievi portano a temi più complessi!")
m1.append("A presto,")
m1.append("  Andrea")
m2 = m1
m3 = Message(m1)
m2.append("P.S.: questa modifica su m2 varrà anche per m1, ma non per m3")
print("--- m1 " + "-" * 60 + "\n", m1)
print("--- m2 " + "-" * 60 + "\n", m2)
print("--- m3 " + "-" * 60 + "\n", m3)
```

```
--- m1 ------------------------------------------------------------
 Mittente . . . . . : andrea.ribuoli@yahoo.com
Destinatario . . . : aicoordinamento@itsturismomarche.it
Messaggio :
  Ciao Riccardo,
    Le domande degli allievi portano a temi più complessi!
  A presto,
    Andrea
  P.S.: questa modifica su m2 varrà anche per m1, ma non per m3
--- m2 ------------------------------------------------------------
 Mittente . . . . . : andrea.ribuoli@yahoo.com
Destinatario . . . : aicoordinamento@itsturismomarche.it
Messaggio :
  Ciao Riccardo,
    Le domande degli allievi portano a temi più complessi!
  A presto,
    Andrea
  P.S.: questa modifica su m2 varrà anche per m1, ma non per m3
--- m3 ------------------------------------------------------------
 Mittente . . . . . : andrea.ribuoli@yahoo.com
Destinatario . . . : aicoordinamento@itsturismomarche.it
Messaggio :
  Ciao Riccardo,
    Le domande degli allievi portano a temi più complessi!
  A presto,
    Andrea
```