Laurea in Ingegneria Gestionale

Corso di Fondamenti di Informatica
A.A. 2017/2018

DIPARTIMENTO DI INGEGNERIA INFORMATICA
AUTOMATICA E GESTIONALE ANTONIO RUBERTI

SAPIENZA
UNIVERSITÀ DI ROMA

classi e oggetti - 2

Slide tratte da:

Pensare da informatico
Versione Python
di Allen B. Downey, Jeffrey Elkner e Chris Meyers
Traduzione di Alessandro Pocaterra

Capitoli 12, 13 e 14

Disponibili al link
https://www.python.it/doc/Howtothink/Howtothink-html-it/index.htm

Copia "profonda"

Fortunatamente il modulo `copy` contiene un altro metodo chiamato `deepcopy` che copia correttamente non solo l'oggetto ma anche gli eventuali oggetti presenti al suo interno:

```
>>> from copy import deepcopy
>>> Oggetto2 = deepcopy(Oggetto1)
```

Ora `Oggetto1` e `Oggetto2` sono oggetti completamente separati e occupano diverse zone di memoria.

Copia "profonda"

Possiamo usare `deepcopy` per riscrivere completamente `AumentaRettangolo` così da non cambiare il `Rettangolo` originale ma restituire una copia con le nuove dimensioni:

```
from copy import *

def AumentaRettangolo(Rett, AumentoLargh, AumentoAlt) :
  NuovoRett = deepcopy(Rett)
  NuovoRett.Larghezza = NuovoRett.Larghezza \
     + AumentoLargh
  NuovoRett.Altezza = NuovoRett.Altezza \
     + AumentoAlt
  return NuovoRett
```

Metodi

Le funzioni viste in precedenza che manipolano istanze (oggetti) della classe `Punto` o `Rettagolo` prendono sempre come input oggetti di queste classi come parametri.

In Python (come in ogni linguaggio che supporta una programmazione orientata agli oggetti) è possibile usare metodi di classe, e quindi le funzioni che prendono in input oggetti possono essere trasformate in metodi.

I metodi sono simili alle funzioni con due differenze:

- I metodi sono definiti all'interno della definizione di classe per rendere più esplicita la relazione tra la classe ed i metodi corrispondenti.
- La sintassi per invocare un metodo è diversa da quella usata per chiamare una funzione (uso della notazione punto)

La classe `Tempo`

Definiamo una classe chiamata `Tempo` con l'obiettivo di registrare un'ora del giorno:

```
class Tempo:
  pass
```

Possiamo creare un nuovo oggetto `Tempo` assegnando gli attributi per le ore, i minuti e i secondi:

```
Time = Tempo()
Time.Ore = 11
Time.Minuti = 59
Time.Secondi = 30
```

La classe `Tempo` e la funzione `StampaTempo`

Definiamo anche una funzione per la stampa di un oggetto di classe Tempo

```
class Tempo:
  pass

def StampaTempo(Orario):
  print(str(Orario.Ore) + ":" + \
        str(Orario.Minuti) + ":" + \
        str(Orario.Secondi))
```

Per chiamare la funzione passiamo un oggetto `Tempo` come parametro:

```
>>> OraAttuale = Tempo()
>>> OraAttuale.Ore = 9
>>> OraAttuale.Minuti = 14
>>> OraAttuale.Secondi = 30
>>> StampaTempo(OraAttuale)
>>> 9:14:30
```

La classe Tempo e il metodo `StampaTempo`

Per rendere `StampaTempo` un metodo, tutto quello che dobbiamo fare è muovere la definizione della funzione all'interno della definizione della classe.

```
class Tempo:
  def StampaTempo(Orario):
    print(str(Orario.Ore) + ":" + \
          str(Orario.Minuti) + ":" + \
          str(Orario.Secondi))
```

Ora possiamo invocare `StampaTempo` usando la notazione punto.

```
>>> OraAttuale.StampaTempo()
```

L'oggetto su cui il metodo è invocato appare prima del punto ed il nome del metodo subito dopo.

L'oggetto su cui il metodo è invocato è automaticamente assegnato al primo parametro, quindi nel caso di `OraAttuale` è assegnato a `Orario`.

Ancora sui metodi

Per convenzione il primo parametro di un metodo è chiamato `self`, traducibile in questo caso come "l'oggetto stesso".

Notiamo che la sintassi di una chiamata di funzione suggerisce che la funzione sia l'agente attivo: invocare `StampaTempo(OraAttuale)` è come dire "StampaTempo: c'è un oggetto per te da stampare!"

Nella programmazione orientata agli oggetti sono gli oggetti ad essere considerati l'agente attivo: `OraAttuale.StampaTempo()` di fatto significa "OraAttuale: invoca il metodo per stampare il tuo valore!"

Lo spostamento della responsabilità dalla funzione all'oggetto rende possibile scrivere funzioni più versatili e rende più immediati il mantenimento ed il riutilizzo del codice.

Un altro esempio

```
class Tempo:
  def StampaTempo(Orario):
       …     # corpo della funzione omesso
             # per semplicità di esposizione
  # La funzione Dopo() restituisce True se l'orario
  # indicato dall'oggetto di invocazione è successivo
  # a quello del parametro Tempo2
  def Dopo(self, Tempo2):
    if self.Ore > Tempo2.Ore:
      return True
    if self.Ore < Tempo2.Ore:
      return False
    if self.Minuti > Tempo2.Minuti:
      return True
    if self.Minuti < Tempo2.Minuti:
      return False
    if self.Secondi > Tempo2.Secondi:
      return True
    return False
```

Un altro esempio

```
>>> t = Tempo()
>>> t.Ore=10
>>> t.Minuti=30
>>> t.Secondi=10
>>> t1 = Tempo()
>>> t1.Ore = 10
>>> t1.Minuti =30
>>> t1.Secondi = 15
>>> t.Dopo(t1)
False
>>> t.StampaTempo()
10:30:10
>>> t1.StampaTempo()
10:30:15
```

Funzioni o metodi con argomenti opzionali

La seguente funzione cerca un carattere in una stringa e restituisce il suo indice se lo trova, -1 altrimenti

```
def Trova(Stringa, Carattere):
  Indice = 0
  while Indice < len(Stringa):
    if Stringa[Indice] == Carattere:
      return Indice
    Indice = Indice + 1
  return -1
```

La versione seguente fa lo stesso, ma inizia la ricerca a partire da un indice dato come ulteriore parametro

```
def Trova(Stringa, Carattere, Inizio=0):
  Indice = Inizio
  while Indice < len(Stringa):
    if Stringa[Indice] == Carattere:
      return Indice
    Indice = Indice + 1
  return -1
```

Funzioni o metodi con argomenti opzionali

Il terzo parametro, `Inizio`, è opzionale perché abbiamo fornito il valore 0 di default. Se invochiamo `Trova` con solo due argomenti usiamo il valore di default per il terzo così da iniziare la ricerca dall'inizio della stringa:

```
>>> Trova("Mela", "e")
1
```

Se forniamo un terzo parametro questo sovrascrive il valore di default:

```
>>> Trova("Mela", "e", 2)
-1
```

Il metodo di inizializzazione

Il metodo di inizializzazione è un metodo speciale invocato quando si crea un oggetto. Il nome di questo metodo è `__init__` (due caratteri di sottolineatura, seguiti da `init` e da altri due caratteri di sottolineatura). Un metodo di inizializzazione per la classe `Tempo` potrebbe essere:

```
class Tempo:
  def __init__(self, Ore=0, Minuti=0, Secondi=0):
    self.Ore = Ore
    self.Minuti = Minuti
    self.Secondi = Secondi
```

Non c'è conflitto tra l'attributo `self.Ore` e il parametro `Ore`. La notazione punto specifica a quale variabile ci stiamo riferendo.

Quando invochiamo il costruttore `Tempo` gli argomenti che passiamo sono girati a `__init__`:

```
>>> OraAttuale = Tempo(9, 14, 30)
>>> OraAttuale.StampaTempo()
>>> 9:14:30
```

Il metodo di inizializzazione

Dato che i parametri sono opzionali possiamo anche ometterli:

```
>>> OraAttuale = Tempo()
>>> OraAttuale.StampaTempo()
>>> 0:0:0
```

Possiamo anche fornire solo il primo parametro:

```
>>> OraAttuale = Tempo(9)
>>> OraAttuale.StampaTempo()
>>> 9:0:0
```

o i primi due parametri:

```
>>> OraAttuale = Tempo(9, 14)
>>> OraAttuale.StampaTempo()
>>> 9:14:0
```

Infine possiamo anche passare un sottoinsieme dei parametri nominandoli esplicitamente:

```
>>> OraAttuale = Tempo(Secondi = 30, Ore = 9)
>>> OraAttuale.StampaTempo()
>>> 9:0:30
```

Rappresentazione di oggetti come stringhe

Ogni oggetto ha di default una rappresentazione sotto forma di stringa (accessibile attraverso la funzione `str()`, o la `print()`).

```
>>> InizioLezione = Tempo(9)
>>> str(InizioLezione)
'<__main__.Tempo object at 0x11215ff60>'
>>> print(InizioLezione)
<__main__.Tempo object at 0x11215ff60>
```

Intuitivamente , sia `print` che `str` "chiedono" all'oggetto `InizioLezione` di fornire la sua rappresentazione come stringa (la `print` poi la stampa a schermo senza apici)

E' possibile modificare il comportamento di una classe in modo da avere per le sue istanze una rappresentazione sotto forma di stringa definita da utente e quindi possibilmente diversa da quella di default?

Il metodo `__str__`

Questo è possibile attraverso la (ri-)definizione di un metodo di classe chiamato `__str__` , che appunto ritorna una rappresentazione di un oggetto sotto forma di stringa. Ad esempio:

```
def __str__(Orario):
    return str(Orario.Ore) + ":" + \
        str(Orario.Minuti) + ":" + \
        str(Orario.Secondi)
```

Quindi avremo:

```
>>> InizioLezione = Tempo(9)
>>> str(InizioLezione)
'9:0:0'
>>> print(InizioLezione)
9:0:0
```

Nota: il codice di `__str__` è simile a quello della funzione `StampaTempo`, ma il comportamento finale è diverso: per stampare un oggetto non dobbiamo più invocare una funzione scritta ad hoc (`StampaTempo`), ma possiamo agire come per tutti gli altri valori ed oggetti Python

La classe `Punto` rivisitata

Riscriviamo la classe `Punto` in uno stile più orientato agli oggetti:

```
class Punto:
  def __init__(self, x=0, y=0):
    self.x = x
    self.y = y

  def __str__(self):
    return '(' + str(self.x) + ', ' + str(self.y) + ')'
```

Esempi di uso

```
>>> P = Punto(3, 4)
>>> print(P)
(3, 4)
```

Quando scriviamo una nuova classe iniziamo quasi sempre scrivendo `__init__` (la funzione che rende più facile istanziare oggetti) e `__str__` (utile per il debug).

Ridefinizione di un operatore

Pyhon, come anche altri linguaggi, consente di cambiare la definizione degli operatori predefiniti quando applicati a tipi definiti dall'utente. Questa caratteristica è chiamata ridefinizione dell'operatore (o "overloading dell'operatore") e si rivela molto utile soprattutto quando vogliamo definire nuovi tipi di operazioni matematiche.

Se vogliamo ridefinire l'operatore `+` scriveremo un metodo chiamato `__add__`

```
class Punto:
  ... # definizione di altre funzioni
      # omessa per semplicità
  def __add__(self, AltroPunto):
    return Punto(self.x + AltroPunto.x, \
                 self.y + AltroPunto.y)
```

Ridefinizione di un operatore

```
class Punto:
  ...
  def __add__(self, AltroPunto):
   return Punto(self.x + AltroPunto.x, \
                self.y + AltroPunto.y)
```

Come al solito il primo parametro è l'oggetto su cui è invocato il metodo. Il secondo parametro è chiamato `AltroPunto`. Ora sommiamo due oggetti `Punto` restituendo la somma in un terzo oggetto `Punto` che conterrà la somma delle coordinate `x` e delle coordinate `y`.

Quando applichiamo l'operatore `+` ad oggetti `Punto` Python invocherà il metodo `__add__`:

```
>>> P1 = Punto(3, 4)
>>> P2 = Punto(5, 7)
>>> P3 = P1 + P2
>>> print( P3 )
(8, 11)
```

L'espressione `P1 + P2` è equivalente a `P1.__add__(P2)` ma più elegante.

Ridefinizione di operatori e metodi: esempi

operator	Function	Description
+	__add__(self, other)	Addition
*	__mul__(self, other)	Multiplication
-	__sub__(self, other)	Subtraction
%	__mod__(self, other)	Remainder
/	__truediv__(self, other)	Division
<	__lt__(self, other)	Less than
<=	__le__(self, other)	Less than or equal to
==	__eq__(self, other)	Equal to
!=	__ne__(self, other)	Not equal to
>	__gt__(self, other)	Greater than
>=	__ge__(self, other)	Greater than or equal to
[index]	__getitem__(self, index)	Index operator
in	__contains__(self, value)	Check membership
len	__len__(self)	The number of elements
str	__str__(self)	The string representation

Esempio*: definizione della classe

```
from math import *
class Circle:
    def __init__(self, radius):
        self.__radius = radius
    def setRadius(self, radius):
        self.__radius = radius
    def getRadius(self):
        return self.__radius
    def area(self):
        return pi * self.__radius ** 2
    def __add__(self, another_circle):
        return Circle( self.__radius +  \
                    another_circle.__radius )
    def __gt__(self, another_circle):
        return self.__radius > another_circle.__radius
    def __lt__(self, another_circle):
        return self.__radius < another_circle.__radius
    def __str__(self):
        return "Circle with radius " + str(self.__radius)
```

Il __ all'inizio del nome di una variabile è usato per indicare che si tratta di una variabile di oggetto, pensata per essere usata solo internamente alla classe.

*http://thepythonguru.com/python-operator-overloading/

Esempio: possibile programma principale che usa la classe

```
# programma di esempio disponibile nel file UsaCircle.py
# classe Circle disponibile nel file Circle.py
# i due file sono nella stessa cartella

from Circle import *

c1 = Circle(4)
print(c1.getRadius())

c2 = Circle(5)
print(c2.getRadius())

c3 = c1 + c2
print(c3.getRadius())

print( c3 > c2) # Became possible because
                # we have added __gt__ method

print( c1 < c2) # Became possible because
                # we have added __lt__ method

print(c3)       # Became possible because
                # we have added __str__ method
```

Esempio: output atteso

```
========== RESTART: /.../UsaCircle.py ==========
4
5
9
True
True
Circle with radius 9
>>>
```

Information hiding (cenni)

Nella programmazione orientata agli oggetti (OO) si tende a "nascondere" all'esterno la struttura interna di una classe, cioè i tipi di dati usati per memorizzare le informazioni che caratterizzano un oggetto, e si rendono *pubblici* solo metodi per la lettura e la modifica dell'oggetto (o di sue parti).

A questo fine, i linguaggi di programmazione OO adottano vari meccanismi, in particolare per impedire ad un programma o funzione che usa la classe (utente della classe) l'accesso diretto alle sue variabili.

In Python c'è solo un supporto limitato a questo tipo di esigenza, attraverso l'uso del prefisso __ (doppio carattere di sottolineatura) nel nome di una variabile di oggetto. Qualsiasi identificatore in questa forma viene rimpiazzato, a livello di codice eseguito, con `_nomeclasse__variab`, dove `nomeclasse` è il nome della classe corrente e `__variab` l'identificatore.

Nell'esempio precedente (classe `Circle`), abbiamo visto questo utilizzo per la variabile `__radius`.

Per consentire ad un utente di leggere o modificare il contenuto di queste variabili, il progettista della classe deve prevedere dei metodi specifici (metodi get e set, come i metodi `setRadius` e `getRadius` dell'esempio)

Information hiding: esempio

```
>>> c1 = Circle(4)
>>> c1.__radius
Traceback (most recent call last):
  File "<pyshell#30>", line 1, in <module>
    c1.__radius
AttributeError: 'Circle' object has no attribute
'__radius'
>>> c1.getRadius()
4
>>> c1._Circle__radius
4
```

L'ultima espressione in realtà ci fa capire che non siamo realmente riusciti ad impedire l'accesso alla variabile `__radius`, ma lo abbiamo solo reso più complicato. In altri termini in Python non c'è un vero meccanismo per garantire l'information hiding quando definiamo una classe, e l'uso di variabili di oggetto precedute da __ è in pratica solo una buona norma da adottare nella scrittura del codice (e che lo rende più leggibile).

Variabili di classe

Finora abbiamo visto che un oggetto istanza di una classe può avere delle variabili (anche chiamate attributi di istanza o di dato in Python), e che è buona norma definire queste variabili all'interno del metodo `__init__` .

Oltre alle variabili di istanza è possibile definire variabili di classe, accessibili sia tramite il nome della classe sia tramite una istanza della classe.

Esempio: Definiamo una classe `Dog`*

```
class Dog:
    # definiamo un attributo di classe
    scientific_name = 'Canis lupus familiaris'

    # definiamo un __init__ che accetta un nome
    def __init__(self, name):
        # creiamo un attributo di istanza per il nome
        self.name = name
```

**Esempio tratto da http://www.html.it/pag/15622/classi-in-python/*

Variabili di classe

```
>>> # creiamo due istanze di Dog
>>> rex = Dog('Rex')
>>> fido = Dog('Fido')
>>> # verifichiamo che ogni istanza ha un nome diverso
>>> rex.name
'Rex'
>>> fido.name
'Fido'
>>> # accediamo all'attributo di classe da Dog
>>> Dog.scientific_name
'Canis lupus familiaris'
>>> # accediamo all'attributo di classe dalle istanze
>>> rex.scientific_name
'Canis lupus familiaris'
>>> fido.scientific_name
'Canis lupus familiaris'
>>> # modifichiamo il valore dell'attributo di classe
>>> Dog.scientific_name = 'Canis lupus lupus'
>>> # verifichiamo il cambiamento dall'istanza
>>> rex.scientific_name
'Canis lupus lupus'
```

Variabili di classe

In questo esempio abbiamo visto che il nome scientifico (`'Canis lupus familiaris'`), comune a tutte le istanze di `Dog`, viene dichiarato durante la definizione della classe, ed è accessibile sia dalle istanze che dalla classe stessa. Ogni istanza ha poi un nome univoco (es. `'Rex'` e `'Fido'`), che viene definito nell'`__init__` e che è accessibile solamente dalle istanze.

E' possibile anche definire variabili di classe al di fuori della definizione della classe, ma non è buona norma

```
>>> Dog.family = 'Canidi'
>>> rex.family
'Canidi'
>>> fido.family
'Canidi'
```

Le variabili di classe sono usate per specificare proprietà generali della classe che valgono per tutte le istanze. Spesso si vuole anche che tali proprietà non siano modificabili. In Python però non esistono variabili costanti, ed è compito del programmatore fare attenzione a non modificare variabili pensate per essere in realtà costanti (ripensate alla variabile `math.pi`)

La Classe `str`

Il tipo di dato `str` (stringa), come del resto anche `int` o `float`, è in effetti una classe.

In Python una stringa è quindi un oggetto istanza della classe `str`.

Tale classe mette a disposizione un costruttore (`str()` !) e dei metodi predefiniti che possiamo invocare sulle stringhe usando la usuale "notazione punto".

Ad esempio, abbiamo già usato i metodi `upper` e `lower:` Le invocazioni `str.upper()` e `str.lower()` (`str` è una variabile a cui abbiamo precedentemente assegnato un oggetto di classe `str`) producono i seguenti effetti:

- restituiscono una stringa con tutte le lettere di `str` convertite rispettivamente in maiuscolo o in minuscolo;
- ogni carattere di `str` che non è una lettera non sarà cambiato nella stringa restituita dal metodo;
- poiché le stringhe sono oggetti immutabili, la stringa restituita sarà una nuova stringa.

Principali metodi sulle stringhe in Python

I seguenti metodi sulle stringhe restituiscono valori Booleani:

`str.isalnum()`	restituisce `True` se `str` consiste di soli caratteri alfanumerici (non ci sono simboli), `False` altrimenti
`str.isalpha()`	restituisce `True` se `str` consiste di soli caratteri alfabetici (non ci sono simboli o numeri), `False` altrimenti
`str.isdecimal()`	restituisce `True` se `str` consiste di soli caratteri numerici (`0,1,2,3,4,5,6,7,8,9`), `False` altrimenti
`str.isspace()`	restituisce `True` se `str` consiste di soli spazi, `False` altrimenti
`str.islower()`	restituisce `True` se i caratteri alfabetici in `str` sono tutti minuscoli, `False` altrimenti
`str.isupper()`	restituisce `True` se i caratteri alfabetici in `str` sono tutti maiuscoli, `False` altrimenti
`str.istitle()`	restituisce `True` se tutte le parole in `str` sono in *"titlecase"* (il primo carattere alfabetico della parola è maiuscolo e tutti gli altri caratteri alfabetici sono minuscoli), `False` altrimenti

Questi metodi restituiscono `False` anche se invocati sulla stringa vuota

Metodi su stringhe per controllare l'input: Esempio

Riprendiamo il programma per il calcolo del MCD (ad esempio quello che implementa il metodo di Euclide disponibile nel file `MCD_3.py`)

```
m = int(input("Primo numero: "))
n = int(input("Secondo numero: "))
while m != n :
    if m > n :
        m = m - n
    else :
        n = n - m
print ("Il MCD e'", m)
```

Cosa succede se m=n=0? ⇒ il risultato è 0
E se m=0 e n!=0 (o viceversa)? ⇒ si entra in **un ciclo infinito**

Se si vuole tenere conto del fatto che l'utente possa immettere un qualsiasi input, è necessario inserire una verifica sui dati in ingresso.

Metodi su stringhe per controllare l'input: Esempio

```
# Programma per il calcolo del MCD di due
# numeri positivi con algoritmo Euclide che
# controlla l'input dell'utente
m = input("Primo numero: ")
if not (m.isdecimal() and int(m) > 0) :
    print("input non corretto")
else :
    n = input("Secondo numero: ")
    if not (n.isdecimal() and int(n) > 0) :
       print("input non corretto")
    else :
        m = int(m)
        n = int(n)
        while (m != n):
           if (m > n):
              m = m - n
           else:
              n = n - m
        print ("Il MCD e'", m)
```

Principali metodi sulle stringhe in Python

Metodo `join`: `str.join(str1)` restituisce uno nuova stringa in cui i caratteri di `str1` sono inframmezzati dai caratteri di `str`.

```
>>> "a".join("bbb")
'babab'
>>> "cc".join("bbb")
'bccbccb'
>>> "".join("bbb")
'bbb'
>>> "bbb".join("")
''
```

Metodo `replace`: `str.replace()` restituisce una nuova stringa uguale ad `str` tranne che per alcune sostituzioni. Se la stringa da sostituire non compare in `str`, il metodo restituisce la stringa stessa (non una sua copia)

```
>>> baloon = "Sammy has a balloon"
>>> baloon.replace("has", "had")
'Sammy had a balloon'
```

Principali metodi sulle stringhe in Python

Metodo `split`: l'invocazione `str.split()` "spezzetta" la stringa `str` in una sequenza di sottostringhe restituita sottoforma di lista. `str` viene spezzata in corrispondenza di ciascun carattere di spaziatura, a meno che non sia data esplicitamente una stringa che funge da separatore in input a `split` (trovate ulteriori dettagli nelle slide sul tipo di dato lista).

Metodo `count`: l'invocazione `str.count(substring)` restituisce il numero di volte che `substring` occorre in `str`

la variante `str.count(substring,i,j)` restituisce il numero di volte che `substring` occorre in `str` nell'intervallo `[i-j)`

Metodo `find`: `str.find(substring)` restituisce l'indice della posizione iniziale della prima occorrenza di `substring` in `str`, se questa occorrenza esiste, restituisce `-1` altrimenti

la variante `str.find(substring,i,j)` opera come `find`, ma considera l'intervallo `[i,j)`

Principali metodi sulle stringhe in Python

Metodo `strip`: `str.strip()` restituisce una copia di `str` in cui tutti i carateri di spaziatura (compreso `"\n"`) iniziali e finali sono eliminati.

La variante `str.strip(substring)` opera come `strip()`, ma invece degli spazi elimina tutte le occorrenze dei caratteri in `substring` (fino a che non trova in `str` un caratere non contenuto in `substring`)

```
>>> "   ci sono 3 spazi all'inizio e due alla fine  ".strip()
"ci sono 3 spazi all'inizio e due alla fine"
>>> "123bbbb12312312".strip("312")
'bbbb'
```

Esistono anche le versioni `lstrip` e `rstrip` che sono analoghe a `strip` ma operano rispettivamente solo all'inizio o solo alla fine della stringa.

Metodo `capitalize`: `str.capitalize()` restituisce una nuova stringa che consiste di tutti i caratteri di `str` trasformati in minuscolo (se alfanumerici), tranne il primo, che è maiuscolo (se alfanumerico).

Il metodo `title`: `str.title()` restituisce la versione in titlecase di `str`.

La classe `list`: il metodo `append`

Anche `list` è di fatto una classe, ed ogni lista è un oggetto istanza di questa classe. Anche in questo caso abbiamo un costruttore (`list()`) e dei metodi predefiniti che facilitano la manipolazione delle liste.

Metodo `append`: `l.append(elem)` aggiunge il nuovo elemento `elem` alla fine della lista `l` (`append` è un *modificatore* dell'oggetto di invocazione e non restituisce risultati; dopo l'invocazione `l` sarà modificata)

```
>>> l = [1,2,3]
>>> l.append(12)
>>> l
[1, 2, 3, 12]
>>> l.append([12,13])
>>> l
[1, 2, 3, 12, [12, 13]]
>>> s = l.append(56)
>>> s
>>>
```

Notate che `s` "punta ad un oggetto vuoto", il cui tipo non è definito

```
>>> type(s)
<class 'NoneType'>
```

Appendere elementi ad una lista

In realtà sapevamo già come "appendere" un elemento alla fine di una lista senza usare il metodo `append`, ma attraverso la concatenazione di liste. In questo caso però non modifichiamo la lista a cui vogliamo aggiungere l'elemento, ma in effetti creiamo una lista nuova con un elemento in più:

```
>>> l=[1,2,3]
>>> id(l)
4360398344
>>> l = l + [4]
>>> id(l)
4360460104
>>>
```

Notate che l'identificatore di `l` è cambiato, perché l'operazione di concatenazione `l + [4]` ha di fatto generato un nuovo oggetto lista in cui sono stati copiati (in sequenza) gli elementi degli operandi

Se l'obiettivo è modificare la lista a cui punta `l`, l'uso di `append` è sicuramente più efficiente.

Appendere elementi ad una lista (approfondimento)

A dirla tutta esiste un altro modo per modificare una lista aggiungendo un elemento alla fine (generalizzabile all'inserimento di elementi in qualsiasi posizione): usando lo slicing.

Prima di tutto notiamo quanto segue

```
>>> l=[1,2,3,4]
>>> l[4]
Traceback (most recent call last):
  File "<pyshell#155>", line 1, in <module>
    l[4]
IndexError: list index out of range
>>> l[4:4]
[]
```

Mentre l'operazione di indicizzazione (`l[4]`) tenta di accedere ad una posizione che non esiste in `l` (la `4`), l'operazione di slicing in cui i due indici sono entrambi pari ad `i` (nell'esempio `l[4:4]`) individua la sottolista di `l` che inizia alla posizione `i`, ma non comprende questa posizione. In questi casi la sottolista è ovviamente vuota (ma va comunque considerata una sottolista di `l` posta "appena prima della posizione" `i`).

Appendere elementi ad una lista (approfondimento)

Possiamo quindi sempre appendere un elemento ad una lista `l` modificando la sua sottolista (vuota) che otteniamo tramite `l[len(l):len(l)]`.

```
Ad esempio
>>> l=[1,2,3,4]
>>> id(l)
4360460104
>>> l[4:4] = [5]
>>> l
[1,2,3,4,5]
>>> id(l)
4360460104
```

Si noti che l'id di `l` non è cambiato (si tratta quindi dello stesso elemento modificato).

Nota: in realtà questo meccanismo funziona anche con `l[x:len(l)]`, per ogni `x >= len(l)`

La classe `List`: i metodi `insert` e `extend`

Metodo `insert`: `l.insert(i,elem)` inserisce l'elemento `elem` nella lista `l` in posizione `i` (che sarà quindi la posizione di `elem` dopo l'inserimento). Tutti gli elementi che si trovavano in posizione `k>=i` prima dell'invocazione della `insert` si troveranno in posizione `k+1` dopo l'invocazione della `insert`. Si noti che `l.insert(0,elem)` inserisce `elem` in testa alla lista `l`, e `l.insert(len(a),elem))` equivale a `l.append(elem)`. Si noti anche che le seguenti istruzioni modificano `l` nello stesso modo:

```
l.insert(i, elem)
l[i:i] = [elem]
```

In particolare, la seconda espressione sostituisce la sottolista `[]` di `l` che si trova prima dell'elemento `i` con la lista `[elem]`.

Il metodo `extend` nell'invocazione `l.extend(L)` estende la lista aggiungendovi in coda tutti gli elementi della lista `L`; equivale a `l[len(l):len(l)] = L`.

Cicli che modificano il numero di elementi in una lista: Attenzione!

La seguente funzione prende in input una lista di stringhe `l` ed un intero `i` ed elimina da `l` tutte le parole che hanno lunghezza maggiore di `i`

```
def eliminaParoleErr(l,i) :
    for elem in l :
        if len(elem) > i :
            l.remove(elem)
```

Consideriamo la seguente invocazione

```
>>> l = ["mamma","sì","ciao","pippo"]
>>> eliminaParoleErr(l,2)
>>> l
['sì', 'pippo']
```

Dopo aver eliminato un elemento in posizione `i`, tutti gli elementi che lo seguono vengono spostati all'indietro di una posizione. All'iterazione successiva il ciclo `for` assegna ad `elem` l'elemento di `l` nella posizione `i+1`, e quindi "salterà" al di là dell'elemento che era il successivo prima dell'eliminazione

Cicli che modificano il numero di elementi in una lista: Attenzione!

La soluzione consiste in questo caso nel bloccare l'avanzamento dell'indice in caso di eliminazione, in modo che all'iterazione successiva si consideri nuovamente la posizione appena analizzata (che contiene un nuovo elemento). Per fare questo è necessario usare un ciclo `while`

```
def eliminaParole(l,i) :
    k = 0
    while k < len(l) :
        if len(l[k]) > i :
            l.remove(l[k])
        else :
            k = k + 1
```

La classe `List`: i metodi `remove` e `pop`

Metodo `remove`: `l.remove(elem)`, rimuove il primo elemento della lista il cui valore è `elem`. L'assenza di tale elemento produce un errore.

Metodo `pop`: `pop(i)`, rimuove l'elemento di indice `i` e lo restituisce come risultato dell'operazione. Si noti che entrambe le seguenti istruzioni modificano `l` nello stesso modo

```
l.pop(i)
l[i-1:i] = []
```

anche se `l.pop(i)` in più fornisce come risultato anche l'elemento eliminato.

Se non è specificato alcun indice, `l.pop()` rimuove l'ultimo elemento della lista e lo restituisce come risultato dell'operazione.

La classe `List`: i metodi `index`, `count`, `sort` e `reverse`

Medoto `index`: `l.index(elem)` restituisce l'indice del primo elemento della lista `l` il cui valore è `elem`. L'assenza di tale elemento produce un errore.

Metodo `count`: `l.count(elem)` restituisce il numero di occorrenze di `elem` nella lista `l`.

Metodo `sort`: `l.sort()` ordina gli elementi della lista `l`, modificando l'oggetto di invocazione

Metodo `reverse`: `l.reverse()` inverte gli elementi della lista `l`, modificando l'oggetto di invocazione

Copia di liste

Possiamo copiare una lista usando la funzione `copy`:

```
>>> a = [1, 2, 3]
>>> from copy import copy
>>> b = copy(a)
>>> print(b)
[1, 2, 3]
```

Oppure possiamo usare il costruttore della classe `list`:

```
>>> a = [1, 2, 3]
>>> b = list(a)
>>> print(b)
[1, 2, 3]
```

Oppure possiamo usare l'operatore di slicing:

```
>>> a = [1, 2, 3]
>>> b = a[:]
>>> print(b)
[1, 2, 3]
```

`a` e `b` si riferiscono sempre a due oggetti diversi che però contengono gli stessi valori. Quindi `a==b` restituirà `True` mentre `a is b` restituirà `False`

La classe tuple

Il costruttore di classe tuple() invocato con una lista in input costruisce una tupla con gli stessi elementi della lista

```
>>> tuple([1,2,[1,2]])
(1, 2, [1, 2])
```

In verità il costruttore può prendere in input un oggetto di tipo `iterator` (oltre la lista, una stringa, o il risultato dell'invocazione di un range)

```
>>> tuple("ciao")
('c', 'i', 'a', 'o')
>>> tuple(range(9))
(0, 1, 2, 3, 4, 5, 6, 7, 8)
```

`tuple()` restituisce la tupla vuota `()`

Altri metodi disponibili sulle tuple: `count` e `index` (si comportano come su oggetti di classe lista)