Table of Contents

Programmazione Object Oriented in Python

Appunti recuperati da vecchie lezioni. Da revisionare, integrare, mettere in ordine, ecc.

Questo testo è distribuito con Licenza Creative Commons Attribuzione - Condividi allo stesso modo 4.0 Internazionale. Autore: Loris Tissino.

Il codice sorgente degli esempi è disponibile su GitHub.

Introduzione

La documentazione ufficiale su Python è disponibile a partire dai siti www.python.it e www.python.org.

In queste lezioni farò riferimento alla versione 3.1 di Python, per cui se si ha occasione di leggere documentazione su versioni 2.x, è consigliabile un'occhiata a Moving from Python 2 to Python 3, che riporta in estrema sintesi le maggiori differenze.

Per chi ha già un po' di esperienza di programmazione, può essere utile dare un'occhiata a Learn Python in 10 minutes.

Alcune guide (alcune forse un po' datate) che possono comunque essere d'aiuto sono le seguenti:

- Dive into Python
- Guida Python su HTML.it
- Pensare da informatico, versione Python
- Python per sopravvivere
- Passo dopo passo impariamo la programmazione in Python
- Python 3 for Scientists

Sulla programmazione OOP in Python vedere:

- Introduction to OOP in Python
- Learn to think like a Computer Scientist
- Python Types and Objects
- Python Attributes and Methods

Primi esempi con cose di base

Introduzione

Il codice seguente va copiato ed incollato in una finestra di IDLE - l'editor di base di Python, per poter essere eseguito e compreso, tramite il confronto fra le istruzioni eseguite e l'output ottenuto.

Commenti

```
"""Un primo esempio di programma Python
In questo esempio vedremo:
- come si scrivono i commenti
- come si formatta l'output
- come si usa la capacità introspettiva del linguaggio
- come si ottiene un input
- come si gestiscono selezioni e cicli
"""

"""I commenti possono essere scritti su una riga a partire dal p
unto in
cui è presente un #, oppure su più righe (come queste)
"""
```

Output e stringhe di formattazione

```
print("--------")

amount=50 # assegna il valore 50 alla variabile importo

print(amount)
# visualizza il valore di importo

print("L'importo è ", amount, ".")
# visualizza la costante tra virgolette e poi il valore
```

```
# questo tipo di output è sconveniente perché spezza la frase
# (ad esempio, sono difficili le traduzioni)

print("L'importo è %d." % amount)
# visualizza il valore usando una stringa di formattazione
# (%d è un segnaposto per i numeri interi)
# questo tipo di output, simile a quello delle funzioni printf e
 sprintf del C,
# è deprecato e non sarà più valido in futuro

rate=.073

print("Il tasso è %4.2f%%." % (rate*100))
# la stringa di formattazione è un po' più complicata, perché st
iamo
# rappresentando un valore in virgola mobile (segnaposto %f) per
 il quale
# diciamo quante cifre destinare alla parte dopo la virgola

print("L'importo è %d e il tasso è %4.2f%%." % (amount, rate*100
))
# stringa di formattazione con due segnaposto: i valori
# vanno inseriti in una tupla

print("L'importo è %(amount)s e il tasso è %(rate)s%%." % {"amou
nt": amount, "rate": rate*100})
# stringa di formattazione con uso di segnaposto aventi un nome
# e un dizionario di associazioni per i valori

print("L'importo è {amount} e il tasso è {rate}%.".format(amount
=amount, rate=rate*100))
# usando il metodo format della classe str, si possono usare par
ametri con nome
```

Questi ultimi due esempi producono il seguente output:

```
L'importo è 50 e il tasso è 7.3%.
```

```
print(id(amount))
# visualizza il codice univoco associato all'oggetto amount
print(type(amount))
# visualizza il tipo dell'oggetto
```

```
<class 'int'>
```

```
name=input("Come ti chiami? ")
print("Ciao, %s." % name)
# per un input da parte dell'utente, è sufficiente usare la funz
ione input
print(len(name))
# visualizza la lunghezza del nome
```

Test

```
age=input("Quanti anni hai? ")
if age.isdigit():
    print("Hai inserito un numero.")
    to100 = 100 - int(age)
    if to100>0:
        print("Ti mancano %d anni per arrivare a 100!" % to100)
else:
    print("Hai inserito qualcosa che non è un numero.")
```

Cicli "while"

```
print("vediamo un ciclo while...")
n=2
while(n<5):
    print(n)
    n+=1 # incremento della variabile n di una unità
print("ciclo while finito...")

# se vogliamo un test in coda (come nel repeat... until del pasc
al)
# possiamo usare la parola chiave break
n=2
print("ciclo con break condizionato")
while(True):
    print(n)
    n+=1
    if(n>=5):
        break
print("fine ciclo con break condizionato")
```

Cicli "for"

```
# I cicli for si basano su oggetti che possono essere iterati:
print("ciclo for")
for fruit in ["arancia", "mela", "pera", "ananas"]:
    print(fruit)
```

`arancia` `mela` `pera` `ananas`

```
fruits=["arancia", "mela", "pera", "ananas"]
for fruit in fruits:
    print(fruit)

arancia
mela
pera
ananas
```

Se si deve iterare su più liste in parallelo si può usare la funzione zip (il ciclo termina con l'esaurimento della lista di lunghezza inferiore).

```
print("*** Ciclo for basato su diversi oggetti iterabili percors
i in parallelo")
colors=["arancio", "bianco", "bianco", "giallo"]
for fruit, color in zip(fruits, colors):
    print('Il frutto %s ha la polpa color %s.' % (fruit, color))
```

```
 *** Ciclo for basato su diversi oggetti iterabili percorsi in
parallelo Il frutto arancia ha la polpa color arancio. Il frutto
mela ha la polpa color bianco. Il frutto pera ha la polpa color
bianco. Il frutto ananas ha la polpa color giallo.
```

```
sentence="ma che bel castello marcondirodirondello..."
for letter in sentence:
    if letter in "aeiou":
        print(letter.upper(), end="") # mettiamo in maiuscolo le
 vocali...
    else:
        print(letter, end="")
print()
```

Cicli con indici numerici

```
print("*** Ciclo for su range(5)")
for i in range(5):
    print(i)

*** Ciclo for su range(5)
0
1
2
3
4

print("*** Ciclo for su range(5, 10)")
for i in range(5,10):
    print(i)

*** Ciclo for su range(5, 10)
5
6
7
8
9
```

Come si può notare, non si raggiunge l'estremo superiore.

```
print("*** Ciclo for su range(10, 20, 2)")
for i in range(10, 20, 2):
    print(i)

*** Ciclo for su range(10, 20, 2)
10
12
14
16
18
```

Cicli alla rovescia

Per fare un ciclo alla rovescia basta usare range con un incremento negativo:

```
print("*** Ciclo for su range(5, 0, -1)")
for i in range(5,0,-1):
  print(i)

*** Ciclo for su range(5, 0, -1)
5
4
3
2
1
```

In questo caso, non si raggiunge il limite inferiore.

Nel caso si abbia la necessità di visualizzare i caratteri (che vengono indicizzati a partire da 0) della stringa s partendo dall'ultimo, dovremo fare un ciclo strutturato in questo modo:

```
print("*** Ciclo for per i caratteri di una stringa alla rovesci
a")
name="Mario"
for i in range(len(name)-1,-1,-1):
  print(name[i])

*** Ciclo for per i caratteri di una stringa alla rovescia
o
i
r
a
M
```

Si tenga presente che, comunque, gli elementi di una sequenza possono essere gestiti anche ricorrendo ad un indice negativo: -1 corrisponde all'ultimo elemento, -2 al penultimo, ecc. Avremmo quindi potuto scrivere anche:

```
name="Mario"
for i in range(1,len(name)+1): # ciclo da 1 alla lunghezza della
 stringa +1
  print(-i, name[-i]) # uso l'opposto di i

-1 o
-2 i
-3 r
-4 a
-5 M
```

Alcune cose sulle funzioni

Introduzione

I commenti posti all'inizio della funzione hanno uno scopo di documentazione, ma anche, se ben predisposti, di semplice test del codice.

Sulla gestione delle eccezioni (blocco try… except) torneremo più avanti.

Nella funzione per il calcolo dell'area si noti l'uso delle asserzioni per la verifica dei parametri.

Nella funzione per le soluzioni di un'equazione di secondo grado, si noti che viene restituito None nel caso di assenza di soluzioni reali, oppure la coppia di soluzioni.

È anche utile sapere che:

- le funzioni, come in Pascal e a differenza del C, possono contenere al loro interno la definizione di altre funzioni (dette locali);
- i parametri delle funzioni possono avere un valore di default e un nome, in modo da poterli omettere (viene preso il valore di default) oppure essere passati in ordine diverso (con il loro nome).

Una semplice funzione per l'input controllato di numeri

```
def input_int(message):
    """Permette l'inserimento di un numero intero in maniera con
trollata.

    Chiede in input una stringa usando *message* come prompt, po
i
    tenta la conversione in numero intero e restituisce il numer
o
    convertito. Se viene inserito un valore che non è un numero,
    viene richiesto di nuovo l'inserimento.
    """
    while(True):
        v=input(message)
        try:
            n=int(v)
            return n
        except ValueError as err:
            print("Sembra che tu non abbia inserito un numero in
tero…")

k=input_int("Inserisci un numero: ")
print(k)
```

Una funzione migliore, che consente di inserire anche numeri in virgola mobile

```
def input_number(message, accept_float=False):
    """Permette l'inserimento di un numero in maniera controllat
a.

    Chiede in input una stringa usando *message* come prompt, po
i
    tenta la conversione in numero e restituisce il numero
    convertito. Se viene inserito un valore che non è un numero,
    viene richiesto di nuovo l'inserimento.
    Se *accept_float* è vero, accetta anche numeri in virgola mo
bile.
    """
    while(True):
        v=input(message)
        try:
            if accept_float:
                n=float(v)
            else:
                n=int(v)
            return n
        except ValueError as err:
            print("Sembra che tu non abbia inserito un numero va
lido…")

k=input_number("Inserisci un numero: ", True)
print(k, type(k))
```

Una funzione per il calcolo dell'area di un cerchio

```
import math
def circle_area(radius):
    """Restituisce l'area di un cerchio, dato il raggio *radius*
.

>>> print(circle_area(12))
452.389342117
>>> print(circle_area(12.3))
475.291552562
"""

    assert(isinstance(radius, (int, float)))
    return radius ** 2 * math.pi


# Le istruzioni seguenti possono essere usate
# per eseguire i test messi nel commento
import doctest
doctest.testmod()
```

Una funzione per il calcolo delle soluzioni reali di un'equazione di secondo grado

```
def equation_solutions(a, b, c):
    """Restituisce una tupla con le soluzioni di un'equazione di
 secondo grado

    I parametri *a*, *b* e *c* sono i coefficienti dell'equazion
e in forma
    normalizzata (devono essere float).
    La funzione restituisce None nel caso in cui l'equazione non
 ammetta
    soluzioni.

    >>> print(equation_solutions(1.0, 3.0, 2.0))
    (-2.0, -1.0)
    >>> print(equation_solutions(1.0, 1.0, 2.0))
    None
    """

    for p in a, b, c:
        assert(isinstance(p, float))
    d = b ** 2 - 4*a*c
    if d<0:
        return None
    else:
        d = math.sqrt(d)
        return ((-b-d)/(2*a), (-b+d)/(2*a))
```

Alcune funzioni con parametri opzionali

```
def sayhello1(message):
    """Visualizza un messaggio di saluto (da indicare obbligator
iamente)

    >>> sayhello1("hello, world!")
    hello, world!
    """
    print(message)

def sayhello2(message="ciao"):
    """Visualizza un messaggio di saluto (in assenza di indicazi
one, "ciao")

    >>> sayhello2("hello, world!")
    hello, world!
    >>> sayhello2()
    ciao
    """

    print(message)


def sayhello3(message="ciao", times=2):
    """Visualizza un messaggio di saluto *times* volte

    >>> sayhello3("ciao")
    ciaociao
    >>> sayhello3(message="ciao", times=3)
    ciaociaociao
    >>> sayhello3(times=1, message="ciao")
    ciao
    """

    print(message*times)
```

Argomenti annotati

È possibile aggiungere delle annotazioni per ciascuno degli argomenti di una funzione. L'annotazione non viene usata da Python, ma può essere utile a scopo di documentazione.

```
def rectangle_area(width: "the width, expressed in cm", height:
"the height, expressed in cm"):
    """Restituisce l'area di un rettangolo, base per altezza
    """

    return (width * height)

print(rectangle_area(5, 12))

help(rectangle_area)
```

L'output sarà il seguente:

```
60
Help on function rectangle_area in module __main__:

rectangle_area(width: 'the width, expressed in cm', height: 'the
 height, expressed in cm')
    Restituisce l'area di un rettangolo, base per altezza
```

Funzioni senza argomenti

Naturalmente, è possibile definire funzioni senza argomenti:

```
def computeUltimateAnswerToTheUltimateQuestionOfLifeTheUniverseA
ndEverything()
   return 42
   # see http://tinyurl.com/ultimateanswer42
```

Nel richiamo, andranno messe le parentesi:

```
answer=computeUltimateAnswerToTheUltimateQuestionOfLifeTheUniver
seAndEverything()
```

Valore restituito in mancanza di esplicitazione

Una funzione restituisce sempre un valore che, se non è indicato esplicitamente, è None. Questo valore di default restituito può essere comodo. Si pensi al seguente esempio di una funzione che deve restituire il rapporto tra due numeri, ma None nel caso in cui il secondo numero sia zero:

```
def Ratio(numerator, denominator):
    if denominator!=0:
        return numerator/denominator
    else:
        return None

print(Ratio(5,2))
print(Ratio(5,0))

2.5
None
```

La funzione avrebbe esattamente lo stesso comportamento se omettiamo di scrivere il ramo "else":

```
def Ratio(numerator, denominator):
    if denominator!=0:
        return numerator/denominator
```

Aiuto sulle funzioni predefinite

Tutte le funzioni predefinite di Python mettono a disposizione della documentazione sul loro uso direttamente nella shell di Python.

```
>>> help(ord)
Help on built-in function ord in module builtins:

ord(...)
    ord(c) -> integer

    Return the integer ordinal of a one-character string.

>>> help(math.sqrt)
Help on built-in function sqrt in module math:

sqrt(...)
    sqrt(x)

    Return the square root of x.

>>> s="python"
>>> help(s.upper)
Help on built-in function upper:

upper(...)
    S.upper() -> str

    Return a copy of S converted to uppercase.
```

Esercizi

1. modificare la funzione `circle_area()` con un asserzione nella quale si verifica che il raggio è non negativo.
2. modificare la funzione `equation_solutions()` aggiungendo l'asserzione mancante (da individuare)
3. scrivere un programma che:
    - chieda in input il raggio di un cerchio in maniera controllata;
    - calcoli l'area;
    - visualizzi un messaggio del tipo "L'area di un cerchio di raggio X è Y."
4. scrivere un programma che:

- chieda all'utente i parametri a, b e c di un'equazione di secondo grado, accettandoli anche interi;
- chiami la funzione equation_solutions() per ottenere le soluzioni;
- visualizzi un messaggio con il risultato, in una forma tipo "L'equazione con parametri 1.0, 1.0, 2.0 non ha soluzione" (oppure "ha le seguenti soluzioni: …")

Tipi di dati

Tipi numerici

I tipi numerici sono *int*, *float*, *decimal.Decimal* e *fractions.Fraction*. Esiste anche il supporto per i numeri complessi.

I valori interi non sono limitati dalle caratteristiche del processore, per cui si ottiene una precisione che dipende solo dalla memoria:

```
>>> n=2**172
>>> print(n)
5986310706507378352962293074805895248510699696029696
```

In Python anche i semplici numeri interi sono oggetti, per i quali la memoria è allocata dinamicamente (oltre un certo valore, la quantità di memoria occupata dipende in maniera proporzionale dal numero di cifre).

Il tipo *decimal.Decimal* può essere utile per gli importi con decimali, che non possono essere approssimati come normalmente si fa per i valori in virgola mobile.

Booleani

I valori booleani sono di tipo *bool*. Le variabili possono essere impostate a `True` o `False`, ed esistono gli operatori `and`, `or` e `not`. Python usa una valutazione "di corto circuito" (ad esempio, se in un `and` il primo operando è falso, non viene valutato il secondo).

Stringhe

Le stringhe sono memorizzate in formato Unicode, per cui le conversioni minuscolo/maiuscolo funzionano anche con le lettere non US-ASCII e si ottengono codici numerici multi-byte:

```
>>> s="perché"
>>> s.upper()
'PERCHÉ'
>>> print(ord("€"))
8364
>>> print(chr(8364))
€
```

Sono a disposizione molte funzioni per operare sulle stringhe, troppe per elencarle qui.

Molto utile è la possibilità di analizzare parti di stringa (il cosiddetto *slicing*):

```
>>> name="Mario Rossi"
>>> print(name[0])
M
>>> print(name[1])
a
>>> print(name[0:4])
Mari
>>> print(name[-3])
s
>>> print(name[-3:])
ssi
```

È molto utile la possibilità di consultare una guida in linea direttamente nell'interprete di Python. Ad esempio, supponiamo di avere una variabile name con il valore "Pippo":

```
name="Pippo"
```

Digitando nell'interprete

```
name.
```

e attendendo qualche secondo, comparirà una lista di funzioni (metodi) associati all'oggetto name (capitalize, center, count, encode, ecc.).

Un elenco delle funzioni disponibili si può ottenere anche con il comando

```
dir(name)
```

oppure, se sappiamo che name è di tipo *str*, con

```
dir(str)
```

Se vogliamo avere informazioni su come usare una di queste funzioni, nell'esempio la funzione *find*, possiamo semplicemente digitare

```
help(name.find)
```

per ottenere informazioni sulla funzione stessa:

```
Help on built-in function find:

find(...)
    S.find(sub[, start[, end]]) -> int

    Return the lowest index in S where substring sub is found,
    such that sub is contained within s[start:end].  Optional
    arguments start and end are interpreted as in slice notation
.

    Return -1 on failure.
```

Sequenze

Le sequenze sono tipi di dati avanzati con i quali è possibile iterare e usare l'operatore di appartenenza in. Inoltre, con esse si può usare la funzione *len()* e lo *slicing* ("affettatura", come con le stringhe, quando si estraggono i caratteri).

Tra le sequenze ricordiamo, oltre alle stringhe:

- le tuple (sequenze immutabili)
- le liste (sequenze modificabili)
- i byte
- gli array di byte

Parleremo delle ultime due in occasione delle lezioni sui file binari.

I valori delle sequenze sono riferimenti a oggetti, e gli oggetti possono essere di tipo diverso. Quindi, è possibile che in una sequenza siano compresi, mescolati tra loro, numeri interi, numeri in virgola mobile, stringhe, tuple, liste, ecc.

Tuple

Le tuple sono insiemi di dati (o, meglio, di riferimenti ad oggetti). Una volta create, non sono modificabili.

```
>>> mynumbers=(2, 10, 12, 17, 18)
>>> 2 in mynumbers
True
>>> 3 in mynumbers
False
>>> mynumbers[0]
2
>>> mynumbers[-1]
18
>>> mynumbers[3:5]
(17, 18)
```

ma...

```
>>> mynumbers[0]=3
Traceback (most recent call last):
  File "<pyshell#41>", line 1, in <module>
    mynumbers[0]=3
TypeError: 'tuple' object does not support item assignment
```

Le tuple possono essere usate anche a sinistra per assegnare più valori in un colpo solo:

```
a,b,c = mynumbers[0:3]   # si estraggono tre valori e li si asse
gna
a,b,c,*others = mynumbers # tutti i valori sono considerati, ma
i rimanenti vengono messi in una lista
```

Per far sì che una tupla contenga un solo elemento, è necessario mettere una virgola dopo il valore:

```
>>> a=(1234,)
>>> a
(1234,)
>>> len(a)
1
```

Diverse sintassi per la creazione di una tupla

```
Per creare una tupla si possono usare diverse notazioni. Queste
due sono equivalenti (ma la seconda è raccomandabile per chiarez
za):
>>> a=12,32,44,67
>>> a=(12,32,44,67)
```

Per creare una tupla a partire da un'altra sequenza, si può usare forzare la conversione:

```
>>> c=[12,34,45] # questa è una lista
>>> b=tuple(c) # la tupla b viene creata con i dati di c
```

Liste

Le liste sono insiemi di riferimenti come le tuple, ma a differenza di queste possono essere modificate (si possono cambiare i valori degli elementi, aggiungerne, toglierli, ecc.).

```
>>> mycats=['Birillo', 'Fuffi', 'Silvestro']
>>> mycats.append('Billy')
>>> mycats
['Birillo', 'Fuffi', 'Silvestro', 'Billy']
>>> print(mycats)
['Birillo', 'Fuffi', 'Silvestro', 'Billy']
>>> mycats[0]='Ribillo'
>>> mycats[0]='Ribillo'
>>> lastcat=mycats.pop()
>>> print(lastcat)
Billy
```

Ad una lista possono essere aggiunti elementi in coda (prendendoli da un'altra lista o uno a uno). Gli elementi possono anche essere inseriti nella posizione desiderata:

```
>>> mydogs=['Fido', 'Bobbie', 'Charlie']
>>> mydogs
['Fido', 'Bobbie', 'Charlie']
>>> lostdogs=['Dunnie', 'Bah']
>>> mydogs.extend(lostdogs)   # aggiunge in coda un'intera lista
>>> mydogs
['Fido', 'Bobbie', 'Charlie', 'Dunnie', 'Bah']
>>> mydogs.insert(1, 'Birillo')   # inserisce un elemento nella
posizione prima di quella indicata
>>> mydogs
['Fido', 'Birillo', 'Bobbie', 'Charlie', 'Dunnie', 'Bah']
```

Ordinamento

Una lista può essere ordinata con il metodo *sort()*:

```
>>> print(mycats)
['Fuffi', 'Ribillo', 'Silvestro', 'Billy']
>>> mycats.sort()
>>> print(mycats)
['Billy', 'Fuffi', 'Ribillo', 'Silvestro']
```

L'ordinamento può avvenire solo a patto che gli oggetti della lista siano comparabili. Ad esempio, una lista contenente tuple e numeri interi non può essere ordinata direttamente:

```
>>> mynumbers=[ (12, 34), 13 ]
>>> mynumbers.sort()
TypeError: unorderable types: int() < tuple()
```

È però possibile definire una funzione da richiamare per riportare a un tipo comune gli oggetti della lista (ad esempio, potremmo decidere di estrarre il primo valore di una tupla e considerarlo per la comparazione), e poi usare la funzione come argomento del metodo sort:

```
def FirstItem(a):
   if isinstance(a, tuple):
       return a[0]
   return a

mynumbers=[ (12, 34), 13 ]
mynumbers.sort(key=FirstItem)
print(mynumbers)

mynumbers=[ (12, 34), 11 ]
mynumbers.sort(key=FirstItem)
print(mynumbers)

[(12, 34), 13]
[11, (12, 34)]
```

Diverse sintassi per la creazione di una lista

Per creare una lista si usano, come visto negli esempi, le parentesi quadre

```
>>> a=[12,32,44,67]
```

Per creare una lista a partire da un'altra sequenza, si può usare forzare la conversione:

```
>>> m=(12,34,45) # questa è una tupla
>>> n=list(m) # la lista n viene creata con i dati di m
```

Organizzazione del codice

I moduli

Il codice di un programma python può essere utilmente organizzato in moduli. Ad esempio, si può pensare di salvare un file *simplemath.py* con il seguente codice (ripreso da una lezione precedente):

```
import math

def input_number(message, accept_float=False):
    """Permette l'inserimento di un numero in maniera controllat
a.

    Chiede in input una stringa usando *message* come prompt, po
i
    tenta la conversione in numero e restituisce il numero
    convertito. Se viene inserito un valore che non è un numero,
    viene richiesto di nuovo l'inserimento.
    Se *accept_float* è vero, accetta anche numeri in virgola mo
bile.
    """
    while(True):
        v=input(message)
        try:
            if accept_float:
                n=float(v)
            else:
                n=int(v)
            return n
        except ValueError as err:
            print("Sembra che tu non abbia inserito un numero va
lido…")


def circle_area(radius):
    """Restituisce l'area di un cerchio, dato il raggio *radius*
```

```
.

    >>> print(circle_area(12))
    452.389342117
    >>> print(circle_area(12.3))
    475.291552562
    """

    assert(isinstance(radius, int) or isinstance(radius, float))
    return radius ** 2 * math.pi


if __name__=="__main__":
    r=input_number("Inserisci il raggio del cerchio: ")
    a=circle_area(r)
    print("L'area di un cerchio di raggio %f è %f." % (r, a))
```

Come si può notare, il codice del modulo ha due parti: la prima contiene le funzioni definite, mentre la seconda quello che in altri linguaggi di programmazione si chiama "programma principale".

In pratica, se questo codice viene eseguito direttamente, vengono eseguite le istruzioni di richiesta di un numero, calcolo dell'area e output. Se invece il modulo viene importato (vedi esempio successivo), queste istruzioni non vengono eseguite.

Importazione di moduli

L'importazione di un modulo avviene semplicemente indicando il nome del file, senza l'estensione. Tutte le funzioni (e le classi, come vedremo) definite nel file vengono messe a disposizione. Il file importato deve trovarsi nella directory del file principale, oppure in una tra quelle predefinite.

Importazione completa

Se il modulo viene interamente importato, tutte le sue funzioni vengono messe a disposizione. In caso di nomi ripetuti sia nel modulo sia nel codice del proprio file, si può specificare quale funzione usare attraverso la specificazione del modulo:

```
import simplemath

"""
Importando il modulo simplemath non vengono eseguite le istruzio
ni
presenti nel "programma principale"
"""

def circle_area(n):
    return 1

if __name__=='__main__':
    for i in range(0,10):
        print(i, simplemath.circle_area(i))
        print(i, circle_area(i))
```

Importazione solo di alcune funzioni

Se lo si desidera, si possono importare solo alcune funzioni:

```
from simplemath import circle_area

if __name__=='__main__':
    for i in range(0,10):
        print(i, circle_area(i))
```

Un esempio completo

Immaginiamo di voler raccogliere in un modulo chiamato *usefulfunctions* alcune funzioni utili. Sarà sufficiente salvare il file *usefulfunctions.py*, con un contenuto simile al seguente:

```
"""
Questo è un modulo che contiene una funzione che somma tutti i
numeri interi presenti in una lista.

Si noti che la funzione contiene un commento che contiene a sua
volta
degli esempi di esecuzione con risultato atteso.

Se si esegue il modulo come codice autonomo i test vengono effet
tivamente svolti.
"""

def sumup(l):
    """Restituisce la somma di tutti i valori interi di una list
a o di una tupla

    Tutti i valori non interi vengono ignorati in silenzio.
    Non è ricorsiva: vengono presi in considerazione solo i valo
ri del primo
    livello della lista (o tupla).

    >>> print(sumup((1,3)))
    4
    >>> print(sumup([12, 20, 40.12, 'foo', 13]))
    45
    """

    assert(isinstance(l, (list, tuple)))
    s=0
    for i in l:
        if isinstance(i, int):
            s+=i
    return s

if __name__=='__main__':
    import doctest
    r = doctest.testmod()
    print(r)
```

Eseguendo questo file viene richiamato il codice indicato alla fine, cioè vengono effettivamente svolti i test di esempio presenti nel commento della funzione:

```
TestResults(failed=0, attempted=2)
```

Se in un altro programma abbiamo bisogno della funzione sumup(), non dovremo fare altro che importare il modulo:

```
import usefulfunctions

def countdown(n, k=100, step=1):
    '''restituisce una lista di *n* valori interi, decrementando
 a partire da *k*'''
    l=[]
    for i in range(n):
        l.append(k-i*step)
    return l


if __name__=='__main__':
    n=countdown(10,250, 3)
    print(n)
    print('La somma è %d.' % sumup(n))
```

Il risultato sarà:

```
[250, 247, 244, 241, 238, 235, 232, 229, 226, 223]
La somma è 2365.
```

Packages

Volendo, quando il codice comincia a essere complesso e richiede un'organizzazione più complessa, i vari moduli possono essere organizzati in pacchetti (*packages*).

Un package è in sostanza una directory contenente diversi moduli e un file (`__init.py__`) che funziona come una sorta di indice e/o file di inizializzazione.

Insiemi e dizionari

Introduzione

I dizionari (*dict*) sono delle strutture dati in cui gli elementi sono memorizzati in coppie chiave/valore. Sono simili a quelli che in altri linguaggi di programmazione vengono chiamati *hash* o *array associativi*.

Negli insiemi (*set*), invece, sono presenti solo le chiavi.

Valori ammissibili come chiavi

Come chiave di un insieme o di un dizionario, possiamo usare qualsiasi valore di tipo immutabile (oggetti di tipo *int*, *str*, *frozenset*, tuple). Teoricamente anche i valori di tipo *float* potrebbero essere usati, ma questo è sconsigliato. Non possono invece essere usati come chiavi i valori di oggetti mutabili (liste, dizionari, insiemi) perché il valore della funzione hash per questi oggetti cambia con il cambiare del valore.

Gli insiemi

Un insieme (*set*) può essere creato usando la notazione delle parentesi graffe, specificando solo le chiavi:

```
>>> days={'monday', 'tuesday', 'wednesday', 'thursday', 'friday'
}
>>> print(days)
{'monday', 'tuesday', 'friday', 'wednesday', 'thursday'}
```

È importante notare che gli insiemi non ammettono chiavi duplicate (non viene segnalato un errore, semplicemente il valore duplicato viene ignorato), e che non è garantito l'ordine (nella visualizzazione l'ordine può essere diverso rispetto a quello in cui sono stati inseriti i dati).

```
>>> n=(12,12,32,33)  # n è una lista
>>> m=set(n)  # creo m come insieme, a partire dai valori di n
>>> print(m)  # visualizzo m, e vedo che il doppio 12 non c'è
{32, 33, 12}
```

Per rimuovere un elemento da un insieme, si usa il metodo remove():

```
>>> m.remove(12)
>>> m
{32, 33}
```

Per aggiungere un elemento a un insieme, si usa il metodo add():

```
>>> m.add(15)
>>> m
{32, 33, 15}
```

Gli insiemi immutabili: *frozenset*

Esiste una variante di insieme di tipo immutabile, nel quale gli elementi non possono essere aggiunti, rimossi o modificati. Si chiama *frozenset*.

```
>>> n=(11,21,31,41) # n è una lista
>>> m=frozenset(n)  # creo m come insieme congelato, a partire d
ai valori di n
>>> m.add(51)

Traceback (most recent call last):
  File "/tmp/frozenset.py", line 3, in <module>
    m.add(51)
AttributeError: 'frozenset' object has no attribute 'add'
```

I dizionari

Per creare (e/o aggiungere elementi a) un dizionario si possono usare diverse sintassi. Presentiamo qui di seguito alcuni esempi, corredati da alcuni esempi di visualizzazione delle coppie:

```
months={ 'jan': 'Gennaio', 'feb': 'Febbraio', 'mar': 'Marzo' }

# equivalente a:
# months=dict(jan='Gennaio', feb='Febbraio', mar='Marzo')

print(months)
{'jan': 'Gennaio', 'mar': 'Marzo', 'feb': 'Febbraio'}

months['apr'] = 'Aprile'
months['may'] = 'Maggio'
months['june'] = 'Giugno'

print(months)
{'mar': 'Marzo', 'feb': 'Febbraio', 'apr': 'Aprile', 'june': 'Gi
ugno', 'jan': 'Gennaio', 'may': 'Maggio'}
```

Il metodo *keys()* restituisce le chiavi (in forma di *object view*, vedi più avanti):

```
print('\nChiavi:')
for key in months.keys():
    print(key)

Chiavi:
mar
feb
apr
june
jan
may
```

Il metodo *values()* restituisce i valori (in forma di *object view*, vedi più avanti):

```
print('\nValori:')
for value in months.values():
    print(value)
Valori:
Marzo
Febbraio
Aprile
Giugno
Gennaio
Maggio
```

Il metodo *items()* restituisce l'insieme delle tuple chiave/valore (in forma di *object view*, vedi più avanti)::

```
print('\nChiavi e valori:')
for key, value in months.items():
    print(key, ' --> ', value)

Chiavi e valori:
mar  -->  Marzo
feb  -->  Febbraio
apr  -->  Aprile
june  -->  Giugno
jan  -->  Gennaio
may  -->  Maggio
```

Se si desidera creare un dizionario a partire da due liste (o due tuple), delle quali una contiene le chiavi e l'altra i valori, è possibile usare la funzione zip:

```
weekdays=dict(zip(
    ('mon', 'tue', 'wed', 'thu', 'fri', 'sat', 'sun'),
    ('Lunedì', 'Martedì', 'Mercoledì', 'Giovedì', 'Venerdì', 'Sa
bato', 'Domenica')
    ))

print(weekdays)

{'wed': 'Mercoledì', 'sun': 'Domenica', 'fri': 'Venerdì', 'tue':
 'Martedì', 'mon': 'Lunedì', 'thu': 'Giovedì', 'sat': 'Sabato'}
```

Per rimuovere un elemento da un dizionario, si usa la parola chiave del:

```
>>> d={'a':100, 'b':200, 'k':400}
>>> d
{'a': 100, 'k': 400, 'b': 200}
>>> del d['b']
>>> d
{'a': 100, 'k': 400}
```

Estrazione di valori

Per accedere in modo sicuro al valore di un dizionario basandosi su una chiave (che potrebbe non esistere) possiamo usare il suo metodo *get()*, che consente anche di definire un valore di default (usato nel caso di mancanza dell'elemento):

```
>>> k=dict(a=12, b=13, c=25)
>>> print(k)
{'a': 12, 'c': 25, 'b': 13}
>>> v=k.get('a')
>>> print(v)
12
>>> v=k.get('n')
>>> print(v)
None
>>> v=k.get('n', 'No value defined')
>>> print(v)
No value defined
```

Operazioni insiemistiche

Sui dizionari e sugli insiemi si possono fare operazioni di tipo insiemistico. La più comune è l'uso dell'operatore in:

```
for month in ('Maggio', 'Giugno', 'Luglio', 'Agosto'):
    if month in months.values():
        print('%s appartiene all\'insieme dei mesi' % month)
    else:
        print('%s NON appartiene all\'insieme dei mesi' % month)
```

Con i dati impostati precedentemente, l'output sarebbe il seguente:

```
Maggio appartiene all'insieme dei mesi
Giugno appartiene all'insieme dei mesi
Luglio NON appartiene all'insieme dei mesi
Agosto NON appartiene all'insieme dei mesi
```

Inoltre, sono supportate le operazioni di intersezione, unione, differenza e differenza simmetrica:

```
evens={2,4,6,8,10,12,14,16,18,20} # questo è un set
multiples_of_three={3,6,9,12,15,18}

print('Intersezione: ', evens & multiples_of_three)
print('Unione: ', evens | multiples_of_three)
print('Differenza simmetrica: ', evens ^ multiples_of_three)
print('Differenza1: ', evens - multiples_of_three)
print('Differenza2: ', multiples_of_three - evens)

Output:
Intersezione:  {18, 12, 6}
Unione:  {2, 3, 4, 6, 8, 9, 10, 12, 14, 15, 16, 18, 20}
Differenza simmetrica:  {2, 3, 4, 8, 9, 10, 14, 15, 16, 20}
Differenza1:  {2, 4, 8, 10, 14, 16, 20}
Differenza2:  {9, 3, 15}
```

Chiavi e valori come *object view*

Quando si estraggono le chiavi e i valori da un dizionario con i metodi keys(), values() e items(), si ottiene un riferimento dinamico al dizionario stesso. Ciò significa che se il dizionario cambia, le modifiche avranno un riflsso sulle chiavi e i valori ottenuti. Se vogliamo una copia statica, dobbiamo farcela esplicitamente, mettendo i valori estratti in una tupla. Chiariamo questo concetto con un esempio pratico:

```
months={ 'jan': 'Gennaio', 'feb': 'Febbraio', 'mar': 'Marzo' }

print(months)
{'jan': 'Gennaio', 'mar': 'Marzo', 'feb': 'Febbraio'}

keys=months.keys()  # creiamo un riferimento dinamico alle chiav
i
keysCopy=tuple(months.keys())  # creiamo una copia delle chiavi
esistenti ora

values=months.values() # come sopra, ma per i valori
valuesCopy=tuple(months.values()) # copia dei valori esistenti o
ra

del months['jan'] # eliminiamo l'elemento con chiave 'jan'
months['apr']='Aprile' # aggiungiamo un elemento con chiave 'apr
'

print(keys) # otteniamo la lista aggiornata
dict_keys(['apr', 'mar', 'feb'])
print(keysCopy) # otteniamo la copia (non aggiornata)
('jan', 'mar', 'feb')

print(values) # come sopra
dict_values(['Aprile', 'Marzo', 'Febbraio'])
print(valuesCopy) # come sopra
('Gennaio', 'Marzo', 'Febbraio')
```

Introduzione

Non ci soffermeremo in questi esempi sulla terminologia relativa alla programmazione orientata agli oggetti, dandola per nota. Ci concentreremo su alcuni esempi di base e su eventuali differenze tra Python e altri linguaggi di programmazione.

Un primo esempio di classe

In un primo esempio, utilizzeremo una classe solo per memorizzarci dei dati relativi ad un certo oggetto (come in una `struct` del C o in un `record` del Pascal).

Il concetto chiave, qui, è che i "campi" sono gli elementi di un dizionario automaticamente predisposto, il cui nome è `__dict__`, per cui possiamo utilizzare le normali operazioni sui dizionari per visualizzarli.

I "campi" vengono chiamati anche "proprietà" o "attributi" dell'oggetto (con alcune piccole sfumature nel significato).

Come al solito, presentiamo il codice inframmezzato dai risultati dell'esecuzione:

```
class Picture():
    pass
    # si usa la parola chiave "pass" quando
    # non c'è un corpo della funzione o della classe

img1=Picture()
img1.filename="gatto.jpg"
img1.filesize=12340
img1.width=320
img1.height=240
img1.filetype='JPEG'

print("Visualizzazione dell'intero dizionario:")
print(img1.__dict__)

Visualizzazione dell'intero dizionario:
```

```
{'width': 320, 'filetype': 'JPEG', 'height': 240, 'filesize': 12
340, 'filename': 'gatto.jpg'}

img1.width=480
img1.__dict__['width']=480  # equivalente alla precedente
print("Visualizzazione dell'intero dizionario (un valore è cambi
ato):")
print(img1.__dict__)

Visualizzazione dell'intero dizionario (un valore è cambiato):
{'width': 480, 'filetype': 'JPEG', 'height': 240, 'filesize': 12
340, 'filename': 'gatto.jpg'}

print("Visualizzazione delle chiavi:")
for key in img1.__dict__.keys():
    print(key)

Visualizzazione delle chiavi:
width
filetype
height
filesize
filename

print("Visualizzazione dei valori:")
for value in img1.__dict__.values():
    print(value)

Visualizzazione dei valori:
480
JPEG
240
12340
gatto.jpg

print("Visualizzazione delle coppie chiave/valore:")
for key, value in img1.__dict__.items():
    print(key, ' -->', value)

Visualizzazione delle coppie chiave/valore:
```

```
width  --> 480
filetype  --> JPEG
height  --> 240
filesize  --> 12340
filename  --> gatto.jpg
```

Il costruttore

Quando viene istanziato un nuovo oggetto della nostra classe, viene automaticamente invocata una funzione membro (detta anche "metodo") speciale, detto costruttore. In Python, tale funzione ha come nome `__init__()`. (Vedremo più avanti che in realtà la creazione di un'istanza in Python è effettuata in due momenti). Usando il costruttore, possiamo, ad esempio, impostare dei valori di default per alcuni dei valori dei "campi" (o, meglio, di quello che abbiamo visto essere il dizionario degli attributi):

```
class Picture():
    def __init__(self, filename, width=320, height=240, filetype
='JPEG'):
        self.filename= filename
        self.width=width
        self.height=height
        self.filetype=filetype

img1=Picture('gatto.jpg', height=480)

print("Visualizzazione dell'intero dizionario:")
print(img1.__dict__)
```

Output:

```
Visualizzazione dell'intero dizionario:
{'width': 320, 'filetype': 'JPEG', 'height': 480, 'filename': 'g
atto.jpg'}
```

Come per tutti i metodi della classe, per motivi che vedremo in seguito, in Python il primo parametro deve essere un riferimento all'oggetto stesso (che tradizionalmente viene chiamato `self`).

Setters e *getters*

Con il codice presentato qui sopra, non c'è nulla che possa evitare di utilizzare nomi non validi per assegnarli a particolari attributi. Ad esempio, sarebbe perfettamente lecito, per quanto sconveniente, scrivere istruzioni del tipo

```
img1.width = "abc"
img1.height = -12
```

Per ovviare a questo problema, può essere una buona idea racchiudere il codice da usare per assegnare un valore ad uno degli attributi in una funzione speciale, in modo da poter fare tutti i controlli che desideriamo.

L'attributo diventa quindi "privato" (cioè, non accessibile direttamente dall'esterno). In Python gli attributi "privati" sono indicati con nomi che iniziano con un doppio segno di sottolineatura. (È importante notare che in Python tali attributi non sono realmente privati, così come invece in C++, visto che in casi particolari — ad esempio per esigenze di debug — è comunque possibile accedere ai valori.)

Esempio di *setter*

Nell'esempio che segue, sono stati resi "privati" gli attributi width e height, e sono state definiti due metodi per poter cambiare loro il valore.

```
class Picture():
    def __init__(self, filename, width=320, height=240, filetype
='JPEG'):
        self.filename = filename
        self.setWidth(width)
        self.setHeight(height)
        self.filetype=filetype

    def setWidth(self, value):
        assert(isinstance(value, int))
        assert(value>=0)
        self.__width = value

    def setHeight(self, value):
        assert(isinstance(value, int))
        assert(value>=0)
        self.__height = value

img1=Picture('gatto.jpg', height=480)

img1.setWidth(200)
img1.__width = -10 # crea un nuovo attributo, non fa accedere a
quello privato

print("Visualizzazione dell'intero dizionario:")
print(img1.__dict__)
```

Output:

```
Visualizzazione dell'intero dizionario:
{'_Picture__height': 480, '__width': -10, 'filetype': 'JPEG', '_
Picture__width': 200, 'filename': 'gatto.jpg'}
```

Esempio di *getter*

Alla nostra classe, per i campi width e height, possiamo aggiungere anche dei metodi getter, che ci consentono di accedere agli attributi privati. Inoltre, già che ci siamo, aggiungiamo un metodo che fornisce una rappresentazione completa

dell'oggetto:

```
class Picture():
    ....

    def getWidth(self):
        return self.__width

    def getHeight(self):
        return self.__height

    def getCompleteDescription(self):
        text = "Immagine %s, larghezza %d, altezza %d" % (
            self.filename,
            self.getWidth(),
            self.getHeight()
            )
        return text

...

print(img1.getCompleteDescription())
```

Output:

```
Immagine gatto.jpg, larghezza 200, altezza 480
```

Più avanti, vedremo che — grazie al decoratore `@property` — saremo in grado di definire un accesso agli attributi anche senza il richiamo esplicito della funzione, ma in maniera più naturale.

Cosa restituisce un setter?

Una funzione che imposta il valore di un attributo, svolto il suo compito, normalmente non deve restituire un valore o un riferimento. Se però si fa in modo che restituisca un riferimento all'oggetto stesso (self), sarà possibile usare una sintassi più concisa per richiamare più metodi sullo stesso oggetto:

```
def setWidth(self, value):
    assert(isinstance(value, int))
    assert(value>=0)
    self.__width = value
    return self

def setHeight(self, value):
    assert(isinstance(value, int))
    assert(value>=0)
    self.__height = value
    return self
```

Questo ci consente di implementare, se lo desideriamo, una interfaccia fluente, grazie alla quale, se vogliamo impostare sia la larghezza sia l'altezza dell'immagine, potremo scrivere:

```
img1.setWidth(2000).setHeight(3000)
```

Il riferimeno a `self` nei parametri dei metodi

Un chiarimento rispetto all'uso della parola chiave `self` nella lista dei parametri dei metodi può essere utile. Immaginiamo di avere una classe definita in questo modo:

```
class Picture():
    def doSomething(self, n):
        print("Devo fare qualcosa con il valore %d." % n)
```

Nel momento in cui vogliamo usare il metodo `doSomething()` abbiamo due possibilità, perfettamente equivalenti.

La prima è di usare il nome dell'istanza:

```
img1=Picture()
img1.doSomething(42)   # uso il nome dell'istanza
```

La seconda è di usare il nome della classe:

```
Picture.doSomething(img1, 42)  # uso il nome della classe,
                               # e passo l'istanza come primo pa
rametro
```

Il primo modo è più pratico, ma viene comunque tradotto internamente in una chiamata al metodo della classe `Picture`, passando il riferimento all'istanza indicata come primo parametro. Nel codice del metodo doSomething() non c'è modo di sapere come si chiama l'istanza (ce ne potrebbero essere molte), quindi si usa un riferimento generico, che è appunto `self`.

Tra l'altro, il nome `self` è semplicemente una convenzione. Sarebbe perfettamente lecito, seppur sconsigliabile per ovvii motivi di leggibilità, scrivere un codice come il seguente:

```
class Picture():
    def setDescription(foo, text):
        foo.description=text

    def getDescription(bar):
        return bar.description

img1=Picture()
img1.setDescription('nice picture of the sea')
print(img1.getDescription())
```

Le classi: slot e controllo dell'input

Introduzione

A volte, per questioni di performance e di risparmio di risorse, si può desiderare di non utilizzare un dizionario per immagazzinare gli attributi di un oggetto, bensì un insieme di campi predefiniti, detti slot.

Se nella definizione della classe scriviamo un'istruzione come la seguente, definiamo l'elenco degli *slot*. Non sarà possibile, durante l'esecuzione, aggiungere altri attributi (nell'esempio che segue, l'elenco è una tupla, non una lista o un dizionario, e pertanto è immutabile; ma la regola varrebbe comunque).

```
__slots__ = ('filename', 'width', 'height', 'filetype', 'quality
')
```

Già che ci siamo, vedremo nell'esempio che segue che è possibile definire anche i tipi associati a ciascuno slot, in modo da poter gestire un input controllato. Inoltre, visto che la tupla di slot è iterabile, possiamo definire una funzione che prende in considerazione tutti i campi per fare l'input dei dati di un oggetto.

La classe Picture

Nel file `picture.py` definiamo la nostra classe per gestire un'immagine.

```
from basic_io import *

class Picture():
    __slots__ = ('filename', 'width', 'height', 'filetype', 'qua
lity')
    # __slots__ è una parola chiave che definisce l'elenco degli
 attributi
    # di una classe in modo vincolante

    __attributeTypes = {
        'filename': str,
```

```
        'width': int,
        'height': int,
        'filetype': str,
        'quality': float
        }
    # __attribuiteTypes è un dizionario privato in cui memorizzo
 i tipi
    # associati a ciascun attributo

    __attributeLabels = {
        'filename': 'nome del file',
        'width': 'larghezza',
        'height': 'altezza',
        'filetype': 'tipo di file',
        'quality': 'qualità'
        }
    # __attributeLabels è un dizionario privato in cui memorizzo
 i tipi
    # associati a ciascun attributo

    # costruttore
    def __init__(self, filename='', width=320, height=240, filet
ype='JPEG', quality=1.0):
        self.filename = filename
        self.width = width
        self.height = height
        self.filetype = filetype
        self.quality = quality

    def getCompleteDescription(self):
        """Return a complete description of the picture as a str
ing
        filled with all attributes' names and values.
        """
        fields=[]
        for attrName in self.__slots__:
            DisplayName = str(self.__attributeLabels[attrName])
            AttributeValue= str(getattr(self, attrName))
            # il metodo getattr serve per accedere al valore di
un
```

```
            # attributo avendo il suo nome
            # getattr(self, 'width') corrisponde a self.width
            fields.append('%s: %s' % (DisplayName, AttributeValu
e))
        return ', '.join(fields)
        # il metodo join applicato ad una stringa fa sì che la s
tringa
        # venga usata come "collante" per unire tutti gli elemen
ti della
        # lista che ha come argomento

    def __setattr__(self, name, value):
        assert(isinstance(value, self.__attributeTypes[name]))
        super().__setattr__(name, value)

    def inputFields(self):
        print("*** INSERIMENTO DATI PER UNA IMMAGINE ***")
        for field in self.__slots__:
            self.inputField(field)

    def inputField(self, field):
        datatype = self.__attributeTypes[field]
        message = 'Inserisci un valore per "%s": ' % self.__attr
ibuteLabels[field]
        data=checked_input(message, datatype)
        self.__setattr__(field, data)

    def showFields(self):
        print("*** VISUALIZZAZIONE DATI PER UNA IMMAGINE ***")
        for attrName in self.__slots__:
            self.showField(attrName)

    def showField(self, attrName):
        '''Prints out a list of field names and values'''
        print('%s --> %s' % (
            self.__attributeLabels[attrName],
            str(getattr(self, attrName))
            ))
```

```
if __name__=="__main__":
    img1=Picture()
    img1.inputFields()
    print(img1.getCompleteDescription())
    img1.showFields()
```

La funzione `checked_input`

Nel file si importa il modulo `basic_io`, che contiene il codice seguente (piccola variante rispetto a quello presentato precedentemente, che consentiva l'input controllato di un numero, e che qui invece è generica ed effettua la verifica per qualsiasi classe, tentando la conversione):

```
def checked_input(message, constraint_class):
    """Permette l'inserimento di un valore in maniera controllat
a.

    Chiede in input una stringa usando *message* come prompt, po
i
    tenta la conversione nella classe *constraint_class* e
    restituisce il valore convertito.
    Se viene inserito un valore non valido, viene richiesto di
    nuovo l'inserimento.
    """
    while(True):
        v=input(message)
        try:
            n = constraint_class(v)
            return n
        except ValueError as err:
            print("Sembra che tu non abbia inserito un valore va
lido…")
```

La classe PictureBox

A questo punto, se vogliamo gestire un elenco di immagini, possiamo pensare di metterle in una lista, o meglio ancora di creare una nostra versione personalizzata di lista (classe derivata). Vediamo un esempio, nel quale, già che c'eravamo,

abbiamo ridefinito i metodi append e insert per controllare che i valori aggiunti alla lista siano di tipo Picture, e il metodo `__setitem__` per far sì che un elemento possa essere cambiato o con un altro elemento di tipo Picture oppure con None:

```
from Picture import *

class PictureBox(list):
    def showPictures(self):
        for i in range(len(self)):
            # NB len(self) ha senso in questo caso, visto che se
lf è una lista
            if isinstance(self[i], Picture):
                # controlliamo, potrebbe essere None
                print(self[i].getCompleteDescription())

    def append(self, item):
        # ridefinizione del metodo append (se si vuole essere si
curi che non
        # si possano aggiungere elementi non di tipo Picture)
        assert(isinstance(item, Picture))
        super().append(item) # uso il metodo della classe padre

    def insert(self, position, item):
        # ridefinizione del metodo insert (se si vuole essere si
curi che non
        # si possano aggiungere elementi non di tipo Picture)
        assert(isinstance(item, Picture))
        super().insert(position, item) # uso il metodo della cla
sse padre

    def __setitem__(self, key, value):
        # ridefinizione del metodo __setitem__ (se si vuole esse
re sicuri che non
        # si possano cambiare elementi con oggetti non di tipo P
icture, a meno
        # che non li si imposti a None, che invece consideriamo
cosa lecita)
        assert(value is None or isinstance(value, Picture))
        super().__setitem__(key, value) # uso il metodo della cl
asse padre
```

```
if __name__=='__main__':
    mybox=PictureBox()

    n = checked_input("Quante immagini vuoi inserire? ", int)

    for i in range(n):
        pic=Picture()
        pic.inputFields()
        mybox.append(pic)

    mybox.showPictures()
```

Un esempio di interazione con l'utente potrebbe essere questo:

```
Quante immagini vuoi inserire? 3
*** INSERIMENTO DATI PER UNA IMMAGINE ***
Inserisci un valore per "nome del file": gatto.jpg
Inserisci un valore per "larghezza": 320
Inserisci un valore per "altezza": 240
Inserisci un valore per "tipo di file": JPEG
Inserisci un valore per "qualità": 0.8
*** INSERIMENTO DATI PER UNA IMMAGINE ***
Inserisci un valore per "nome del file": cane.jpg
Inserisci un valore per "larghezza": 800
Inserisci un valore per "altezza": 600
Inserisci un valore per "tipo di file": JPEG
Inserisci un valore per "qualità": 0.7
*** INSERIMENTO DATI PER UNA IMMAGINE ***
Inserisci un valore per "nome del file": pulce.png
Inserisci un valore per "larghezza": 16
Inserisci un valore per "altezza": 16
Inserisci un valore per "tipo di file": PNG
Inserisci un valore per "qualità": 1
Immagine "gatto.jpg", larghezza 320, altezza 240
Immagine "cane.jpg", larghezza 800, altezza 600
Immagine "pulce.png", larghezza 16, altezza 16
```

Tipi personalizzati, per così dire

Introduzione

Spesso capita di dover gestire "tipi di dati" personalizzati. Questa esigenza può essere gestita facilmente mediante la definizione di proprie classi, da zero oppure basandosi su classi predefinite a disposizione. Qui di seguito presentiamo due semplici esempi, quello di un valore intero positivo e quello di una data.

Un classe per valori interi positivi

Immaginiamo, per le nostre immagini, di voler porre il vincolo che la larghezza e l'altezza siano valori positivi. Anziché reinventare la ruota, possiamo basarci sulla classe int e derivarne una classe personalizzata, che chiameremo PosInt, in questo modo:

```
class PosInt(int):
    """Una classe per la rappresentazione di valori interi posit
ivi"""
    def __init__(self, v):
        try:
            self = int(v)
        except ValueError as err:
            return ValueError
        if self<=0:
            raise ValueError
```

In pratica, ridefiniamo il metodo `__init__()` in modo che prenda il valore del parametro e lo converta in un intero. Se non ci riesce, restituirà un errore di tipo ValueError. Ma lo stesso errore verrà restituito anche nel caso di valore minore o uguale a zero.

In una successiva lezione approfondiremo la questione della gestione degli errori e vedremo cosa significa l'istruzione `raise`.

Si noti che, essendo la classe Posint derivata dalla classe int, tutti i metodi della classe int sono disponibili anche per gli oggetti della classe PosInt, come si vede con l'output della funzione dir(), che elenca i metodi disponibili:

```
n=PosInt(12)
print(dir(n))

['__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__cla
ss__', '__delattr__', '__dict__',
[...snip...]
 'denominator', 'imag', 'numerator', 'real']
```

Sulle classi derivate torneremo più avanti.

Una classe per la gestione delle date

Immaginiamo di volere una classe per la gestione di una data, il cui costruttore abbia come parametro una stringa nella forma 'gg/mm/aaaa' (che rappresenta una data valida).

Ad esempio, vogliamo essere in grado di scrivere

```
d1 = Date('21/12/2010')
```

ma non deve invece funzionare

```
d2 = Date('34/12/2010')
```

Inoltre, sulla data memorizzata avremo bisogno di compiere delle operazioni (come minimo, estrarre l'anno, il mese e il giorno).

I possibili approcci sono:

1. definire una classe propria, partendo da zero
2. cercare nella documentazione se esiste una classe apposita tra quelle predefinite, che faccia al caso nostro
3. definire una classe propria, derivandola da qualcosa di esistente

Il primo approccio sarebbe interessante, ma solo a scopo didattico (ottimo esercizio). Qui lo tralasciamo.

Con il secondo approccio verifichiamo velocemente che esiste già una classe, nel modulo `datetime`, che potrebbe essere utile. Il problema è che il costruttore non riceve una stringa, bensì una tupla con i valori dell'anno, del mese, ecc., e quindi non fa al caso nostro.

Dovremo quindi fare ricorso al terzo approccio, ovvero scrivere una nostra classe. Presento di seguito due soluzioni possibili.

Una classe che usa al suo interno un oggetto datetime (classe *wrapper*)

Con il codice seguente definiamo una classe che al suo interno ha l'attributo privato `__date` di tipo `datetime`. Le varie funzioni non sono altro che rinvii alle funzioni predefinite dell'oggetto datetime (metodi proxy). Per comprendere il funzionamento della classe, è necessario leggere la documentazione riguardante i metodi `strptime()` e il metodo `strftime()`.

```
from datetime import *

class Date():
    def __init__(self, value):
        self.__date=datetime.strptime(value, "%d/%m/%Y")
    def getYear(self):
        return self.__date.year
    def getMonth(self):
        return self.__date.month
    def getDay(self):
        return self.__date.day
    def __str__(self):
        return self.__date.strftime("%d/%m/%Y")

print("Esempio con classe wrapper (1)")
n=Date('20/10/2010')
print(n, type(n))
print(n.getYear())

Esempio con classe wrapper (1)
20/10/2010 <class '__main__.Date'>
2010
```

Si noti che il metodo `__str__()`, se definito, serve a fornire una raprresentazione testuale standard di un oggetto, senza doverla richiamare esplicitamente. Nell'esempio qui sopra, se `n` è una data, si può scrivere tranquillamente `print(n)`.

Si può migliorare il codice di questa classe ridefinendo la funzione `__getattr__()`, in modo che venga sempre richiamato il corrispondente attributo della data presente in `__date`:

```
class Date():
    def __init__(self, value):
        self.__date=datetime.strptime(value, "%d/%m/%Y")
    def __getattr__(self, name):
        return getattr(self.__date, name)
    def __str__(self):
        return self.__date.strftime("%d/%m/%Y")

print("Esempio con classe wrapper (2)")
n=Date('20/10/2010')
print(n, type(n))
print(n.year)

Esempio con classe wrapper (2)
20/10/2010 <class '__main__.Date'>
2010
```

Il codice diventa più snello e abbiamo acquistato la possibilità di accedere a tutte le proprietà della data.

Una classe che deriva dalla classe `datetime`

Un'altra possibilità è di definire una classe derivata dalla classe `datetime`, semplicemente ragionando sul costruttore (in pratica, se ci si accorge che il costruttore riceve un parametro unico di tipo stringa, si invoca la funzione della classe che trasforma la stringa in data). La comprensione di questo esempio richiede qualche conoscenza fino ad ora non acquisita (ad esempio l'unpacking di sequenze, che vedremo successivamente, o la ridefinizione del metodo `__new__()`). Però si può verificare facilmente che la classe si comporta come dovrebbe.

```
class Date(datetime):
    def __new__(cls, *args):
        if len(args) == 1 and isinstance(args[0], str):
            return cls.strptime(args[0], "%d/%m/%Y")
        return datetime.__new__(cls, *args)
    def __str__(self):
        return self.strftime("%d/%m/%Y")

print("Esempio con classe derivata")
n=Date('12/11/2010')
print(n, type(n))
print(n.year)

n=Date(2010, 12, 31)
print(n, type(n))
print(n.year)
```

Output:

```
Esempio con classe derivata
12/11/2010 <class '__main__.Date'>
2010
31/12/2010 <class '__main__.Date'>
2010
```

Il vantaggio è che così possiamo comunque usare anche il costruttore tipico (basato su una tupla di numeri).

Esercizi

1. Creare due classi, `OddInt` e `EvenInt`, che consentano solo valori rispettivamente dispari e pari.
2. Creare una classe `CenturyDate` che consenta solo date di questo secolo.

Elenco di campi in `named tuple`

Introduzione

Con le conoscenze acquisite, possiamo procedere a un refactoring del codice scritto precedentemente, in modo da migliorare alcuni aspetti della gestione della "tabella di dati".

Gli obiettivi che ci proponiamo sono:

- avere la possibilità di definire i diversi campi di cui un elemento della tabella è composto (come se si trattasse di un record in Pascal o di una struct in C);
- poter accedere al valore del campo o per nome o per posizione;
- far sì che i vari record vengano inseriti all'interno di una lista.

Per realizzare questi obiettivi, ci converrà partire definendo una struttura dati per la memorizzazione di quello che ci serve per un "campo". Per iniziare, memorizzeremo la classe associata (il "tipo"), l'etichetta, la dimensione (per questioni di formattazione), il tipo di allineamento. A tale scopo definiremo una apposita classe, `Field`.

Successivamente, dovremo memorizzare i campi del nostro elemento (ossia quelli che abbiamo visto si chiamano *slots*) in una struttura dati un po' più complessa di una tupla, in modo da avere la possibilità di accedere ad essi sia attraverso un indice numerico, sia attraverso una chiave. Per questo motivo, useremo una struttura dati apposita, chiamata namedtuple, che fa parte del modulo collections.

La classe Field

La classe Field è molto semplice. Ci consente di memorizzare le informazioni necessarie per gestire un campo del nostro elemento:

```
class Field():
    def __init__(self, class_constraint, label='',  size=20, ali
gnment='left'):
        self.__class_constraint = class_constraint
        self.__label = label
        self.__size = size
        self.__alignment = alignment

    def getClassConstraint(self):
        return self.__class_constraint
    def getLabel(self):
        return self.__label
    def getSize(self):
        return self.__size
    def getAlignment(self):
        return self.__alignment
    def getMethod(self):
        return self.__method

    def setLabel(self, v):
        self.__label = v
        return self
```

La classe Picture

La ridefinizione della classe Picture si basa sul fatto che i dati relativi agli slot vengono memorizzati in una namedtuple. Inoltre, è stato aggiunto un metodo di classe per ottenere l'elenco dei campi, in modo da poterli visualizzare nell'intestazione quando i dati di più elementi vengono presentati in forma tabellare (vedi codice di PictureBox).

```
from basic_io import *
from PosInt import *
from Field import *
from Date import *

import collections
```

```
class Picture(object):
    nt = collections.namedtuple('fields', 'filename, width, heig
ht, filetype, quality, date_taken')

    __fields = nt(
        Field(str, 'nome del file', 30),
        Field(PosInt, 'larghezza', 8, 'right'),
        Field(PosInt, 'altezza', 8, 'right'),
        Field(str, 'tipo di file', 5),
        Field(float, 'qualità immagine', 4, 'right'),
        Field(Date, 'data di ripresa', 8, 'right')
        )

    __slots__ = nt._fields

    def __setattr__(self, name, value):
        assert(isinstance(value, getattr(self.__fields, name).ge
tClassConstraint()))
        super().__setattr__(name, value)

    def getAttrByName(self, name):
        return self.__getattribute__(name)

    def getAttrByPosition(self, position):
        name = self.__slots__[position]
        return self.getAttrByName(name)

    def inputData(self):
        print("*** INSERIMENTO DATI PER UNA IMMAGINE ***")
        for fieldname, field in zip(self.__slots__, self.__field
s):
            self.inputField(fieldname, field)

    def inputField(self, fieldname, field):
        message = 'Inserisci un valore per "%s": ' % field.getLa
bel()
        data=checked_input(message, field.getClassConstraint())
        self.__setattr__(fieldname, data)

    def outputData(self):
```

```
        print('*** DATI DI UNA IMMAGINE ***')
        for fieldname, field in zip(self.__slots__, self.__field
s):
            self.outputField(fieldname, field)

    def outputField(self, fieldname, field):
        print(field.getLabel(),': ', self.getAttrByName(fieldnam
e))

    @classmethod
    # un metodo "di classe" riceve come primo parametro un rifer
imento
    # alla classe
    def getFields(cls):
        return cls.__fields
```

La classe PictureBox

Nella classe PictureBox abbiamo solo aggiunto il codice per la visualizzazione dell'intestazione, nel quale si richiama un metodo statico della classe Picture.

```
class PictureBox(list):
    def showHeader(self):
        fields=list()
        # creiamo una lista dove mettere le etichette dei campi
        for field in Picture.getFields():
            fields.append(field.getLabel())
            # aggiungiamo un'etichetta alla lista
        print(*fields, sep=' | ')
        # visualizziamo tutte le etichette della lista
        # *fields trasforma l'oggetto fields in un elenco, come
se
        # scrivessimo
        # print(fields[0], fields[1], fields[2], ...)

    def showPictures(self):
        for i in range(len(self)):
            self[i].outputData()
```

L'interazione con l'utente avviene come nel seguente esempio:

```
Quante immagini vuoi inserire? 2
*** INSERIMENTO DATI PER UNA IMMAGINE ***
Inserisci un valore per "nome del file": gatto.jpg
Inserisci un valore per "larghezza": 300
Inserisci un valore per "altezza": 200
Inserisci un valore per "tipo di file": JPEG
Inserisci un valore per "qualità immagine": f
Sembra che tu non abbia inserito un valore valido…
Inserisci un valore per "qualità immagine": 0.9
Inserisci un valore per "data di ripresa": 20/100/2009
Sembra che tu non abbia inserito un valore valido…
Inserisci un valore per "data di ripresa": 20/10/2010
*** INSERIMENTO DATI PER UNA IMMAGINE ***
Inserisci un valore per "nome del file": cane.png
Inserisci un valore per "larghezza": 40
Inserisci un valore per "altezza": 40
Inserisci un valore per "tipo di file": PNG
Inserisci un valore per "qualità immagine": 1
Inserisci un valore per "data di ripresa": 12/10/2010
*** DATI DI UNA IMMAGINE ***
nome del file:  gatto.jpg
larghezza:  300
altezza:  200
tipo di file:  JPEG
qualità immagine:  0.9
data di ripresa:  20/10/2010
*** DATI DI UNA IMMAGINE ***
nome del file:  cane.png
larghezza:  40
altezza:  40
tipo di file:  PNG
qualità immagine:  1.0
data di ripresa:  12/10/2010
```

Attributi privati e pubblici, di classe e di istanza

Attributi pubblici di istanza

Gli attributi di istanza sono quelli che sono legati al singolo oggetto. Agli attributi pubblici si può accedere direttamente anche dal programma chiamante, senza invocare una funzione della classe. Si veda il seguente esempio:

```
class Fruit(object):

    def __init__(self, name):
        self.setName(name)

    def setName(self, value):
        self.name = value   # name è un attributo pubblico
        return self

    def setQuality(self, value):
        self.quality = value   # quality è un attributo pubblico
        return self

    def getName(self):
        return self.name

    def getQuality(self):
        return self.quality

def main():
    myapple = Fruit('mela granny smith')
    myapple.setQuality(12)
    print('Nome: %s\nqualità: %d' % (myapple.getName(), myapple.
getQuality()))

    myapple.quality = 20
    print('Nome: %s\nqualità: %d' % (myapple.getName(), myapple.
getQuality()))
    # accesso agli attributi tramite funzione "getter"
```

```
    print(myapple.quality)
    # accesso all'attributo diretto

    myorange = Fruit('arancia siciliana')
    myorange.quality = -20
    # non viene fatto nessun controllo sul valore impostato...
    print('Nome: %s\nqualità: %d' % (myorange.getName(), myorang
e.getQuality()))

if __name__ == '__main__':
    main()
```

che produce il seguente output:

```
Nome: mela granny smith
qualità: 12
Nome: mela granny smith
qualità: 20
20
Nome: arancia siciliana
qualità: -20
```

Per inciso, quando si chiama una funzione definita all'interno della classe, si può usare la versione con indicazione esplicita della classe:

```
Fruit.setQuality(myapple, 12)
```

oppure la versione con l'indicazione dell'oggetto e la notazione puntata:

```
myapple.setQuality(12)
```

Attributi privati di istanza

Se si vuole che ad un determinato attributo (ma il discorso vale anche per le funzioni definite all'interno di una classe) sia accessibile solo dalle funzioni interne alla classe, la convenzione in Python è di usare un nome che inizia con due segni di sottolineatura (ma vedi nota in fondo al riguardo). In questo modo viene a crearsi un attributo privato. (Si noti che in Python gli attributi privati sono comunque accessibili, usando particolari meccanismi, e questo viene considerato utile nel caso, ad esempio, di sessioni di debugging.)

Modifichiamo la classe precedente come segue:

```
class Fruit(object):

    def __init__(self, name):
        self.setName(name)

    def setName(self, value):
        self.name = value   # name è un attributo pubblico
        return self

    def setQuality(self, value):
        self.__quality = value   # quality è un attributo privat
o
        return self

    def getName(self):
        return self.name

    def getQuality(self):
        return self.__quality

def main():
    myapple = Fruit('mela granny smith')
    myapple.setQuality(12)
    # equivalente a Fruit.setQuality(myapple, 12)
    print('Nome: %s\nqualità: %d' % (myapple.getName(), myapple.
getQuality()))

    # myapple.__quality = 20
    # non possibile, visto che l'attributo è privato

    # print(myapple.__quality)
    # accesso diretto all'attributo
    # non possibile, visto che l'attributo è privato

if __name__ == '__main__':
    main()
```

Controllo dell'attributo

In questo modo, possiamo far sì che qualsiasi modifica dell'attributo `__quality` passi attraverso il richiamo di una funzione della classe. Ad esempio, se volessimo forzare un controllo sul valore, potremmo ridefinire la funzione `setQuality()` così:

```
class Fruit(object):

    def __init__(self, name):
        self.setName(name)

    def setName(self, value):
        self.name = value
        return self

    def setQuality(self, value):
        if 0 <= value <= 100:
            self.__quality = value
        else:
            self.__quality = 50
        return self

    def getName(self):
        return self.name

    def getQuality(self):
        return self.__quality

def main():
    myapple = Fruit('mela granny smith')
    myapple.setQuality(12)
    print('Nome: %s\nqualità: %d' % (myapple.getName(), myapple.
getQuality()))

    myapple.setQuality(-10)
    # impostiamo un valore fuori dal range valido
    print('Nome: %s\nqualità: %d' % (myapple.getName(), myapple.
getQuality()))
    # ... e vediamo che viene impostato a 50

if __name__ == '__main__':
    main()
```

L'output sarà il seguente:

```
Nome: mela granny smith
qualità: 12
Nome: mela granny smith
qualità: 50
```

Funzioni private

Anche le funzioni, come già accennato, possono essere private. Immaginiamo di voler aggiungere una funzione che aggiunga un messaggio di allarme ogni qual volta il valore della quantità viene impostato in modo non corretto. Sarà sufficiente che il nome della funzione inizi con il solito doppio underscore:

```
class Fruit(object):

    def __init__(self, name):
        self.setName(name)
        self.__alerts = [] # un attributo privato (lista degli a
llarmi)

    def setName(self, value):
        self.name = value
        return self

    def setQuality(self, value):
        if 0 <= value <= 100:
            self.__quality = value
        else:
            self.__quality = 50
            self.__addAlert('bad quality: %d' % value)
        return self

    def __addAlert(self, value):
        self.__alerts.append(value)
        return self

    def getName(self):
        return self.name
```

```
    def getQuality(self):
        return self.__quality

    def getAlerts(self):
        return self.__alerts

def main():
    myapple = Fruit('mela granny smith')
    myapple.setQuality(12)
    print('Nome: %s\nqualità: %d' % (myapple.getName(), myapple.
getQuality()))

    myapple.setQuality(-10)
    # impostiamo un valore fuori dal range valido
    print('Nome: %s\nqualità: %d' % (myapple.getName(), myapple.
getQuality()))
    # ... e vediamo che viene impostato a 50
    print("Lista degli allarmi: ", myapple.getAlerts())

if __name__ == '__main__':
    main()
```

L'output sarà come il seguente:

```
Nome: mela granny smith
qualità: 12
Nome: mela granny smith
qualità: 50
Lista degli allarmi:  ['bad quality: -10']
```

Attributi legati alla classe

Può a volte essere necessario gestire degli attributi che hanno a che fare non con le singole istanze, ma con la classe in quanto tale. Si immagini ad esempio di voler memorizzare un elenco degli elementi istanziati nell'attributo di classe `salad`. Si può scrivere un codice come il seguente:

```
class Fruit(object):
```

```
    salad = [] # attributo di classe, vale per tutte le istanze
    # in questo esempio, è una lista di tutti i frutti istanziat
i

    def __init__(self, name):
        self.setName(name)
        self.__alerts = []
        self.salad.append(self)

    def __str__(self):
        return self.getName()

    def getName(self):
        return self.name

    def setName(self, value):
        self.name = value
        return self

    def getIdx(self):
        return self.__idx

    def setQuality(self, value):
        if 0 <= value <= 100:
            self.__quality = value
        else:
            self.__quality = 50
            self.__addAlert('bad quality: %d' % value)
        return self

    def __addAlert(self, value):
        self.__alerts.append(value)
        return self

    def getName(self):
        return self.name

    def getQuality(self):
        return self.__quality
```

```
    def getAlerts(self):
        return self.__alerts

def main():
    myapple = Fruit('mela granny smith')
    print('macedonia a cui appartiene la mela:')
    print(*myapple.salad, sep=", ")

    myorange= Fruit('arancia siciliana')
    print('macedonia a cui appartiene l\'arancia:')
    print(*myorange.salad, sep=", ")

    mykiwi= Fruit('kiwi friulano')
    print('macedonia a cui appartiene il kiwi:')
    print(*mykiwi.salad, sep=", ")

    print('==========')
    print('macedonia a cui appartiene la mela:')
    print(*myapple.salad, sep=", ")
    print('macedonia a cui appartiene l\'arancia:')
    print(*myorange.salad, sep=", ")
    print('macedonia a cui appartiene il kiwi:')
    print(*mykiwi.salad, sep=", ")


if __name__ == '__main__':
    main()
```

Come si può vedere, per qualsiasi frutto istanziato, la lista degli elementi risulta la medesima:

```
macedonia a cui appartiene la mela:
mela granny smith
macedonia a cui appartiene l'arancia:
mela granny smith, arancia siciliana
macedonia a cui appartiene il kiwi:
mela granny smith, arancia siciliana, kiwi friulano
==========
macedonia a cui appartiene la mela:
mela granny smith, arancia siciliana, kiwi friulano
macedonia a cui appartiene l'arancia:
mela granny smith, arancia siciliana, kiwi friulano
macedonia a cui appartiene il kiwi:
mela granny smith, arancia siciliana, kiwi friulano
```

Va notato che, affinché questo metodo funzioni, l'attributo di istanza deve essere di tipo mutabile (liste, dizionari, ecc.) e non di tipo immutabile (numeri, stringhe, tuple, ecc.) -- a meno che non si implementi un metodo di classe (vedi più avanti).

Una possibile implementazione è la seguente, che usa un dizionario come attributo privato di classe:

```
class Fruit(object):
    __salad = {'quantity': 0} # attributo di classe, vale per tu
tte le istanze
    # in questo esempio, è un dizionario, di cui ci interessa un
 solo valore
    # qui lo rendiamo privato

    def __init__(self, name):
        self.setName(name)
        self.__alerts = []
        self.__salad['quantity'] += 1

        ...

    def getSaladInfoByKey(self, key):
        return self.__salad[key]

def main():
    myapple = Fruit('mela granny smith')
    print('Numero di frutti fino ad ora: %d' % myapple.getSaladI
nfoByKey('quantity'))

    myorange= Fruit('arancia siciliana')
    print('Numero di frutti fino ad ora: %d' % myorange.getSalad
InfoByKey('quantity'))

    mykiwi= Fruit('kiwi friulano')
    print('Numero di frutti fino ad ora: %d' % mykiwi.getSaladIn
foByKey('quantity'))
```

L'output sarà il seguente:

```
Numero di frutti fino ad ora: 1
Numero di frutti fino ad ora: 2
Numero di frutti fino ad ora: 3
```

In alternativa, è possibile definire un metodo di classe (che, a differenza dei metodi di istanza, riceve come primo parametro un riferimento alla classe e non all'oggetto istanziato):

```
class Fruit(object):
    __salad = 0

    @classmethod
    def __incSaladCount(cls):
        cls.__salad += 1

    @classmethod
    def getSaladCount(cls):
        return cls.__salad

    def __init__(self, name):
        self.setName(name)
        self.__alerts = []
        self.__incSaladCount()

....

def main():
    myapple = Fruit('mela granny smith')
    print('Numero di frutti fino ad ora: %d' % myapple.getSaladC
ount())

    myorange= Fruit('arancia siciliana')
    print('Numero di frutti fino ad ora: %d' % myorange.getSalad
Count())

    mykiwi= Fruit('kiwi friulano')
    print('Numero di frutti fino ad ora: %d' % mykiwi.getSaladCo
unt())
    print('Numero di frutti fino ad ora: %d' % myapple.getSaladC
ount())
    print('Numero di frutti fino ad ora: %d' % Fruit.getSaladCou
nt())
```

Si noti che richiamare

```
mykiwi.getSaladCount()
myapple.getSaladCount()
Fruit.getSaladCount()
```

è esattamente equivalente.

Uno o due *underscore*? (piccola nota sul *name mangling*)

Piccolo approfondimento necessario. In Python non esiste un vero e proprio meccanismo di protezione degli attributi, come in Java o C++. Ci si basa semplicemente sulla convenzione. Un programmatore sa che, quando il nome di un attributo o di una funzione inizia con il simbolo di underscore, questo attributo o funzione deve essere considerato privato, e di conseguenza sa che se cerca di accedere a quell'attributo "da fuori" sta facendo una cosa "sporca" e poco raccomandabile (sebbene lecita).

Se il nome di un attributo inizia con un doppio underscore, a quell'attributo non è direttamente (facilmente) possibile accedere da fuori, a causa del *name mangling*. Si consideri il seguente esempio:

```
class Foo():
    def __init__(self):
        self.bar = 1
        self._bar = 2
        self.__bar = 3

f = Foo()
print("f.bar: %d" % f.bar)
print("f._bar: %d" % f._bar)
print("f.__bar: %d" % f.__bar)
```

Il risultato è questo:

```
f.bar: 1
f._bar: 2
Traceback (most recent call last):
  File "...private.py", line 10, in <module>
    print("f.__bar: %d" % f.__bar)
AttributeError: 'Foo' object has no attribute '__bar'
```

Come si vede, l'accesso all'attributo `__bar` dall'esterno della classe non è stato possibile, mentre quello all'attributo `_bar` sì. Per rendersi conto del motivo, si deve sapere che tutti gli attributi di un oggetto vengono memorizzati in un dizionario di nome `__dict__`, nel quale le chiavi sono costituite dai nomi degli attributi, ma -- nel caso di attributi il cui nome inizia con un doppio underscore -- viene anteposto ad esso il nome della classe. Noto questo, è comunque possibile accedere agli attributi "privati":

```
class Foo():
    def __init__(self):
        self.bar = 1
        self._bar = 2
        self.__bar = 3

f = Foo()
print("f.bar: %d" % f.bar)
print("f._bar: %d" % f._bar)
#print("f.__bar: %d" % f.__bar)

print(f.__dict__)
print("f.__bar (accesso indiretto): %d" % f.__dict__['_Foo__bar'
])
print("f.__bar (accesso con nome della classe): %d" % f._Foo__ba
r)
```

Output:

```
f.bar: 1
f._bar: 2
{'_bar': 2, 'bar': 1, '_Foo__bar': 3}
f.__bar (accesso indiretto): 3
f.__bar (accesso con nome della classe): 3
```

Il *name mangling* è stato introdotto soprattutto per evitare conflitti sui nomi quando si usa l'ereditarietà, non tanto per la protezione reale degli attributi. È importante tenere presente questo fatto, poiché si potrebbe avere qualche problema nell'uso di classi derivate (di cui parleremo approfonditamente più avanti). Si consideri questo esempio:

```
class Foo():
    def __init__(self):
        self.bar = 1
        self._bar = 2
        self.__bar = 3

class DerivedFromFoo(Foo):
    def update(self):
        self.bar = 10
        self._bar = 20
        self.__bar = 30

f = DerivedFromFoo()
print(f.__dict__)

f.update()
print(f.__dict__)
```

Come si può vedere osservando l'output qui sotto, la funzione update() della classe derivata non modifica il valore di __bar, ma crea nel dizionario un nuovo elemento:

```
{'_bar': 2, 'bar': 1, '_Foo__bar': 3}
{'_bar': 20, '_DerivedFromFoo__bar': 30, 'bar': 10, '_Foo__bar':
 3}
```

In conclusione, se gli attributi devono essere accessibili dalle classi derivate, ma non dall'esterno, si fa iniziare il loro nome con un singolo underscore (quasi equivalente agli attributi protected di Java o C++), mentre se si vuole che siano privati si fa iniziare il loro nome con un doppio underscore.

Nomi dei metodi speciali

Tutte le classi dispongono di alcuni metodi predefiniti che si possono ridefinire in caso di necessità. Questi metodi hanno nomi preceduti e seguiti da doppio underscore. Ad esempio, la funzione `__init__()` contiene le istruzioni di inizializzazione di un'istanza, la funzione `__str__()` restituisce informazioni sull'istanza in forma di stringa, ecc. Ulteriori informazioni si possono trovare nella documentazione ufficiale di Python.

Proprietà

Proprietà vs Attributi

Le proprietà sono modi per ottenere e impostare il valore di alcuni attributi (ma non solo, possono esserci proprietà calcolate) senza dover richiamare le funzioni *getter* e *setter*, come abbiamo visto negli esempi fino a qui. Per i programmatori Python, questa è una tecnica usata molto comunemente, considerata l'estrema semplicità di implementazione.

```
class Fruit(object):
    .....

    @property
    def quality(self):
        return self.__quality

    @quality.setter
    def quality(self, value):
        if 0 <= value <= 100:
            self.__quality = value
        else:
            self.__quality = 50
            self.__addAlert('bad quality: %d' % value)


def main():
    myapple = Fruit('mela granny smith')
    myapple.quality = 10
    print('quality %d' % myapple.quality)
    print('Allarmi: ', *myapple.getAlerts())
    myapple.quality = -20
    print('quality %d' % myapple.quality)
    print('Allarmi: ', *myapple.getAlerts())
```

L'output è come il seguente:

```
quality 10
Allarmi:
quality 50
Allarmi:  bad quality: -20
```

Come si vede, nonostante l'uso sia simile a quello degli attributi pubblici, in realtà vengono richiamate le funzioni specifiche impostate.

Uso dell'attributo di classe `__slots__`

Come abbiamo già visto, si può usare l'attributo di classe `__slots__` per indicare una lista di attributi di istanza validi. È sufficiente inserire nel codice la riga

```
__slots__ = ('name', '__alerts', '__quality')
```

per far sì che il codice seguente, altrimenti valido, provochi un errore:

```
myapple = Fruit('mela granny smith')
myapple.quality = 10
myapple.color='verde'
print(myapple.color)

>>>
Traceback (most recent call last):
  File ".../Fruit.py", line 58, in <module>
    main()
  File "/.../Fruit.py", line 54, in main
    myapple.color='verde'
AttributeError: 'Fruit' object has no attribute 'color'
```

Proprietà calcolate

Alcune proprietà possono essere calcolate a partire dal valore degli attributi. Per esse si può impostare solo il getter, e non il setter. Ad esempio, si consideri il seguente codice:

```
@property
def is_good(self):
    return self.quality > 70
```

Sarà possibile eseguire queste istruzioni:

```
myapple = Fruit('mela granny smith')
myapple.quality = 10
print(myapple.quality, myapple.is_good)
myapple.quality = 75
print(myapple.quality, myapple.is_good)
```

ma non questa:

```
myapple.is_good = False
# non funziona (manca il setter)
```

Eliminazione di un attributo

Gli attributi possono essere eliminati. Per farlo, è sufficiente usare la parola chiave del:

```
>>> f=Foo()  # immaginiamo di avere una classe Foo.
>>> f.bar=1
>>> print(f.bar)
1
>>> del f.bar
>>> print(f.bar)
Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    print(f.bar)
AttributeError: 'Foo' object has no attribute 'bar'
```

Se usiamo le proprietà, possiamo definire una funzione che verrà invocata in occasione della cancellazione:

```
@quality.deleter
def quality(self):
    del self.__quality
```

Visto che consentiamo la cancellazione, sarà necessario gestire anche il *getter* in maniera più articolata, ad esempio con un try... except:

```
@property
def quality(self):
    '''Quality of the picture. Can be any value between 0 and 10
0 (included).'''
    try:
        return self.__quality
    except AttributeError as err:
        return None
```

Già che ci siamo, possiamo aggiungere una stringa di documentazione sulla proprietà, da poter richiamare in caso di necessità:

```
print(Fruit.quality.__doc__)
Quality of the picture. Can be any value between 0 and 100 (incl
uded).
```

Proprietà impostate senza uso di decoratori

Una proprietà può essere impostata senza uso di decoratori, tramite la funzione predefinita `property`, che permette di indicare, nell'ordine, *getter*, *setter*, *deleter* e *docstring*:

```
def getQuality(self):
    try:
        return self.__quality
    except AttributeError as err:
        return None


def setQuality(self, value):
    if 0 <= value <= 100:
        self.__quality = value
    else:
        self.__quality = 50
        self.__addAlert('bad quality: %d' % value)

def delQuality(self):
    print('removing quality')
    del self.__quality

quality=property(
    getQuality,
    setQuality,
    delQuality,
    'Quality of the picture. Can be any value between 0 and 100
(included).'
    )
```

Spacchettamento di sequenze

Introduzione

Lo spacchettamento di sequenze è un'operazione molto comune in Python e vale la pena di soffermarcisi un attimo per presentare qualche esempio.

Richiamo di una funzione

Immaginiamo di avere una funzione come la seguente:

```
def describe_person(name, age):
    assert(isinstance(name, str))
    assert(isinstance(age, int))
    print('La persona di nome %s ha %d anni' % (name, age))
```

Essa potrà essere richiamata in uno dei seguenti modi:

```
describe_person('mario', 32)
# chiamata con parametri nel giusto ordine

describe_person(age=32, name='mario')
# chiamata con parametri in forma chiave/valore
# si noti che l'ordine non è importante

info = ('mario', 32)
describe_person(*info)
# chiamata con i parametri in una tupla e "sequence unpacking"

info = ['mario', 32]
describe_person(*info)
# chiamata con i parametri in una lista e "sequence unpacking"

info = {'age': 32, 'name': 'mario'}
describe_person(**info)
# chiamata con i parametri in un dizionario e "dict unpacking"
# notare che in questo caso si mettono due asterischi
```

In ogni caso, i parametri ricevuti, e di conseguenza l'output, sarà il medesimo:

```
La persona di nome mario ha 32 anni
```

Numero arbitrario di parametri per una funzione

Immaginiamo di avere bisogno di avere un numero arbitrario di parametri per una funzione. Possiamo definirla in questo modo:

```
def do_something(*args):
    print('Elenco degli argomenti:')
    for v in args:
        print(v)
    print('Elenco degli argomenti (con indice numerico):')
    for i in range(len(args)):
        print(i, args[i])

if __name__=='__main__':

    do_something('uno', 'due', 'tre')
```

L'output sarà:

```
Elenco degli argomenti:
uno
due
tre
Elenco degli argomenti (con indice numerico):
0 uno
1 due
2 tre
```

Parametri in forma chiave/valore

Se vogliamo che la funzione ammetta parametri nella forma chiave/valore possiamo procedere in questo modo:

```
def do_something(*args, **kwargs):
    print('Elenco degli argomenti senza parola chiave:')
    print(args)
    for v in args:
        print(v)
    print('Elenco degli argomenti con parola chiave:')
    print(kwargs)
    for k,v in kwargs.items():
        print(k, '=', v)

if __name__=='__main__':

    do_something('uno', 'due', 'tre', valore=100, totale=200)
```

Output:

```
Elenco degli argomenti senza parola chiave:
('uno', 'due', 'tre')
uno
due
tre
Elenco degli argomenti con parola chiave:
{'totale': 200, 'valore': 100}
totale = 200
valore = 100
```

Uso di un dizionario per stringhe di formattazione

Può essere comodo usare un dizionario per impostare i valori dei segnaposto in una stringa di formattazione. Come visto in uno dei primi esempi in una precedente unità, potrei scrivere l'istruzione

```
print("L'utente si chiama {name} e ha {age} anni.".format(name='
Giorgio', age=22)
```

per ottenere

```
L'utente si chiama Giorgio e ha 22 anni.
```

Il medesimo risultato può essere ottenuto con i valori presenti in un dizionario, spacchettato quando esso diventa argomento del metodo format della classe str:

```
dic={'name':'Giorgio', 'age':22}
print("L'utente si chiama {name} e ha {age} anni.".format(**dic)
)
```

Si noti che in questi esempi la stringa "L'utente si chiama {name} e ha {age} anni." è l'oggetto di cui chiamiamo il metodo format(). Avremmo potuto ovviamente assegnare la stringa ad una variabile e poi richiamare il metodo usando la variabile.

Spacchettamento parziale

Può essere necessario prelevare da una sequenza una parte degli elementi, come negli esempi che seguono.

```
a = ['arancia', 'mela', 'pera', 'fragola', 'banana', 'ananas']

primo, secondo, *altri = a
print('primo: ', primo)
print('secondo: ', secondo)
print('altri: ', altri)

primo: arancia
secondo: mela
altri: ['pera', 'fragola', 'banana', 'ananas']

primo, *centrali, ultimo = a
print('primo: ', primo)
print('centrali: ', centrali)
print('ultimo: ', ultimo)

primo: arancia
centrali: ['mela', 'pera', 'fragola', 'banana']
ultimo: ananas
```

Le stringhe sono sequenze, per cui il metodo funziona anche con esse:

```
nome="Mario"
primalettera,*successive = nome
print('prima lettera: ', primalettera)
print('successive: ', successive)

prima lettera: M
successive: ['a', 'r', 'i', 'o']
```

Spacchettamento di oggetti che implementano iteratori

Se una classe implementa un iteratore (in sintesi, ci permette di fare un ciclo for con i valori via via generati), è possibile spacchettare i valori delle sue istanze. L'esempio classico è quello della classe range:

```
>>> print(range(2,8))
range(2, 8)
>>> print(*range(2,8))
2 3 4 5 6 7
```

Ovviamente vale anche

```
>>> r=range(2,8)
>>> print(r)
range(2, 8)
>>> print(*r)
2 3 4 5 6 7
```

Come faccio a sapere se un dato oggetto è una sequenza?

Gli oggetti sono sequenze se implementano l'*iteration protocol* o il *sequence protocol* (in sostanza, devono definire alcuni metodi speciali, quali `__iter__()`, `__next__()`, ecc.). Per sapere se un oggetto è una sequenza, l'approccio più semplice è di vedere se con esso si può fare un'iterazione. Volendo, si può anche scrivere una funzione come la seguente:

```
def isIterable(object):
   try:
       k=iter(object)
       return True
   except TypeError:
       return False
```

e poi provarla con diversi tipi di oggetti:

```
>>> isIterable(2)
False
>>> isIterable('abc')
True
>>> isIterable('')
True
>>> isIterable((1,2,3))
True
>>> isIterable(open('/etc/services'))
True
```

File di testo

Introduzione

Vedremo qui di seguito diversi modi per accedere a file di testo in lettura e scrittura. Negli esempi, le stringhe provengono dal testo latino Lorem Ipsum.

Lettura, primo approccio

Si può aprire un file e fare un ciclo che termina quando la riga letta è vuota (ossia, quando non c'è nemmeno il carattere di 'a capo')

```
file_in = open('loremipsum.txt')
while True:
    line = file_in.readline()
    if line == '':
        break
    print(line, end='')
file_in.close()
```

Scrittura

In questo esempio, scriviamo in un file le stringhe presenti in una tupla.

```
lines = (
    'Aenean non nulla ut neque pharetra faucibus feugiat sit ame
t erat.',
    'Nullam sed elit ante.',
    'Praesent aliquet dictum mauris, non pretium leo luctus eu.'
    )

file_out = open('aenannon.txt', 'w')
for line in lines:
    file_out.write(line+'\n')

file_out.close()
```

Aggiunta

Se vogliamo aggiungere righe ad un file, la modalità di apertura è 'a' (per append).

```
newlines = (
    'Vestibulum viverra molestie nulla, sed aliquet urna malesua
da in.',
    'Etiam porttitor quam ut massa rutrum vitae volutpat erat in
terdum.',
    'Etiam lorem purus, sollicitudin ac pretium in, pellentesque
 ac magna.',
    'Nam laoreet, turpis nec feugiat mollis, metus augue sollici
tudin augue, in dictum nulla dolor ac nulla.'
    )

file_out = open('aenannon.txt', 'a')
for line in newlines:
    file_out.write(line+'\n')
file_out.close()
```

Lettura, secondo approccio

Per leggere tutte le righe di un file, si può iterare sull'oggetto file:

```
file_written = open('aenannon.txt')  # leggiamo il file appena s
critto
for line in file_written:
    print(line, end='')
file_written.close()
```

Lettura, terzo approccio

In questo esempio, usiamo un contesto con la parola riservata with e il metodo readlines che legge tutte le righe del file mettendole in una lista. Si noti che, usando il contesto, il file viene automaticamente chiuso.

```
with open('aenannon.txt') as f:
    lines = f.readlines()
for line in lines:
    print(line, end='')
```

Lettura, ordinamento, scrittura

Immaginiamo di avere un file di testo contenente delle righe che vogliamo ordinare alfabeticamente per produrre un file di testo ordinato. Possiamo aprire un doppio contesto con i due file, in questo modo:

```
with open('unordered.txt') as f_in, open('ordered.txt', 'w') as
f_out:
    lines=f_in.readlines()
    lines.sort()
    f_out.writelines(lines)
```

Link alla documentazione ufficiale

- open
- readline
- close
- write

- with
- readlines

Serializzazione e deserializzazione dati con Pickle

Serializzazione e deserializzazione

Per serializzazione si intende un procedimento attraverso il quale dati di qualsiasi tipo presenti in memoria vengono rappresentati in una sequenza di byte che può essere poi salvata in un file per poter essere recuperata successivamente (con l'operazione inversa, detta deserializzazione).

Possono essere serializzati valori semplici, ma anche istanze complete di classi, come si può vedere in questi due esempi, in cui per la serializzazione è stato usato il modulo `pickle`.

Semplice lista

```
import pickle

data=('foo', 'bar', 'baz')

f = open('mydata', 'wb')
pickle.dump(data, f, pickle.HIGHEST_PROTOCOL)
f.close()

data=()

f = open('mydata', 'rb')
data = pickle.load(f)
f.close()

print(data)
```

Output:

```
('foo', 'bar', 'baz')
```

Oggetto complesso

In questo esempio vediamo come la serializzazione possa operare su una tupla che contiene al suo interno una tupla di istanze della classe Foo, una lista e un dizionario:

```
class Foo():
    def __init__(self, name):
        self.name = name
    def __str__(self):
        return 'Foo «%s»' % self.name

data = (
    Foo('abc'), Foo('def'), Foo('ghi'),
    ['uno', 'due', 'tre'],
    {'a': 'primo', 'b': 'secondo', 'c': 'terzo'}
    )

f = open('mydata', 'wb')
pickle.dump(data, f, pickle.HIGHEST_PROTOCOL)
f.close()

data=()

f = open('mydata', 'rb')
data = pickle.load(f)
f.close()

print(data)
print(data[0])
```

Output:

```
(<__main__.Foo object at 0xac8a98c>, <__main__.Foo object at 0xa
c0ceac>, <__main__.Foo object at 0xac86d4c>, ['uno', 'due', 'tre
'], {'a': 'primo', 'c': 'terzo', 'b': 'secondo'})
Foo «abc»
```

Uso dei contesti per la gestione del file

Come per i file di testo, è possibile usare la parola chiave `with` per aprire un contesto:

```
import pickle

data=('foo', 'bar', 'baz')

with open('mydata', 'wb') as f:
    pickle.dump(data, f, pickle.HIGHEST_PROTOCOL)

data=()

with open('mydata', 'rb') as f:
    data = pickle.load(f)

print(data)
```

Nota sulla sicurezza

Quando si caricano i dati memorizzati in un pickle, si deve tenere presente che in esso potrebbero essere memorizzate anche funzioni da richiamare. Ciò comporta un rischio: mai caricare dati da un pickle di fonte non fidata!

File binari

Rappresentazione dei dati in forma binaria

Per poter scrivere dati in un file binario è necessario trasformarli in sequenze di byte con esplicitazione del modo in cui queste sequenze sono rappresentate (ad esempio, quanti byte per un numero intero, se il numero è con o senza segno, se l'ordine dei byte è big-endian o little-endian, ecc.).

Il modulo `struct` di Python ci viene in soccorso.

Si consideri il seguente esempio:

```
import struct

FORMAT='<2hf'
data = struct.pack(FORMAT, 30000, -12, 2.35)  #packing
items = struct.unpack(FORMAT, data)   #unpacking
print(items)
```

Output:

```
(30000, -12, 2.3499999046325684)
```

Il formato FORMAT, da interpretare alla luce delle informazioni presenti nella pagina struct della documentazione, significa:

- '<': che l'ordine dei byte è little-endian
- 'h': che segue un intero corto di due byte
- 'f': che segue un numero in virgola mobile di quattro byte

Se si ha a che fare con stringhe, bisogna scegliere il tipo di rappresentazione della stringa e gestire la codifica e la decodifica:

```
name="Mario Rossi Albertucci"
age=20
FORMAT='<20sb'
data = struct.pack(FORMAT, name.encode('UTF-8'), age)  #packing
items = struct.unpack(FORMAT, data)  #unpacking
print(items[0].decode('UTF-8'), items[1])
```

Output:

```
Mario Rossi Albertuc 20
```

Si noti che in questo caso la stringa viene codificata in formato UTF-8 e poi messa in una stringa di 20 caratteri (per cui il valore viene troncato).

Scrittura di dati in un file binario

Un primo semplice esempio di scrittura di dati in un file binario potrebbe essere il seguente (nel quale abbiamo aggiunto l'uso di un oggetto di tipo `struct.Struct`).

```
import struct

class Person():
    struct_format = struct.Struct('<20sb')

    def __init__(self, name, age):
        self.name = name
        self.age = age

    def getPacked(self):
        return self.struct_format.pack(
            self.name.encode('UTF-8'),
            self.age
            )

people = (
    Person('Mario Rossi', 20),
    Person('Alberto Alberti', 21),
    Person('Barbara Balducci', 22),
    Person('Carla Carletti', 23)
    )

with open('people.dat', 'wb') as f:
    for person in people:
        f.write(person.getPacked())
```

Lettura di dati da un file binario

Per la lettura è sufficiente leggere il file e fare l'*unpacking*; abbiamo però modificato la funzione `__init__()` della classe Person per far sì che la stringa venga compattata, rimuovendo spazi e valori nulli finali che erano stati aggiunti in fase di impacchettamento.

```
import struct

class Person():
    struct_format = struct.Struct('<20sb')

    def __init__(self, name, age):
        self.name = name.strip('\0x00 ')
        self.age = age

    def getPacked(self):
        return self.struct_format.pack(
            self.name.encode('UTF-8'),
            self.age
            )

    def __str__(self):
        return 'Nome: %s, età: %d' % (self.name, self.age)

with open('people.dat', 'rb') as f:
    while True:
        data = f.read(Person.struct_format.size)
        if not data:
            break
        name, age = Person.struct_format.unpack(data)
        p = Person(name.decode('UTF-8'), age)
        print(p)
```

Accesso diretto (*random access*)

Se durante la lettura si vuole posizionare il cursore in un determinato punto del file, si può usare il metodo `seek()`.

Ad esempio, supponendo di avere un file people.idx contenente un indice e un file people.dat contenente i dati di varie persone, potremmo leggere il file indice sequenzialmente e, per ogni indice letto, fare un posizionamento sul file dati, per leggere in quest'ultimo il record che ci interessa:

```
index_format = struct.Struct('<h')

with open('people.idx', 'rb') as i:
    with open('people.dat', 'rb') as f:
        while True:
            data = i.read(index_format.size)
            if not data: break
            idx = index_format.unpack(data)[0]
            print('Indice %d: ' % idx, end='')
            f.seek(idx*Person.struct_format.size)
            data = f.read(Person.struct_format.size)
            name, age = Person.struct_format.unpack(data)
            p = Person(name.decode('UTF-8'), age)
            print(p)
```

Output:

```
Indice 3: Nome: Carla Carletti, età: 23
Indice 1: Nome: Alberto Alberti, età: 21
Indice 0: Nome: Mario Rossi, età: 20
Indice 2: Nome: Barbara Balducci, età: 22
```

Se si intende modificare un record, sarà necessario fare due posizionamenti, uno per la lettura e uno, successivamente, per la scrittura. Il file dovrà essere aperto in modalità 'r+':

```
with open('people.dat', 'r+b') as f:
    idx=2 # vogliamo modificare il record in posizione 2
    f.seek(idx*Person.struct_format.size)
    data = f.read(Person.struct_format.size)
    name, age = Person.struct_format.unpack(data)
    p = Person(name.decode('UTF-8'), age)
    print(p)
    p.name = 'Giorgio Giacomelli'
    f.seek(idx*Person.struct_format.size)
    f.write(p.getPacked())
```

Considerazioni sulla qualità del codice

Negli esempi sopra riportati il codice è stato ridotto all'osso per semplicità. In realtà bisognerà sempre effettuare dei controlli sulla possibilità o meno di leggere/scrivere da/su un file, e inserire il codice più problematico in blocchi `try... except... finally`.

Programmazione GUI con Tkinter

Introduzione

I programmi che vogliono mettere a disposizione una GUI (graphical user interface) possono essere sviluppati in diversi modi usando Python.

Essendo un linguaggio multipiattaforma, i programmi Python devono tendenzialmente essere eseguibili su diversi tipi di calcolatore e con diversi sistemi operativi, garantendo funzionalità e, se possibile, *look&feel* analoghi (anche se adattati all'interfaccia utente standard delle diverse piattaforme).

Di seguito presenteremo qualche esempio che sfrutta la libreria *Tkinter*, che probabilmente non è la più semplice da usare né quella che garantisce i risultati migliori, ma ha due indubbi vantaggi:

- è fornita insieme a Python;
- funziona anche con Python versione 3 e successive.

Documentazione (e guida alla sua lettura)

La documentazione su *Tkinter* c'è ed è sufficientemente accurata da poter essere utilizzata per scrivere almeno qualche programma di base, anche se a volte gli esempi forniti riguardano Python 2.x, e quindi non sono perfettamente compatibili con Python 3.x. Piccole modifiche al codice sorgente e qualche ricerca su web possono però risolvere i problemi.

Si può iniziare sicuramente da Tkinter per sopravvivere, di Stephen Ferg (con traduzione di Massimo Piai). La documentazione ufficiale ha una sezione su Tkinter che può essere d'aiuto. Altro documento interessante può essere An Introduction to Tkinter di Fredrik Lundh.

Nello studiare gli esempi che si trovano su Internet, bisognerà considerare che con Python 3.x:

- il modulo da importare si chiama tkinter (minuscolo)
- la funzione print richiede le parentesi

- alcuni componenti, come filedialog, messagebox, ecc. vanno importati in maniera leggermente diversa

Altre cose degne di nota sono che:

- IDLE (l'editor spesso usato per gestire il codice python) è esso stesso un'applicazione tkinter, e ciò può comportare dei problemi durante l'esecuzione. Si consiglia di avviare le applicazioni GUI direttamente e non premendo F5 dall'interno di IDLE;
- è bene che i file abbiano estensione .pyw, in modo che le finestre vengano aperte direttamente;
- per essere eseguiti direttamente in ambiente linux, la prima riga dovrebbe contenere la scritta #!/usr/bin/env python3.1 e il file dovrebbe essere reso eseguibile (chmod +x)

Esempi di base

Hello, world!

Un primo esempio fa comparire una finestra di dialogo (senza nulla dentro).

```
import tkinter
root = tkinter.Tk()
root.mainloop()
```

Avremmo potuto ottenere lo stesso risultato importando direttamente nel namespace corrente le componenti di tkinter:

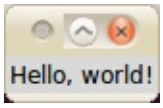
```
from tkinter import *
root = Tk()
root.mainloop()
```

Quando il ciclo principale si esegue esso attende gli eventi che accadono nell'oggetto radice.

Se avviene un evento allora esso viene gestito e il ciclo prosegue, continuando la sua attesa per il successivo evento. L'esecuzione del ciclo continua sinché nella finestra radice non si verifica un evento «destroy» (ad esempio, quando viene chiusa la finestra). Quando la radice è distrutta, il ciclo di gestione degli eventi termina.

Se vogliamo aggiungere un'etichetta nella finestra, dobbiamo prima definirla e poi sistemarla (con il metodo pack o con altri) nella finestra principale o in un altro contenitore. Il processo serve a stabilire una relazione visuale tra il widget e il suo genitore. In sua assenza, il widget esiste ma non viene visualizzato.

```
from tkinter import *
root = Tk()
MyLabel = Label(root, text="Hello, world!")
MyLabel.pack()
root.mainloop()
```



Gli oggetti *GUI* (*widget*, da *windows gadget*) vengono organizzati in gerarchie. Nell'esempio, l'etichetta MyLabel è figlia dell'oggetto principale, root.

Per i widget possono essere definiti i valori di alcuni attributi in diversi modi. Il più semplice è di impostarli come se fossero valori di un dizionario:

```
from tkinter import *
root = Tk()
MyLabel = Label(root, text="The quick brown fox jumps over the l
azy dog.")
MyLabel['background']="#FFFFFF"
MyLabel['foreground']="red"
MyLabel.pack()
root.mainloop()
```

Avremmo potuto impostare anche il testo scrivendo un codice simile al seguente:

```
MyLabel['text']="The quick brown fox jumps over the lazy dog."
```

Si può avere un po' di controllo su come i widget vengono sistemati all'interno della finestra usando l'attributo "side", come nell'esempio che segue:

```
from tkinter import *
root = Tk()
MyLabel1 = Label(root, text="Prima etichetta.")
MyLabel1['background']="#FFFFFF"
MyLabel1['foreground']="red"
MyLabel1.pack({"side":"right"})
MyLabel1.pack()
MyLabel2 = Label(root, text="Seconda etichetta.")
MyLabel2['background']="#FFFFFF"
MyLabel2['foreground']="blue"
MyLabel2.pack({"side":"left"})
root.mainloop()
```

Il metodo `pack()` serve ad accatastare i widget uno dopo l'altro a partire dalla posizione indicata. Vedremo ulteriori esempi più avanti.

Gestione degli eventi

Per i pulsanti è possibile impostare un gestore di eventi, in modo da richiamare una determinata funzione al clic del mouse.

```
from tkinter import *

def MyButton_Click():
    print("Mi hai cliccato... (standard output)")
    StatusBar["text"]="mi ha cliccato... (status bar)"

root = Tk()

MyButton = Button(root, text="Fai clic qui")
MyButton['background']="#FFFFFF"
MyButton['foreground']="red"
MyButton['command']=MyButton_Click
MyButton.pack({"side":"top", "padx": 10, "pady": 20})

StatusBar = Label(root, text="...")
StatusBar['background']="#FFFFFF"
StatusBar['foreground']="blue"
StatusBar.pack({"side":"bottom", "expand":"yes", "fill":"x"})

root.mainloop()
```

prima…

dopo…

Input di testo

Per poter chiedere in input un testo all'utente, si può usare un widget di tipo Entry, che viene associato ad una variabile.

```
from tkinter import *
def MyButton_Click():
    StatusBar["text"]=name.get()

root = Tk()
name = StringVar()
MyInputBox = Entry(root, textvariable=name)
MyInputBox.pack()
MyButton = Button(root, text="Fai clic qui")
MyButton['background']="#FFFFFF"
MyButton['foreground']="red"
MyButton['command']=MyButton_Click
MyButton.pack({"side":"top", "padx": 10, "pady": 20})
StatusBar = Label(root, text="...")
StatusBar['background']="#FFFFFF"
StatusBar['foreground']="blue"
StatusBar.pack({"side":"bottom", "expand":"yes", "fill":"x"})

root.mainloop()
```

Implementazione Object-Oriented

Per applicazioni con un minimo di complessità, conviene definire una propria classe, come nell'esempio seguente:

```
from tkinter import *

class Application(object):
    def __init__(self, parent):
        self.name = StringVar()
        self.MyInputBox = Entry(parent, textvariable=self.name)
        self.MyInputBox.pack()
        self.MyButton = Button(parent, text="Fai clic qui")
        self.MyButton['background']="#FFFFFF"
        self.MyButton['foreground']="red"
        self.MyButton['command']=self.MyButton_Click
        self.MyButton.pack({"side":"top", "padx": 10, "pady": 20
})
        self.StatusBar = Label(parent, text="...")
        self.StatusBar['background']="#FFFFFF"
        self.StatusBar['foreground']="blue"
        self.StatusBar.pack({"side":"bottom", "expand":"yes", "f
ill":"x"})

    def MyButton_Click(self):
        self.StatusBar["text"]=self.name.get()

def main():
    root = Tk()
    myapp = Application(root)
    root.mainloop()

if __name__=='__main__':
    main()
```

Creazione e gestione dinamica dei widget

Può essere utile creare dinamicamente dei widget durante l'esecuzione del programma (ad esempio, a seconda dell'input dell'utente). Se si vogliono creare dei pulsanti, si pone il problema di come associare una funzione che gestisce gli eventi associati ai vari pulsanti (visto che il gestore di eventi è una funzione che non ha parametri). La soluzione, come nell'esempio seguente, è di creare una funzione dinamicamente (tramite la parola chiave lambda). Approfondiremo l'uso

delle espressioni lambda più avanti, nella lezione 22. Vedremo anche, nella lezione 23, che attraverso `functools.partial()` si può ottenere lo stesso risultato in maniera leggermente più semplice.

```
from tkinter import *

def str2int(s):
    try:
        k=int(s)
        return k
    except:
        return 0

class Application(object):
    def __init__(self, parent):

        self.name = StringVar()
        self.buttons=[]
        self.parent = parent

        self.MyInputBox = Entry(parent, textvariable=self.name)
        self.MyInputBox.pack()

        self.MyButton = Button(parent, text="Fai clic qui")
        self.MyButton['background']="#FFFFFF"
        self.MyButton['foreground']="red"
        self.MyButton['command']=self.MyButton_Click
        self.MyButton.pack({"side":"top", "padx": 10, "pady": 20
})

        self.StatusBar = Label(parent, text="...")
        self.StatusBar['background']="#FFFFFF"
        self.StatusBar['foreground']="blue"
        self.StatusBar.pack({"side":"bottom", "expand":"yes", "f
ill":"x"})

    def MyButton_Click(self):
        self.StatusBar["text"]=self.name.get()
        n=str2int(self.name.get())
        for i in range(n):
```

```
            b=Button(self.parent, text="pulsante " + str(i))
            b['background']='yellow'
            b['command']=self.BuildButtonAction(i)
            b.pack()
            self.buttons.append(b)

    def BuildButtonAction(self, number):
        return lambda : self.ChangeButtonColor(number)

    def ChangeButtonColor(self, number):
        self._SwitchColor(self.buttons[number])

    def _SwitchColor(self, button):
        button['background']='yellow' if button['background']=='
red' else 'red'

def main():
    root = Tk()
    myapp = Application(root)
    root.mainloop()

if __name__=='__main__':
    main()
```

Controllo avanzato dei widget

Widget particolari

Presenteremo in questa sezione alcuni widget utili, illustrandoli con semplici esempi.

Pulsante con icona

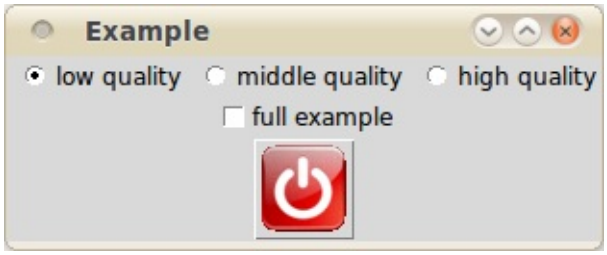
Ad un pulsante può essere associata un'icona. È sufficiente caricare un'immagine (in formato GIF) e poi associarla al pulsante tramite la proprietà image.

```
self.imq=PhotoImage(file='exit.gif')

self.QuitButton=Button(
    self.parent,
    image=self.imq,
    command=self.Quit
    )
self.QuitButton.pack()
```

Pulsanti di selezione e di opzione (*radio buttons* e *check buttons*)

I pulsanti di selezione e di opzione si ottengono con i widget *Radiobutton* e *Checkbutton*, rispettivamente, associati a variabili di tipo `IntVar` o `BooleanVar`.

```
self.quality = IntVar()
self.quality.set(10)

f=Frame(self.parent)

self.qualityRadiobutton_Low=Radiobutton(
    f,
    text='low quality',
    variable=self.quality,
    value=10
    )
self.qualityRadiobutton_Middle=Radiobutton(
    f,
    text='middle quality',
    variable=self.quality,
    value=20
    )
self.qualityRadiobutton_High=Radiobutton(
    f,
    text='high quality',
    variable=self.quality,
    value=30
    )

self.qualityRadiobutton_Low.pack({'side':'left'})
self.qualityRadiobutton_Middle.pack({'side':'left'})
self.qualityRadiobutton_High.pack({'side':'left'})

f.pack()

self.full=BooleanVar()
self.full.set(False)

self.fullCheckbutton=Checkbutton(
    self.parent,
    text='full example',
    variable=self.full
    )
self.fullCheckbutton.pack()
```



Menù di opzioni

Nell'esempio che segue vediamo come si possono impostare alcune proprietà dei widget attraverso un menù di opzioni.

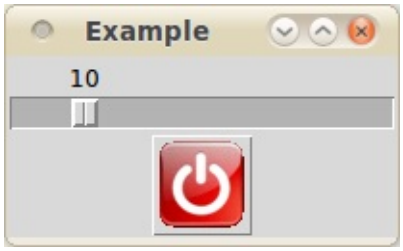
```
from tkinter import *

class Application(object):
    def __init__(self, parent):

        self.propertyname = StringVar()
        self.propertyname.set('foreground') # valore di default
        self.propertyvalue = StringVar()
        self.propertyvalue.set('green')
        self.parent = parent

        properties = ['background',
                      'foreground',
                      'activebackground',
                      'disabledforeground',
                      'highlightcolor',
                      'highlightbackground',
                      'width',
                      'height',
                      'padx',
                      'pady',
                      ]

        self.PropertyMenu = OptionMenu(parent, self.propertyname
, *properties)
        self.PropertyMenu.pack()

        self.MyInputBox = Entry(parent, textvariable=self.proper
tyvalue)
```

```
        self.MyInputBox.pack()

        self.MyButton = Button(parent, text="Fai clic qui")
        self.MyButton['background']="#FFFFFF"
        self.MyButton['foreground']="red"
        self.MyButton['command']=self.MyButton_Click
        self.MyButton.pack({"side":"top", "padx": 10, "pady": 20
})

        self.StatusBar = Label(parent, text="...")
        self.StatusBar['background']="#FFFFFF"
        self.StatusBar['foreground']="blue"
        self.StatusBar.pack({"side":"bottom", "expand":"yes", "f
ill":"x"})

    def MyButton_Click(self):
        print('name:', self.propertyname.get())
        print('value', self.propertyvalue.get())
        self.MyButton[self.propertyname.get()]=self.propertyvalu
e.get()


def main():
    root = Tk()
    myapp = Application(root)
    root.mainloop()

if __name__=='__main__':
    main()
```

Regolatore

Un regolatore consente di variare il valore di una variabile facendo scorrere una sorta di manopola.

```
self.quality = IntVar()
self.quality.set(10)

self.qualityScale = Scale(
    self.parent,
    variable=self.quality,
    from_=0,
    to=60,
    orient=HORIZONTAL,
    length=200,
    sliderlength=15
    )

self.qualityScale.pack()
```



Caselle di testo multilinea (con barra di scorrimento)

Le caselle di testo possono essere multilinea e consentono un sacco di operazioni (estrazioni di parte del testo, sostituzioni, formattazione, ecc.). Esempi e documentazione si possono trovare nel Tk Tutorial.

Alle caselle di testo, così come ai canvas e ad altri widget "grandi" possono essere associate le barre di scorrimento (verticali o orizzontali).

Una barra di scorrimento ha un doppio legame con il widget associato (bisogna impostare il parametro `command` della barra alla funzione `yview` o `xview` del widget e il parametro `yscrollcommand` o `xscrollcommand` del widget alla funzione `set` della barra). Può essere conveniente inserire widget principale e barra di scorrimento in un frame:

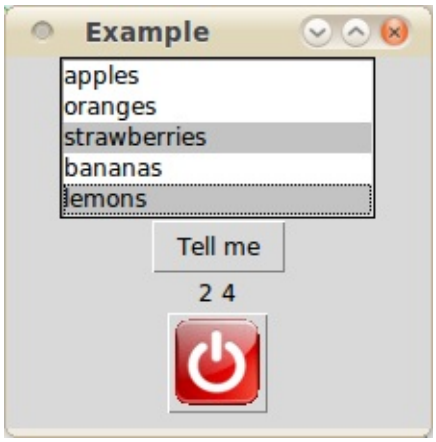
```
self.lyricsFrame=Frame(self.parent)

self.lyricsText = Text(self.lyricsFrame,
    width=80,
    height=4
    )
self.lyricsText.pack({'side':'left'})

self.lyricsText.insert("1.0",
    "Lorem ipsum dolor sit amet, consectetur adipiscing elit.\nN
am consectetur dictum nisl, eget blandit magna suscipit tristiqu
e.\nFusce sit amet nunc eu lorem varius luctus ac id eros.\nClas
s aptent taciti sociosqu ad litora torquent per conubia nostra,\
nper inceptos himenaeos.")

self.lyricsScrollbar=Scrollbar(self.lyricsFrame)
self.lyricsScrollbar.pack({'side':'left', 'expand':'yes', 'fill'
:'y'})

self.lyricsFrame.pack()

self.lyricsScrollbar['command']=self.lyricsText.yview
self.lyricsText['yscrollcommand']=self.lyricsScrollbar.set
```

Casella di elenco

Una casella di elenco contiene un elenco di opzioni tra cui l'utente può scegliere (tipicamente preso da una tupla o da una lista). L'indice o gli indici degli elementi selezionati partono da 0.

La casella può essere di tipo single, browse, multiple o extended, a seconda del numero di elementi selezionabili e dalla modalità di interazione possibile.

```
self.fruitListbox=Listbox(
    self.parent,
    height=5,
    selectmode='extended'
    )

self.fruits=('apples', 'oranges', 'strawberries', 'bananas', 'le
mons')

for f in self.fruits:
    self.fruitListbox.insert("end", f)

self.fruitListbox.pack()
```



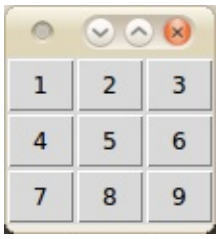
Canvas

Un canvas è una tavolozza dove si può disegnare.

```
class Application(object):
    def __init__(self, parent):

        self.Canvas = Canvas(parent, width=300, height=400)
        self.Canvas['background'] = 'white'
        self.Canvas.pack()

        self.r1_id = None

        self.MyButton = Button(parent, text="Fai clic qui")
        self.MyButton['background']="#FFFFFF"
        self.MyButton['foreground']="red"
        self.MyButton['command']=self.MyButton_Click
        self.MyButton.pack({"side":"top", "padx": 10, "pady": 20
})

        self.StatusBar = Label(parent, text="...")
        self.StatusBar['background']="#FFFFFF"
        self.StatusBar['foreground']="blue"
        self.StatusBar.pack({"side":"bottom", "expand":"yes", "f
ill":"x"})

    def MyButton_Click(self):
        if self.r1_id:
            self.Canvas.move(self.r1_id, 2, 5)
        else:
            self.r1_id = self.Canvas.create_rectangle(2, 10, 8,
20)
```

Posizionamento dei widget

Vedremo con tre esempi semplicissimi come posizionare i nostri widget nella finestra utilizzando le tre modalità che Tkinter ci mette a disposizione. Il nostro obiettivo è di avere una serie di nove pulsanti numerati.



Pack()

Con il metodo `pack()` specifichiamo dove mettere un widget in relazione agli altri (a sinistra, a destra, in alto, in basso). Può essere comodo avere dei widget invisibili (frame) che ne contengono altri.

```
class Application(object):
    def __init__(self, parent):

        self.UpperFrame=Frame(parent)
        self.MiddleFrame=Frame(parent)
        self.BottomFrame=Frame(parent)

        self.b1 = Button(self.UpperFrame, text="1")
        self.b2 = Button(self.UpperFrame, text="2")
        self.b3 = Button(self.UpperFrame, text="3")

        for widget in self.b1, self.b2, self.b3:
            widget.pack({'side':'left'})

        self.b4 = Button(self.MiddleFrame, text="4")
        self.b5 = Button(self.MiddleFrame, text="5")
        self.b6 = Button(self.MiddleFrame, text="6")

        for widget in self.b4, self.b5, self.b6:
            widget.pack({'side':'left'})

        self.b7 = Button(self.BottomFrame, text="7")
        self.b8 = Button(self.BottomFrame, text="8")
        self.b9 = Button(self.BottomFrame, text="9")

        for widget in self.b7, self.b8, self.b9:
            widget.pack({'side':'left'})

        self.UpperFrame.pack({'side':'top'})
        self.MiddleFrame.pack({'side':'top'})
        self.BottomFrame.pack({'side':'top'})
```

Grid()

Con il metodo `grid()` organizziamo la nostra finestra come se fosse una tabella, e inseriamo i nostri widget all'interno delle diverse celle.

```
class Application(object):
    def __init__(self, parent):

        self.parent= parent

        self.digitButtons = []

        for digit in range(9):
            digitButton = Button(parent, text=str(digit+1))
            digitButton.grid(row=digit//3, column=digit%3)
            self.digitButtons.append(digitButton)

        self.QuitButton=Button(parent, text='quit')
        self.QuitButton['command'] = self.Quit
        self.QuitButton.grid(row=4, column=4)

    def Quit(self):
        self.parent.destroy()
```

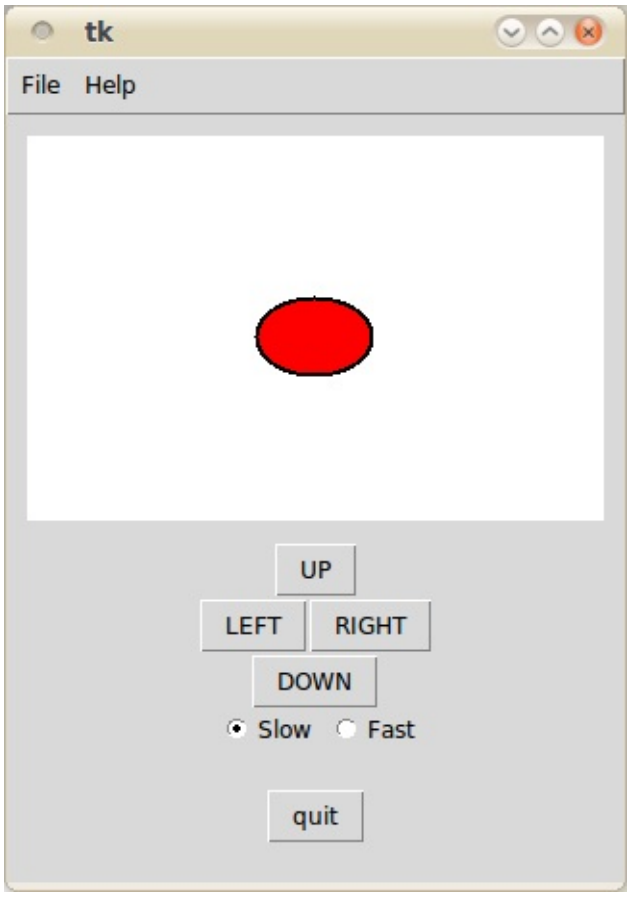
Place()

Con il metodo `place()` indichiamo esplicitamente dove posizionare i widget, specificandone le coordinate in termini assoluti o relativi.

```
class Application(object):
    def __init__(self, parent):

        self.parent= parent

        SCALE_X=30
        OFFSET_X=10
        SCALE_Y=40
        OFFSET_Y=60

        self.digitButtons = []

        for digit in range(9):
            digitButton = Button(parent, text=str(digit+1))
            digitButton.place(
                y=(digit//3)*SCALE_Y+OFFSET_Y,
                x=(digit%3)*SCALE_X+OFFSET_X
                )
            self.digitButtons.append(digitButton)

        self.QuitButton=Button(parent, text='quit')
        self.QuitButton['command'] = self.Quit
        self.QuitButton.place(relx=0.7, rely=0.8)

    def Quit(self):
        self.parent.destroy()
```

Menù, timer, eventi di tastiera, finestre di dialogo

Un'applicazione con interfaccia grafica spesso consente un'interazione sofisticata con l'utente, attraverso i pulsanti, le finestre di dialogo (informative o per la scelta di un file), la gestione degli "eventi di tastiera" (ossia le pressioni dei tasti). Inoltre, può essere utile avere a disposizione un timer per controllare il passare del tempo. Per esemplificare l'uso di queste cose, presenteremo qui una piccola applicazione gioco (non completa), con il seguente aspetto:



Menù

I menù sono realizzati predisponendo una barra agganciata alla finestra principale, nella quale vengono "inserite" le voci principali (`add_cascade`), alla quale appartengono le voci che possono essere effettivamente scelte (`add_command`):

```
def setMenu(self):
    MenuBar = Menu(self.parent)
    self.parent['menu']=MenuBar

    FileMenu=Menu(MenuBar)
    FileMenu.add_command(label='Open…', command=self.FileLoad)
    FileMenu.add_command(label='Save as…', command=self.FileSave
)
    FileMenu.add_command(label='Quit', command=self.Quit)

    HelpMenu=Menu(MenuBar)
    HelpMenu.add_command(label='About', command=self.HelpAbout)

    MenuBar.add_cascade(label='File', menu=FileMenu)
    MenuBar.add_cascade(label='Help', menu=HelpMenu)
```

Gestione degli eventi di tastiera

La pressione di tasti particolari della tastiera si gestisce con il collegamento (bind) della finestra principale ad una funzione.

```
self.parent.bind('<Up>', self.UpButton_Click)
self.parent.bind('<Down>', self.DownButton_Click)
self.parent.bind('<Left>', self.LeftButton_Click)
self.parent.bind('<Right>', self.RightButton_Click)
self.parent.bind('q', self.Quit)
```

Timer

Per usare il timer di tkinter si deve impostare una funzione con il metodo after della finestra principale, specificando il numero di millisecondi dopo i quali la funzione dev'essere richiamata. Se l'evento è ricorrente, nella funzione chiamata bisognerà reimpostare la funzione.

```
self.parent.after(500, self.MoveObjectDown)
```

Finestre di dialogo

Per usare le finestre di dialogo, bisogna importare da tkinter esplicitamente le componenti necessarie:

```
from tkinter import messagebox
from tkinter import filedialog
```

Messagebox

Messagebox consente di visualizzare messaggi:

```
messagebox.showinfo(title='THE BIG GAME', message='--- GAME OVER
 ---')
```

Filedialog

Filedialog consente di chiedere il nome di un file (da aprire o su cui salvare):

```
def FileLoad(self):
    ftypes = [('Text files', '*.txt'), ('XML Files', '*.xml'), (
'All files', '*.*')]
    filename = filedialog.askopenfilename(title='Chose a file to
 open', filetypes=ftypes)
    if filename:
        messagebox.showinfo(title='File opening', message='I sho
uld open file ' + filename)

def FileSave(self):
    ftypes = [('Text files', '*.txt'), ('XML Files', '*.xml'), (
'All files', '*.*')]
    filename = filedialog.asksaveasfilename(title='Chose where t
o save data', filetypes=ftypes)
    if filename:
        messagebox.showinfo(title='File saving', message='I shou
ld save to file ' + filename)
```

Finestre secondarie

Una finestra secondaria può essere generata con `Toplevel()`. Ad esempio, la finestrella di informazioni sull'autore e sulla licenza può essere richiamata visualizzata con un codice simile al seguente:

```
def HelpAbout(self):
    self.AboutWindow = Toplevel(self.parent)
    self.AboutWindow.title('info')
    self.AuthorLabel = Label(self.AboutWindow, text='Author: ' +
 AUTHOR)
    self.AuthorLabel.pack()
    self.LicenceLabel = Label(self.AboutWindow, text='Licence: '
 + LICENCE)
    self.LicenceLabel.pack()
    self.CloseAboutWindowButton = Button(self.AboutWindow, text=
'Close', command=self.CloseAboutWindowButton_Click)
    self.CloseAboutWindowButton.pack()

def CloseAboutWindowButton_Click(self):
    self.AboutWindow.destroy()
```

Se si desidera che la finestra sia "modale" (ossia che l'input della finestra principale sia inibito fino a quando essa non viene chiusa), sarà necessario aggiungere l'istruzione

```
self.AboutWindow.grab_set()  # questo rende la finestra "modale"
```

Creazione di un widget personalizzato e convalida dell'input

Introduzione

In questa lezione vedremo come preparare un proprio widget personalizzato mettendo insieme alcuni widget base. Approfitteremo dell'esempio preparato anche per vedere come effettuare una convalida dei dati.

Immaginiamo di voler creare un banalissimo programma per la semplificazione di frazioni, in cui però decidiamo di voler preparare un widget personalizzato per l'input del numeratore e del denominatore della frazione.

Il risultato che vogliamo ottenere è il seguente:

Programma principale

Vogliamo fare in modo di poter aggiungere la frazione nella finestra principale come se fosse un widget normale, scrivendo qualcosa tipo:

```
self.parent = parent

self.Fraction=FractionWidget(self.parent, background='red')
self.Fraction['borderwidth']=10
self.Fraction['relief']='ridge'
self.Fraction['numerator']=12
self.Fraction['denominator']=24

self.Fraction.pack({'side':'top', 'padx':10, 'pady':10})
```

Come si vede, il nostro widget, chiamato `FractionWidget`, si comporta a tutti gli effetti come un widget di base (il costruttore riceve dei parametri che vengono impostati, le proprietà possono essere modificate con il metodo dell'accesso tramite parentesi quadre. ecc.). Inoltre, è possibile impostare i valori per il numeratore e il denominatore.

La classe FractionWidget

Per il nostro widget personalizzato definiamo una classe, in cui il costruttore prepara un frame in cui sono presenti due widget Entry separati da un widget Label con i trattini (si può fare di meglio). Per il numeratore e denominatore impostiamo la proprietà justify a right.

```
def __init__(self, parent, height=10, width=10, background=None)
:
    self.parent=parent
    self.height=height
    self.width=width

    self.NumeratorValue=IntVar()
    self.NumeratorValue.set(1)
    self.DenominatorValue=IntVar()
    self.DenominatorValue.set(1)

    self.MainFrame=Frame(self.parent, width=self.width, height=s
elf.height)
    self.Numerator=Entry(self.MainFrame,
                         textvariable=self.NumeratorValue,
                         width=self.width,
                         justify="right")
    self.FractionLine=Label(self.MainFrame, text="------")
    self.Denominator=Entry(self.MainFrame,
                           textvariable=self.DenominatorValue,
                           width=self.width,
                           justify="right")
    self.Numerator.pack(side="top", padx=5, pady=5)
    self.FractionLine.pack(side="top")
    self.Denominator.pack(side="top", padx=5, pady=5)

    if background:
        self.MainFrame['background']=background
```

Pack, grid e place

Per fare in modo che sia possibile usare i metodi pack, grid e place per il nostro widget, essi andranno ridefiniti nella classe, eseguendoli con la stessa lista di parametri sul frame che abbiamo creato:

```
def pack(self, *args, **kwargs):
    self.MainFrame.pack(*args, **kwargs)

def grid(self, *args, **kwargs):
    self.MainFrame.grid(*args, **kwargs)

def place(self, *args, **kwargs):
    self.MainFrame.place(*args, **kwargs)
```

Convalida dell'input

Se vogliamo effettuare una convalida dell'input durante la digitazione (per evitare che vengano inseriti caratteri non numerici, nel nostro caso) bisogna adottare una tecnica un po' particolare, che deriva dal fatto che si è costretti a richiamare funzionalità di basso livello di Tk.

Si inizia definendo una tupla contenente il comando di validazione, composto da una funzione e da una serie di parametri che si possono tenere in considerazione per la validazione stessa:

```
vcmd=(self.parent.register(self.doValidation), '%P')
```

Poi si impostano per il widget in cui fare la validazione i parametri validate (quando effettuare la validazione) e validatecommand (la tupla impostata prima):

```
self.Numerator=Entry(self.MainFrame,
    ....
    validate="key",
    validatecommand=vcmd)
```

Per `validate` i valori validi sono `key` (pressione di un tasto), `focus`, `focusin`, `focusout` e `all`.

I parametri che si possono impostare per la validazione sono i seguenti:

- %d = tipo di azione (1=inserimento, 0=cancellazione, -1 altro)
- %i = indice del carattere da inserire/cancellare, oppure -1

- %P = valore della casella di input, se la modifica è consentita
- %s = valore della casella prima della modifica
- %S = stringa di testo che dovrebbe essere inserita o cancellata, se esiste
- %v = tipo di validazione correntemente impostata
- %V = tipo di validazione che ha fatto scattare la funzione
- %W = nome Tk del widget

Ridefinizione metodi degli operatori per l'accesso agli attributi

Se si vuole fare in modo che sia possibile cambiare le proprietà del widget mediante la sintassi delle parentesi quadrate, si devono ridefinire gli operatori mediante le funzioni `__setitem__` e `__getitem__` :

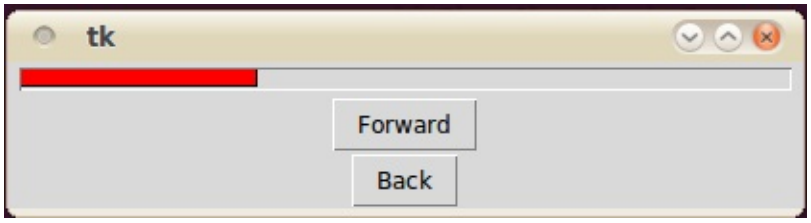
```
def __setitem__(self, key, value):
    if key in ('background', 'bg', 'borderwidth', 'bd', 'relief'
):
        self.MainFrame[key]=value
    if key=='numerator':
        self.setNumerator(value)
    if key=='denominator':
        self.setDenominator(value)


def __getitem__(self, key):
    if key in ('background', 'bg', 'borderwidth', 'bd', 'relief'
):
        return self.MainFrame[key]
    if key=='numerator':
        return self.getNumerator()
    if key=='denominator':
        return self.getDenominator()
```

Barra di progressione

Una barra di progressione può essere creata basandosi su un canvas in cui si disegna un rettangolo colorato, come nel seguente esempio:

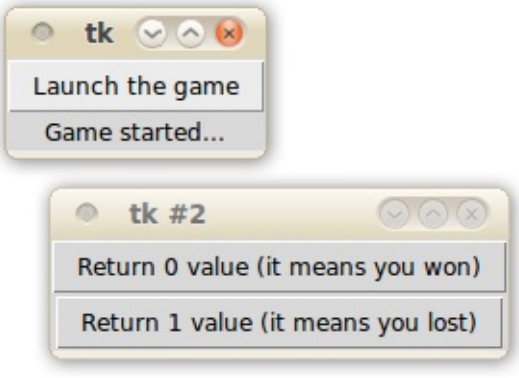
```
class ProgressBar(object):
    def __init__(self, parent, height=10, width=300, background=
None, foreground='black', progress=0):
        # adapted from "Python 2.1 Bible" (by Dave Brueck and St
ephen Tanner)
        self._parent=parent
        self._height=height
        self._width=width
        self._background=background
        self._foreground=foreground

        self.barcanvas=Canvas(
            self._parent,
            width=self._width,
            heigh=self._height,
            background=self._background,
            borderwidth=1,
            relief=SUNKEN
            )
        self.barcanvas.pack(padx=5, pady=2)
        self.rectangleValue=self.barcanvas.create_rectangle(0,0,
0, self._height)
        self.barcanvas.itemconfigure(
            self.rectangleValue,
            fill=self._foreground
            )

        self.setProgressPercent(progress)

    def getProgressPercent(self):
        return self._progress

    def setProgressPercent(self, level):
        self._progress = min(100, max(0, level))
        self.DrawProgress()

    def DrawProgress(self):
        pixels=round(self._width * self.getProgressPercent()/100
.0)
```

```
            self.barcanvas.coords(
                self.rectangleValue,
                0,
                0,
                pixels,
                self._height
                )

    def pack(self, *args, **kwargs):
        self.barcanvas.pack(*args, **kwargs)

    def grid(self, *args, **kwargs):
        self.barcanvas.grid(*args, **kwargs)

    def place(self, *args, **kwargs):
        self.barcanvas.place(*args, **kwargs)
```



Esecuzione di un programma esterno

Può capitare di dover eseguire un programma esterno per verificare un valore. Ad esempio, si immagini di volere che facendo clic su un pulsante venga avviato un programma in cui l'utente gioca a qualcosa. Il programma chiamante deve venire informato dell'esito del programma chiamato.



Per ottenere questo risultato, il programma chiamato deve uscire con un esito, che nel nostro esempio può essere 0 (il giocatore ha vinto), 1 (il giocatore ha perso) oppure 2 (il giocatore ha chiuso la finestra).

```
def main():
    root = Tk()
    myapp = Application(root)
    root.mainloop()
    return myapp.returnValue

if __name__=='__main__':
    sys.exit(main())
```

Il programma chiamante, d'altro canto, deve essere in grado di avviare un altro programma e capire qual ne è stato l'esito. Per farlo, useremo il modulo subprocess:

```
try:
    subprocess.check_output([interpretername(), 'thegame.pyw'])
    self.statusbar['text']='You won!'
except:
    self.statusbar['text']='You lose!'
```

Il nome dell'interprete varia da sistema a sistema, per cui possiamo definire una funzione interpretername() che ce lo restituisca a seconda di alcune considerazioni (file di configurazione, sistema operativo usato, o altro).

Si noti che questo metodo consente di far interagire programmi completamente indipendenti (anche scritti in altri linguaggi di programmazione).

Funzioni anonime (lambda)

Introduzione

Abbiamo visto nella lezione 4 che se si vuole ordinare una lista contenente riferimenti a oggetti di tipo diverso è necessario definire una funzione che li renda comparabili e passare la funzione al metodo sort come suo argomento.

Anche in altri casi può essere necessario passare una funzione al metodo `sort`. Ad esempio, si supponga di avere una lista di tuple, dove ogni tupla contiene nome, cognome e codice di una persona.

```
people=[
    ('John', 'Smith', 123),
    ('James', 'Baird', 231),
    ('Rebecca', 'Campbell', 457),
    ('Diane', 'Kelly', 125)
    ]
```

Se usiamo il metodo `sort()` sulla lista, otteniamo un ordinamento basato sul primo elemento di ogni tupla:

```
print('*** ordinamento per nome ***')
people.sort()
showList(people)

*** ordinamento per nome ***
('Diane', 'Kelly', 125)
('James', 'Baird', 231)
('John', 'Smith', 123)
('Rebecca', 'Campbell', 457)
```

Se vogliamo ordinare le persone per cognome, dobbiamo creare una funzione che estrae il cognome (secondo elemento della tupla) come primo valore da considerare per la comparazione:

```
def surname_first(data):
    return (data[1], data[0], data[2])
```

… e poi passare al metodo sort() la funzione definita:

```
print('*** ordinamento per cognome ***')
people.sort(key=surname_first)
showList(people)
```

Il risultato è quello che ci aspettiamo:

```
*** ordinamento per cognome ***
('James', 'Baird', 231)
('Rebecca', 'Campbell', 457)
('Diane', 'Kelly', 125)
('John', 'Smith', 123)
```

Se volessimo un ordinamento per codice, dovremmo definire un'altra funzione, che questa volta mette al primo posto il terzo elemento della tupla:

```
def code_first(data):
    return (data[2], data[1], data[0])
```

e richiamare il metodo `sort()` con questa nuova funzione:

```
print('*** ordinamento per codice ***')
people.sort(key=code_first)
showList(people)

*** ordinamento per codice ***
('John', 'Smith', 123)
('Diane', 'Kelly', 125)
('James', 'Baird', 231)
('Rebecca', 'Campbell', 457)
```

Funzioni anonime

Ovviamente, con l'aumentare del numero dei campi da prendere in considerazione, avremmo bisogno di definire sempre nuove funzioni, il che può costituire un indubbio svantaggio. In queste occasioni, vengono in soccorso le cosiddette funzioni anonime, che si creano al volo quando se ne presenta la necessità.

Ordinamento di una lista

Potremo così risolvere il problema dell'ordinamento in maniera più brillante:

```
print('*** ordinamento per cognome con funzione lambda ***')
people.sort(key=lambda x: (x[1], x[0], x[2]))
showList(people)
```

oppure

```
print('*** ordinamento per codice con funzione lambda ***')
people.sort(key=lambda x: (x[2], x[1], x[0]))
showList(people)
```

Come si vede in questi esempi, la parola chiave lambda consente di creare una funzione istantaneamente. Il risultato è naturalmente esattamente equivalente:

```
*** ordinamento per cognome con funzione lambda ***
('James', 'Baird', 231)
('Rebecca', 'Campbell', 457)
('Diane', 'Kelly', 125)
('John', 'Smith', 123)
*** ordinamento per codice con funzione lambda ***
('John', 'Smith', 123)
('Diane', 'Kelly', 125)
('James', 'Baird', 231)
('Rebecca', 'Campbell', 457)
```

Possiamo in questo modo gestire anche una scelta da parte dell'utente (qui con un minimale controllo di validità):

```
k=int(input("In base a quale campo vuoi l'ordinamento? "))
print('*** ordinamento su campo a scelta ***')
if not 0<k<3: k=0
people.sort(key=lambda x: x[k])
showList(people)

In base a quale campo vuoi l'ordinamento? 2
*** ordinamento su campo a scelta ***
('John', 'Smith', 123)
('Diane', 'Kelly', 125)
('James', 'Baird', 231)
('Rebecca', 'Campbell', 457)
```

Le funzioni anonime non supportano i cicli, però supportano le selezioni. Immaginiamo di avere una lista di clienti in cui per ogni cliente è specificato, come quarto campo, se si tratta di una persona fisica o di una persona giuridica ('p' e 'j', rispettivamente):

```
customers=[
    ('John', 'Smith', 123, 'p'),
    ('James', 'Baird', 231, 'p'),
    ('Acme SA', None, 458, 'j'),
    ('Diane', 'Kelly', 125, 'p'),
    ('Rebecca', 'Campbell', 457, 'p'),
    ('Alphagamma Ltd', None, 318, 'j'),
    ('Blueskies Inc.', None, 941, 'j')
    ]
```

Se desideriamo che l'ordinamento prenda in considerazione il nome per le persone giuridiche e il cognome per le persone fisiche, potremo scrivere una funzione lambda che contiene una condizione:

```
customers.sort(key=lambda x: x[0] if x[3]=='j' else x[1])
```

Il risultato sarà quello che ci attendiamo:

```
*** ordinamento per nome o cognome con funzione lambda e selezio
ne ***
('Acme SA', None, 458, 'j')
('Alphagamma Ltd', None, 318, 'j')
('James', 'Baird', 231, 'p')
('Blueskies Inc.', None, 941, 'j')
('Rebecca', 'Campbell', 457, 'p')
('Diane', 'Kelly', 125, 'p')
('John', 'Smith', 123, 'p')
```

Uso di una funzione su un'intera lista

Se vogliamo applicare una funzione a tutti gli elementi della lista, possiamo usare la funzione `map()`, che restituisce un oggetto iterabile (su cui si può fare un ciclo for, o che si può convertire in lista).

Ad esempio, immaginiamo di volere una lista di tutti i nomi dei nostri clienti (nome e cognome in maiuscolo per le persone fisiche, nome senza nessuna modifica per le persone giuridiche).

Potremmo scrivere un codice come questo:

```
def capitalizePersonNames(c):
  if c[3]=='p':
    return ' '.join(c[0:2]).upper()
    # mettiamo in maiuscolo i primi due campi, dopo averli
    # uniti con uno spazio
  else:
    return c[0]

names=list(map(capitalizePersonNames, customers))
print(names)

['JOHN SMITH', 'JAMES BAIRD', 'Acme SA', 'DIANE KELLY', 'REBECCA
 CAMPBELL', 'Alphagamma Ltd', 'Blueskies Inc.']
```

In alternativa, possiamo definire una funzione anonima, con lo stesso risultato:

```
names=list(map(
    lambda v: v[0] if v[3]=='j' else ' '.join(v[0:2]).upper(),
    customers))
print(names)
```

In maniera analoga, possiamo usare una funzione anonima per filtrare con la funzione filter() gli elementi di una lista in base ad una condizione (nell'esempio qui sotto, vogliamo solo le persone fisiche):

```
print('*** Elenco delle persone fisiche ***')
people=list(filter(lambda v: v[3]=='p', customers))
print(people)

*** Elenco delle persone fisiche ***
[('John', 'Smith', 123, 'p'), ('James', 'Baird', 231, 'p'), ('Di
ane', 'Kelly', 125, 'p'), ('Rebecca', 'Campbell', 457, 'p')]
```

Si noti che usando `map()` la lista originale non viene modificata. Se desideriamo che effettivamente i valori vengano cambiati, possiamo calcolare una nuova lista. Si consideri il seguente esempio, in cui una lista di prezzi deve essere aggiornata (con la maggiorazione del 10%):

```
prices=[100.0, 200.0, 340.0, 25.0, 180.0]
print('prezzi di partenza:')
print(prices)
print('prezzi aumentati (calcolati):')
print(list(map(lambda x: round(x*1.1, 2), prices)))
print('i prezzi di partenza non sono cambiati:')
print(prices)

prezzi di partenza:
[100.0, 200.0, 340.0, 25.0, 180.0]
prezzi aumentati (calcolati):
[110.0, 220.0, 374.0, 27.5, 198.0]
i prezzi di partenza non sono cambiati:
[100.0, 200.0, 340.0, 25.0, 180.0]

prices=list(map(lambda x: round(x*1.1, 2), prices))
print('i prezzi ora sono aggiornati:')
print(prices)

i prezzi ora sono aggiornati:
[110.0, 220.0, 374.0, 27.5, 198.0]
```

Definizione di funzioni tramite espressioni lambda

Può essere comodo, in alcuni contesti, definire una funzione come risultato di un'espressione lambda. Supponiamo ad esempio di voler definire una funzione che calcola l'area di un settore circolare, dato il raggio del cerchio e l'angolo, espresso in gradi.

Come sappiamo, potremmo scrivere la funzione in questo modo:

```
import math
def sector_area(radius, degrees):
    return radius**2*math.pi*degrees/360
```

Ma sarebbe esattamente equivalente scrivere anche così:

```
import math
sector_area = lambda radius, degrees: radius**2*math.pi*degrees/
360
```

In entrambi i casi, potremo chiamare la nostra funzione nel modo seguente:

```
print(sector_area(3,90))
7.06858347058
```

Funzioni lambda come gestori di evento

Un caso molto comune di uso di funzioni anonime si ha quando si desidera definire dinamicamente un gestore di evento per widget Tkinter. Consideriamo il caso di una serie di dieci pulsanti per i quali dobbiamo gestire l'evento click. Per dieci pulsanti, dovremmo creare dieci funzioni (con tkinter le funzioni di gestione di eventi non possono avere parametri). Le funzioni anonime, create al volo con l'espressione lambda, ci consentono di superare facilmente la difficoltà:

```
class Application(object):
    def __init__(self, parent):
        self.parent=parent
        self.buttons=[]

        for i in range(10):
            b=Button(self.parent, text="pulsante %d" %i)
            b['background']='yellow'
            b['command']=self.BuildButtonAction(i)
            b.pack()
            self.buttons.append(b)

    def BuildButtonAction(self, number):
        return lambda : self.ChangeButtonColor(number)

    def ChangeButtonColor(self, number):
        self._SwitchColor(self.buttons[number])

    def _SwitchColor(self, button):
        button['background']='yellow' if button['background']=='
red' else 'red'
```

Strumenti legati alle funzioni

Il modulo `functools` mette a disposizione del programmatore Python alcuni strumenti molto utili che possono facilitare il suo lavoro.

Parametri di default con *functools.partial()*

Immaginiamo di avere una funzione con una lunga serie di parametri obbligatori o con valori di default che non fanno al caso nostro. Partial() ci consente di definire una nuova funzione che ne richiama un'altra indicando gli argomenti che ci interessa fissare, con i rispettivi valori.

In questo esempio, immaginiamo di avere una funzione `foo()` con i tre parametri *a*, *b* e *c*, e di essere soliti chiamare questa funzione con il parametro *b* impostato a 10 e con il parametro *c* a 22 (diverso dal default):

```
import functools

def foo(a, b, c=0):
    print('a=', a)
    print('b=', b)
    print('c=', c)
    return 1

customfoo=functools.partial(foo, b=10, c=22)

v=customfoo(2)
print(v)
```

Il risultato sarà:

```
a= 2
b= 10
c= 22
1
```

Ovviamente, avremmo potuto raggiungere un risultato analogo anche definendo una funzione wrapper nel modo consueto (ma si noti che così saremmo costretti a definire comunque tutti i parametri, anche quelli per i quali non vogliamo fare nessun cambiamento):

```
def customfoo(a):
    return foo(a, b=10, c=22)
```

Gestori di eventi con functools.partial() in Tkinter

La possibilità di definire funzioni in questo modo può essere molto comoda nello sviluppo di applicazioni Tkinter, quando dobbiamo definire gestori di eventi, visto che ci consente di creare le funzioni senza espressioni lambda:

```
class Application(object):
    def __init__(self, parent):
        self.parent=parent
        self.buttons=[]

        for i in range(5):
            b=Button(self.parent, text="pulsante %d" %i)
            b['background']='yellow'
            b['state']='disabled'
            b['command']=functools.partial(self.ChangeButtonColo
r, i)
            b.pack()
            self.buttons.append(b)

        self.PlayButton=Button(self.parent,
               text='Gioca',
               command=functools.partial(self.ExecuteCommand, 'p
lay')
                )
        self.PlayButton.pack()
        self.QuitButton=Button(self.parent,
               text='Esci',
               command=functools.partial(self.ExecuteCommand, 'q
uit')
```

```
                )
        self.QuitButton.pack()

    def ChangeButtonColor(self, number):
        self._SwitchColor(self.buttons[number])

    def _SwitchColor(self, button):
        button['background']='yellow' if button['background']=='
red' else 'red'

    def ExecuteCommand(self, command):
        if command=='quit':
            self.parent.destroy()
        if command=='play':
            self.StartGame()

    def StartGame(self):
        for button in self.buttons:
            button['state']='active'
```

Riduzione di una lista con *functools.reduce()*

Se si ha la necessità di estrarre dati da una lista facendone una sintesi esprimibile in un unico valore (ad esempio, la somma dei valori, la media, ecc.) può essere comodo utilizzare la funzione reduce() del modulo functools.

Immaginiamo di avere una lista contenente le età di alcune persone, e di voler calcolare l'età totale. Possiamo usare `functools.reduce` a cui passiamo una funzione da usare e la lista su cui lavorare. I due parametri della funzione da usare per la "riduzione" corrispondono rispettivamente al valore del risultato progressivo e al valore dell'elemento considerato:

```
ages=(22, 24, 21)
print('--- Calcolo somma età con functools.reduce (e funzione la
mbda) ---')
total_age=functools.reduce(lambda x, y: x+y, ages)
print(total_age)

--- Calcolo somma età con functools.reduce (e funzione lambda) -
--
67
```

Per le operazioni più comuni, il modulo `operator` ci mette a disposizione le funzioni da usare, in modo da evitare la definizione di una funzione anonima tramite espressione lambda e migliorare la leggibilità del codice:

```
print('--- Calcolo somma età con functools.reduce (e operator.ad
d) ---')
total_age=functools.reduce(operator.add, ages)
print(total_age)

--- Calcolo somma età con functools.reduce (e operator.add) ---
67
```

Nel caso di problemi un po' più complessi, è necessario ricorrere a proprie funzioni. Consideriamo il caso di un a tupla di tuple, dove per ogni persona sono presenti nome ed età:

```
people=(
    ('Alice', 22),
    ('Bob', 24),
    ('Charlie', 21)
    )
```

Vogliamo sempre sommare le età, ma non possiamo sommare le tuple tra loro, dobbiamo estrarre il secondo valore di ogni tupla.

```
print('--- Calcolo somma età da una serie di tuple con apposita
funzione ---')
def sumUp(a,b):
    # sommma al valore corrente (a) il secondo elemento della tu
pla b
    return a+b[1]

total_age=functools.reduce(sumUp, people, 0)
# si noti che in questo caso bisogna inizializzare il sommatore
# (altrimenti viene preso il primo valore, che è una tupla
print(total_age)

--- Calcolo somma età da una serie di tuple con apposita funzion
e ---
67
```

Anche in questo caso, naturalmente, sarebbe possibile usare l'operatore lambda:

```
print('--- Calcolo somma età da una serie di tuple con funzione
lambda ---')
total_age=functools.reduce(lambda x, y: x+y[1], people, 0)
print(total_age)

--- Calcolo somma età da una serie di tuple con funzione lambda
---
67
```

Come ultimo esempio, vediamo il caso del calcolo di una media ponderata, basata su valori e pesi definiti in due tuple "parallele":

```
values=(10,8,8,9,4)
weights=(1,1,2,3,2)
```

Tradizionalmente, dovremmo inizializzare a zero un paio di variabili e fare un ciclo per scorrere le tuple e sommare i prodotti e i pesi:

```
print('--- Calcolo di una media ponderata in maniera tradizional
e ---')
ps=0 # somma dei prodotti
ws=0 # somma dei pesi
for v, w in zip(values, weights):
    ps+=v*w
    ws+=w
weighted_average = ps/ws
print(weighted_average)

--- Calcolo di una media ponderata in maniera tradizionale ---
7.66666666667
```

Attraverso `functools.reduce()` possiamo fare l'intera operazione in un'unica riga di codice:

```
print('--- Calcolo di una media ponderata con functools.reduce -
--')
weighted_average = functools.reduce(
  operator.add,
  map(operator.mul, values, weights), 0
  )/functools.reduce(
    operator.add,
    weights
    )
print(weighted_average)

--- Calcolo di una media ponderata con functools.reduce ---
7.66666666667
```

Il funzionamento è analogo all'uso delle funzioni *MATR.SOMMA.PRODOTTO()* e *SOMMA()* con il foglio elettronico.

Ereditarietà e polimorfismo

Introduzione

Sulla programmazione orientata agli oggetti e i suoi concetti fondamentali (ereditarietà, polimorfismo, incapsulamento), esiste abbondante documentazione in rete, per cui non mi ci soffermerò qui.

Ereditarietà

L'ereditarietà nella programmazione orientata agli oggetti consiste nella possibilità di definire delle classi sulla base di altre classi già esistenti.

Vediamo un semplice esempio in Python. Immaginiamo di voler definire delle classi per la rappresentazione di semplici figure geometriche piane, di cui ci interessano il nome, l'area, il perimetro, ecc.

Possiamo procedere definendo una classe base e poi le classi derivate, come nel seguente esempio:

```
class Shape(object):
    '''Un oggetto di tipo Shape rappresenta una figura geometric
a generica'''

    @property
    def area(self):
        '''Restituisce l'area della figura'''
        pass

    @property
    def perimeter(self):
        '''Restituisce il perimetro della figura'''
        pass

    @property
    def name(self):
        return getattr(self, '_name', 'Untitled')
        # restituisce l'attributo _name se esiste, altrimenti 'U
ntitled'
```

La classe Shape deriva dalla classe object, la classe antenata di tutte le classi.

Se poi volessimo definire le classi Rectangle e Circle potremmo farle derivare dalla classe Shape:

```
class Rectangle(Shape):
    def __init__(self, width, height, name="A rectangle"):
        self._width=width
        self._height=height
        self._name=name

    @property
    def perimeter(self):
        return 2*(self._height+self._width)

    @property
    def area(self):
        return self._height * self._width


class Circle(Shape):
    def __init__(self, radius):
        self._radius=radius

    @property
    def perimeter(self):
        return self._radius*math.pi*2

    @property
    def area(self):
        return math.pi * self._radius ** 2
```

Nel programma principale potremo quindi creare delle istanze di Rectangle e Circle, e richiamare i metodi delle classi specifiche oppure quelli della classe base:

```
c=Circle(4)
print('Area del cerchio:', c.area)
print('Perimetro del cerchio:', c.perimeter)
print('Nome:', c.name)
print(type(c))

Area del cerchio: 50.2654824574
Perimetro del cerchio: 25.1327412287
Nome: Untitled
<class '__main__.Circle'>

r=Rectangle(10,20)
print('Area del rettangolo:', r.area)
print('Perimetro del rettangolo:', r.perimeter)
print('Nome:', r.name)
print(type(r))

Area del rettangolo: 200
Perimetro del rettangolo: 60
Nome: A rectangle
<class '__main__.Rectangle'>
```

Polimorfismo

Il polimorfismo fa sì che gli oggetti si comportino in maniera diversa a seconda della classe a cui appartengono, quando viene invocato uno specifico metodo (o quando viene richiesto l'accesso ad una proprietà).

Consideriamo il seguente esempio, in cui abbiamo una lista di figure geometriche, per tutti gli elementi della quale vogliamo visualizzare nome e area:

```
shapes=[
    Circle(5),
    Rectangle(9,10),
    Rectangle(11,21,'R1'),
    ]
for s in shapes:
    print(s.name, s.area)
```

L'output sarà, come ci aspettiamo, il seguente:

```
Untitled 78.5398163397
A rectangle 90
R1 231
```

Si noti che viene sempre richiamato il metodo o la proprietà della classe di appartenenza dell'oggetto, mai quello della classe base. In altri linguaggi, come il C++, questo comportamento, chiamato late-binding, è presente solo nel caso di definizione esplicita dei metodi come virtuali nella classe base.

Overriding

Quando si ridefinisce un metodo in una classe derivata, si parla di *overriding*. A volte può essere utile e/o necessario richiamare comunque, nell'implementazione del metodo nella classe derivata, le azioni della classe base. Si può procedere come nell'esempio seguente, in cui è stato aggiunto alla classe base il metodo place() per indicare le coordinate dell'estremo in alto a sinistra della figura geometrica. Nelle classi Rectangle e Circle si sfrutta il metodo della classe base e poi si imposta il valore del punto centrale della figura.

```
class Shape(object):
    ...
    def place(self, x, y):
        '''Imposta le coordinate dell'estremo NW della figura'''
        self._x=x
        self._y=y

    @property
    def center(self):
        '''Restituisce la tupla di coordinate del centro della f
igura'''
        return self._center

class Rectangle(Shape):
    ...
    def place(self, x, y):
        super().place(x, y)
        self._center=(self._x+self._width/2, self._y+self._heigh
t/2)

class Circle(Shape):
    ...
    def place(self, x, y):
        super().place(x, y)
        self._center=(self._x+self._radius, self._y+self._radius
)
```

Nel programma principale creeremo un'istanza e poi potremo impostarne le coordinate, visualizzando poi la tupla delle coordinate del centro:

```
c=Circle(4)
c.place(-1,12)
print('Centro del cerchio:', c.center)

Centro del cerchio: (3, 16)

r=Rectangle(10,20)
r.place(4,-2)
print('Centro del rettangolo:', r.center)

Centro del rettangolo: (9.0, 8.0)
```

Ereditarietà multipla

Introduzione

A volte capita di voler definire delle classi che ereditano alcune caratteristiche da una determinata classe ed altre da una classe diversa. Si parla in questo caso di ereditarietà multipla.

Supponiamo ad esempio di avere la nostra classe `Rectangle` come presentata nalla lezione precedente, e un'altra classe, chiamata `Logged`, che ci consente di registrare in un file su disco alcuni messaggi, con identificativo dell'oggetto e data e ora dell'evento, come nel seguente esempio

```
(176041292) 2011-02-06 15:04:52.738374: creato oggetto
(176041292) 2011-02-06 15:04:52.738412: assegnato nome a oggetto
```

ottenuto eseguendo questo codice

```
l=Logged('/tmp/llog')
l.startlogger()
l.write('creato oggetto')
l.name='Foo'
l.write('assegnato nome a oggetto')
l.stoplogger()
```

Una possibile implementazione della classe Logged è la seguente:

```
class Logged(object):
    def __init__(self, path):
        self._path=path

    def startlogger(self):
        self._file=open(self._path, 'a')

    def stoplogger(self):
        self._file.close()

    def write(self, line):
        if not line.endswith('\n'):
            line+='\n'
        output = '(%d) %s: %s' % (id(self), str(datetime.now()),
 line)
        self._file.write(output)
```

Ereditarietà multipla

A questo punto, desideriamo creare un'istanza di un oggetto che abbia tutte le caratteristiche dei rettangoli, ma anche la possibilità di sfruttare le abilità di logging della classe Logged. Per ottenere questo risultato, sarà sufficiente definire una classe che deriva da entrambe le classi a nostra disposizione:

```
class LoggedRectangle(Rectangle, Logged):
    def __init__(self, width, height, path, name="A logged recta
ngle"):
        Rectangle.__init__(self, width, height, name)
        Logged.__init__(self, path)

    def place(self, x, y):
        super().place(x, y)
        self.write('Placed in (%f, %f) position' %(x,y))
```

Come si può vedere nel codice, è stato definito un costruttore che coniuga i costruttori delle due classi di base, nel quale essi vengono richiamati in maniera esplicita. Per l'overriding del metodo `place()`, invece, si può fare

tranquillamente uso di `super()`, visto che non c'è nessun problema di ambiguità.

Quando vogliamo creare un'istanza di tipo `LoggedRectangle`, e usarne i vari metodi, non dovremo fare altro che scrivere un codice simile al seguente:

```
lr=LoggedRectangle(20, 30, '/tmp/rlog', 'MyRect')
lr.startlogger()
lr.write('Object "%s" with area %f has been created.' % (lr.name
, lr.area))
lr.place(-15,12)
lr.stoplogger()
```

Il file di log conterrà righe come le seguenti:

```
(175305164) 2011-02-06 15:15:50.284788: Object "MyRect" with are
a 600.000000 has been created.
(175305164) 2011-02-06 15:15:50.284857: Placed in (-15.000000, 1
2.000000) position.
```

Ordine di risoluzione dei metodi

Se, per ipotesi, esistesse un metodo con un determinato nome in entrambe le classi base, quale verrebbe invocato?

Immaginiamo, ad esempio, di avere definito nella classe Rectangle e nella classe Logged il metodo `setName()`, con un'implementazione diversa:

```
class Rectangle(Shape):
    ...
    def setName(self, value):
        self._rname=value

class Logged(object):
    ...
    def setName(self, value):
        self._lname=value
```

Un codice come il seguente

```
lr=LoggedRectangle(20, 30, '/tmp/rlog', 'MyRect')
lr.setName('Foo')
```

eseguirà il metodo definito in Rectangle, in Logged, o entrambi? La risposta dipende unicamente dall'ordine in cui sono specificate le classi di base quando si definisce la classe derivata. Se, come nel nostro caso, abbiamo scritto

```
class LoggedRectangle(Rectangle, Logged):
```

allora verrà eseguito esclusivamente il codice di Rectangle. Se invece avessimo scritto

```
class LoggedRectangle(Logged, Rectangle):
```

verrebbe eseguito il codice di Logged.

Possiamo avere informazioni sull'ordine in cui i metodi vengono cercati nella gerarchia delle classi visualizzando la proprietà speciale `__mro__` (*method resolution order*) della classe che ci interessa.

```
print(LoggedRectangle.__mro__)

(<class '__main__.LoggedRectangle'>, <class '__main__.Rectangle'
>, <class '__main__.Shape'>, <class '__main__.Logged'>, <class '
object'>)
```