Laboratorio di Programmazione

Con il linguaggio Python

ARTIFICIAL INTELLIGENCE
& DATA ANALYTICS

UNIVERSITÀ
DEGLI STUDI
DI TRIESTE

Stefano Alberto Russo

Corso di Laurea Triennale in Intelligenza Artificiale e Data Analytics (AIDA)

Dipartimento di Matematica, Informatica e Geoscienze
Università di Trieste

Dispense trascritte dalle lezioni grazie ai contributi cumulativi di: Gianluca Guglielmo (stesura iniziale), Michele Rispoli, Pietro Morichetti, Nicolas Solomita, Andrea Mecchina, Elena Buscaroli, Federico Pigozzi, Lucrezia Valeriani e Valentina Blasone.

Versione 1.0.1 (15 dicembre 2023)

[1] La licenza dice grossomodo che, a condizione di citare l'autore e di utilizzare la stessa tipologia di licenza, questi materiali possono essere liberamente copiati, modificati, usati per creare opere derivate e ridistribuiti.

Indice

PART

I

Strumenti di lavoro

Per affrontare il *Laboratorio di Programmazione in Python* è necessario usare alcuni strumenti, che vediamo brevemente.

File manager Il *File Manager* è quello strumento, spesso integrato nel sistema operativo, che permette all'utente di vedere, modificare, copiare ed eliminare file e cartelle.

Shell La *shell* permette di eseguire programmi e navigare il file system attraverso una *Command-Line Interface (CLI)*, ovvero di interagire direttamente con la macchina. Nei sistemi Windows le shell a disposizione sono il *Prompt dei Comandi (CMD)* e la *PowerShell*, mentre nei sistemi Unix si ha *BASH* (in genere accessibile tramite l'applicazione *Terminale*).

Editor del codice L'*Editor del Codice* può essere un qualsiasi programma per la scrittura del testo *semplice*, ovvero che non aggiunga della formattazione come il grassetto, corsivo, paragrafi etc. È possibile usare un editor che interpreti il linguaggio utilizzato per colorare le diverse sintassi e facilitarne la lettura. Per il corso è necessario impostare l'editor in modo che utilizzi 4 spazi al posto dei tab, che serviranno per indentare il codice.

Git Il software per il versionamento *Git* è uno strumento per tenere traccia delle modifiche fatte al codice, può essere sincronizzato con dei collaboratori ed è decentralizzato. Non è fondamentale per il corso, ma serve se si vuole puntare alla lode. Le *Repository* (repo) sono le cartelle dei progetti e possono contenere sia codice che file di altro tipo. L'operazione base è il *commit*, che crea un "salvataggio" del codice in Git permettendo di avere dei checkpoint, univocamente identificati da un cosiddetto *hash*. È possibile trovare qui una guida su Git scritta da Michele Rispoli.

IDE L'*Integrated Development Environment* è un editor che supporta lo sviluppatore attraverso l'integrazione con sistemi di debugging, File Mangaer, Shell, talvolta Git e svariate altre agevolazioni, le quali rendono la programmazione più scorrevole. Gli IDE più diffusi supportano diversi linguaggi e possono adattare i propri suggerimenti a seconda della sintassi utilizzata. Tra i più utilizzati troviamo Sublime, Visual Studio Code e Code::Blocks.

Replit Per semplificare lo sviluppo, durante il corso verrà usato l'IDE Cloud-based *Replit* (replit.com). Questo mette a disposizione l'acceso ad una "micro macchina virtuale" basata sul sistema operativo Linux nella Cloud, tramite un' interfaccia che si compone di tre parti principali (nel layout di default, da sinistra a destra): il file manager, l'editor del codice e la shell.

Per inizializzare l'ambiente eseguire i seguenti passi:

1. registrarsi su Replit e (opzionale) registrarsi su GitHub;
2. (opzionale) creare un nuvo repository su GitHub, impostandolo come pubblico e spuntando la casella per aggiungere un file di README;
3. creare un nuovo Repl impostando il template a **Python** e, nel caso si sia creato il repositoy su GitHub negli step sopra, importando quest'ultimo con l'apposita funzione (concedere le autorizzazioni se richiesto);
4. infine, configurare il nuovo Repl in modo da indentare il codice con 4 spazi e disabilitare l'autocompletamento tramite AI.

Dopo l'inizializzazione, modificare il file `main.py` (che sarà già automaticamente aperto) e scriverci dentro il seguente contenuto:

```
print("Hello world!")
```

Recarsi quindi sulla shell e digitare `python main.py`: questo eseguirà il vostro codice e stamperà a schermo "Hello world!".

Infine, se è stata configurata l'integrazione con Git, digitare i seguenti comandi nella shell per creare un "checkopoint", o "salvataggio":

```
git add main.py
git commit -m "Esempio hello world"
git push
```

...che aggiungerà il file `python main.py` a quelli di cui tenere traccia (`git add`), creerà un salvataggio con commento "Esempio hello world" (`git commit`) e lo inoltrerà (`git push`) su GitHub.

Pseudocodice Un ultimo strumento concettuale è costituito dal cosiddetto *pseudocodice*. Prima di scrivere il codice vero e proprio, in qualsiasi linguaggio di programmazione, è utile scriverne una bozza, anche a carta e penna, focalizzandosi non sul come, ma sul *cosa* fare. Non c'è uno standard con cui scrivere questa bozza: lo pseducodice è del tutto inventato, usando parole anche in linguaggio naturale (italiane o inglesi) che permettano di trasporre il problema che si cerca di risolvere in logica di programmazione imperativa ma senza, appunto, preoccuparsi di dettagli come la sintassi, i tipi dati etc. per non interrompere il flusso di ragionamento logico.

PART

II

Introduzione a Python

Cos'è Python *Python* è un linguaggio di programmazione interpretato di alto livello, cioè "distante" dal linguaggio macchina. Durante il corso useremo la versione 3, e in particolare ≥ 3.6. Python nasce nel 1991 come linguaggio di programmazione ad oggetti. Il suo utilizzo è in costante crescita, grazie anche alla sua semplicità e intuitività. È un linguaggio *interpretato*, quindi non ha bisogno di compilazione e può essere anche usato in modalità interattiva dall'interprete Python, molto utile ogniqualvolta si voglia testare delle cose in rapidità.

Python è anche il linguaggio "de facto" standard per la Data Science, grazie ad un enorme ecosistema di strumenti per il calcolo numerico, scientifico e statistico.

In questo corso si assume che si abbia già familiarità con i costrutti e gli operatori base di un linguaggio di programmazione, come ad esempio:

- Assegnazione di variabili e stampa a schermo (`=, print`)
- Operatori condizionali (`if, else`)
- Operatori aritmetici (`+,-,*,` etc.)
- Operatori logici (`and, or`)
- Operatori di confronto (`==, <, >,` etc.)
- Cicli (`for, while,` etc.)

Nei prossimi capitoli verrano comunque presentati questi costrutti e operatori nel linguaggio Python.

Ciao, mondo! Come ogni buona trattazione di un linguaggio di programmazione, iniziamo con un classico "Ciao mondo".

Funzione 1 (Print)**.** La funzione `print` permette di stampare una stringa a schermo. La sintassi è la seguente:

```
print('Ciao, Mondo!') # Stampa "Ciao, Mondo!"
```

È anche possibile inserire un valore personalizzato nella stringa usando la sintassi `.format`:

```
x = 25
print('Ho {} anni.'.format(x)) # Stampa 'Ho 25 anni."
```

Tipi dati In Python non è necessario definire il tipo dati delle variabili durante la dichiarazione. Infatti, Python deduce automaticamente il tipo durante l'inizializzazione grazie al valore che le viene assegnato. E' infatti un linguaggio *non tipizzato*. Al contrario dei linguaggi *tipizzati* come ad esempio il C, in Python una variabile può cambiare tipo dati in continuazione, basta assegnarle un nuovo valore di tipo diverso.

In Python sono presenti i seguenti tipi dati principali (ma ce ne sono anche altri più complessi, come liste e dizionari, che vedremo in seguito):

```
mia_var = 1       # variabile di tipo intero (int)
mia_var = 1.1     # variabile di tipo floating point (float)
mia_var = 'ciao'  # variabile di tipo stringa (str)
mia_var = True    # variabile di tipo booleano (bool)
mia_var = None    # "niente" si rappresenta con il None
```

Nel codice sopra, i cancelletti indicano un commento: ogni carattere successivo ad un cancelletto viene ignorato dall'interprete Python, fino a nuova riga.

In Python il tipo dati `None` viene usato, oltre al valore nullo, anche per semplicemente indicare qualcosa di non rilevante, come ad esempio una variabile che è necessario inizializzare per motivi sintattici ma che non è ancora stata utilizzata (p.es. `somma_totale=None`).

Il tipo dato di una variabile può essere controllato utilizzando ad esempio l'istruzione `type(var) is int`, nel caso in cui si voglia controllare che la variabile `var` abbia tipo dato intero (`int` può ovviamente essere sostituito con il tipo dato desiderato, quindi `float`, `str`, `bool`, `None`). L'istruzione ritornerà `True` se il tipo dato della variabile è quello indicato, `False` in caso contrario.

Operatori Python dispone di diversi operatori built-in per effettuare operazioni su valori e variabili.

Sintassi 1 (Operatori di confronto)**.** Gli operatori di confronto ritornano `True` o `False` a seconda che la condizione indicata sia rispettata o no. Sono usati soprattutto nei costrutti `if`.

Operatore	Nome	Esempio
==	Uguale	x==y
!=	Diverso	x!=y
>	Maggiore	x>y
<	Minore	x<y
>=	Maggiore o uguale	x>=y
<=	Minore o uguale	x<=y

Sintassi 2 (Operatori aritmetici)**.** Gli operatori aritmetici effettuano le semplici operazioni matematiche indicate dal simbolo.

Operatore	Nome	Esempio
`+`	`Addizione`	`x+y`
`-`	`Sottrazione`	`x-y`
`*`	`Moltiplicazione`	`x*y`
`/`	`Divisione`	`x/y`
`%`	`Modulo` [a]	`x%y`
`**`	`Esponenziale`	`x**y`

[a] L'operazione modulo ritorna il resto della divisione x/y.

Sintassi 3 (Operatori logici)**.** Gli operatori logici permettono di collegare due (o più) condizioni attraverso le logiche `and`, `or` e `not`.

Operatore	Descrizione	Esempio
`and`	`True se entrambe True`	`x<5 and x<10`
`or`	`True se almeno una True`	`x<5 or x<4`
`not`	`Inverte il risultato`	`not(x<5 and x<10)`

Liste La *lista* è un tipo dati built-in che permette di rappresentare una sequenza mutabile di oggetti. Gli oggetti possono essere di tipo eterogeneo, anche se questo tipo di lista non serve in quasi nessun caso. Una lista si definisce usando le parentesi quadre.

Esempio 0.1. Alcune liste in Python:

```
mia_lista = [1,2,3] # Lista di numeri
mia_lista = ['Marco', 'Irene', 'Paolo'] # Lista di stringhe
mia_lista = [15, 32, 'ambo'] # Lista eterogenea
```

Le liste ci introducono agli operatori di appartenenza, all'accesso per posizione e allo *slicing*.

Sintassi 4 (Operatori di appartenenza)**.** Gli operatori di appartenenza servono a controllare se un valore si trova in una lista, stringa o tupla[a].

Operatore	Descrizione	Esempio
`in`	`True se x è nella lista y`	`x in y`
`not in`	`Inverte risultato di in`	`x not in y`

[a] Una tupla è una lista *immutabile* definita usando le parentesi tonde.

Sintassi 5 (Accesso per posizione)**.** Posso accedere agli elementi di una lista per posizione usando la notazione con parentesi quadre:

```
mia_lista[0] # Primo elemnto
mia_lista[1] # Secondo elemento
mia_lista[-1] # Ultimo  elemento
```
Con la stessa sintassi si può accedere agli elementi delle stringhe, che si possono immaginare come una serie (lista) di caratteri.

Sintassi 6 (Slicing)**.** È possibile effettuare lo *slicing* delle liste, ovvero tagliarne una fetta, usando la seguente sintassi[a]:
```
mia_stringa[0:50]  # Dal 1° al 50° carattere
mia_stringa[30:50] # Dal 30° al 50° carattere
mia_stringa[0:-1]  # Dal 1° al penultimo carattere
```
Si può applicare lo slicing anche alle stringhe, che sono immaginabili come una serie di caratteri.

[a] In questa sintassi, l'estremo sinistro dell'intervallo è compreso, l'estremo destro è escluso.

Dizionari Il *dizionario* è un tipo dati built-in che permette di associare un valore `value` ad una chiave `key`. Si definisce usando le parentesi graffe e separando i `value` e le `key` con i due punti. Si accede al valore associato ad una chiave utilizzando le parentesi quadre.

Esempio 0.2. Nel seguente esempio, alcuni dizionari in Python:
```
mio_diz = {'Trieste': 34100, 'Padova': 35100} # Diz. di numeri
mio_diz = {'Trieste': 'TS', 'Padova': 'PD'} # Diz. di stringhe

sigla_ts = mio_diz['Trieste'] # A sigla_ts sarà assegnato il
                              # valore legato alla chiave
                              # 'Trieste' nel dizionario my_dict
```

Le chiavi devono essere tipi dati che possono essere univocamente identificati ("hashabili"). Non approfondiamo ora questo concetto, come prima approssimazione diciamo che possiamo usare come chiave tutti i tipi dati "base" non composti, ovvero interi, stringhe, floating point ma non le liste. I valori, invece, possono essere di un tipo dati qualsiasi, incluso liste e dizionari: possiamo infatti tranquillamente creare un dizionario di liste o un dizionario di altri dizionari, a patto che le chiavi siano impostate come descritto sopra. Come per le liste è possibile usare gli operatori di appartenenza per controllare se una chiave è contenuta nel dizionario. Non è possibile utilizzare gli operatori di appartenenza per i valori.

Indentazione Nella sintassi di C e C++ un blocco di codice contenente le istruzioni di un costrutto `if` o di un ciclo è delimitato da due parentesi

graffe. In Python invece, l'inizio e la fine di un blocco di codice è definito dall'*indentazione* del blocco stesso.

Indentare significa "far rientrare" alcune parti del listato di codice, in genere tramite degli spazi. È una pratica comune in tutti i linguaggi di programmazione, ma in Python è strettamente legata al funzionamento del linguaggio stesso ed è da considerarsi parte della sintassi. Di norma, vengono usati 4 spazi o un carattere `tab` (l'importante è essere coerenti in tutto il codice), ma gli spazi sono da preferirsi. Vanno indentanti anche i commenti allo stesso livello del blocco di codice cui si riferiscono, per semplificare la lettura.

Istruzioni condizionali Un'*istruzione condizionale* (il classico `if-else`) permette di inserire una condizione, che deve essere verificata per poter eseguire le istruzioni contenute nelle successive righe di codice.

Nota bene: la riga contenente l'istruzione condizionale termina con due punti, che la separano dal blocco di codice che contiene le istruzioni da eseguire nel caso in cui la condizione sia verificata e che deve essere correttamente indentato.

Esempio 0.3. Un semplice costrutto `if`:

```
if (mia_var > tua_var):
    # Indentazione per segnalare che queste
    # istruzioni fanno parte del blocco if
    print("La mia var è più grande della tua")
    if (mia_var-tua_var) <= 1:
        # Ulteriore indentazione per il secondo if
        print("...Non così tanto...")
```

Note bene: in Python è stata introdotta la parola chiave `elif` per comprimere la scrittura di `else if`.

Esempio 0.4. Un costrutto `if-elif-else`:

```
if (mia_var > tua_var):
    print("La mia var è più grande della tua")
    if (mia_var-tua_var) <= 1:
        print("...Non così tanto...")
    elif (mia_var-tua_var) <= 5:
        print("...Abbastanza")
    else:
        # L'ultima condizione ha bisogno solo dell'else, che
        # gestisce tutti i casi che non sono stati considerati
        print("...Di molto")
```

Cicli Anche in Python abbiamo i classici cicli `for` e `while`. Anche in questo caso la prima riga termina con i due punti ed è seguita da un blocco indentato, contenente le istruzioni da eseguire per ogni iterazione del ciclo.

Esempio 0.5. Un ciclo `for`:

```
for i in range(10):
    # Indentazione per identificare il blocco del for
    print(i)
```

Esempio 0.6. Un ciclo `while`:

```
i = 0
while i<10:
    # Indentazione per identificare il blocco del while
    print(i)
    i = i + 1
```

Funzione 2 (Range)**.** La funzione built-in `range`, introdotta nel `for` dell'esempio precedente, crea "al volo" una lista di numeri su cui poi si può effettuare un ciclo, nel seguente modo[a]:

```
range(10)        # Lista di numeri da 0 a 9
range(3, 10)     # Lista di numeri da 3 a 9
range(3, 10, 2) # Lista di numeri da 3 a 9
                 # con passo 2: [3, 5, 7, 9]
```

[a] Anche in questo caso si ha l'estremo destro escluso.

Esempio 0.7. Un modo più comodo per ciclare sui tipi dati iterabili (come liste, stringhe e tuple), senza usare `range`, è il seguente:

```
for elemento in mia_lista:
    # ogni elemento è preso in modo ordinato dalla lista mylist
    print(elemento)
```

Questo modo di ciclare è detto "pythonico", perché sfrutta appieno le potenzialità del linguaggio. Essere "pythonici" vuol dire proprio questo: scrivere codice che sfrutta quello che ci mette a disposizione il linguaggio evitando di fare le cose a mano quando non serve.

Funzione 3 (Enumerate)**.** La funzione built-in `enumerate` ritorna sia l'elemento che il suo indice nella lista, sotto forma di *tupla*:

```
for (i, elemento) in enumerate(mia_lista):
    print("Posizione {}: elemento {}".format(i, elemento))
```

Funzioni Le *funzioni* in Python sono dichiarate usando la parola chiave `def` e indicando eventuali paramenti di input tra parentesi. Dopo i due punti che

terminano la prima riga della definizione, si passa poi a un blocco indentato, come già visto sia per le istruzioni condizionali che per i cicli. La parola chiave `return`, alla fine del blocco di istruzioni, precede i valori da ritornare al chiamante. Non è necessario, al contrario di C e C++, indicare il tipo dati degli input, ne tanto mento quello dei valori da ritornare.

Esempio 0.8. Funzione con due argomenti e nessun ritorno. Se eseguo il codice verrò stampato a schermo "Argomenti: Pippo e Pluto":

```
def mia_funzione(argomento1, argomento2):
    print("Argomenti: {} e {}".format(argomento1, argomento2))

print(mia_funzione('Pippo', 'Pluto')) # Chiamata della funzione
```

Esempio 0.9. Funzione con un argomento e un ritorno. Se eseguo il codice verrà stampato a schermo "Risultato: 16"':

```
def eleva_al_quadrato(numero):
    return numero*numero

risultato = eleva_al_quadrato(4)
print('Risultato: {}'.format(risultato))
```

Python dispone di molte funzioni *built-in*, ovvero disponibili senza dover importare nessuna libreria. È possibile consultarne un elenco qui.

Scope Si può pensare ad uno *scope* come ad una regione di codice. In Python si stabilisce una gerarchia tra diversi scope. Una variabile, infatti, è disponibile solo all'interno dello scope in cui è stata creata e in tutti gli scope di livello più basso nella gerarchia. È importante saper prevedere quali variabili apparterranno a quali scope, in modo da non far confusione con il loro utilizzo. Python segue la regola *LEGB*, che indica la seguente gerarchia:

- Local: scope locale relativo a ciascuna classe e funzione (ogni classe e funzione ha il suo local scope). È il livello più basso nella gerarchia.
- Enclosed: scope che racchiude la regione di codice attuale, come ad esempio la funzione esterna nel caso di due funzioni annidate, o più semplicemente anche il "corpo" esterno del programma in cui sto definendo una funzione.
- Global: scope più esterno, contiene i nomi visibili in tutto il codice. È sconsigliato l'utilizzo diretto di variabili globali all'interno di funzioni, in quanto si può perdere in interpretabilità e creare bug indesiderati.
- Built-in: scope che contiene i nomi riservati ai moduli built-in di Python. È importante non sovrascrivere i nomi built-in per evitare malfunzio-

namenti nel codice [1].

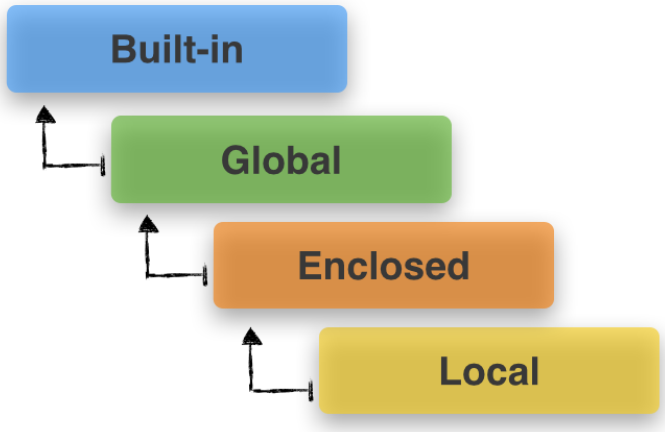

Figura 1. Struttura LEGB

È buona norma, in una funzione, utilizzare variabili locali. Si noti che un argomento in input è considerato locale dalla funzione, in quanto il suo scope è la funzione stessa a cui viene passato. Per salvare un risultato, è prassi passarlo ad una variabile dello scope superiore usando `return`, mentre è sconsigliato modificare variabili dello scope superiore direttamente dall'interno della funzione. Una funzione dovrebbe essere vista come un elemento isolato, che lavora solo sulle variabili locali e comunica con lo scope esterno solo tramite l'input (i suoi argomenti) e l'output (ciò che segue il return).

Moduli Un *modulo* è un file che funge da libreria di funzioni. Le funzioni presenti in un modulo possono essere *importate*, usando la parola chiave `import`, ossia `import modulo`, nel file principale. Di norma, per utilizzare una funzione di un determinato modulo, è necessario precedere il nome della funzione con il nome del modulo importato. Alternativamente, si può importare direttamente una singola funzione con la sintassi `from modulo import funzione`. In quest'ultimo caso, il nome della funzione non deve essere preceduto da quello del modulo. I moduli non verranno usati durante il corso.

Esempio 0.10. Due modalità diverse per usare la stessa funzione `sqrt` del modulo `math`:

```
import math
var = math.sqrt(4) # var = 2

from math import sqrt
var = sqrt(4)      # var = 2
```

[1] Ad esempio, mai chiamare una variabile `sum`, perchè in Python esiste una funzione built-in `sum` che verrrebbe in questo modo sovrascritta.

Essere pythonici Quando si scrive codice Python, in particolare se si arriva da altri linguaggi di programmazione, si potrebbe essere portati ad adottare una filosofia "classica" nella scrittura del codice.

Per esempio, per iterare su di una lista si potrebbe essere portati ad usare gli indici:

Esempio 0.11. Iterazione su di una lista usando un indice di supporto

```
mia_lista=['a','b','c']
i=0
while (i < len(mia_lista)):
    print(mia_lista[i])
    i = i + 1
```

Il codice qui sopra riportato funziona, ma non è *pythonico*. Essere Pythonici vuol dire sfruttare appieno le potenzialità del linguaggio, che porta ad una sintassi più semplice e un codice più compatto ed efficiente.

Con Python si può infatti quasi rasentare lo pseudocodice, e programmare in modo quasi discorsivo. Vediamo come possiamo iterare su di una lista in un modo più pythonico, riprendendo quanto già esposto nelle sezioni precedenti:

Esempio 0.12. Iterazione su di una lista in modo pythonico

```
mia_lista=['a','b','c']
for elemento in mia_lista:
    print(elemento)
```

Ci sono molti trucchetti per scrivere codice in modalità più pythonica, e in generale laddove ci si ritrova ad usare degli indici o del codice poco facilmente comprensibile, questo è segno che c'è qualcosa di ampiamente migliorabile.

È una capacità che si fa propria con gli anni, ma che è bene inquadrare fin da subito, poichè facilita notevolmente nella programmazione con Python. Su https://coady.github.io si può trovare un blog con vari spunti su come essere più pythonici (in inglese).

File e database I file sono sequenze di bit dove vengono salvate informazioni. Possono essere di due tipologie primarie, ovvero di testo semplice (*plain text*) oppure binari. Un codice sorgente in C, così come un listato Python, è un testo semplice, mentre un codice sorgente C compilato in formato macchina è un file binario. I file di testo semplice si possono editare sia nella Shell tramite interfacce testuali, che con programmi ad interfaccia grafica come ad esempio un generico "blocco note" (Notepad sui sistemi Windows e TextEdit sui sistemi Apple). I file di testo semplice sono in realtà nati per immagazzinare documenti di testo, e agli inizi dell'infromatica erano l'unico formato supportato per il testo (da qui, l'estensione `.txt`). Successivamente sono poi stati introdotti formati di testo che supportavano la formattazione, come il grassetto, il corsivo, i paragrafi, le tabelle etc. Questi possono essere sempre dei file di testo semplice arricchiti con dei "tag" secondo delle logiche chiamate di Markup (l'HTML ricade in questa categoria) oppure in formato binario, come i file di Microsoft Word (`.doc`).

Per salvare dei dati strutturati, come ad esempio dei dati temporali e numerici, si potrebbe semplicemente far ricorso a dei documenti di testo, indifferentemente se di testo semplice o più sofisticati. Potremmo per esempio salvare in una tabella di un documento Word i valori di una determinata azione in borsa giorno per giorno, oppure in un file di testo semplice annotare il prezzo e la data e andare a capo per ogni settimana.

Consultare dei dati in questa modalità sarebbe però estremamente scomodo, se non per quantità veramente limitate. Per semplificare il salvataggio e il successivo recupero di dati strutturati sono stati allora inventate le basi di dati (*Database*, in inglese). I database forniscono un modo estremamente completo per salvare dati e poi consultarli, e sono ad oggi disponibili in varie tipologie, a seconda del contesto in cui vengono utilizzati. La creazione e il mantenimento di un database hanno però bisogno di un attento lavoro di design e gestione delle risorse, che può risultare eccessivo per compiti semplici.

Il formato CSV Un *file* in formato *CSV*, ovvero "Comma-Separated Values", è un buon compromesso tra l'utilizzo di un database vero e proprio e il salvataggio dei dati in un documento di testo. In un CSV, ogni riga è un'entrata ("entry" in inglese) con valori separati da una virgola, che formano le colonne. È possibile inserire un'intestazione (opzionale) per dare un nome alle colonne. L'estensione di questi file è `.csv`, o anche semplicemente `.txt` in quanto sono a tutti gli effetti documenti di testo semplice. Durante il corso verrà usato il file `shampoo_sales.csv`, che riporta le vendite di shampoo per un periodo di tre anni di cui qui sotto si riporta una anteprima e il cui andamento è mostrato graficamente in Figura 2.

```
Date,Sales
01-01-2012,266.0
```

```
01-02-2012,145.9
01-03-2012,183.1
01-04-2012,119.3
...
```

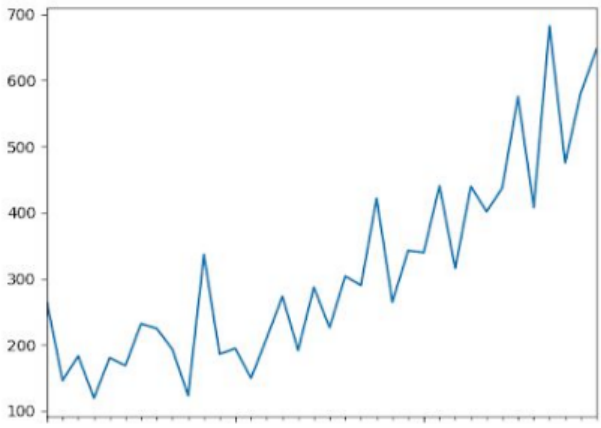

Figura 2. Andamento dei valori contenuti nel file `shampoo_sales.csv`

I files in Python Python mette a disposizione varie modalità per interagire con i files, tutte disponibili in seguito ad aver aperto il file tramite la funzione built-in *open()*, che apre il file passato come argomento [1].

[1]Se il file non si trova nella stessa directory del codice, si deve indicare il percorso da seguire per trovarlo, relativamente alla directory in cui ci si trova o assoluto.

Funzione 4 (`open`)**.** Per aprire un file in Python si usa la funzione built-in `open()`, che ritorna un entità di tipo file. Gli argomenti da passare alla funzione sono il nome del file e una stringa per indicare se si vuole aprire il file in modalità di lettura, scrittura, aggiunta o combinazioni di queste. In particolare:

- `'r'` indica modalità *read*;
- `'w'` indica modalità *write*, si sovrascrive file;
- `'rw'` indica modalità mista *read* e *write*;
- `'a'` indica modalità *append*, in cui aggiungo alla fine del file.

Funzione 5 (`file.close`)**.** Per chiudere un file si usa la funzione dell'entità file `close()`. È necessario chiudere un file per liberare memoria e per salvare alcuni cambiamenti che potrebbero essere ancora non stati scritti sul disco.

Funzione 6 (`file.read`)**.** La funzione `read()` legge l'intero file e ne ritorna il contenuto.

Funzione 7 (`file.readline`)**.** La funzione `readline()` legge una riga per volta e ne torna il contenuto. Chiamando `readline()` più volte sulla stessa entità file si andrà automaticamente alle righe successive.

Esempio 0.13. Il seguente snippet di codice apre il file in modalità lettura, ne stampa i primi 50 caratteri usando lo *slicing* delle stringhe e lo chiude:

```
mio_file = open('shampoo_sales.csv', 'r')
print(mio_file.read()[0:50])
mio_file.close()
```

Esempio 0.14. Il seguente snippet invece, apre il file in modalità lettura, legge e stampa le prime 5 righe e lo chiude:

```
mio_file = open('shampoo_sales.csv', 'r')
for i in range(5):
    print(mio_file.readline())
mio_file.close()
```

Funzione 8 (`file.write`)**.** Per scrivere su file, dopo averlo aperto in modalità scrittura ('w'), si usa la funzione dell'entità file `write()`.

Per leggere tutte le righe di un file, si può sia caricarne tutto il contenuto in memoria con la funzione `read()` e poi trovare un modo di processarlo riga per riga, o in alternativa si può usare la funzione `readline()` ripetutamente, finché il valore tornato non sarà `None`, a significare che non ci sono più nuove righe. Tuttavia, il modo pythonico di leggere un file riga per riga è più semplice, e va a sfruttare l'iterabilità dell'entità file, come esposto nell'esempio seguente:

Esempio 0.15. Un file letto riga per riga in modo pythonico

```
mio_file = open('shampoo_sales.csv', 'r')
for linea in mio_file:
    print(linea)
mio_file.close()
```

Si noti che Python "sa" che l'unità dell'iterazione sono le righe: riconosce il carattere che indica l'andare a capo e lo usa come criterio per passare ad una nuova iterazione.

Un'altra pratica pythonica è di aprire un file non in maniera a sè stante come fatto sopra, ma all'interno di un cosiddetto *contesto di esecuzione*, che si apre tramite la funzione built-in `with()`. Questo è un blocco di codice che fa delle cose al posto nostro per aiutarci e in particolare laddove lo usiamo per interagire con un file, chiude lui per noi il file. In pratica, non dovremo chiamare la chiusura del file a mano: lo farà già il contesto di esecuzione per noi. Possiamo sempre usare questo "aiutante" per leggere i file.

Esempio 0.16. Un file letto all'interno di un contesto di esecuzione, riga per riga in modo pythonico

```
with open('shampoo_sales.csv', 'r') as mio_file:
    for linea in mio_file:
        print(linea)
```

Leggere un file CSV Python mette a disposizione un modulo per la lettura di file CSV, ma a fini didattici scriveremo noi il codice necessario. Per prima cosa dobbiamo introdurre alcune nuove funzioni, e in particolare la funzione `split()` dell'entità stringa.

Funzione 9 (`str.split`)**.** La funzione `split()` dell'entità stringa divide la stringa quando incontra il carattere indicato come argomento tra parentesi e ritorna una lista contenente i vari "pezzi".

Esempio 0.17. Un esempio di come dividere una stringa in base ad uno spazio come divisore dei pezzi. La variabile `pezzi` è una lista.

```
mia_stringa = 'Ciao mondo'
pezzi = mia_stringa.split(' ') # pezzi = ['Ciao', 'mondo']
```

Divideremo quindi ogni riga del file in base al separatore virgola. quello che viene letto da un file CSV (che ricordiamo è salvato come testo semplice) sono sempre stringhe, anche se rappresentano informazioni numeriche. Per farne uso sarà quindi necessaria una conversione, che ci introduce al concetto di *casting* di tipi dati.

Sintassi 7 (Casting)**.** Per convertire valori tra diversi tipi dati si usa la sintassi di casting, in cui si usa direttamente il tipo dati di destinazione, passandoci il valore (o variabile) da convertire tra parentesi:

```
float('9.1') # Da stringa numerica a floating point: 9.1
int(9.1)     # Da floating point a intero: 9
float(9)     # Da intero a foating point: 9.0
str(9)       # Da intero a stringa: '9'
float('hey') # Da stringa testuale a floating point: errore
float([])    # Da lista a floating point: errore
```

Se il tipo dati di destinazione non può rappresentare il valore originale, questo può essere troncato perdendo informazione (come per la conversione da floating point a intero), oppure può venire modificato anche aggiungendone di nuova (come per la conversione da intero a floating point, in cui si aggiunge una cifra decimale, posta a zero). Se invece il tipo dati di destinazione non può in nessun modo rappresentare l'originale (come ad esempio cercando di convertire una lista a floating point), allora verrà generato un errore.

Una volta divisi i valori di ogni linea del file in corrispondenza della virgola e convertiti quelli numerici a intero o a floating point, dobbiamo aggiungerli ad una struttura dati adatta per immagazzinarli tutti. Per semplicità scegliamo la lista, in cui ogni elemento sarà a sua volta una lista con un elemento per ogni valore trovato nella riga del file CSV: avremo quindi una *lista di liste*.

Un primo approccio, spesso seguito in C, sarebbe quello di pre-allocare le liste in base a quante righe ed elementi ha il file. Questo comporta dover sapere a priori quanti elementi saranno presenti, e ciò non è sempre possibile, nonché scomodo e poco flessibile. Possiamo invece aggiungere dati ad una lista già esistente, compresa la lista vuota, man mano che li processiamo, utilizzando la funzionalità di `append` delle liste.

Funzione 10 (`list.append`)**.** La funzione `append()` dell'entità lista aggiunge un elemento alla fine della lista, modificandone la lunghezza:

```
mia_lista = []
mia_lista.append(1) # mia_lista = [1]
mia_lista.append(2) # mia_lista = [1, 2]
```

Abbiamo ora tutti gli strumenti per leggere informazioni da un file CSV e tornarli sotto forma di lista, vediamo come:

Esempio 0.18. Nel seguente esempio, si legge da `shampoo_sales.csv` e si popola la lista `dati` che ne conterrà le informazioni:

```
# Inizializzo una lista vuota per salvare i dati
dati = []
# Apro e leggo il file, linea per linea
with open( 'shampoo_sales.csv' , 'r' ) as mio_file:
    for linea in mio_file:
        # Faccio lo split di ogni riga sulla virgola
        elementi = linea.split(',')
        # Se NON sto processando l'intestazione...
        if elementi[0] != 'Date':
            # Setto le variabili data e il valore,
            # convertendo il valore a floating point
            data = elementi[0]
            valore = float(elementi[1])
            # Aggiungo alla lista dei dati
            dati.append([data,valore])
```

PART

III

Gli oggetti

La *Programmazione orientata agli Oggetti* (in inglese *Object-Oriented Programming, OOP*), o semplicemente *Programmazione a oggetti* è un paradigma di programmazione che permette di ragionare in termini di entità e non più solo di funzioni con input/output. Queste entità prendono il nome di *oggetti* e hanno due grossi vantaggi:

1. possono mantenere uno stato, e
2. sono organizzabili gerarchicamente.

Oggetti e Classi Gli oggetti si definiscono tramite le *classi*. Una classe è una generica definizione di un oggetto, ovvero uno schema che permette di descrivere un'entità con le caratteristiche desiderate. Ad esempio, una persona con un nome e un cognome. Una specifica entità è invece chiamata *istanza*, come ad esempio la persona specifica Mario Rossi. L'operazione di passare da classe ad istanza di oggetto si chiama *istanziazione*. Il concetto di oggetto è generico e racchiude sia classi che istanze, è un po' una parola "jolly" per esprimere un concetto o l'altro a seconda dei contesti. In genere che si parli di classe o istanza è infatti abbastanza evidente, ma nei casi dubbi è sempre meglio specificare a cosa ci si sta riferendo, sia nei commenti del codice che nel descriverlo o discuterlo. Non c'è un limite a quante istanze di classe si possono creare, e possono essere create anche dinamicamente e salvate in strutture di supporto come ad esempio le liste.

Le funzioni contenute in una classe prendono il nome di *metodi*, mentre le variabili prendono il nome di *attributi*. Sono sempre funzioni e variabili, ma in questo contesto cambiano nome. Non è un errore parlare di funzioni e variabili nel contesto della programmazione ad oggetti, ma è piuttosto una imprecisione.

Le classi permettono quindi di definire in modo astratto ma intuitivo delle entità con caratteristiche ben definite. Ad esempio, una classe `Persona` può avere un attributo `nome` e un metodo `saluta()`. Se si volesse salvare il nome di una persona senza usare l'attributo di una classe lo si dovrebbe salvare in una struttura a parte, come una lista o un dizionario. Questo può risultare scomodo, in quanto è necessario tenere a mente quali strutture hanno quale compito. In più, gli indici per accedere ai vari elementi nelle liste potrebbero velocemente diventare troppi. Quindi, le classi offrono una soluzione semplice per salvare in modo compatto le informazioni legate ad un'entità logica, come una persona.

Python e gli oggetti In Python **tutto**, tranne le parole chiave, **è un oggetto**: numeri, stringhe, liste, persino le funzioni e, ovviamente, gli oggetti stessi. Per esempio, fino ad ora abbiamo parlato di "entità stringa", e della funzione `split()` dell'entità. In realtà la stringa è un oggetto, e `split()` è un suo metodo (che ne fa la divisione in pezzi in base al separatore specificato come argomento). Similmente, abbiamo parlato di "entità file", ma in realtà quello che ci ritorna la funzione built-in `open()` è proprio un oggetto, ti tipo `file`. E quando, per leggere un file, abbiamo invocato la "funzione `read()`", abbiamo in realtà usato un metodo dell'oggetto file (che ci torna il contenuto del file rappresentato dal suo oggetto).

Per convincerci che in Python è tutto un oggetto e controllarne il tipo, possiamo usare le funzioni built-in `dir()` e `type()`.

Funzione 11 (`dir`)**.** La funzione `dir()` ritorna una lista con tutti gli attributi e i metodi di un oggetto

Esempio 0.19. La funzione `dir()` applicata ad una stringa. Si può vedere listato, tra gli altri, anche il metodo `split()`.

```
>>> dir('ciao')
['__add__', '__class__', '__contains__', '__delattr__',
... 'split', 'splitlines', 'startswith', 'strip', ...
'swapcase', 'title', 'translate', 'upper', 'zfill']
```

Funzione 12 (`type`)**.** La funzione `type()` ritorna la classe dell'oggetto. Poiché in Python tutto è un oggetto, la stessa definizione di "tipo" passa per il concetto di classe.

Esempio 0.20. La funzione `type()` applicata ad un numero, ad una stringa, e ad una funzione.

```
>>> type(3.14)
<class float>
>>> type('ciao')
<class str>
>>> type(print)
<class builtin_function_or_method>
```

Come già anticipato sopra, tutto in Python, tranne le parole chiave, è un'istanza di una classe ovvero un oggetto, ed è sempre possibile esplorarne le caratteristiche con le funzioni appena presentate.

Quando per esempio si converte da un tipo dati ad un altro, con una sintassi del tipo `float('3.14')`, si sta creando una nuova istanza dell'oggetto `float` a partire da una stringa (dal contenuto 3.14).

Per controllare se una variabile è istanza di un determinato tipo di classe (ovvero se un oggetto è di un certo tipo), si può usare la funziona built-in *isinstance()*:

Funzione 13 (isinstance)**.** La funzione `isinstance()` verifica se il suo primo argomento è istanza della classe specificata dal secondo argomento.

```
print(isinstance(variabile, MiaClasse))
# True se variabile è istanza di MiaClasse, False altrimenti
```

Operazioni in-place Quando di chiama un metodo di un oggetto, a differenza delle funzioni a sé stanti, questo può agire direttamente sull'oggetto senza tornare niente. Non è una pratica consigliata, in quanto dovrebbe sempre valere il paradigma `input -> computazione -> output`, ma è una pratica accettata laddove ci siano ragioni specifiche, come per esempio di performance. Se vogliamo ad esempio rendere una stringa tutta in maiuscolo, possiamo assumere che le stringhe non saranno mai poi così lunghe da avere problemi di memoria nel creare una copia della stringa e tornare quella. Invece, nell'operazione di inversione di una lista, questa può essere anche molto lunga: ed ecco allora che anziché crearne una copia, invertirla e tornare la copia, invertiamo direttamente la lista originale. In realtà la motivazione in questo caso è più algoritmica, ma è più semplice da pensare alla questione come appena esposto.

Nel caso un metodo modifichi direttamente un oggetto non tornando niente (`None`), questa operazione prende il nome di *in-place*, ovvero che viene eseguita sul posto. Se invece il metodo non modifica lo stato dell'oggetto allora non parlerò di operazione in-place. Bisogna stare attenti con le operazioni in-place perché possono portare a non accorgersi di star cambiando gli oggetti originali, causando bug e mal di testa.

Esempio 0.21. Un'operazione in-place e una non in-place:

```
mia_lista = [1,2,3]
print(mia_lista.reverse()) # Stampa "[3,2,1]"
print(mia_lista)           # Stampa "[3,2,1]",
                           # mia_lista è stata modificata


mia_stringa = 'ciao mondo'
print(mia_stringa.upper()(',')) # Stampa "CIAO MONDO".
print(mia_stringa)              # Stampa "ciao mondo",
                                # mia_stringa è rimasta invariata.
```

Un'ultima questione importante nell'approcciare la programmazione ad oggetti in Python è la notazione, in termini di formato, da usare nel codice. Di norma, si usa il *camelcase* (es. `MyClass`) per i nomi delle classi e lo *snake-*

case (es. `my_method`) per i nomi dei metodi e delle variabili. Ci sono alcune eccezioni a questa convenzione, come ad esempio tutti i tipi dati built-in (parliamo di `float` e non di `Float`) e alcuni moduli storici (come quello di logging). Negli altri casi, è sempre buona norma seguire questa convenzione poiché altrimenti si perderebbe di interpretabilità del codice.

Definire una classe Per definire una classe in Python si usa la parola chiave `class`. L'*inizializzatore*, ovvero il metodo usato per inizializzare gli attributi di una classe, si definisce usando `__init__`. Di norma, i metodi con il doppio underscore seguono una convenzione secondo cui sono di uso tecnico interno, privato, e non dovrebbero essere invocati esplicitamente. Per accedere agli attributi (o i metodi) all'interno della classe, ovvero nel suo corpo di codice, si usa la parola chiave `self`, che rappresenta l'istanza, seguita dal nome dell'attributo (o metodo) cui si vuole accedere, separati da un punto. Nella definizione dei metodo delle classi, usare `self` come primo argomento è obbligatorio, mentre nella loro chiamata va omesso.

Esempio 0.22. Di seguito, la dichiarazione della classe `Persona`:

```
class Persona:
    def __init__(self, nome, cognome):
        self.nome = nome
        self.cognome = cognome
    def saluta(self):
        print("Ciao, sono {} {}".format(self.nome, self.cognome))
```

Per creare un'istanza della classe, ossia un oggetto della classe, si invoca il nome della classe stessa, passando gli eventuali argomenti per l'inizializzazione. Python creerà uno spazio dedicato in memoria per ospitare l'istanza della classe, a cui farà puntare `self`, e chiamerà l'inizializzatore della classe definito con `__init__`.

Per accedere agli attributi (o i metodi) dell'istanza della classe, si usa il nome assegnato alla variabile che "contiene" l'istanza della classe, seguita da un punto e dal nome dell'attributo (o metodo) cui si vuole accedere.

Esempio 0.23. Istanziazione di una classe e accesso a metodi e attributi dall'esterno. Le variabile `persona1` sarà un'istanza della classe `Persona`, così come `persona2`.

```
persona1 = Persona('Mario', 'Rossi')
persona2 = Persona('Irene', 'Bianchi')
print(persona1.nome) # 'Mario'
print(persona2.nome) # 'Irene'
persona1.saluta()    # 'Ciao, sono Mario Rossi'
persona2.saluta()    # 'Ciao, sono Irene Bianchi'
```

Esempio 0.24. Utilizzo di oggetti come componenti di una lista

```
persone = [ Persona('Mario', 'Rossi'),
            Persona('Irene', 'Bianchi')]
print(persone[0].nome) # Mario
persone[1].saluta()    # 'Ciao, sono Irene Bianchi'
```

Estendere oggetti Come accennato, oltre a permetterci di mantenere uno stato (come il valore degli attributi nome e persona nella sezione precedente), gli oggetti sono anche molto comodi per rappresentare gerarchie.

Il concetto di *ereditarietà* permette, a partire da una classe *genitore*, di creare classi *figlie*, che ereditano tutti gli attributi e metodi della classe genitore. Le classi figlie possono sovrascrivere i metodi ereditati per personalizzarli e possono aggiungerne di nuovi per estenderne le funzionalità. Le modifiche effettuate ad una classe figlia non impattano la classe genitore. L'operazione di estensione di una classe prende anche il nome di `specializzazione`, nel senso che la si rende più "specifica" per una determinata entità che si vuole rappresentare (vedi esempio dopo sui bicchieri).

Possiamo interpretare le gerarchie tra classi in termini di insiemi e relazioni di inclusione, come mostrato in Figura 3.

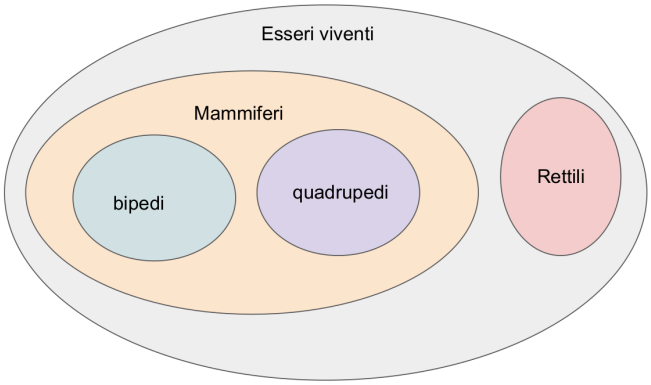

Figura 3. Esempio di gerarchia

Non esiste un metodo "corretto" di rappresentare le gerarchie, dipende dal contesto. Pensiamo per esempio ai bicchieri: si potrebbe creare una gerarchia tra classi dove una generica classe `Bicchiere` viene estesa da due sottoclassi `BicchiereAlcolico` e `BicchiereAnalcolico`, oppure da `BicchiereVetro` e `BicchierePlastica`. La prima è probabilmente più adatta nel contesto della ristorazione, la seconda nella logistica e nei magazzini.

Una classe figlia si definisce mettendo tra parentesi il nome della classe genitore nella definizione della classe, come si vede nel seguente esempio.

Esempio 0.25. Estensione dell'oggetto Persona declinandolo in Studente con un saluto diverso. Tutti i metodi e gli attributi definiti nella classe Persona sono automaticamente ereditati, incluso ___init___, mentre `saluta()` viene sovrascritto.

```
class Studente(Persona):
    # Sovrascrivo il metodo saluta()
    def saluta(self):
        print("Ciao, sono {} e sono uno studente.".format(self.nome))
```

Quando si va ad estendere classi e sovrascrivere metodi, si può sempre tornare ad accedere ai metodi definiti originariamente nella classe genitore tramite una dicitura particolare, ovvero `super()`, che sta per "superclasse", la classe genitore. Questo può essere comodo per riutilizzare metodi sovrascritti, sia così come stanno che richiamandoli dal novo metodo che li sta sovrascrivendo (ad esempio per elaborarne l'output).

Funzione 14 (`super`)**.** La funzione built-in `super()`, quando usata all'interno del corpo di una classe, permettere di accedere alla classe genitore e conseguentemente a tutti i suoi metodi.

Esempio 0.26. Uso del metodo `super()` per recuperare il metodo della classe genitore:

```
class Professore(Persona):
    def __str__(self):
        return 'Prof. "{} {}"'.format(self.nome, self.cognome)'

    # Si sovrascrive il metodo "saluta"
    def saluta(self):
        print ('Ciao, sono il Prof. {} {}'.format(self.nome, self.cognome))

    # Si ri-accede al metodo "saluta" originale usando super()
    def saluta_informale(self):
        super().saluta()
```

Per controllare se una classe è sottoclasse di un'altra, ovvero se l'ha estesa ereditandone metodi e attributi, si può usare la funzione built-in *(*issubclass()):

Funzione 15 (issubclass)**.** La funzione `issubclass()` verifica se il suo primo argomento è sottoclasse del secondo, cioè se é una classe figlia.

```
print(issubclass(MiaClasse1, MiaClasse2))
# True se MiaClasse1 è sottoclasse, cioè classe figlia,
# di MiaClasse2, False altrimenti.
```

PART

IV

Gestione degli errori

Le eccezioni In Python, una qualsiasi interruzione del normale flusso del codice è chiamata *eccezione*. Queste interruzioni sono quasi sempre errori, ma non è sempre così. Per esempio, internamente a Python, per segnalare la fine di un ciclo si usa una eccezione che rappresenta la fine del ciclo. Detto questo, possiamo in prima approssimazione pensare alle eccezioni come agli errori, perché è in effetti quasi sempre così.

Le eccezioni, come tutto in Python, sono oggetti, e in particolare estendono sempre la classe base. *Exception*. Vediamone alcuni esempi:

- `Exception`:
  - `ArithmeticError`: problemi nel calcolo numerico;
    - `FloatingPointError`: problemi nel calcolo numerico
    - `ZeroDivisionError`: divisione per zero
  - `AttributeError`: l'attributo dell'oggetto non esiste
  - `NameError`: variabile non definita
  - `SyntaxError`: il codice non è semanticamente corretto
  - `TypeError`: tipo dati inappropriato
  - `ValueError`: valore inappropriato

Quando si verifica un'eccezione, il comportamento di default è di interrompere l'esecuzione del codice dove è avvenuta, e di stamparla a schermo assieme al suo *traceback*. Questo è un "report" automaticamente generato da Python che serve ad identificare dove è avvenuta l'eccezione. Quello che fa è proprio di *rintracciare all'indietro* le chiamate delle funzioni che hanno portato al suo verificarsi.

```
> python lezione5.py
Traceback (most recent call last):
  File "lezione5.py", line 3, in <module>
    float(my_var)
ValueError: could not convert string to float: 'Ciao'
```

Figura 4. Una eccezione con il suo traceback.

Errori recoverable e non Un errore *recoverable* non causa un crash del programma ma può portare ad una perdita di dati.

Nel caso di input non corretto, infatti, si hanno due opzioni:

- stampare l'errore, usare un valore di default e continuare con l'esecuzione del programma (errore recoverable);
- stampare l'errore o un'eccezione personalizzata e uscire dal programma (errore non recoverable).

Riprendendo l'esempio della stampa delle prime `n` righe di un file, si può pensare che un utente passi come valore `3.5`. A tutti gli effetti questo è un errore, in quanto ci si aspettava un int, ma lo si può *sanitizzare* usando il casting. Invece, nel caso in cui venisse passata la stringa `'a'`, allora l'errore sarebbe non recoverable se non si disponesse di un valore di default da sostituire e il programma non potrebbe continuare.

Si noti che l'uscita dal programma si effettua solo in caso di input proveniente dall'esterno, in quanto sarebbe sintomo di design errato uscire da un programma i cui valori vengono passati dal programma stesso.

Il costrutto try-except Il costrutto *try-except* serve a gestire le eccezioni che possono insorgere all'interno del suo corpo. Si può così reagire ad un errore, in vari modi. In alcuni casi si riesce a "recuperare": se si deve leggere un file con dei parametri ottimizzati per una simulazione, si può probabilmente comunque proseguire con i parametri di default. In altri casi, si può segnalare l'errore ma proseguire con l'esecuzione del programma senza farlo terminare brutalmente: pensiamo per esempio alle interfacce grafiche o agli applicativi Web. In altri casi ancora, non si può fare niente e allora si lascia che l'eccezione "salga" a Python, senza gestirla, che interromperà l'esecuzione del programma e la stamperà a schermo con il suo traceback, come nell'immagine precedente.

Più in dettaglio, il costrutto `try-except` prova ad eseguire il blocco di istruzioni indentato che segue la parola chiave `try`, e:

- se non si presentano eccezioni, salta alla prima istruzione successiva al costrutto stesso (cioè dopo il blocco di codice dell' `except`);
- se invece si presenta un'eccezione, il costrutto esegue il blocco di codice dell' `except`.

Esempio 0.27. Esempio di utilizzo del costrutto `try-except`.

```
mia_var = 'ciao'
try:
    mia_var = float(mia_var)
except:
    print('Non posso convertire "mia_var" a numerio!)
```

```
    print('Uso il valore di default di "0.0"')
    mia_var = 0.0
print(mia_var) # Stampa a schermo 0.0
```

Nell'esempio appena visto l'eccezione `ValueError`, che si presenta quando si prova a convertire una stringa alfabetica a float, viene catturata e gestita dall' `except`. Si noti che non è stato specificato il tipo di eccezione da gestire, quindi Python è pronto a gestirne una qualsiasi. Possiamo però gestire solo specifiche eccezioni, specificandone il tipo nell' `except`.

In questo modo il costrutto `try-except` passerà in rassegna tutti i tipi di eccezioni specificate dalle varie `except`, e verificherà se corrisponde ad una di esse (la prima che trova):

- se corrisponde, eseguirà il relativo blocco di codice, senza bloccare l'esecuzione del programma;
- se invece non corrisponde a nessuna di esse, Python bloccherà l'esecuzione del programma e riporterà l'eccezione a schermo con il suo traceback, come se non ci fosse stato nessun costrutto `try-ecept`.

Si può inoltre "nominare" un'eccezione gestita in un `except` usando la parola chiave *as*: questo la rende disponibile nel blocco, ed essendo un'eccezione un oggetto come un altro, può essere manipolata, stampata a schermo o altro, come ad esempio essere trascritta su un file di log.

Esempio 0.28. Nel seguente snippet, si chiede a Python di gestire l'errore in modo diverso a seconda dell'eccezione, e di gestire un'eccezione non coperta dai casi previsti stampandola a schermo.

```
try:
    mia_var = float(mia_var)
except ValueError:
    print('Non posso convertire "mia_var" a valore numerico!')
    print('Errore di VALORE, "mia_var" valeva "{}".'.format(mia_var))
except TypeError:
    print('Non posso convertire "mia_var" a valore numerico!')
    print('Errore di TIPO, "mia_var" era "{}".'.format(type(mia_var)))
except Exception as error:
    print('Non posso convertire "mia_var" a valore numerico!')
    print('Errore generico: "{}".'.format(error))
```

L'ultima clausola, effettuando il controllo sulla classe base `Exception`, andrà a gestire tutte le eccezioni, al pari dell' `except` semplice, ma l'utilizzato di questo costrutto ci ha permesso di dare un nome all'eccezione (variabile `error`) per poi stamparla a schermo.

Si possono usare anche due ulteriori funzionalità del costrutto `try-except`,

che sono rispettivamente le parole chiave *else* e *finally*. Il blocco di codice che segue l' `else` viene eseguito nel caso non ci sia stata nessuna eccezione, mentre quello che segue il `finally` viene eseguito sempre, anche in caso di errore, prima di interrompere l'esecuzione del programma. Questo è molto utile per "far pulizia", come ad esempio chiudere i file aperti. In generale, il costrutto nella sua interezza è da intendersi così:

- Prova (try):
  - fai questa cosa
- Se c'è questo tipo di errore (primo except):
  - fai quest'altra cosa
- Se c'è quest'altro tipo di errore (secondo except):
  - fai quest'altra cosa ancora
- Altrimenti, se non ci sono errori (else):
  - fai questo
- E in ogni caso (finally):
  - fai quest'altro

Un esempio pratico potrebbe per esempio essere la lettura di un paramento da un file, come esposto nell'esempio sottostante.

Esempio 0.29. Lettura di un parametro da file con gestione della maggior parte delle casistiche di errore.

```
parametro = 0
nome_file_parametro = 'parametro.txt'
try:
    file_parametro = open(nome_file_parametro)
    parametro_come_stringa = file_parametro.read()
    parametro_come_float = float(parametro_come_stringa)
except IOError:
    print('Non posso leggere il file!')
    print('Userò il vaore di default per il parametro')
except ValueError:
    print('Non posso convertire il parametro a valore numerico!')
    print('Errore di valore, il parametro valeva "{}".'.format(parametro_come_stringa))
    print('Userò il valore di default per il parametro')
except Exception as error:
    print('Errore generico: "{}".'.format(error))
    print('Userò il valore di default per il parametro')
else:
```

```
    parametro = parametro_come_float
finally:
    file_parametro.close()
```

Da notare che questo costrutto `try-except` gestisce eccezioni multiple e diverse anche nella loro natura, poiché potrebbero insorgere sia dall'apertura del file che dalla conversione del parametro a float. Si sarebbero potuti scrivere due costrutti `try-except` distinti nel caso fosse servito essere più granulari.

Da notare anche che si sarebbero potuti *annidare* i costrutti: per prima cosa si poteva provare ad aprire il file e poi si poteva scrivere un ulteriore `try-excpt` annidato, da eseguire solo nel caso in cui l'apertura del file fosse avvenuta senza errori, per provare a convertire il parametro in float.

Generare eccezioni Per generare un'eccezione si usa la parola chiave *raise*, seguita da una istanza dell'eccezione. Si potrebbe voler generare (o, *alzare*) un'eccezione per svariati motivi, come ad esempio quando un valore di input non è tra quelli ammessi, o perché si sono verificate delle condizioni particolari che non sono gestite nel codice.

L'argomento da passare all'eccezione quando viene istanziata è il "messaggio che porta con sé', ovvero quello che verrà stampato a schermo nel caso in cui nel codice questa eccezione non venga gestita da nessun costrutto `try-except`.

Si potrebbe anche voler definire un'eccezione personalizzata. In tal caso basterebbe estendere la classe base `Exception` con un corpo vuoto (cioè usando l'istruzione nulla, che in Python si indica con *pass*).

Esempio 0.30. Definizione e generazione di un'eccezione personalizzata.

```
class InvalidParameter(Exception):
    pass

parametro = -5
if parametro < 0:
    raise  InvalidParameter('Il parametro è minore di zero')
```

PART

V

Controllo degli input

Un *input* è un qualsiasi dato che "entra da qualche parte". Una funzione riceve degli argomenti in input, la lettura di un file porta dei dati come input al programma e anche la scrittura dell'utente su linea di comando è un tipo di input. Un input può essere sia già presente nel codice che passato dall'esterno.

Non fidarsi degli input Non ci si deve mai fidare della bontà degli input, perché non si ha il controllo su chi o cosa li genera. Se i dati sono in formato sbagliato, danneggiati, fuori range o in generale non adatti per l'uso che se ne deve fare, il programma può andare in errore oppure, peggio, non mostrare nessun segno di problema ma dare risultati sbagliati. A seconda dei casi può quindi essere desiderabile controllare che gli input rispettino certe *ipotesi di lavoro* prefissate.

Ad esempio, si supponga di creare una funzione che legga le prime `n` righe di un file, dove `n` è passato come input dall'utente. Si dovrebbe controllare che:

- `n` sia un numero;
- `n` sia positivo;
- `n` sia minore o uguale al numero delle righe del file;
- `n` sia intero (si potrebbe accettare anche un `float` ma servirebbe decidere a priori come interpretarlo).

Ogni situazione necessita di ipotesi di lavoro adeguate, da definire in base al contesto, ed è fondamentale prendere in considerazione anche *edge cases*[2], ovvero non solo i casi ideali in cui tutto va come sperato, ma prepararsi a tutte le eventuali condizioni rare che possono verificarsi, perché è su quelle che in genere sorgono la maggior parte dei problemi. Questo concetto è vero in generale e non solo nella gestione degli input, ma essendo gli input una sorgente enorme di incertezza ci focalizziamo su questa.

[2]Di norma si cita l'esempio delle mucche sferiche.

Prevenire o correggere? L'approccio alla gestione degli input e degli errori (cioè, di fatto, la maggior parte della logica scritta nei programmi) può seguire due grandi filosofie: prevenire o correggere.

In genere, i linguaggi tipizzati (come C e Java) preferiscono prevenire tutto: viene richiesto che l'input sia esattamente di quel tipo, oppure il codice non può proprio funzionare. In Python, che non è un linguaggio tipizzato, è invece in genere preferita la logica del "prima provo, poi chiedo scusa". Ovvero non si

cerca sempre di prevenire, ma piuttosto di correggere gli errori (cioè, gestendo le eccezioni) laddove si verifichino.

Tuttavia, anche in Python può spesso avere molto senso controllare prima alcune cose. Se infatti controllare a priori il tipo dati ricevuto in input per verificare che sia corretto può in generale portare a posizioni diverse su quale sia il miglior approccio da utilizzare e quali siano i diversi controlli da fare, ci sono invece casi in cui l'approccio è forzato. Ad esempio, se un programma è pensato per lavorare con un parametro entro un determinato range, allora va sempre verificato che questa condizione sia soddisfatta. Un caso di questo tipo si verifica ad esempio in presenza di un parametro numerico di cui poi si va a fare il logaritmo. In questo caso, il valore del parametro deve per forza essere positivo. Un altro esempio è quello di una stringa che identifica il sesso biologico di una persona, che può esser solo 'M' o 'F'. E' vero che anche in questo caso potrei non controllare niente e "provare" per poi eventualmente "chiedere scusa", ma è anche vero che la gestione diventa molto più complicata e diventa sempre più difficile riuscire ad identificare la causa dell'errore. Scrivere buon codice in Python significa trovare il giusto equilibrio tra controllare a priori l'aderenza dei dati in input alle ipotesi di lavoro e non controllare niente, reagendo laddove si generi un eventuale errore.

Validità degli input Per testare il tipo degli input si usano alcune funzioni built-in che lavorano sugli oggetti, come ad esempio `type`, `isinstance` o `issubclas`.

Esempio 0.31. Controllo a priori del tipo di input di una funzione che somma tutti gli elementi di una lista.

```
def somma_lista(lista_input):
    if not type(lista_input) == list:
        raise TypeError('Il tipo di lista_input non è lista ' +
            'ma "{}"'.format(type(lista_input)))
    somma = 0
    for elemento in lista_input:
        somma += elemento
    return somma
```

Nell'esempio sopra in realtà controllare a priori il tipo dati potrebbe non essere una buona idea, perchè la funzione `somma_lista()` potrebbe tranquillamente funzionare anche su una tupla, o una qualsiasi altra struttura che ne permette l'iterazione. Si potrebbe rinominare la funzione in qualcosa tipo `somma_tutti()` e togliere il controllo, ammettendo anche le tuple, i set, e magari anche altro, lasciando all'utilizzatore il compito di passare in input una struttura dati appropriata. Tuttavia, possiamo sempre decidere come in questo caso (in modo del tutto arbitrario) di volere una funzione che sommi solo ed esclusivamente gli elementi di strutture di tipo lista. Si potrebbe però

rilassare un po' il controllo, accettando non solo il tipo dati lista ma anche le sue estensioni (classi ottenute estendendo la classe `lista`). Ad esempio, uno potrebbe voler estendere la classe `lista` con l'obiettivo di aggiungere determinate funzionalità, come nell'esempio sottostante:

Esempio 0.32. Definisco una nuova classe `DataSet` che estende la classe lista aggiungendo una funzione che ritorna il valore assoluto della media degli elementi della lista.

```
class DataSet(list):
    def get_avg_abs_value(self):
        return sum([abs(element) for element in self])/len(self)
```

A questo punto, il check effettuato con il `type` non passerebbe più, poiché la classe `DataSet` non è di tipo lista. E' una classe diversa, che però la estende, quindi mantiene tutte le funzionalità della lista e ne aggiunge altre (il metodo `get_avg_abs_value`, in questo caso). Per ovviare a questo inconveniente, è in generale preferibile usare `isinstance` nei controlli di tipo, a meno che non si voglia fissare intenzionalmente il tipo dati a solo una classe specifica. L'esempio sottostante mostra un possibile utilizzo della funzione `isinstance`:

Esempio 0.33. Controllo a priori del tipo di input di una funzione che somma tutti gli elementi di una lista accettando anche le sottoclassi del tipo dati lista.

```
def somma_lista(lista_input):
    if not isinstance(lista_input, list):
        raise TypeError('Il tipo di lista_input non è lista, ' +
                        'ma "{}"'.format(type(lista_input)))
    somma = 0
    for elemento in lista_input:
        somma += elemento
    return somma
```

In alcuni casi potrei voler controllare anche il tipo dei singoli elementi della lista, come nell'esempio sottostante:

Esempio 0.34. Controllo a priori del tipo di input di una funzione che somma tutti gli elementi di una lista, controllando che i suoi elementi siano di tipo intero o float.

```
def somma_lista(lista_input):
    if not isinstance(lista_input, list):
        raise TypeError('Il tipo di lista_input non è lista', +
                        'ma "{}"'.format(type(lista_input)))
    for i, element in enumerate(lista_input):
        if not (isinstance(element, int) or isinstance(element, float)):
```

```
            raise TypeError('Il tipo di elemento in posizione {} di '.format(i) +
                            'lista_input non è nè intero nè float, ma ' +
                             '"{}"'.format(type(element)))
    somma = 0
    for elemento in lista_input:
        somma += elemento
    return somma
```

Tuttavia, come precedentemente accennato, decidere se effettuare o meno questo tipo di controlli dipende dal caso specifico, da quanto si vuole essere "rigidi" e da come si prevede di utilizzare il codice, perché introducono, tra l'altro, dei rallentamenti nell'esecuzione.

Si noti inoltre come in tutti gli esempi precedenti quando si genera un'eccezione si riporta nel messaggio di errore anche il tipo avuto come non valido: è fondamentale dare sempre più contesto possibile nella gestione degli errori derivanti da input non ammessi, per aiutarci nel debugging evitando messaggi di errore sterili del tipo "Errore", che non ci danno alcun indizio sul cosa possa essere andato storto.

Sanitizzazione degli input In alcuni casi ci si trova di fronte ad input che risulterebbero in prima battuta non validi ma che potrebbero portare comunque l'informazione necessaria per l'esecuzione del codice. In questo caso, è possibile provare a pulirli, o meglio, a *sanitizzarli*. Consideriamo ad esempio il caso accennato in precedenza in cui si richiede in input il sesso biologico di una persona. Un controllo stretto dell'input potrebbe essere implementato come nell'esempio che segue:

Esempio 0.35. Nel seguente esempio si vuole controllare se la stringa contenuta nella variabile `sesso` rappresenta uno dei due sassi biologici validi (`'M'` o `'F'`).

```
if sesso not in ['M', 'F']:
    raise ValueError('Il valore del sesso non è nè M ne F, ' +
                     'ma invece vale "{}"'.format(sesso))
```

Notiamo però che questo controllo è molto restrittivo. Infatti, basterebbe anche solo presentare come input il valore `' m'` e il controllo non passerebbe più, poiché tale stringa contiene uno spazio ed è in minuscolo. Con alcune semplici considerazioni si può però considerare tale input come comunque valido:

Esempio 0.36. Nel seguente esempio si vuole controllare se la stringa contenuta nella variabile `sesso` rappresenta uno dei due sassi biologici validi (`'M'` o `'F'`), sanitizzandola prima di effettuare il controllo.

```
# Si trasforma l'input in maiuscolo
```

```
sesso = sesso.upper()

# Si rimuovono gli spazi
sesso = sesso.strip()

# Ora il controllo passa anche per stringhe del tipo ' m'
if sesso not in ['M', 'F']:
    raise ValueError('Il valore del sesso non è nè M ne F, ' +
                     'ma invece vale "{}"'.format(sesso))
```

In generale, ci si deve sempre chiedere cosa può andare storto quando si utilizza un input, e decidere se non controllarlo per niente (che è una scelta valida, se intenzionale e giustificata) oppure se controllare che sia valido e in che misura. Va in ogni caso fatto uno studio a priori dei possibili problemi, in modo da aver sempre chiaro cosa può, per l'appunto, andare storto.

PART

VI

Il Testing

Il *testing* è una parte fondamentale di qualsiasi prodotto. In contesti più classici prende anche il nome di "controllo qualità", ma il concetto è sempre quello: verificare che il prodotto sia conforme alle *specifiche*.

Prendiamo ad esempio una penna. Per testarne il funzionamento prima di mandarla sugli scaffali dei negozi vorrei sicuramente verificare che scriva correttamente. Ma su che materiali? Carta, cartoncino, legno, plastica, pelle..? E in che condizioni? Solo a temperatura ambiente e all'asciutto, o magari anche al freddo, al caldo, in ambienti saturi di umidità? E in che posizione? Solo in orizzontale, o anche in verticale, o sottosopra?

La risposta a queste domande, che va a braccetto con la gestione degli edge cases a cui si è fatto riferimento nel capitolo precedente, definisce come deve essere fatto il testing.

Testing di codice In programmazione, il testing è automatizzato, e si implementa definendo un input noto, un output noto, e confrontando l'output del codice con quest'ultimo. In pseudocodice:

```
dato un input e un output noto
input noto → CODICE → output
if output != output noto:
    errore
```

Il testing del codice è principalmente di due tipi: il testing *end-to-end* indica un test automatico che determina se l'intero codice funziona nel modo corretto, mentre lo *unit testing* verifica il funzionamento delle unità minime del programma.

Il testing end-to-end equivale un po' ad allacciare un cinturone di sicurezza: se qualcosa non funziona, perlomeno ne sono consapevole, ma non so *dove* è l'errore. So solo che c'è, che è comunque importantissimo. Lo unit-tetsing invece, essendo molto più granulare, non solo mi dice che qualcosa non funziona, ma di dice anche cosa. Chiaramente è molto più impegnativo scrivere unit test rispetto ad un paio di tests end-to-end, ma in genere ripaga sempre.

Il motivo principale per scrivere dei test sul codice non è tanto infatti la verifica del suo corretto funzionamento prima di mandarlo agli utenti (che è comunque importante) ma piuttosto di aiutarci nella cosiddetta *rifattorizzazione* del codice. Ovvero, quando vado ad aggiungere una funzionalità o

sopratutto a modificarne una esistente, il testing mi dice se funziona ancora tutto come prima o se è saltato qualcosa. La stessa cosa si applica all'ambiente inteso come dipendenze del codice: se voglio aggiornare un pacchetto Python che ho installato, il testing mi dice se anche con la nuova versione tutto funziona correttamente.

Il testing in Python Un primo modo semplice per implementare una logica di testing con Python è di utilizzare uno dei vari operatori di confronto e generare un'eccezione laddove una delle condizioni del test non sia soddisfatta.

Esempio 0.37. Esempio di testing semplice:

```
def somma(a,b):
    return a+b

# Testing
if not somma(1,1) == 2:
    raise Exception('Test 1+1 non passato')

if not somma(1.5,2.5) == 4:
    raise Exception('Test 1.5+2.5 non passato')
```

Si noti che i due test appena effettuati vanno a controllare degli aspetti diversi della funzione somma, ovvero il funzionamento con `int` e quello con `float`. Queste sono, in altre parole, le specifiche che ci si è dati: che funzioni con numeri interi e float. Ma avrei per esempio potuto voler supportare anche i numeri complessi, o magari le stringhe (come abbiamo visto la somma fra stringhe è definita come concatenazione delle stesse).

A livello di strutturazione del codice, in genere è sempre meglio scrivere i test in un file a parte, e importare le funzioni o gli oggetti da testare, come mostrato nei due esempi sottostanti:

Esempio 0.38. File `somma.py`

```
def somma(a,b):
    return a+b
```

Esempio 0.39. File `test_somma.py`

```
from somma import somma

if not somma(1,1) == 2:
    raise Exception('Test 1+1 non passato')

if not somma(1.5,2.5) == 4:
    raise Exception('Test 1.5+2.5 non passato')
```

In ambito professionale, il testing con Python si fa di norma utilizzando dei *framework di test*, come ad esempio *unittest*, che fa parte della standard library, oppure *PyTest* o *Nose*, che vanno installati come pacchetti.

Nel framework unittest, per esempio, si definiscono della classi di test (estendendo la classe base TestCase) e dei test case (con dei metodi delle classi che devono iniziare necessariamente per `"test_"`). I controlli si fanno con degli *assert*, metodi presenti già nella classe di test perché ereditati dalla classe base TestCase. Per motivazioni storiche, i metodi per gli assert nel modulo unittest non seguono la convenzione Python snake case (usando gli underscore), ma utilizzano la notazione camel case.

Vediamo come avremmo potuto implementare il testing per la nostra funzione somma con il framework unittest:

Esempio 0.40 (`unittest`)**.** File `test_somma.py` utilizzando il framework unittest per testare la funzione somma del file `somma.py`. Il nome del file contente i test deve iniziare necessariamente per "`test_`".

```
from unittest import TestCase
from somma import somma

class TestSomma(TestCase):
    def test_somma(self):
        self.assertEqual(somma(1,1), 2)
        self.assertEqual(somma(1.5,2.5), 4)
```

Gli `assertEqual()` possono essere sostituiti da `assertTrue()`, per verificare una condizione booleana, o (con una leggera modifica) `assertRaises()`, per verificare che una particolare eccezione viene lanciata, o altri ancora.

Per lanciare il processo di testing si scrive da linea di comando:

```
python -m unittest discover
```

L'output è una serie di puntini che indicano il successo dei test. Si possono visualizzare più informazioni sui test aggiungendo `-v` alla fine della riga.

PART

VII

Lavorare veramente

Per mettere assieme quanto esposto nei capitoli precedenti, vediamo un esempio concreto che, anche se molto semplificato, è sulla falsariga di come si poi lavora veramente con Python in ambito professionale.

L'esempio che vederemo è un piccolo modello predittivo per andare a prevedere le vendite di shampoo caricate dal file `shampoo_sales.csv`.

Creare un modello Un *modello* è una rappresentazione concettuale di un fenomeno che ne spiega il suo funzionamento. Ad esempio, se si lancia un oggetto, si riesce a prevedere dove cadrà grazie ad un modello fisico noto come moto parabolico, il cui funzionamento è spiegato tramite un'equazione.

Tutti i modelli si basano su delle ipotesi di funzionamento e vengono costruiti basandosi su delle leggi: un *modello fisico* viene costruito basandosi sulla validità delle leggi della fisica, un *modello epidemiologico* su quelle della biologia, mentre un *modello comportamentale* su quelle della psicologia o delle scienze cognitive.

Il fenomeno può essere sia temporale (come le vendite di shampoo nel tempo) ma anche spaziale: ad esempio si potrebbe prevedere la temperatura di un luogo se si conoscessero sia il modello che descrive la temperatura nello spazio, sia la temperatura nei luoghi adiacenti.

Il metodo più semplice per creare un modello è assumere di conoscere il comportamento del fenomeno e poterlo descrivere a tavolino. Ovviamente non è il metodo più efficace, e non sarà sempre possibile, ma iniziamo da qui.

Ipotizziamo quindi che l'andamento delle vendite dello shampoo segua una semplice regola, ovvero che le vendite al passo $t + 1$ siano date dalla somma delle due quantità seguenti:

1. la variazione media[3] negli ultimi 3 mesi
2. il valore delle vendite al passo t.

Se per esempio abbiamo i seguenti dati:

- Settembre: passo temporale = t-2, vendite = 50
- Ottobre: passo temporale = t-1, vendite = 52
- Novembre: passo temporale = t, vendite = 60

[3] La *variazione media* è la media delle differenze tra le vendite a un determinato passo temporale e quello precedente, per ogni passo temporale da considerare.

Allora possiamo chiederci quanto potrebbero essere le vendite a Dicembre, e rispondere applicando la logica appena descritta, ovvero:
$60 + ((52 - 50) + (60 - 52))/2 = 65$

Per implementare questa logica in Python possiamo ricorrere ad una classe che rappresenti il modello e che chiameremo `TrendModel`, la quale avrà un metodo `predict()` per fornire la previsione delle vendite.

Esempio 0.41. (file `models.py`) Modello che predice le vendite applicando una semplice logica che considera la variazione degli ultimi 3 mesi.

```
class TrendModel():
    def predict(self, data):
        if len(data) != 3:
            raise ValueError('Passati {} mesi'.format(len(data)) +
                             ' ma il modello richiede di averne 3.')
        variations = []
        item_prev= None
        for item in data:
            if item_prev is not None:
                variations.append(item-item_prev)
            item_prev = item
        avg_variation = sum(variations)/len(variations)
        prediction = data[-1] + avg_variation
        return prediction
```

A questo punto scrivo un semplice test per testare il mio modello:

Esempio 0.42. (file `test_models.py`) Un semplice test per il modello.

```
from models import TrendModel

# Definisco dei dati di test
test_data = [50,52,60]

# Istanzio il modello
trend_model = TrendModel()

# Applico il modello
prediction = trend_model.predict(test_data)

# Testo che il risultato sia corretto
if prediction != 65:
    raise Exception('Il modello sbaglia la prediction: ha ' +
                    'dato {} invece di 65'.format(prediction))
```

```
# Testo anche che il modello non possa funzionare
# su dei dati con un numero di mesi diverso da 3:
wrong_test_data = [50,52,60,70]
try:
    trend_model.predict(wrong_test_data)
except ValueError:
    pass
else:
    raise Exception('Il modello non alza alcuna eccezione ' +
                    'se applicato su un numero di mesi ' +
                    'diverso da 3')

print('ALL TESTS PASS')
```

Se a questo punto eseguo il file `test_models.py` dovrei vedere stampato a schermo il messaggio `ALL TESTS PASS`.

Da notare che non stiamo ancora usando il file vero e proprio delle vendite dello shampoo, e non ci interesserà farlo fino alla fine: è sempre bene sviluppare con dei piccoli dati di esempio e non lavorare sui dati veri, che sarebbero molto più difficili da usare.

Si noti anche che stiamo solo controllando una caratteristica specifica dell'input, ovvero la lunghezza dei dati. Non stiamo imponendo né che sia di tipo lista né che il suo contenuto sia di tipo numerico. Come detto nel capitolo sul controllo degli input, questa è una scelta arbitraria: in questo caso questo compromesso ci va bene.

Possiamo ora decidere di evolvere un po' il modello, ad esempio parametrizzando il numero di mesi da usare per la previsione, così da vedere come si comporta il modello a seconda dei casi. Aggiungiamo quindi un parametro *opzionale* al modello, il cui valore di default è proprio 3, così da lasciarlo compatibile con il codice scritto fino ad ora. Siccome questi modelli che operano su una porzione di dati subito antecedente la previsione si chiamano *modelli a finestra*, chiamerò questo parametro proprio "finestra". L'esempio sottostante mostra una possibile implementazione di questa logica.

Esempio 0.43. (file `models.py`) Modello che predice le vendite applicando una semplice logica che considera la variazione degli ultimi n mesi.

```
class TrendModel():

    def __init__(self, window=3):
        self.window = window
```

```
    def predict(self, data):
        if len(data) != self.window:
            raise ValueError('Passati {} mesi'.format(len(data)) +
                             ' ma il modello richiede di' +
                             ' averne {}.'.format(self.window))
        variations = []
        item_prev = None
        for item in data:
            if item_prev is not None:
                variations.append(item-item_prev)
            item_prev = item
        avg_variation = sum(variations)/len(variations)
        prediction = data[-1] + avg_variation
        return prediction
```

A questo punto la prima cosa da fare è eseguire nuovamente i test, per verificare che tutto funzioni come prima nel caso in cui la lunghezza della finestra di previsione non sia specificata, ma venga presa di default uguale a 3.

Se tutto funziona correttamente, possiamo procedere con il testare la nuova funzionalità aggiungendo i test come nell'esempio sottostante:

Esempio 0.44. (file `test_models.py`) Un semplice test per il modello con finestra di lunghezza variabile.

```
from models import TrendModel

# Definisco dei dati di test
test_data = [50,52,60]

# Istanzio il modello
trend_model = TrendModel()

# Testo che la lungezza della finestra di default venga settata a 3
if trend_model.window != 3:
    raise Exception('Il modello non ha una finestra di default pari a 3,' +
                    ' ma di {}'.format(trend_model.window))

# Applico il modello
prediction = trend_model.predict(test_data)

# Testo che il risultato sia corretto
if prediction != 65:
    raise Exception('Il modello sbaglia la previsione: ha ' +
                    'dato {} invece di 65'.format(prediction))
```

```
# Testo anche che il modello non possa funzionare
# su dati con un numero di mesi diverso da
# quelli della finestra (3 in questo caso):
wrong_test_data = [50,52,60,70]
try:
    trend_model.predict(wrong_test_data)
except ValueError:
    pass
else:
    raise Exception('Il modello non alza alcuna eccezione ' +
                    'se applicato su un numero di mesi ' +
                    'diverso da quelli della finestra')

# Infine tetso il modello con una finstra di lunghezza
# diversa da quella di default di 3 mesi
trend_model = TrendModel(window=4)

# Testo che la lungezza della finestra sia stata settata a 4
if trend_model.window != 4:
    raise Exception('Il modello non ha settata una finestra pari a 4,' +
                    ' ma di {}'.format(trend_model.window))

# Applico il modello con finestra di 4 mesi
test_data_new = [50,52,60,65]
prediction = trend_model.predict(test_data_new)

# Testo che il risultato sia corretto
if prediction != 70:
    raise Exception('Il modello sbaglia la previsione: ha ' +
                    'dato {} invece di 70'.format(prediction))

print('ALL TESTS PASS')
```

Come si vede, il testing entra subito in gioco nel "lavorare veramente": non appena modifichiamo qualcosa, se abbiamo scritto dei test, possiamo utilizzarli per essere sicuri di non aver creato nessun cambiamento che "rompe" qualcosa. Come vedremo nella prossima sezione, avere dei test è fondamentale per effettuare le cosiddette operazioni di *refactoring* del codice, ovvero quando si va a riorganizzarne la struttura del codice a fronte di nuovi requisiti.

Fittare un modello Il modello appena visto non è molto utile, perché guarda solo i dati degli ultimi mesi, cioè di quella che abbiamo chiamato finestra, e non considera minimamente quelli storici. Se per esempio avessimo avuto anni

di incremento nelle vendite di shampoo, seguiti da una riduzione solo negli ultimi mesi, è verosimile pensare che questa diminuzione non permanga a lungo, poiché probabilmente parte solo delle normali fluttuazioni di mercato.

Inoltre non si capisce bene perché abbiamo usato una classe per definire il modello, dato che al momento non stiamo sfruttando nessuno dei due grossi vantaggi della programmazione ad oggetti: mantenere degli stati e gestire gerarchie.

Tutto questo ora cambierà radicalmente. Innanzitutto cominciamo con il considerare il fatto che per fare una previsione più ragionevole dobbiamo tenere conto non solo dell'andamento *locale*, ossia degli ultimi mesi, ma anche di quello *globale*, ovvero storico. Quando un modello viene "calato" sui dati, si dice che si effettua il *fit* del modello. La parola "fit" in inglese significa aderire, ed è in effetti proprio quello che andremo a fare: far aderire il nostro modello ai dati.

In generale, far aderire un modello a dei dati (o *fittare*) significa calcolare, o stimare, dei coefficienti. Vediamo come esempio due modelli matematici, uno lineare e uno esponenziale, e cosa succede se li proviamo a fittare sui dati delle vendite dello shampoo.

Definizione 0.1 (Modello Lineare)**.** Un modello *lineare* ha la forma:

$$f(x_i) = ax_i + b$$

Definizione 0.2 (Modello Esponenziale)**.** Un modello *esponenziale* ha la forma:

$$f(x_i) = ae^{b(x_i - c)} + d$$

Per fittare questi due modelli si cercano i coefficienti che minimizzino la distanza tra il modello e i dati. In Python si può usare la libreria `scikit-learn` (che non vediamo), e si ottengono i seguenti coefficenti e grafici:

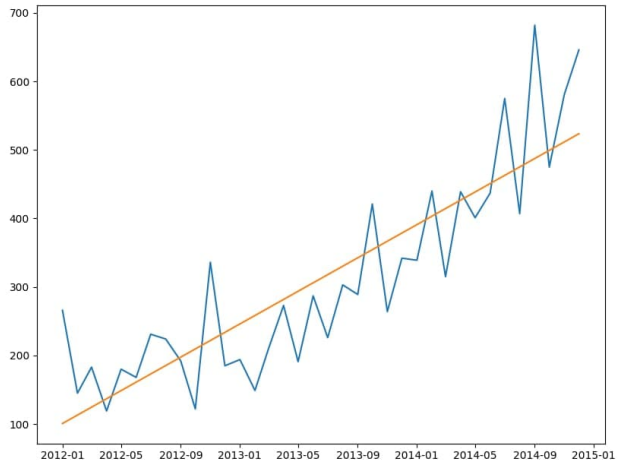

Figura 5. Modello lineare ($a = 3.97 \cdot 10^{-1}, b = -5.98 \cdot 10^{3}$)

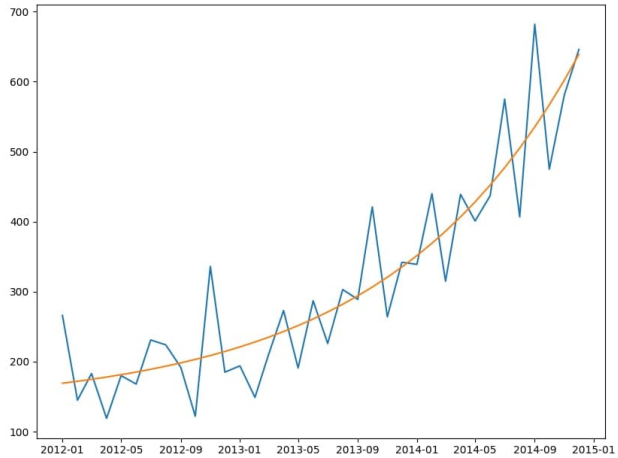

Figura 6. Modello esponenziale ($a = 4.65 \cdot 10^{-16}, b = 2.53 \cdot 10^{-03}, c = 1.35 \cdot 10^{2}$)

Noi non andremo a fare il fit di una retta o di una curva, poiché implicherebbe scrivere una buona dose di codice o utilizzare una libreria esterna che al momento ci farebbe più confusione che altro.

Scegliamo invece, a tavolino, di calcolare un altro parametro di fit anziché i coefficienti di formule matematiche, e in particolare andremo a considerare la *variazione storica* . È un coefficiente tanto quanto un altro, ma è molto più semplice da gestire. Useremo poi questa informazione per effettuare la previsione, pensandolo come la variazione degli ultimi n mesi che vogliamo considerare. In particolare, decidiamo che la variazione che vogliamo considerare al tempo t sarà data dalla media tra la variazione storica e quella della finestra. In altre parole, andiamo a ridefinire il modello in modo che le vendite al passo $t + 1$ siano ottenute dai seguenti passi:

1. si calcola la variazione media storica
2. si media il risultato di 1. con la variazione media negli ultimi n mesi
3. si somma il risultato di 2. con il valore delle vendite al passo t.

Consideriamo i seguenti dati di prova:

- Maggio: passo temporale = non rilevante, vendite = 8
- Giugno: passo temporale = non rilevante, vendite = 19
- Luglio: passo temporale = non rilevante, vendite = 31
- Agosto: passo temporale = non rilevante, vendite = 41
- Settembre: passo temporale = t-2, vendite = 50
- Ottobre: passo temporale = t-1, vendite = 52
- Novembre: passo temporale = t, vendite = 60

Effettuando i conti vediamo come la variazione media storica sia 11, quelle locali all'interno della finestra sono 2 e 8, quindi hanno media 5 e quindi la media è 8. Applicata al valore delle vendite al tempo t ottengo una predizione di 68.

Volendo trasporre in codice quanto detto e ipotizzando di aggiungere al file `models.py` la classe `FitTrendModel`, ci accorgiamo di due cose *fondamentali*:

1. ci troveremmo a scrivere lo stesso codice per il calcolo della variazione media tre volte (una l'abbiamo già scritta e dovremmo ripeterla quasi identica sia per la parte di `fit` che di `predict` del nuovo modello)
2. il modello che ci troveremmo ad aggiungere sarebbe sempre comunque un modello, e quindi è probabile che ci torni comodo usare una gerarchia di qualche tipo

Possiamo quindi pensare di riconsiderare la struttura del codice e di astrarre un po' meglio le diverse entità. In particolare, ci accorgiamo che è probabilmente una buona idea avere una generica classe `Modello` che predisponga dei metodi sia per la funzionalità di fit che quella di predict, anche se poi i modelli specifici non le andranno ad implementare, come nell'esempio seguente:

Esempio 0.45. Classe base `Model`, che prevede due metodi fit e predict non implementati. Starà ai modelli specifici farlo.

```
class Model():

    def fit(self, data):
        # Fit non implementato nella classe base
        raise NotImplementedError('Metodo non implementato')

    def predict(self, data):
        # Predict non implementato nella classe base
        raise NotImplementedError('Metodo non implementato')
```

Modificheremo poi la classe `TrendModel` in modo che estenda Modello, andando ad implementare la predict (ma non la fit). Questo passaggio avrà ancor più senso quando, tra poco, andremo a definire la classe `FitTrendModel` come figlia (cioè che estende `TrendModel`).

Per quanto riguarda il calcolo della variazione media, possiamo definire una funzione di supporto `compute_avg_variation` che lo calcoli. Andiamo a definire questa funzione come metodo della classe `TrendModel`, così poi anche tutte le sue sottoclassi la erediteranno (tra cui, anticipando dove andremo a parare, anche la classe `TrendModel`). Ne approfittiamo per disaccoppiare anche la logica per la validazione dei dati in entrata alla predict, con un metodo `validate_data` che in caso di incompatibilità tra la lunghezza della finestra prefissata e i dati generi un'eccezione. Ci tornerà comoda per il nuovo modello che creeremo fra poco.

Mettendo assieme queste modifiche, il nuovo file `models.py` apparirà come qualcosa del tipo:

Esempio 0.46. (file `models.py`) Rifattorizzazione del codice con introduzione della classe base Model e disaccoppiamento della logica per il calcolo della variazione media e per la validazione dei dati in entrata.

```
class Model():

    def fit(self, data):
        # Fit non implementato nella classe base
        raise NotImplementedError('Metodo non implementato')

    def predict(self, data):
        # Predict non implementato nella classe base
        raise NotImplementedError('Metodo non implementato')
```

```
class TrendModel(Model):

    def __init__(self, window=3):
        self.window = window

    def validate_data(self, data):
        if len(data) != self.window:
            raise ValueError('Passati {} mesi'.format(len(data)) +
                             ' ma il modello richiede di' +
                             ' averne {}.'.format(self.window))

    def compute_avg_variation(self, data):
        variations = []
        item_prev = None
        for item in data:
            if item_prev is not None:
                variations.append(item-item_prev)
            item_prev = item
        avg_variation = sum(variations)/len(variations)
        return avg_variation

    def predict(self, data):
        self.validate_data(data)
        prediction = data[-1] + self.compute_avg_variation(data)
        return prediction
```

A questo punto avviene uno dei passaggi chiave nella metodologia di lavoro corretta: si girano i test! Non abbiamo cambiato le specifiche del codice, né tanto meno il comportamento del modello `TrendModel`. Abbiamo solamente riorganizzato il codice per renderci la vita molto più facile nell'implementazione della nuova classe `FitTrendModel` che ora aggiungeremo, quindi i test *devono* passare. Altrimenti vuol dire che abbiamo introdotto un bug nel processo di rifattorizzazione e come prima cosa prima di procedere dobbiamo risolverlo.

Se abbiamo fatto tutto correttamente dobbiamo quindi ottenere `ALL TESTS PASS` all'esecuzione dei test, e possiamo proseguire con la sicurezza di non aver "rotto niente".

L'aggiunta della nuova classe `FitTrendModel` diventa a questo punto molto semplice, perché siamo stati furbi nell'organizzazione del codice e quasi tutto viene ereditato dalla classe `TrendModel` che estendiamo. Vediamola da sola:

Esempio 0.47. (classe `FitTrendModel` del file `models.py`) la nuova classe per il modello che considera anche la variazione storica.

```
class FitTrendModel(TrendModel):

    def fit(self, data):
        self.historical_avg_variation = self.compute_avg_variation(data)

    def predict(self, data):
        try:
            self.historical_avg_variation
        except AttributeError:
            raise Exception('Il modello non è fittato!')
        self.validate_data(data)

        prediction = data[-1] + (self.historical_avg_variation +
                                 self.compute_avg_variation(data))/2
        return prediction
```

Come si vede stiamo utilizzando tutto quello che abbiamo ereditato dalla classe genitore, ovvero l' `__init__`, che setta la lunghezza della finestra, e i metodi di supporto `compute_avg_variation` e `validate_data`. Possiamo quindi aggiungere la logica di fit andando, con una sola riga di codice, a calcolare l'incremento su tutti i dati (che devono essere l'argomento della fit), che riutilizziamo anche per la predict implementando la logica necessaria in poche righe, incluso il controllo sulla compatibilità fra la lunghezza della finestra impostata nell'istanziazione del modello e i dati passati alla predict (che devono essere solo i dati corrispondenti alla finestra, appunto). Aggiungiamo anche un controllo sul fatto che prima di poter usare un modello con fit, questo debba essere stato effettuato: nella predict proviamo ad accedere all'attributo `historical_avg_variation`, e se abbiamo un errore di attributo questo può solo voler dire che il modello non è ancora stato fittato, errore che segnaliamo in modo chiaro. Potremmo (o meglio, dovremmo) aggiungere anche dei test sul nuovo modello, ma per il momento ci possiamo fermare qui.

Questo esempio conclude, di fatto, tutto il giro del "lavorare veramente": scriviamo un pezzo di codice, verifichiamo che funzioni, lo miglioriamo o aggiungo altre funzionalità, torniamo a verificare che funzioni e aggiungiamo dei test per le nuove parti, e così via. Meglio organizziamo il codice e meno lavoro facciamo, più siamo furbi nella gestione dei casi inattesi e chiari nei messaggi di errore, meno tempo perderemo con il debugging.

Valutare un modello Come ultima parte, e per concludere anche a livello logico l'esempio sui due modelli, vediamo come li possiamo valutare per capire

quale dei due, dati alla mano, sia meglio. Potrebbe succedere infatti che un'ipotesi che abbiamo fatto non sia fondata, come ad esempio l'ipotesi che sia meglio considerare la variazione storica, e trovarci ad aver fatto del cosiddetto overengineering poiché magari il modello che non la considera, ovvero quello senza fit, ci bastava o addirittura si comportava meglio, .

Per dare una valutazione della bontà dei due modelli e quindi rispondere a questa domanda, ci serve prima una *metrica*, che altro non è che una misura di qualcosa. A esempio, una metrica di quanto va "bene" un paese è il prodotto interno lordo (anche se sarebbe tutto da dimostrare), mentre la metrica per quantificare quanto è efficiente un'automobile a combustione interna è in genere quanti chilometri fa con un litro. Per valutare quanto sono buoni i modelli predittivi, si guarda in prima battuta all'*errore medio* che commettono.

La metrica più semplice per quantificare l'errore è considerare l'errore assoluto medio, anche detta MAE (da Mean Absolute Error).

Definizione 0.3 (MAE)**.** Il Mean Absolute Error è così definito:

$$MAE = \sum_{i=1}^{n} \frac{|y_i - f(x_i)|}{n}$$

dove n è il numero di osservazioni considerate, y_i è l'i-esima osservazione e $f(x_i)$ è la previsione del modello legata all'i-esima osservazione.

Un'altra metrica per quantificare l'errore, un minimo più sofisticata, è considerare la radice dell'errore quadratico medio, che in questo contesto viene anche chiamato scarto quadratico medio, portando all'abbreviazione RMSE (da Root Mean Square error).

Definizione 0.4 (RMSE)**.** Il Root Mean Squared Error è così definito:

$$RMSE = \sqrt{\sum_{i=1}^{n} \frac{(y_i - f(x_i))^2}{n}}$$

dove n è il numero di osservazioni considerate, y_i è l'i-esima osservazione e $f(x_i)$ è la previsione del modello legata all'i-esima osservazione.

Se torniamo a considerare la retta e la curva che abbiamo fittato sui dati nella sezione precedente, questi sono, rispettivamente: 76.3 e 60.9. Sembrerebbe quindi che l'andamento esponenziale fitti meglio i dati delle vendite dello shampoo.

Per valutare un modello predittivo quando siamo interessati alla bontà delle sue previsioni piuttosto che a quanto bene aderisce ai dati (che sono due cose diverse), si adotta un approccio che prevede la divisione dei dati in due parti:

- Fit / training set: parte dei dati usata per fittare il modello, circa il 70-80% dei dati;
- Test / evaluation set: parte dei dati su cui calcolare il valore della metrica di valutazione (come il MAE o l'RMSE), circa il 20-30% dei dati.

È importante che questi due set siano divisi perché il test deve avvenire su una parte dei dati che non è stata ancora "osservata" dal modello: se il modello fornisce buone previsioni anche per questa parte di dati, allora è una buona indicazione che il modello è riuscito a catturare la dinamica che si vuole modellare, e che non ha "imparato a memoria" certi comportamenti o, peggio, che è andato nel cosiddetto *overfitting*.

Questo succede quando il modello si comporta (fin troppo) bene sui dati di training ma non va per niente bene sui dati di test, ed è il motivo principale per cui si dividono i dati in queste due parti e non si applica il modello su tutti i dati nel loro insieme come fatto per valutare se avevano un andamento lineare piuttosto che esponenziale.

Se invece il modello non va bene neanche sulla parte di dati riservata al fit, (e sicuramente non andrà meglio su dati mai visti ovvero quelli di test), allora si parla di *underfitting*.

Entrambi i casi di overfitting e underfitting sono rappresentati in Figura 7), assieme anche al caso di fit "corretto".

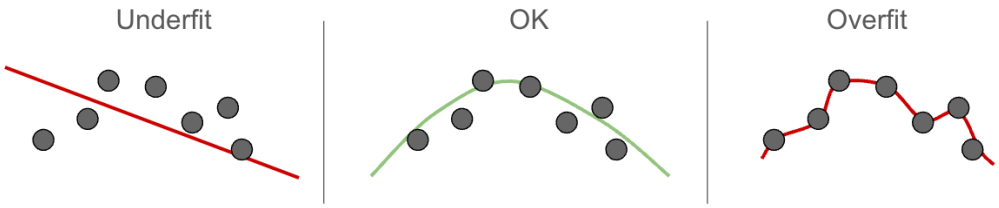

Figura 7. Possibili risultati di un fit

Per concludere il nostro esempio logico andiamo allora a capire come potremmo valutare i due modelli che abbiamo implementato, ovvero le classi `TrendModel` e `FitTrendModel`.

Per prima cosa notiamo che la valutazione del modello è una funzionalità condivisa da entrambi, e quindi decidiamo di implementarla nella classe base `Model`: `TrendModel` e `FitTrendModel` la erediteranno gratis e useranno la loro logica per la predizione in modo automatico.

In secondo luogo, possiamo provare a ragionare in un modo leggermente diverso rispetto al solito, ovvero scrivendo *prima i test*. Potremmo per esempio voler aggiungere un metodo per la valutazione che chiameremo proprio `evaluate()`. E come dovrebbe comportarsi? Decidiamo che il suo input sarà tutto il dataset, e che effettuerà lui lo split tra fit e test set, il fit del modello (se previsto) e il calcolo dell'errore.

Definiamo allora un test set di esempio, ancora più semplice rispetto ai modelli che avevamo già visto per il modello con fit: `[1,2,3,4,5,6,7,8,9,10.5, 11.5,11.2]`. Sono dati inventati di sana pianta! Non è importante che ci siano dati sensati nel testing, ci servono solo per verificare che i conti tornino).

Su questi dati ora, a mano, facciamo i conti: ed è proprio così che si fa anche in ambito professionale. Non importa quanto complesso sia l'argomento, si può (quasi) sempre riportarsi ad un esempio rappresentativo risolvibile a mano.

Facendo i conti a mano, dovremo dividere i nostri dati nel 70% di fit e 30% di test. Siccome abbiamo 12 valori, questo vorrà dire che useremo i primi 8 per il fit e gli ultimi 4 per il test.

Quindi: il fit andrebbe fatto su `[1,2,3,4,5,6,7,8]` e poi predict su `[9,10.5, 11.5,11.2]`. Useremo un modello con finestra pari a 2, il che vuol dire che avremo solo *due* predizioni nel test set: la prima usando come finestra i valori`[9,10.5]` e la seconda usando come valori `10.5,11.5`. Poi avremo finito i dati, ma non ce ne servono altri: c'è tutto il necessario per testare il metodo `evaluate()`.

Faremo i conti i conti solo per il modello `TrendModel`: non ci serve testare il nuovo metodo `evalaute()` che ci stiamo accingendo a scrivere anche sulla classe `FitTrendModel`, poiché in questa classe semplicemente si troverà a chiamare (automaticamente) un altra logica di predizione: cambieranno i numeri, non come funziona il codice. Il testing quindi neanche guarderà i dati del fit, ma verrà testato indirettamente che venga effettuato il corretto split, poichè il risultato verrà tornato correttamente solo se lo split sarà stato fatto in modo corretto.

Facendo i conti per il modello `TrendModel` con la logica spiegata nel relativo capitolo otteniamo, per le due predizioni:

1. `(10.5 - 9)` $10.5 = 12$
2. `(11.5 - 10.5)` $11.5 = 12.5$

A questo punto dobbiamo definire la metrica d'errore che vogliamo calcolare. Scegliamo il MAE per semplicità, e lo calcoliamo a mano:

- per la prima predizione il modello ha previsto 12 ma il valore vero era 11.5, quindi l'errore è 0.5

- per la seconda predizione il modello ha previsto 12.5 ma il valore vero era 11.1, quindi l'errore è -1.3

L'errore medio assoluto è quindi dato da $MAE = (|0.5| + |-1.3|)/2 = 0.9$

Posso procedere quindi a scrivere il test, che includerò nel file `test_models.py`:

Esempio 0.48. Test per la valutazione dei modelli su un determinato dataset.

```
# Definisco dei dati di test
test_data = [1,2,3,4,5,6,7,8,9,10.5,11.5,11.2]

# Istanzio il modello
trend_model = TrendModel(window=3)

# Chiamo la evaluate
evaluation_result = trend_model.evaluate(test_data)

# Verifico il risultato
if evaluation_result != 1.8:
    raise Exception('La valutazione del modello non ha dato 1.8 '
                    ' ma ha dato {}'.format(evaluation_result))
```

Questa modalità di lavoro i chiama *test-driven development*, ed è particolarmente interessante perché permette di focalizzarsi prima esclusivamente sui risultati attesi e poi spostarsi sull'implementazione del codice.

Andrò quindi ora a scrivere il codice vero e proprio per la `evaluate()` nella classe base `Model()`, che dovrà poter gestire anche i modelli senza fit. Vediamo come potrei approcciare la cosa:

Esempio 0.49. Classe base `Model` con lo scheletro di implementazione del metodo evaluate().

```
class Model():

    def fit(self, data):
        # Fit non implementato nella classe base
        raise NotImplementedError('Metodo non implementato')

    def predict(self, data):
        # Predict non implementato nella classe base
        raise NotImplementedError('Metodo non implementato')
```

```
    def evaluate(self, data):

        # Setto l'indice di cutoff per dividere
        # i dati nel 70% di fit e 30% di test
        fit_data_cutoff = int(len(data)*0.7)

        # Divido i dati
        fit_data = data[:fit_data_cutoff]
        test_data = data[fit_data_cutoff:]

        # Provo a fare il fit del modello, e
        # nel caso non sia supportato, passo!
        try:
            self.fit(fit_data)
        except Exception as e:
            if isinstance(e,NotImplementedError):
                pass
            else:
                raise Exception('Il metodo fit è implementato ma' +
                                'ha sollevato una eccezione {}'.format(e))

        # Far fare al modello le previsioni sul
        # test set e confrontarle con i dati veri,
        # calcolando il MAE (evaluation_mae) è
        # invece lasciato al lettore
        evaluation_mae = None

        # Torno il risultato della valutazione
        return evaluation_mae
```

Nonostante l'implementazione della logica della `evaluate()` sia lasciata al lettore, riportiamo comunque i risultati per la valutazione del modello con e senza fit sui dati veri `shampoo_sales.csv`:

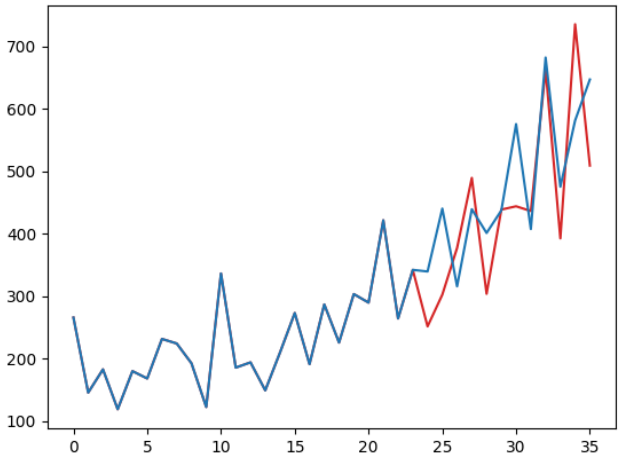

Figura 8. Modello senza fit applicato sui dati veri delle vendite dello shampoo, dividendo i dati in un parte di fit (non utilizzata in questo caso) e una di evaluation: il MAE è 82.

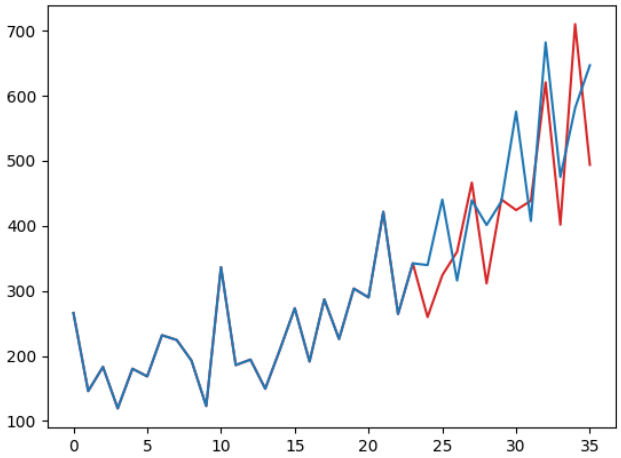

Figura 9. Modello con fit applicato sui dati veri delle vendite dello shampoo, dividendo i dati in un parte di fit e una di evaluation: il MAE è 79.

Sembra quindi che vada leggermente meglio il modello con fit, anche se di talmente poco che probabilmente non è significativo. D'altronde, erano pur sempre solo due "toy-model".

Anche se abbiamo usato dei modelli giocattolo, molto semplici e non particolarmente furbi nelle predizioni, la struttura di quanto visto in questo capitolo può essere usata per incapsulare logiche predittive complesse a piacere, da modelli statistici a reti neurali, permettendo di gestire il codice in modo efficiente, modulare e riutilizzabile.