Laboratorio di Programmazione [DRAFT]

Stefano Alberto Russo

Corso di Laurea Triennale in Intelligenza Artificiale e Data Analytics (AIDA)
Departimento di Matematica e Geoscienze
Università di Trieste

Dispense scritte da Gianluca Guglielmo con i contributi di Michele Rispoli, Pietro Morichetti, Nicolas Solomita, Andrea Mecchina, Elena Buscaroli e Federico Pigozzi.

15 novembre 2022

Indice

PART

I

Programmazione in Python

1 Strumenti utilizzati

Per affrontare il *Laboratorio di Programmazione in Python* è necessario preliminarmente presentare alcuni strumenti di cui si farà uso.

File Manager Il *File Manager* è il comune strumento, integrato nel sistema operativo, che permette all'utente di vedere, modificare, copiare ed eliminare file e cartelle. Per il corso è necessario impostare la visualizzazione dei file nascosti.

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 il *BASH* (o *Terminale*), che può essere aperto con lo shortcut `CTRL + ALT + T`.

Editor del Codice L'*Editor del Codice* può essere un qualsiasi programma per la scrittura del testo *plain*, con encoding UTF-8. È possibile usare un editor che interpreti il linguaggio utilizzato per colorare le diverse sintassi e facilitare la programmazione. Per il corso è necessario impostare l'editor in modo che utilizzi gli spazi al posto dei tab. In Python sono comunemente usati 4 spazi per l'indentazione.

Git Il software per il versionamento *Git* è uno strumento fondamentale per programmare in modo sicuro e pulito. Git tiene traccia delle modifiche fatte al codice, può essere sincronizzato con dei collaboratori ed è decentralizzato. 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 permette di creare dei checkpoint del codice indicati da codici deterministici detti *hash*, che individuano univocamente la versione. 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, Code::Blocks e Replit.

Replit Per semplificare lo sviluppo, durante il corso verrà usato l'IDE cloud-based *Repl.it*. Per inizializzare l'ambiente seguire questi passi:

1. registrarsi su GitHub;
2. creare un repo su GitHub chiamato *ProgrammingLab*. Impostare il repo come pubblico e aggiungere il file ReadMe;
3. registrarsi su ReplIt;
4. creare un repo su Repl importandolo da GitHub (concedere le autorizzazioni se richiesto);
5. configurare il file `.replit` nel seguente modo:

```
language = "bash"
run = "/bin/bash"
```

6. committare il file `.replit` inserendo un commento di commit e cliccando "Commit and Push".

Dopo l'inizializzazione, creare un file `hello.py` con dentro:

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

ed eseguire di nuovo commit + push.

PART

II

Introduzione a Python

Python è un linguaggio di programmazione interpretato di alto livello. Durante il corso useremo la versione 3, in prticolare ≥ 3.6.

Pseudocodice Prima di scrivere un codice, in qualsiasi linguaggio di programmazione, è comodo scrivere una bozza usando parole (italiane o inglesi) con indentazione, focalizzandosi non sul come, ma sul *cosa* fare. Il risultato è chiamato *pseudocodice* ed è buona prassi utilizzarlo come schema. Non esiste uno standard per lo pseudocodice.

Cos'è Python Python nasce nel 1991 come linguaggio di programmazione ad oggetti. Il suo utilizzo è in costante crescita, grazie anche alla sua semplicità ed intuitività. È un linguaggio *interpretato*, quindi non ha bisogno di compilazione e può essere anche usato in live dalla shell Python, molto utile ogniqualvolta si voglia testare delle cose in rapidità. Essendo un linguaggio interpretato, Python è più lento di C nell'eseguire lo stesso tipo di istruzione, ma è possibile utilizzare alcuni accorgimenti per velocizzare l'esecuzione, come sfruttare la libreria *pandas* (che non vedremo in questo corso) per velocizzare i cicli ed eseguirli in stile C.

Python è il linguaggio standard per la Data Science grazie ad un enorme ecosistema di strumenti per il calcolo scientifico/statistico (come la stessa libreria *pandas*, nominata in precedenza).

Si assume che si abbia già familiarità[1] con i costrutti e gli operatori base di un linguaggio di programmazione, come:

- 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 ed operatori nel linguaggio Python.

Tipi dati In Python non è necessario definire il tipo dati della variabile durante la dichiarazione. Infatti, Python deduce automaticamente il tipo durante l'inizializzazione grazie al valore che le viene assegnato. Al contrario di C, in Python una variabile può cambiare tipo dati quando le si assegna un nuovo valore. È possibile pensare ad una variabile, quindi, come ad un'"etichetta" che è possibile spostare su diversi valori e che assume il tipo dati del valore a cui è assegnata in quel momento.

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

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

In Python il tipo dato `None` viene di norma usato ogniqualvolta si vuole rappresentare un valore non rilevante, ad esempio nei valori di default delle funzioni. Questo concetto è utile anche per controllare se una variabile è stata inizializzata o meno.

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.

Nota bene: i cancelletti aggiungono un commento: è buona prassi commentare il codice il più possibile.

Di seguito, vengono presentate due funzioni utili per comprendere gli esempi che verranno mostrati nei prossimi capitoli:

[1] Per un tutorial extra è possibile cliccare qui.

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

```
print('Hello, World!') #stampa 'Hello, World!'
```

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

```
x = 'there'
print('Hello, {}!'.format(x)) #stampa 'Hello, there!"
```

Sintassi 1 (Slicing). *È possibile effettuare lo* slicing *delle stringhe, 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
mia_string[-1] # l'ultimo carattere
```

Si può applicare lo slicing anche alle liste, che verranno presentate a breve.

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

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

Sintassi 2 (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 3 (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 4 (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 1.1. Nel seguente esempio, alcune liste in Python:

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

Le liste ci introducono agli operatori di appartenenza:

Sintassi 5 (Operatori 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.

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 1.2. Nel seguente esempio, alcuni dizionari in Python:

```
my_dict = {'Trieste': 34100, 'Padova': 35100} # diz. di numeri
my_dict = {'Trieste': 'TS', 'Padova': 'PD'} # diz. di stringhe

CAP_ts = my_dict['Trieste'] # a CAP_ts sarà assegnato il valore
                            # legato alla chiave 'Trieste' nel
                            # dizionario my_dict
```

Le chiavi devono essere di tipo *immutabile*[2], mentre i valori possono essere di un tipo dato qualsiasi. In più, è importante ricordare che i dizionari non sono mai ordinati nelle chiavi. È possibile usare gli operatori di appartenenza come per le liste per controllare se una chiave è contenuta nel dizionario. Non è possibile utilizzare gli operatori di appartenenza per i valori.

[2] Per approfondire il concetto di chiavi-valori è possibile fare riferimento a questo link.

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, lo scopo (concetto che verrà chiarito più avanti in questo capitolo) di queste istruzioni è legato all'*indentazione* del blocco.

Indentare significa aggiungere degli spazi prima di alcune righe. 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. Si noti che i commenti non hanno bisogno di essere indentati.

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 1.3. Nel seguente esempio, un costrutto if:

```
if (my_var > your_var):
    # indentazione per segnalare che queste
    # istruzioni fanno parte del blocco if
    print("My var is bigger than yours")
    if (my_var-your_var) <= 1:
        # ulteriore indentazione per il secondo if
        print("...but not so much")
```

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

Esempio 1.4. Nel seguente esempio, un costrutto if-elif-else:

```
if (my_var > your_var):
    print("My var is bigger than yours")
    if (my_var-your_var) <= 1:
        print("...but not so much")
    elif (my_var-your_var) <= 5:
        print("...quite a bit")
    # l'ultima condizione ha bisogno solo dell'else che identifica
    # tutti i casi che non sono ancora stati considerati
    else:
        print("...a lot")
```

Cicli I cicli in Python usano le classiche sintassi `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 1.5. Nei seguenti esempi, i costrutti `for` e `while`:

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

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 una lista di numeri nel seguente modo*[a]*:*

```
list_01 = range(10) # lista di numeri da 0 a 9
list_02 = range(3, 10) # lista di numeri da 3 a 9
list_03 = range(3, 10, 2) # lista di numeri da 3 a 9
                          # con passo 2: [3, 5, 7, 9]
```

[a]Anche in questo caso l'argomento passato alla funzione come estremo destro viene escluso.

Esempio 1.6. Un modo pythonico per *ciclare* sugli oggetti iterabili (come liste, stringhe e tuple), senza usare `range`, è il seguente:

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

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

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

[a]Per "spacchettare" una tupla la si assegna a più variabili, separandole con delle virgole, come fatto per ricavare `i` e `item` da `enumerate(mylist)`.

Funzioni Le *funzioni* in Python sono dichiarate usando la keyword `def`, indicando eventuali input tra parentesi, seguite dai due punti. Per le istruzioni si usa un blocco indentato, come già visto sia per le istruzioni condizionali che per i cicli. La keyword `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 1.7. Nei seguenti esempi, una funzione con return ed una senza:

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

```
mia_funzione('Pippo', 'Pluto') # chiamata della funzione
                               # 'Argomenti: Pippo e Pluto'

# funzione con return
def eleva_al_quadrato(numero):
    return numero*numero

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

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/-funzione ha il suo local scope). È il livello più basso nella gerarchia.
- Enclosed: scope speciale, ad esempio la funzione esterna nel caso di due funzioni una dentro l'altra, o anche il "corpo" esterno del programma.
- 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 malfunzionamenti nel codice [3].

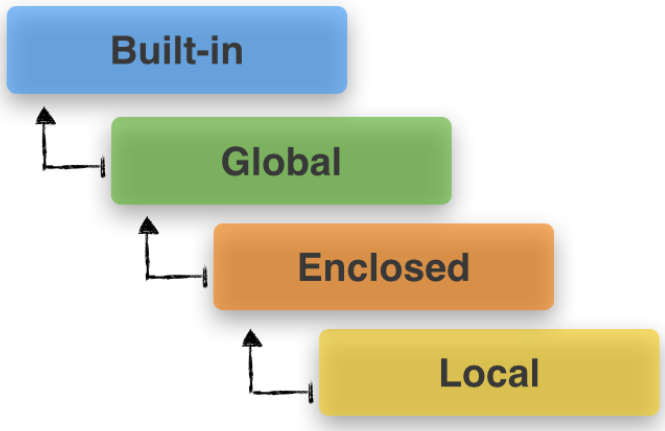

Figura 1. Struttura LEGB

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

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 tramite solo tramite l'input e il return.

Moduli Un *modulo* è un file che funge da libreria di funzioni. Le definizioni presenti in un modulo possono essere *importate*, usando la keyword `import`, ossia `import module`, 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. Si può importare un modulo sotto diverso alias usando la sintassi `import modulo as mod`, oppure importare direttamente una singola funzione con la sintassi `from modulo import function`. In quest'ultimo caso, il nome della funzione non deve essere preceduto da quello del modulo. I moduli non verrano usati durante il corso.

Esempio 1.8. Nei seguenti esempi sono presentati 3 modi per usare la stessa funzione `sqrt` dal modulo `math`:

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

import math as m
var = m.sqrt(4)    # var = 2

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

Essere pythonici Python offre una sintassi semplice e molti trucchetti "pythonici" per rendere il codice il più compatto ed efficiente possibile. L'obiettivo di Python è di rasentare lo pseudocodice, quindi dare allo sviluppatore gli strumenti per programmare in modo quasi discorsivo.

Esempio 1.9. Come accennato in precedenza, per iterare sugli elementi di una lista o, più in generale, di un oggetto iterabile è sufficiente usare il costrutto `for item in list`. Si può estendere questo concetto nel seguente modo. Si supponga di voler stampare ogni numero contenuto in un tensore cubico[a]:

```
tensor = [[[1,2], [3,4]], [[5,6], [7,8]]]
for matrix in tensor:
    for line in matrix:
        for number in line:
            print(number)
```

[a] Una lista di liste di liste.

In generale, il modo pythonico di scrivere un costrutto ne velocizza l'esecuzione, oltre che renderlo più intuitivo per il programmatore. È possibile trovare altri trucchi pythonici qui.

PART

III

Interagire con i File

Database Un *Database* permette di salvare dati in modo strutturato, come un archivio dati digitale. La creazione e il mantenimento di un database, però, hanno bisogno di un attento lavoro di design e gestione delle risorse, che può risultare eccessivo per compiti semplici.

File CSV Un *file* in formato *CSV*, ovvero "Comma-Separated Values", è un buon candidato per importare o esportare dati da un foglio elettronico o da un DB. È infatti possibile esportare un CSV direttamente da programmi come Excel. L'estensione di questi file è `.csv` o `.txt`; l'estensione `.doc` non è adeguata, in quanto il testo potrebbe non essere in formato `plain`[4].

[4]Un testo plain è senza formattazioni di alcun tipo.

In un CSV ogni riga è una entry ed è possibile inserire un'instestazione (opzionale) per dare un nome alle colonne. Durante il corso verrà usato il file `shampoo_sales.csv` (Figura 2) come riferimento.

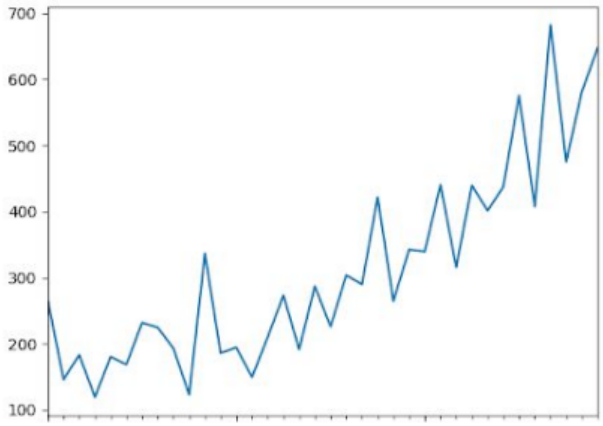

Figura 2. Plot di `shampoo_sales.csv`

File in Python Python mette a disposizione varie funzioni built-in per interagire con i file.

> **Funzione 4** (`open` e *close*)**.** *Per aprire un file in Python si usa la funzione* `open()`*, che ritorna un oggetto file. Per chiudere un file si usa la funzione* `close()`*. È necessario chiudere un file per liberare memoria e per salvare alcuni cambiamenti che potrebbero essere ancora in buffering.*

Gli argomenti da passare alla funzione sono il nome del file[5]ed una keyword per indicare se si vuole usare il file per lettura o scrittura. In particolare:

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

- ‘`r`’ indica modalità *read*;
- ‘`w`’ indica modalità *write*, si sovrascrive file;
- ‘`rb`’ indica modalità mista *read* e *write*;
- ‘`a`’ indica modalità *append* (le entry vengono aggiunte alla fine del file).

Aprire un file in modalità mista causa un uso della memoria superiore rispetto ad una modalità semplice.

Funzione 5 (`read` e `readline`)**.** *La funzione* ***`read()`*** *legge l'intero file e lo passa sotto forma di stringa. La funzione* ***`readline()`*** *legge una riga per volta e la passa sotto forma di stringa. Chiamando* ***`readline()`*** *più volte sullo stesso oggetto si andrà automaticamente alle righe successive.*

Esempio 1.10. Il seguente snippet di codice apre il file in modalità lettura, ne stampa i primi 50 caratteri usando lo *slicing* e lo chiude:

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

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

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

Il modo pythonico di interagire con il file e leggere una riga per volta è quello di sfruttare l'iterabilità dell'oggetto file, nel seguente modo:

```
my_file = open('shampoo_sales.txt', 'r')
for line in my_file:
    print(line)
my_file.close()
```

Si noti che Python riconosce il carattere che indica l'andare a capo e lo usa come criterio per dividere le righe.

Funzione 6 (`write`)**.** *Per scrivere su file, dopo averlo aperto in modalità 'w', si usa la funzione* ***`write()`****.*

CSV in Python Per leggere un CSV in Python è necessario introdurre la funzione `split()`.

Funzione 7 (`split`)**.** *La funzione* ***`.split()`*** *divide una riga quando incontra il carattere indicato tra parentesi e ritorna una lista contenente i vari "pezzi", ovvero gli elementi della entry del CSV.*

Gli oggetti letti da un CSV sono sempre stringhe, anche se rappresentano informazioni numeriche. Per farne uso potrebbe, quindi, essere necessaria

una conversione.

Sintassi 6 (Casting). *Per convertire un valore in Python si usa la sintassi di* casting, *in cui si indica il nome del tipo in cui convertire il valore:*

```
val = '9.1' # stringa, '9.1'
val = float(val) # float, 9.1
val = int(val) # int, 9
val = str(val) # stringa, '9'
```

Per aggiungere le informazioni lette dal CSV ad una lista si possono seguire due strade. Il modo *naive* di procedere sarebbe quello di inizializzare una lista lunga quanto il numero di elementi da leggere. Questo comporta dover sapere a priori quanti elementi saranno presenti, e ciò non è sempre possibile. Per ovviare a questo problema si può usare la funzione `append()`.

Funzione 8 (`append`). *La funzione* `.append()` *aggiunge un elemento alla fine della lista, modificandone la lunghezza:*

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

Si hanno ora tutti gli strumenti per leggere informazioni da un file CSV e trascriverle su una lista.

Esempio 1.11. Nel seguente esempio, si legge da shampoo_sales.csv e si passano le informazioni a due liste `date` e `value`:

```
# Inizializzo una lista vuota per salvare i valori
values = []
date = []
# Apro e leggo il file, linea per linea
my_file = open( 'shampoo_sales.csv' , 'r' )
    for line in my_file:
    # Faccio lo split di ogni riga sulla virgola
    elements = line.split(',')
    # Se NON sto processando l'intestazione...
    if elements[0] != 'Date':
        # Setto la data e il valore
        date = elements[0]
        value = elements[1]
        # Aggiungo alla lista dei valori questo valore
        date.append(date)
        values.append(float(value))
```

PART

IV

Gli Oggetti in Python

La *Programmazione ad Oggetti* (*OOB*) è un paradigma di programmazione che permette di ragionare in termini di *oggetti*, e non solo di funzioni con input/output.

Oggetti e Classi Una classe è una generica definizione di un oggetto, ovvero uno schema che permette di generare un elemento con le caratteristiche desiderate. Un oggetto è un'istanza di una *classe*. Le funzioni relative ad una classe si chiamano *metodi*, le variabili si chiamano *attributi*. Può esistere un numero illimitato di istanze di una classe.

Le classi sono utili per definire in modo sensato ed intuitivo delle categorie di oggetti con caratteristiche ben definite. Ad esempio, una classe `Persona` può avere un attributo `nome` e un metodo `saluta()`. Se si volesse ricordare 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 oggetto logico, come una persona.

Di norma, si usa il *camelcase* (es. `MyClass`) per i nomi delle classi e lo *snakecase* (es. `my_method`) per i nomi dei metodi e delle variabili.

Funzione 9 (`dir` e `type`)**.** *La funzione* `dir()` *ritorna una lista con tutti gli attributi di un oggetto, mentre la funzione* `type()` *ritorna la classe dell'oggetto.*

È importante ricordare che in Python **tutto è un oggetto**, quindi tutto è un'istanza di una classe ed è sempre possibile esplorarne le caratteristiche con le funzioni appena presentate. Si pensi all'esempio del casting: si sta creando un oggetto del tipo indicato usando il costruttore di quella classe.

Un'operazione *in-place* modifica l'oggetto ritornando `None`.

Esempio 1.12. Ecco due esempi di funzioni in-place e non:

```
my_string = 'a,b,c'
print(my_string.split(',')) # ['a','b','c']
print(my_string)            # 'a,b,c', my_string invariata

my_list = [1,2,3]
print(my_list.reverse()) # [3,2,1]
```

```
print(my_list)            # [3,2,1], my_list è stata modificata
```

È sconsigliato l'utilizzo di operazioni in-place perché si può perdere in termini di interpretabilità.

Definire una classe Per definire una classe in Python si usa la keyword `class`. Il *costruttore*, 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 agire su un attributo della classe si usa la keyword `self`, che rappresenta l'istanza, seguita dal nome dell'attributo.

Esempio 1.13. 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, mi chiamo {} {}".format(self.nome, self.cognome))
```

Per definire una classe vuota si usa la keyword `pass` dopo la dichiarazione della classe.

Per istanziare un oggetto da una classe lo si pone uguale al nome della classe con eventuali attributi per l'inizializzazione. Per accedere ad uno dei suoi attributi o metodi si usa la dot notation.

Esempio 1.14. Creazione ed accesso alle proprietà di un oggetto della classe `Persona`:

```
# persona_1 è un'istanza di persona
persona_1 = Persona('Mario', 'Rossi')
print(persona_1.nome) # 'Mario'
persona_1.saluta()    # 'Ciao, mi chiamo Mario Rossi'
```

Ereditarietà Negli OOB esiste il concetto di *ereditarietà*. Essa 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 che non appartengono alla classe genitore.

Possiamo interpretare le gerarchie tra classi in termini di insiemi e relazioni di inclusione ed intersezione:

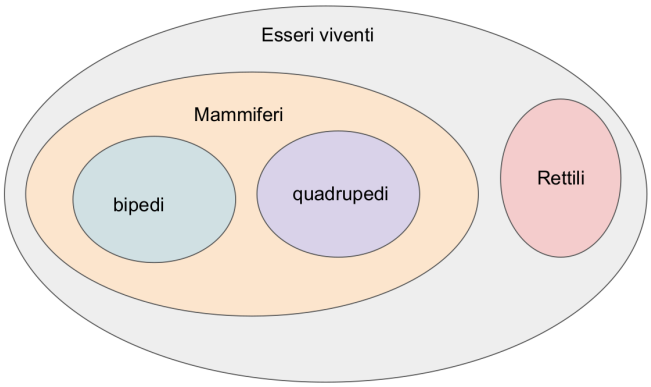

Figura 3. Esempio di gerarchia

Una classe figlia si definisce mettendo tra parentesi il nome della classe genitore.

Esempio 1.15. Creazione di una classe `Studente` che eredità da Persona:

```
class Studente(Persona):
    # Si estende l'oggetto Persona declinandolo in Studente.
    # Tutti i metodi che possedeva l'oggetto Persona sono
    # automaticamente ereditati dagli oggetti Studente.
    def __str__(self):
        return 'Studente "{} {}"'.format(self.nome, self.cognome)
```

Funzione 10 (`super`). *La funzione* `.super()` *chiama il metodo della classe padre su cui è chiamato.*

Esempio 1.16. 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 ricrea il metodo "saluta" originale usando .super()
    def saluta_originale(self):
        saluta.super()
```

PART V

Eccezioni e Flusso Try-Except

Gli errori in Python sono chiamati *eccezioni*, ovvero degli oggetti che estendono la classe base *exception* (Figura 7).

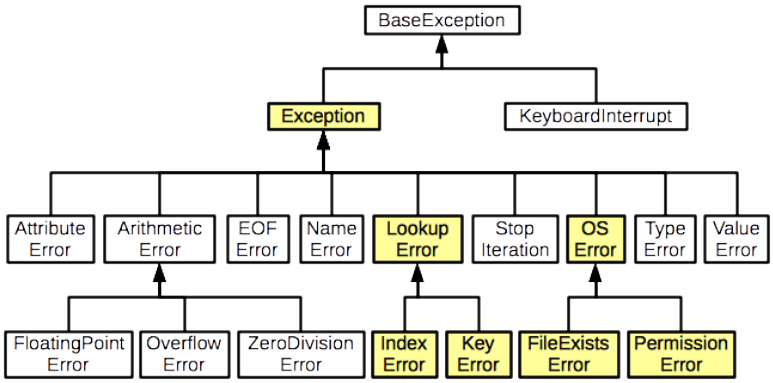

Figura 4. Gerarchia delle eccezioni

Alcune eccezioni utili da tenere a mente sono:

- `ArithmeticError`: problemi nel calcolo numerico;
- `AttributeError`: l'attributo dell'oggetto non esiste;
- `NameError`: variabile non definita;
- `ValueError`: la funzione riceve un valore inappropriato;
- `SyntaxError`: il codice non è semanticamente corretto;
- `TypeError`: la funzione non può essere applicata a quel tipo dato.

Il *traceback* è un report automatico di Python che serve ad identificare dove è avvenuto un errore. Quello che fa è proprio *rintracciare all'indietro* le chiamate delle funzioni in cui si è verificato l'errore. L'ultima riga mostra l'errore, mentre le righe successive mostrano le chiamate delle funzioni che hanno portato alla funzione con l'errore (Figura 5.

```
>>> a = (1, 2, 3)
>>> last = get_last_item(a)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in get_last_item
IndexError: tuple index out of range
```

Figura 5. Un `traceback`

Costrutto try-except Durante l'esecuzione del codice possono verificarsi delle eccezioni ed il costrutto *try-except* serve a gestirle e far fluire l'esecuzione senza interruzioni. Grazie a questo controllo si evita la propagazione di un errore nel codice.

La struttura *try* prova ad eseguire il blocco di istruzioni indentato:

- se non ci sono errori lo esegue, poi salta alla prima istruzione successiva alla struttura;
- se si presenta un errore, l'istruzione try verifica se il codice di errore corrisponde a quello documentato dopo la clausola except:
  - se corrisponde, esegue la relativa istruzione indentata, poi salta alla prima istruzione successiva alla struttura try-except, senza bloccare l'esecuzione del programma;
  - se non corrisponde, l'interprete Python va in errore e blocca l'esecuzione del programma.

Esempio 1.17. Di seguito, un esempio del funzionamento del costrutto:

```
my_var = 'ciao'

try:
    my_var = float(my_var)
except:
    print('Non posso convertire "my_var" a valore numerico!)
    print('Uso il valore di default "0.0" per "my_var")
    my_var = 0.0

print(my_var) # 0.0
```

Nello snippet appena visto l'eccezione `ValueError`, lanciata quando si prova a convertire una stringa alfabetica a float, viene catturata e gestita nel blocco except. Si noti che non è stato specificato il tipo di eccezione da catturare, quindi Python è pronto a catturarne una qualsiasi.

Esempio 1.18. Nel seguente snippet, si chiede a Python di gestire l'errore in modo diverso a seconda dell'eccezione:

```
try:
    my_var = float(my_var)
except ValueError:
    print('Non posso convertire "my_var" a valore numerico!')
    print('Ho avuto un errore di VALORE. "my_var" valeva "{}".'.format(my_var))
except TypeError:
    print('Non posso convertire "my_var" a valore numerico!')
    print('Ho avuto un errore di TIPO. "my_var" era di tipo "{}".'.format(type(my_var)))
except Exception as e:
    print('Non posso convertire "my_var" a valore numerico!')
    print('Ho avuto un errore generico: "{}".'.format(e))
```

La keyword `Exception` intercetta tutti i tipi di eccezione. È stato printato il contenuto usando il metodo `__str__` della classe `exception`, collegato alla funzione `print()`.

È importante notare che il costrutto try-except si comporta come l'if-else, ovvero che appena un blocco intercetta l'eccezione, i successivi vengono saltati. È possibile *annidare* dei try-except.

PART

VI

Gestione degli input

Un *input* è un dato che viene immesso in una struttura logica. Una funzione riceve degli argomenti, la lettura di un file porta dei dati come input al programma e anche la scrittura dell'utente su linea di comando può essere richiesta ed usata come informazione. In generale, un input può essere sia quello che è già presente nel codice che quello che viene passato dall'esterno.

Fidarsi degli input Non ci si deve fidare della bontà degli input, perché non si ha il controllo su chi usa la funzione. Se i dati sono fuori range, danneggiati, in formato sbagliato o in generale non adeguati alla struttura che ne deve fare uso, il programma può andare in *crash* oppure dare risultati sbagliati. Per questo, è necessario controllare che siano corretti.

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 cosa farci).

Ogni situazione necessita dei test adatti che coprano tutte le possibili falle. È utile non pensare solo agli *edge cases*[6], ovvero ai casi ideali in cui tutto va come sperato, ma prepararsi a tutte le eventuali imperfezioni in entrata.

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

Operatori di test Per testare l'input si usano gli operatori di confronto classici, gli operatori di identità e gli operatori di membership.

Sintassi 7 (Operatori di identità)**.** *Gli operatori* `is` *e* `is not` *permettono di controllare che la variabile testata abbia un valore non nullo:*

```
print(var is None)      # True se var ha valore None
print(var is not None)  # True se var non ha valore None
print(var == '')        # True se var è una stringa vuota
```

Sintassi 8 (Operatori di membership)**.** *Gli operatori* `in` *e* `not in` *permettono di controllare che il valore della variabile sia presente all'interno di una sequenza come una lista, stringa, tupla o dizionario:*

```python
# True se età ha un valore tra i 30 e i 39
print(età in range(30, 40))

# True se la stringa lettera non è presente in 'parola'
print(lettera not in 'parola')
```

Un altro modo per testare la correttezza dell'input è quello di verificarne la classe usando delle funzioni apposite.

Funzione 11 (Funzioni legate alla classe)**.** *La funzione* `isinstance()` *permette di controllare se un oggetto è istanza di una classe. La funzione* `issubclass()` *verifica se una classe è sottoclasse di un'altra, ovvero se l'ha ereditata.*

```python
# True se nome_variabile è istanza di Classe
print(isinstance(nome_variabile, Classe))

# True se SottoClasse eredita SuperClasse
print(issubclass(SuperClasse, SottoClasse))
```

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 ed 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 ed 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.

Lanciare un'eccezione Se si vuole lanciare un'eccezione quando si incontra un errore e bloccare l'esecuzione del programma non basta usare il costrutto try-except, in quanto l'esecuzione continuerebbe dopo l'intercettazione. Per fare ciò si introduce l'operatore `raise`.

Sintassi 9 (`raise`). *L'operatore* `raise` *consente di lanciare un'eccezione built-in o custom seguita da un messaggio personalizzato:*

```
# Si lancia un'eccezione generica con un messaggio
raise Exception('Messaggio di errore')

# Si lancia un'eccezione generica con un messaggio contenente
# un'informazione sulla variabile che lo ha generato
raise Exception('Ho avuto un errore, ecco il parametro che
                lo ha generato: "{}"'.format(parametro))

# Si crea l'eccezione custom definendola come classe
class ErroreAIDA(Exception)
    pass

# Si lancia un'eccezione custom
raise ErroreAIDA('Messaggio di errore')
```

Sanitizzazione degli input È possibile provare a pulire o *sanitizzare* gli input quando sono in una forma simile ad una forma utile.

Esempio 1.19. Nel seguente esempio si vuole controllare se una stringa `sesso` appartiene ad una lista contenente `['M','F']`:

```
# Si presenta un input con una lettera minuscola
# ed uno spazio
sesso = 'm '

# Si trasforma l'input in maiuscolo
sesso = sesso.upper()

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

print(sesso in ['M', 'F']) # True
```

Dall'input sono stati rimossi eventuali errori di formattazione per intercettare l'informazione collegata alla stringa `'M'`.

Questo lavoro può essere fatto con ogni tipo di input a patto di uno studio a priori dei possibili problemi.

PART VII

II Testing

Il *testing* è una parte integrata del lavoro di progettazione di qualsiasi prodotto. Tutto ciò che viene mandato in produzione subisce prima una fase di testing in cui si verifica il corretto funzionamento del tutto o delle parti in

varie condizioni di interesse.

In programmazione il testing *end-to-end* indica un test automatico che determina se l'intero codice funziona nel modo corretto, mentre lo *Unit Testing* è legato al testing di un'unità minima del programma.

Esempio 1.20. Il seguente *pseudocodice* mostra il funzionamento di un test:

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

Si confronta l'output con uno precalcolato che si considera corretto. Se i due non corrispondono, si lancia un errore.

Testing in Python Un testing *naive* si fa usando un controllo condizionale e l'operatore `raise`.

Esempio 1.21. Esempio di testing *naive*:

```python
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`. Una buona fase di testing considera tutte le possibili varianti degli input.

Python offre anche un modulo nativo per il testing.

Sintassi 10 (`unittest`). *Il modulo* ***`unittest`*** *si importa in un file il cui nome inizia con* ***`test_`***. *Lo si usa per verificare da linea di comando quanti e quali test hanno successo. È necessario anche importare le funzioni da provare, con la seguente sintassi:*

```python
import unittest

# Si suppone che la funzione 'somma'
# sia stata definita nel file 'mathfun.py'
from mathfun import somma
# Testing
class TestSomma(unittest.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* `assertRaises()`*, per verificare che una particolare eccezione viene lanciata.*
Per lanciare un processo di test 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 query.*

Codice Test-Driven Un paradigma di programmazione possibile è quello *Test-Driven*, in cui si scrivono prima i test e poi si costruisce il codice attorno.

PART

VIII

Creare un Modello

Un *modello* è un'ipotesi di comportamento del mondo reale. Ad esempio, si riesce a lanciare un oggetto e prevedere dove cadrà grazie ad un modello fisico noto come *moto parabolico*. In generale, un *modello fisico* è una rappresentazione concettuale di un fenomeno che ne spiega il funzionamento, ma non la causa. Un *modello fisico* viene quantificato grazie ad un modello matematico che dà gli strumenti per effettuare misurazioni e previsioni.

Modello Statistico Un *modello statistico* è un modello matematico che incarna un insieme di ipotesi statistiche riguardanti la generazione di dati campione. Per comprendere bene questo concetto, si pensi a cosa succede quando vengono collezionati dei dati: un meccanismo probabilistico nascosto li genera con una certa distribuzione e se ne raccolgono alcuni. Successivamente, si prova a fare un'*inferenza*, ovvero partire dai dati per approssimare il modello nascosto.

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

È necessaria una *metrica* per determinare quanto un modello approssima i bene i dati che si hanno.

Definizione 1.1 (Errore)**.** L'*errore* è una metrica definita nel seguente modo:

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

dove n è il numero di osservazioni, y_i è l'i-esima osservazione ed $f(x_i)$ è

la previsione del modello legata all'i-esima osservazione.

Si può creare un modello anche senza usare i dati, ipotizzandolo a priori e verificandone la bontà.

Plottare un modello significa graficarlo e mostrarne l'andamento, eventualmente sovrapponendolo alle osservazioni.

Un Modello Semplice I modelli statistici, in genere, si basano su *equazioni differenziali* [7] ma per questo corso ci si concentrerà su modelli più semplici. Si noti che molti dei modelli di Machine Learning e Deep Learning sono definiti *Black Box*: sono utilizzabili senza comprendere perfettamente i meccanismi al loro interno.

Si prenda come esempio il dataset `shampoo_sales.txt`. Un modello semplice per questi dati è uno tale per cui le vendite al tempo `t+1` sono date dall'incremento medio[8] sugli `n` mesi precedenti applicati al tempo `t`.

Esempio 1.22. In Python si può costruire un modello usando una classe base `Model`, nel seguente modo:

```
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')
```

Si può poi creare un classe figlia `IncrementModel` che eredita `Model` e definisce il metodo `predict`:

```
class IncrementModel(Model)
    # data è una lista di valori per gli n mesi passati
    def predict(self, data):
        for item in data:
            # Logica per la predizione
            ...
            prediction = ...
        return prediction
```

[7] Il modello base che descrive l'andamento di un'epidemia, ad esempio, è chiamato *modello SIR* ed è basato su un'equazione differenziale che considerea l'andamento degli individui *Susceptible*, *Infected* e *Recovered*.

[8] L'*incremento medio* è la media delle differenze tra le vendite al tempo `t` e le vendite al tempo `t-1`, per `t` che va da 2 alle vendite totali.

PART IX

Fittare un Modello

Quando si parte da un modello statistico con coefficienti e li si stima per approssimare al meglio i dati, si parla di *fitting* del modello.

Vediamo, come esempio, due modelli possibili per il dataset `shampoo_sales.txt`: lineare ed esponenziale.

Definizione 1.2 (Modello Lineare)**.** Un modello *lineare* ha la forma:

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

Definizione 1.3 (Modello Esponenziale)**.** Un modello *esponenziale* ha la forma:

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

Per fittare si cercano i coefficienti che minimizzino la distanza (l'errore) tra il modello e i dati. In Python si può fare ciò usando la libreria `scikit-learn` oppure modificando il metodo `fit` della classe creata in predenza.

Fittando i modelli sul dataset si ottengono i seguenti grafici:

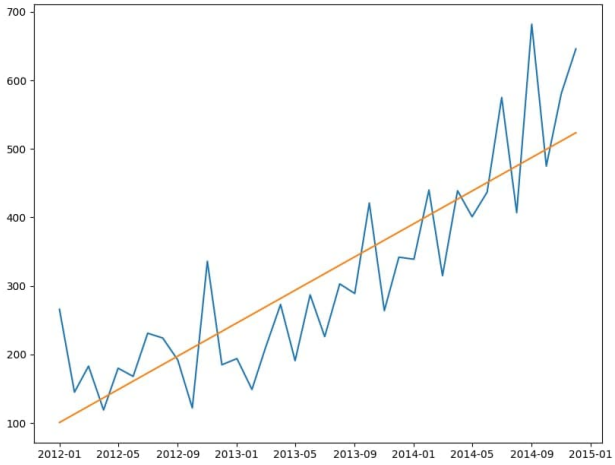

Figura 6. Coefficienti: $a = 3.97 \cdot 10^{-1}, b = -5.98 \cdot 10^{3}$

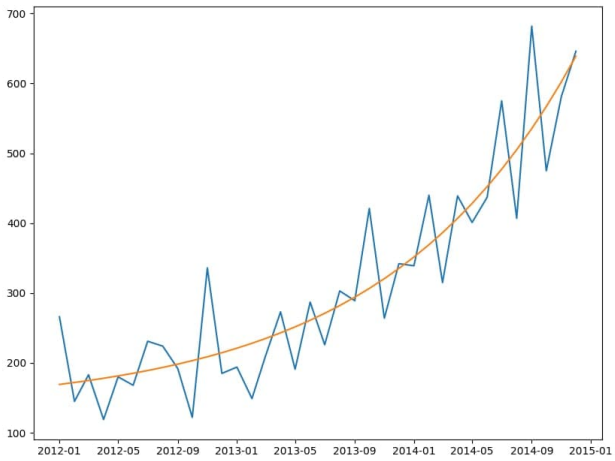

Figura 7. Coefficienti: $a = 4.65 \cdot 10^{-16}, b = 2.53 \cdot 10^{-03}, c = 1.35 \cdot 10^{2}$

Gli *RMSE*[9] sono:

$$E_{lin} = 76.3$$
$$E_{exp} = 60.9$$

Si potrebbe quindi dire che il modello esponenziale fitta meglio i dati presentati.

La classe `FitIncrementModel` Si può estendere la classe `IncrementModel` per includere il metodo `fit()`. Un modo semplice per creare un fit, quindi aderire ai dati precedenti, è prevedere le vendite al tempo `t+1` come l'incremento medio negli `n` mesi precedenti mediato con l'incremento medio su tutto il resto del dataset, applicato alle vendite al tempo `t-1`.

[9] L'RMSE è lo scarto quadratico medio: $RMSE = \sqrt{\sum_{i=1}^{n} \frac{(y_i - f(x_i))^2}{n}}$

PART X

Valutare un Modello

Per valutare un modello si divide il dataset in due (talvolta tre) parti:

- Fit/Training set: parte del dataset usata per fittare il modello;
- Evaluation/Test set: parte del dataset su cui calcolare il valore della metrica di valutazione.

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 aderisce bene a questi dati allora si avrà una buona intuizione di come funziona il fenomeno sottostante.

Se il modello si comporta bene sui dati di training ma non ha una buona performance sul dataset di test, ci si trova in una condizione di *overfitting*. Il modello non ha cercato di aderire al fenomeno sottostante ma ha attribuito troppa importanza alle informazioni dei dati che è riuscito a vedere, che normalmente portano una componente di errore. Al contrario, quando il modello aderisce troppo poco ai dati si parla di *underfitting* (Figura 8).

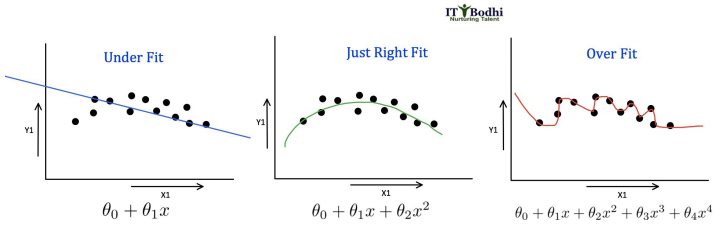

Figura 8. Possibili risultati di un fit

Talvolta, il dataset di test viene affiancato da un dataset di *validazione*. Ciò avviene quando il processo di fit ha bisogno di più valutazioni intermedie:

se ne verifica la performance parziale sul dataset di test e quella globale sul dataset di validazione, ancora non osservato.

Plottando gli errori di training e test in funzione di qualche parametro che indica la complessità del modello, come il grado del polinomio approssimante, si trova un minimo dell'errore sul dataset di test in corrispondenza della complessità migliore per il modello. In Figura 9 è possibile vedere la zona di underfitting e quella di overfitting, rispettivamente a sinistra e a destra della migliore complessità.

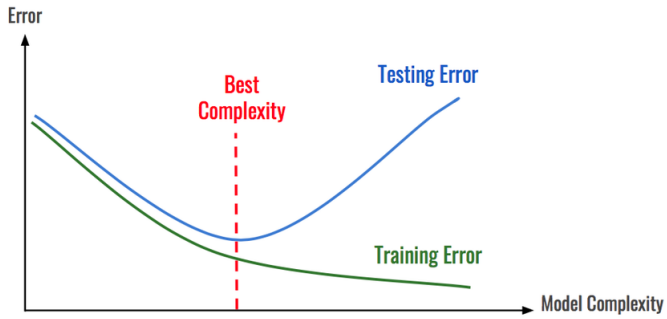

Figura 9. Training/Test error

Una *rule of thumb* per la divisione del dataset è quella di allocarne il $70-80\%$ per il training e il restante per il test.