

Introduzione a Python

Programmazione di Applicazioni Data Intensive

Laurea in Ingegneria e Scienze Informatiche
DISI – Università di Bologna, Cesena

Proff. Gianluca Moro, Roberto Pasolini
nome.cognome@unibo.it



Outline

-
- Introduzione e versioni di Python
 - Sintassi di base
 - Tipi di dati di base (numeri, booleani, stringhe)
 - Collezioni (liste, tuple, insiemi, dizionari)
 - Costrutti if, while, for
 - Definizione di funzioni
 - Moduli e package
 - Installazione di librerie e ambienti virtuali
 - Lettura e scrittura di file
 - Funzioni e moduli utili della libreria standard



Python

- Linguaggio interpretato cross-platform
 - disponibile per i principali SO (Linux, Mac, Windows, ...)
 - un'implementazione di riferimento (*CPython*) più altre alternative
 - integrabile in altri linguaggi (C, C++, Java, ...)
- Creato alla fine degli anni '80, divenuto popolare nei 2000
- Multi-paradigma
 - imperativo, object-oriented, funzionale
 - sintassi facilmente estendibile ad altri paradigmi
- Enfasi sulla facilità di lettura e scrittura del codice
 - “there should be one—and preferably only one—obvious way to do it”

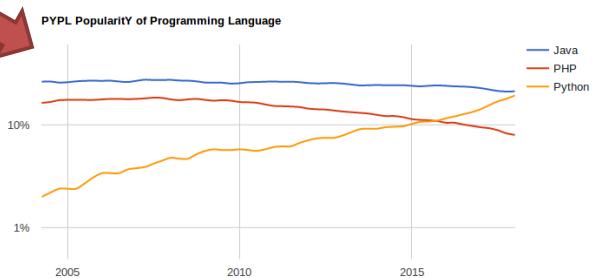
Data Intensive Applications - G. Moro, R. Pasolini - UNIBO, DISI (Cesena)

3



Perché Python?

- Linguaggio *general-purpose*, usato per molteplici scopi
 - scripting, sviluppo Web, data science, ...
- Facile da imparare
- Usato per prototipazione e cicli di sviluppo rapidi
- **Popolarità in rapido aumento**
negli ultimi anni
- Include una libreria standard di
molte funzioni di uso comune
- Ampia disponibilità di
librerie esterne



Trend ricerche su Google di tutorial sui linguaggi (da <http://pypl.github.io>)



Python 2 e Python 3

- Sono diffuse due diverse versioni *major* di Python
- Su **Python 2** si basa molto software tutt'ora in uso
 - usato di default in molte distribuzioni Linux (es. Ubuntu) e in Mac OS X
 - l'ultima versione *minor* prevista è la 2.7, rilasciata nel 2010
 - il termine del supporto è previsto nel 2020
- **Python 3** introduce novità incompatibili con Python 2
 - prima release nel 2008, ultima versione minor 3.7 del 2018
- Molte librerie di uso comune sono state (ri)sritte per funzionare con entrambe le versioni

Nel corso useremo Python 3



Uso di Python

- È possibile eseguire uno script Python contenuto in un file `.py`
 - con un doppio clic sul file corrispondente (se abilitato nel SO)
 - da linea di comando scrivendo: `python nomeFile.py`
 - in Linux/Mac usare “`python3`” per distinguere da Python 2
- In più, Python può essere eseguito in modalità interattiva
 - l'utente digita istruzioni una ad una, l'interprete le valuta e stampa il valore che viene restituito
 - modalità nota in generale come *REPL* (*Read-Eval-Print Loop*)
- Per avviare l'interprete in questa modalità
 - Linux/Mac: comando `python3` senza specificare un file
 - Windows: Start → Python 3.x → IDLE
- Per uscire dall'interprete digitare `exit()` o (Linux/Mac) Ctrl+D



Sintassi di Base

- Un'istruzione Python è contenuta di default in una riga

```
print("Hello, world")
```

- Si possono scrivere più istruzioni in riga separate con ";"

```
print("Hello"); print("world")
```

- I commenti sono introdotti da "#" e finiscono a fine riga

```
# Questo è un commento
```

```
print("Hello, world") # altro commento
```

commenti multiriga con 3 doppi apice """ in chiusura e apertura commento

- Si può far continuare un'istruzione in una riga successiva

- esplicitamente se la riga termina in "\\"

- implicitamente se ci sono parentesi non chiuse (più comune)

```
print("Hello, "
      + "world")
```

Data Intensive Applications - G. Moro, R. Pasolini - UNIBO, DISI (Cesena)

7



Sintassi di Base: Blocchi di Codice Indentati

- In altri linguaggi i blocchi di codice (if, for, ...) sono delimitati da simboli specifici (spesso "{}")

- l'indentazione è usata convenzionalmente per migliore leggibilità

- Python usa l'indentazione come sintassi per i blocchi

- ogni riga che introduce un blocco (es. if) termina in ":"

- le righe a pari livello sono indentate con pari numero di spazi

- per indicare un blocco vuoto si usa la parola chiave "pass"

```
// esempio in Java
```

```
nums = getNumbers();
for (int x: nums) {
>   if (x < 0) {
>     System.out.println(x);
>   }
}
```

```
System.out.println("end");
```

```
# esempio in Python
```

```
nums = get_numbers()
for x in nums:
>   if x < 0:
>     println(x)
println("end")
```

Data Intensive Applications - G. Moro, R. Pasolini - UNIBO, DISI (Cesena)

8



Oggetti, Tipi e Variabili in Python

- Python è **puramente object-oriented**: **ogni costrutto** è un oggetto (e.g. numeri, stringhe, liste, funzioni, ...) – es.: in Java valori di tipo primitivo (`int`, `float`, ...) **non** sono oggetti
- Ogni oggetto ha per **tipo** una determinata *classe*, che determina quali operazioni si possono compiere su di esso
- Esiste un oggetto predefinito `None` (di tipo `NoneType`), usato per indicare **assenza di valore**
 - simile a `null` in Java (che però è un valore, *non* un oggetto)
- Le **variabili** in Python sono sempre **riferimenti ad oggetti**
 - come in Java, più variabili si possono riferire al medesimo oggetto



Assegnare Valori a Variabili

- Una **variabile** è **creata al primo assegnamento di un valore**
 - non serve dichiararla esplicitamente come in Java o C
- Per l'assegnamento si usa la tipica sintassi “`nome = valore`”


```
>>> foo = 21*2    # assegno un valore alla variabile
>>> foo          # ottengo il valore della variabile
42
```
- Le **variabili non hanno tipo**: è possibile sostituire un valore assegnato ad una variabile con uno di tipo diverso
- Si può **eliminare** una variabile con l'istruzione `del`

```
>>> del foo
      – l'accesso ad una variabile non definita o eliminata solleva un errore
```
- Per convenzione, le variabili Python hanno nomi in minuscolo con underscore “`_`” al posto degli spazi, es. “`file_name`”



Valori Numerici

- Python supporta numeri interi (tipo `int`) e frazionari (tipo `float`)
- Sono supportate le ordinarie operazioni aritmetiche: `+` `-` `*` `/`
 - operazioni tra `int` e `float` restituiscono sempre `float`
 - la divisione con `/` da sempre `float` (*non in Python 2!*)
- Altri operatori:
 - `%`: modulo (resto della divisione)
 - `//`: divisione intera (arrot. per difetto)
 - `**`: elevamento a potenza

```
>>> 2 + 2
4
>>> (4-1.5) * 4
10.0
>>> 6 / 3
2.0
>>> 13 // 5
2
>>> 13 % 5
3
>>> 2 ** 8
256
```



Operatori di Assegnamento

- Anche in Python esistono gli operatori di assegnamento “esteso” per **cambiare il valore di una variabile col risultato di un’operazione**
 - esistono operatori di assegnamento per tutti gli operatori binari
 - sono utilizzabili con **qualsiasi oggetto** supporti l’operazione usata
 - non esistono però gli operatori `++` e `--` presenti in Java

```
>>> x = 5
>>> x += 2    # x = x+2
>>> x
7
>>> y = 123456
>>> y %= 100
>>> y
56
```



Valori Logici (Booleani)

- Python definisce le costanti `True` e `False` di tipo `bool`
 - notare l'iniziale maiuscola!
- Sono il risultato di operazioni di comparazione `== != > < >= <=`
 - le comparazioni si possono concatenare
- Si possono applicare ad essi gli operatori logici `and`, `or` e `not`
- `bool` è sottotipo di `int`: se trattati come numeri, `True` vale 1 e `False` 0

```
>>> 2 == 2
True
>>> 3 != 3
False
>>> 0 <= 42 <= 100
True
```

`0 <= 42 and 42 <= 100`

```
>>> not True
False
>>> True and False
False
>>> 3 * (4 > 2)
3
```

`= True = 1`



Stringhe

- Una stringa (tipo `str`) è delimitata da apici singoli o doppi
 - è equivalente scrivere `'hello'` o `"hello"`
- Si possono usare le tipiche sequenze di `escape` con “\”
 - `\n` = a capo, `\t` = tabulazione, `\\"` = un carattere “\”, ...

```
>>> print("Hello\nworld")
```

Hello
world

- Per scrivere una stringa su più righe, delimitarla con tre apici

```
>>> print("""Hello,
... world!""")
```

Hello,
world!



Funzioni

- Una **funzione** è un oggetto che può essere *chiamato*, usando la tipica sintassi “`nome(argomento1, argomento2, ...)`”
 - Python definisce diverse funzioni standard di uso comune
 - Tra queste `print` stampa uno o più oggetti in output
- ```
>>> print("Hello, world!")
Hello, world!
```
- ```
>>> print("Hello,", "world", "!")
Hello, world !
```
- *in Python 2 print è un’istruzione del linguaggio, non usa le parentesi*

- `input` restituisce una stringa digitata dall’utente
 - utile negli script piuttosto che in modalità interattiva
- ```
user_name = input("What's your name? ")
```



# Oggetti: Attributi e Metodi

- Ogni oggetto ha degli *attributi* e dei *metodi*, paragonabili a variabili e funzioni definite localmente all’oggetto stesso
  - analoghi a *campi* e *metodi* in Java
- Attributi e metodi di un oggetto sono *definiti dalla sua classe*
  - ma possono anche essere aggiunti dinamicamente ai singoli oggetti
- Per accedere ad un attributo (o metodo) di un oggetto, si usa la tipica sintassi “`oggetto.attributo`”
- Ad esempio su una stringa si può invocare il metodo `upper` per ottenerne una copia con tutte le lettere maiuscole

```
>>> "Hello!".upper()
'HELLO!'
```



# Oggetti Immutabili e Mutabili

- Oggetti quali numeri, stringhe, ecc. rappresentano dei *valori*, in modo simile ai tipi primitivi in Java
- Oggetti come questi sono *immutabili*: non hanno uno stato interno che possa essere cambiato
 

```
>>> x = 40 # si ha un oggetto che rappresenta il 40
>>> x += 2 # si ottiene un nuovo oggetto per il 42
```
- Oggetti con uno stato che si può cambiare sono invece *mutabili*
- In chiamate a funzioni gli oggetti sono *passati per riferimento*
  - per oggetti immutabili questo è equivalente al passarli per valore
  - gli oggetti mutabili invece *possono essere modificati* dalla funzione



# Collezioni

- Python definisce diversi tipi di *collezioni* di oggetti
  - una collezione può contenere *oggetti di tipi eterogenei*
  - le collezioni *solo* oggetti a loro volta, si possono quindi innestare
- Le collezioni si dividono in *mutabili e immutabili*
  - solo nelle collezioni mutabili è possibile aggiungere, rimuovere e sostituire elementi
  - NB: una collezione immutabile non costituisce un oggetto immutabile se contiene oggetti mutabili!
- Le collezioni includono anche le *stringhe (str)*, che sono sequenze *immutabili* di caratteri
  - un carattere è una stringa lunga 1, non c'è un tipo di dato apposito
- Python fornisce *funzionalità comuni* a tutti i tipi di collezioni



# Liste e Tuple

- Una *lista* (tipo `list`) è una **sequenza mutabile** di oggetti
  - si crea indicando gli oggetti tra parentesi quadre, separati da virgole

```
>>> names = ["Alice", "Bob", "Carol"]
```

```
>>> matrix = [[1, 0, 0], [0, 1, 0], [0, 0, 1]]
```

```
>>> nothing = []
```
- Una *tupla* (tipo `tuple`) è una **sequenza immutabile** di oggetti
  - si crea indicando gli oggetti separati da virgole
  - è comune (a volte necessario) racchiudere le tuple tra parentesi tonde
  - tupla con un solo elemento x: “`x,`”, tupla vuota: “`()`”

```
>>> x = (42, True) # parentesi opzionali
```

```
>>> single = (8,) # virgola necessaria!
```

```
>>> nothing = ()
```
- Iterare una *tupla* è più rapido di iterare una *lista*

Data Intensive Applications - G. Moro, R. Pasolini - UNIBO, DISI (Cesena)

19



# Insiemi

- Gli oggetti `set` sono **insiemi mutabili** di oggetti **immutabili**
  - possono contenere ad es. numeri, stringhe e tuple, ma non liste
  - ogni oggetto può apparire **una sola volta** in un insieme e **non ha una posizione** specifica al suo interno
  - per creare un insieme con alcuni elementi si usano le parentesi graffe
  - per creare un insieme vuoto bisogna scrivere “`set()`”

```
>>> fruits = {"apple", "banana", "orange"}
```

```
>>> nothing = set()
```
- I `frozenset` sono la versione **immutabile** dei `set`
  - si creano con l'omonima funzione, passando una collezione

```
>>> fruit = frozenset(["apple", "banana", "orange"])
```

Data Intensive Applications - G. Moro, R. Pasolini - UNIBO, DISI (Cesena)

20



# Operazioni sulle Collezioni

- La funzione `len` dà il **numero di elementi** in una collezione

```
>>> len("Hello") # numero di caratteri
5
>>> len([-12.345, "Hello", False]) # num. oggetti
3
```

- L'operatore `in` verifica la presenza di un elemento

- esiste anche la versione negata `not in`
- nelle stringhe si possono cercare anche sequenze di più caratteri

```
>>> "B" in ("A", "B", "C")
True
>>> "lo" in "hello"
True
```



## Sequenze: Accesso agli Elementi

- In collezioni sequenziali (stringhe, liste, tuple) è possibile accedere agli elementi per posizione
  - Per estrarre un elemento, si indica l'indice tra quadre
    - il primo elemento ha indice 0
- ```
>>> foo = "Hello"
>>> foo[1]    # secondo carattere della stringa
'e'
```
- Usando un indice negativo $-n$, si accede al n -ultimo elemento
 - -1 = ultimo elemento, -2 = penultimo, ...
- ```
>>> foo[-1] # ultimo carattere della stringa
'o'
```
- Sia  $N$  il numero di elementi, usando un indice non compreso tra  $-N$  e  $N-1$  si ha errore (`IndexError`)

|    |    |    |    |    |
|----|----|----|----|----|
| 0  | 1  | 2  | 3  | 4  |
| H  | e  | I  | I  | o  |
| -5 | -4 | -3 | -2 | -1 |



# Sequenze: Intervalli

- Per estrarre elementi in un intervallo, si usa al posto dell'indice la notazione “*inizio: fine*”
  - l'indice iniziale è incluso, quello finale escluso
  - se non indicati, si considerano inizio e fine dell'intera sequenza
  - si può aggiungere “*: passo*” per saltare elementi  $N$  a  $N$

```
>>> foo = "ABCDEFGHI"
>>> foo[2:5] # dal terzo al quinto carattere
'CDE'
>>> foo[:3] # primi tre caratteri
'ABC'
>>> foo[-3:] # ultimi tre caratteri
'GHI'
>>> foo[4::2] # ogni due caratteri dal quinto
'EGI'
```



# Sequenze: *Unpacking*

- Se un'espressione restituisce una sequenza di valori (es. una tupla), è possibile **assegnarne i singoli valori a variabili distinte**

```
>>> x = [1, 1, 2, 3, 5, 8, 13, 21]
>>> a, b = x[4:6] # ovvero [5, 8]
>>> a
5
>>> b
8
```

- Questa funzionalità si usa spesso per creare funzioni che **restituiscono più valori con una singola chiamata**
- Utile anche per **assegnamenti contemporanei** a più variabili
  - ad es. per scambiare due variabili senza usarne una intermedia:  
`a, b = b, a`



# Modifica di Liste

- Per **sostituire** un elemento in una lista, usare l'assegnamento

– è possibile sostituire un intero intervallo

```
>>> foo = ["A", "X", "Y", "Z", "F"]
>>> foo[1] = "B" # sostituisci secondo elemento
>>> foo
['A', 'B', 'Y', 'Z', 'F']
>>> foo[2:4] = ["C", "D", "E"] # togli 2, metti 3
>>> foo
['A', 'B', 'C', 'D', 'E', 'F']
```

- Il metodo **append** aggiunge un elemento in fondo alla lista

```
>>> foo.append("G")
```

- Per **rimuovere** invece un elemento, usare l'istruzione **del**

```
>>> del foo[0] # rimuovi primo elemento
```



## Modifica di Liste: Altri Metodi

- **insert(index, elem)** – inserisce elem nella posizione data da index, spostando i successivi elementi a destra
- **extend(list2)** – appende list2 alla lista; analogo a + oppure +=
- **index(elem)** – restituisce l'indice di elem nella lista; solleva l'eccezione ValueError se elem non esiste ValueError (nessuna eccezione con "in")
- **remove(elem)** – rimuove la prima occorrenza di elem; eccezione ValueError se elem è assente
- **sort()** – ordina la lista modificandola; restituisce None
- **sorted()** – restituisce una nuova lista ordinata; è meno efficiente di sort() ma è applicabile ad ogni iterabile (stringhe, tuple, dizionari)
- **reverse()** – inverte la lista modificandola; restituisce None
- **pop(index)** – restituisce e rimuove l'elemento in posizione index; se index è omesso restituisce e rimuove l'ultimo elemento
  - eccezione IndexError se index è fuori range



# Operazioni tra Sequenze

- Due sequenze dello stesso tipo (liste, tuple o stringhe) possono essere **concatenate** tra loro con l'operatore “`+`”

```
>>> (1, 2) + (3, 4, 5)
(1, 2, 3, 4, 5)
>>> "hell" + "o"
'hello'
```

- Si può usare l'operatore di assegnamento esteso “`+=`”
  - per sequenze immutabili, si modifica il valore della variabile
  - per sequenze mutabili, è modificato l'oggetto stesso
- Si può inoltre **ripetere una sequenza** più volte moltiplicandola con “`*`” (o con “`*=`” come sopra) per un numero intero

```
>>> 3 * "Knock! "
'Knock! Knock! Knock! '
```



# Operazioni sugli Insiemi

- Negli insiemi modificabili (`set`) si possono aggiungere e rimuovere singoli elementi con i metodi `add` e `remove`

```
>>> abc = {"a", "c"}
>>> abc.add("b")
>>> abc
{'c', 'b', 'a'}
>>> abc.remove("c")
>>> abc
{'b', 'a'}
```

- Gli insiemi (sia `set` che `frozenset`) supportano le tipiche operazioni di intersezione “`&`”, unione “`|`” e differenza “`-`”

```
>>> {1, 2, 3} & {2, 3, 4} → {2, 3}
>>> {1, 2, 3} | {2, 3, 4} → {1, 2, 3, 4}
>>> {1, 2, 3} - {2, 3, 4} → {1}
```



# Uguaglianza tra Oggetti

- L'operatore “`==`” in generale indica se due espressioni (es. due variabili) hanno lo **stesso valore**
  - ad es. se due liste diverse hanno esattamente lo stesso contenuto
  - equivale al metodo `equals` in Java
  - negazione: “`!=`”
- L'operatore “`is`” verifica invece se due espressioni si riferiscono al **medesimo oggetto**
  - equivale all'operatore “`==`” in Java
  - negazione: “`is not`”

```
>>> a = [4, 2]
```

```
>>> b = a
```

```
>>> c = [4, 2]
```

```
>>> a == b
```

True

```
>>> a == c
```

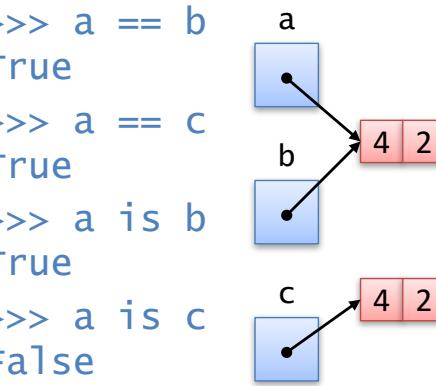
True

```
>>> a is b
```

True

```
>>> a is c
```

False



# Conversioni tra Tipi di Oggetti

- I tipi degli oggetti (`str`, `int`, `list`, ...) possono essere chiamati come funzioni per **convertire oggetti da altri tipi**
- Le operazioni possibili includono:
  - convertire una stringa in numero
 

```
>>> int("2") + 2
```

```
4
```
  - ottenere una lista modificabile dei caratteri in una stringa
 

```
>>> list("Hello!")
```

```
['H', 'e', 'l', 'l', 'o', '!']
```
  - convertire collezioni tra tipi diversi, es. da tupla a insieme
 

```
>>> set((1, 2, 1, 4, 1, 3)) # duplicati rimossi
```

```
{1, 2, 3, 4}
```



# Metodi delle Stringhe

Le stringhe (oggetti `str`) forniscono alcuni metodi utili, ad es.:

- `lower/upper` convertono la stringa in minuscolo/maiuscolo  
`>>> "Hello!".upper()`  
`'HELLO!'`
- `startswith/endswith` indica se la stringa inizia/finisce con la sottostringa data  
`>>> "How are you?".endswith("?)")`  
`True`
- `join` usa la stringa per unire le stringhe in una collezione data  
`>>> " x ".join(["A", "B", "C"])`  
`'A x B x C'`
- `split` separa la stringa in parti secondo un delimitatore dato  
`>>> "How are you?".split(" ")`  
`['How', 'are', 'you?']`

Data Intensive Applications - G. Moro, R. Pasolini - UNIBO, DISI (Cesena)

31



## Inserire Valori in una Stringa

- Come inserire nelle stringhe valori di variabili o espressioni
- Un approccio comune è usare la concatenazione di stringhe...

```
>>> x = 42
>>> y = "integer"
>>> "The number " + str(x) + " is " + y
'The number 42 is integer'
```

Gli oggetti che non sono stringhe vanno convertiti in tali per essere concatenati

- Si consiglia però di usare il metodo `format`, che inserisce valori dati in corrispondenza di segnaposto “`{}`” nella stringa
  - nel segnaposto si può inserire una specifica sul formato

```
>>> "The number {} is {}".format(x, y)
'The number 42 is integer'
>>> "The number {:.2f} is float".format(123.7777)
'The number 123.78 is float'
```

10 caratteri totali,  
2 cifre decimali

Data Intensive Applications - G. Moro, R. Pasolini - UNIBO, DISI (Cesena)

32



# Dizionari

- Un **dizionario** (tipo `dict`) è una **collezione mutabile di coppie chiave-valore** non ordinata con chiavi distinte (e.g. `Map` di Java)
    - per recuperare efficientemente valori quando è nota la chiave
  - Le **chiavi** devono essere oggetti **immutabili** (es. stringhe)
  - Un dizionario si crea specificando tra parentesi graffe una sequenza di coppie “`chiave: valore`”
- ```
>>> prices = {"water": 1.5, "soda": 2, "beer": 2.5}
>>> nothing = {}
```
- Con l’operatore “[]” si ottiene il valore associato a una chiave
 - se alla chiave non è associato nulla, si ha un `KeyError`
- ```
>>> prices["beer"]
2.5
```
- Con la funzione `len` si ottiene il numero di coppie presenti

Data Intensive Applications - G. Moro, R. Pasolini - UNIBO, DISI (Cesena)

33



## Dizionari: Operazioni

- Si possono **assegnare valori** sia a chiavi esistenti (sostituendo il valore corrente) che nuove
- ```
>>> prices = {"water": 1, "soda": 2}
>>> prices["water"] = 1.5; prices["beer"] = 2.5; prices
{'water': 1.5, 'soda': 2, 'beer': 2.5}
```
- Si può **rimuovere** un’associazione con `del`
- ```
>>> del prices["water"]; prices
{'soda': 2, 'beer': 2.5}
```
- `in` verifica la **presenza di una chiave** (**non** di un valore!)
- ```
>>> "soda" in prices, 2 in prices
True, False
```
- Creazione di dizionari anche da sequenze
- ```
my_dict = dict([(1, 'apple'), (2, 'lemon')])
```

Data Intensive Applications - G. Moro, R. Pasolini - UNIBO, DISI (Cesena)

34



# Controllo di Flusso: if-elif-else e while

- Il costrutto `if` esegue un blocco di istruzioni condizionalmente
  - si possono accodare una o più clausole `elif (else if)` e/o una clausola `else`
- `if` e `else` si possono usare anche in linea per scegliere **una di due espressioni** su una condizione
  - “`a if x else b`” è pari ad `a` se `x` è `True`, altrimenti è `b`
  - equivalente a “`x ? a : b`” in Java
- Il costrutto `while` ripete un blocco fintanto che una condizione è vera

```
if vote == 30:
 msg = "Perfect!"
elif vote >= 18:
 msg = "Passed"
else:
 msg = "Retry"
println(msg)

passed = ("Yes" if
 vote >= 18 else "No")
```

```
n = 10
while n > 0:
 print(n)
 n -= 1
```



35

## Convertire Espressioni in Valori Booleani

- In Python **qualsiasi valore** può essere convertito in booleano
  - il numero 0, stringhe e collezioni vuote e `None` valgono come `False`
  - tutto il resto vale come `True`
- Un valore `x` è convertito in `bool`:
  - implicitamente se usato ad es. come condizione in `if` o `while`
  - esplicitamente con “`bool(x)`”
- Gli operatori `and` e `or` si possono usare su valori arbitrari e restituiscono sempre per risultato uno degli operandi

```
n = 10
while n: equivale a
 n != 0
 print(n)
 n -= 1
```

```
items = [???]
first = (items[0]
 if items
 else None)
```

*ovvero “se items non è vuota”*

```
>>> 1 or 2
1
>>> 0 or 3
3
>>> 5 and 0
0
```



36

# Oggetti Iterabili e Costrutto for

- Tutte le collezioni (sequenziali e non) sono oggetti **iterabili**: è possibile **scorrere in sequenza** i loro contenuti
  - iterando stringhe, liste e tuple si ottengono gli elementi nel loro ordine
  - iterando un insieme se ne ottengono gli elementi in ordine indefinito
- Il costrutto **for** esegue un blocco di codice una volta per **ogni elemento** estratto da un **oggetto iterabile**

```
foo = [1, 5, 21]
for x in foo: # per ogni x nella lista foo...
 print(x*2) # ...stampa il doppio di x
```

- Oltre alle collezioni, esistono molti altri oggetti iterabili
  - ad es. si possono iterare le righe di un file leggendole una ad una, senza trasferire in memoria in una volta sola l'intero file



## Esempio di Oggetti Iterabili: **range**

- La funzione **range** restituisce un oggetto iterabile che fornisce in ordine i numeri da *a* incluso (default 0) a *b* escluso
  - come per altri oggetti iterabili, i valori **non sono rappresentati esplicitamente in memoria**, ma generati progressivamente
- Gli oggetti iterabili possono essere usati nei costrutti **for...**

```
>>> for x in range(3): # da 0 incluso a 3 escluso
... print(x)
0
1
2
```

- ...o anche ad es. per costruire liste, tuple, ecc.

```
>>> list(range(4, 8)) # da 4 incluso a 8 escluso
[4, 5, 6, 7]
```

usando `list`, i valori del `range` sono inseriti in una lista modificabile (quindi rappresentati esplicitamente in memoria)



# Funzioni Utili su Oggetti Iterabili

- La funzione `enumerate` associa ad ogni elemento di un iterabile il suo indice, in modo da iterare su entrambi
  - si usa l'`unpacking` per separare la tupla indice+oggetto in due variabili

```
>>> for index, value in enumerate(["A", "B", "C"]):
... print("item {} is {}".format(index, value))
item 0 is A
item 1 is B
item 2 is C
```

- La funzione `zip` unisce  $N$  iterabili in uno unico che ne restituisce gli elementi abbinati in tuple di  $N$  elementi

```
>>> zip([1,2,3], [10,20,30])
[(1,10),(2,20),(3,30)]
```



# Iterabili: Funzioni d'Aggregazione

- Le funzioni `min` e `max` restituiscono l'elemento minimo o massimo di un iterabile

```
>>> max(range(10))
9
```

- `all` e `any` indicano se tutti o almeno uno degli elementi dell'iterabile è `True` (convertendo tutti gli oggetti in `bool`)
  - utilizzabile ad es. per verificare assenza di `None` o stringhe vuote

```
>>> all(["OK", "also OK", ""])
False
```

- `sum` esegue la somma di elementi

```
>>> sum(range(5)) # 0+1+2+3+4
10
```



# Ordinamento di Elementi

- La funzione `sorted` crea una lista con gli elementi dell'iterabile dato in ordine crescente...

```
>>> sorted([3, 1, 5, 2, 8, 1])
[1, 1, 2, 3, 5, 8]
```

- ...oppure decrescente, specificando “`reverse=True`”

```
>>> sorted([3, 1, 5, 2, 8, 1], reverse=True)
[8, 5, 3, 2, 1, 1]
```

- Le liste forniscono il metodo `sort` per ordinare gli elementi modificando la lista stessa invece di crearne una nuova

```
>>> x = [3, 1, 5, 2, 8, 1]
>>> x.sort()
>>> x
[1, 1, 2, 3, 5, 8]
```

Data Intensive Applications - G. Moro, R. Pasolini - UNIBO, DISI (Cesena)

41



# Iterazione di un Dizionario

- Usando un dizionario (`dict`) come iteratore o convertendolo ad altra collezione, se ne estraggono le chiavi

```
>>> prices = {"water": 1, "soda": 2}
>>> list(prices)
['water', 'soda']
```

- Per ottenere un iteratore sui valori, usare il metodo `values`

```
>>> list(prices.values())
[1, 2]
```

- Per iterare su tuple (`chiave, valore`), usare `items`

```
>>> list(prices.items())
[('water', 1), ('soda', 2)]
```

con l'*unpacking* ad ogni iterazione ottengo chiave e valore corrispondente in due variabili distinte

```
>>> for product, price in prices.items():
... print("{}: {}".format(product, price))
```

Data Intensive Applications - G. Moro, R. Pasolini - UNIBO, DISI (Cesena)

42



# Controllo di Flusso: `break`, `continue`, `else`

In un ciclo `while` o `for`...

- Un’istruzione `break` provoca l’uscita immediata dal ciclo
- Un’istruzione `continue` causa invece l’interruzione dell’iterazione corrente e l’inizio della successiva (se c’è)
- Si può aggiungere al termine un blocco `else`, eseguito alla fine del ciclo se e solo se non è stato interrotto da `break`

```
for x in numbers: # con numbers lista di numeri
 if x < 0:
 println("Found negative number")
 break # esce dal ciclo e salta il blocco else
 else:
 # se break non è stato eseguito
 println("No negative number found")
```



# Gestione Errori ed Eccezioni

- Alcune operazioni possono sollevare errori o eccezioni, ad es.
  - lettura di una variabile non esistente (`NameError`)
  - accesso a dizionario con chiave non presente (`KeyError`)
  - istruzione esplicita, es: `raise Exception("invalid input")`
- Un errore interrompe l’esecuzione del blocco di codice corrente e, se non gestito, fa terminare il programma
- Per gestire gli errori, si usa un blocco `try-except-finally`, con semantica analoga a `try-catch-finally` in Java

```
try:
 istruzioni da eseguire
except Exception as ex: # "as ex" opzionale
 gestione dell'errore, rappresentato da oggetto ex
 print(ex); print(str(ex)); print(ex.args) # esempi
finally: # opzionale
 istruzioni di chiusura eseguite in ogni caso
```



## List Comprehensions

- Una *list comprehension* crea in modo compatto una lista da un altro iterabile, **trasformando e filtrando elementi**

- Per trasformare elementi da un'altra collezione si usa “[*espressione for elemento in iterabile*]”

```
>>> nums = [1, 5, 21]
>>> [x*2 for x in nums] # doppio di ogni x in nums
[2, 10, 42]
```

- Con “*if condizione*” si selezionano solo alcuni elementi

```
>>> [x*2 for x in nums if x > 3] # 1 viene saltato
[10, 42]
```

- Si possono concatenare più *for...in* (somma delle 9 combinazioni)

```
>>> [x+y for x in nums for y in nums]
[2, 6, 22, 6, 10, 26, 22, 26, 42]
```



## Altri tipi di Comprehensions

- Le comprehensions possono essere usate per creare altri tipi di collezioni, quali tuple e insiemi

```
>>> tuple(x//2 for x in range(5))
(0, 0, 1, 1, 2)
```

```
>>> set(x//2 for x in range(5))
{0, 1, 2}
```

Trattandosi di un insieme, i duplicati sono eliminati

- E' anche possibile creare dizionari con una coppia chiave-valore per ogni elemento iterato

```
>>> {x: 10**x for x in range(4)}
{0: 1, 1: 10, 2: 100, 3: 1000}
```



# Definizione di Funzioni

- Per definire una funzione si usa un blocco `def`
- Una funzione può accettare un numero arbitrario di **parametri**
- Una funzione può **restituire un valore** usando `return`

```
def factorial(x):
 if x > 1:
 return x * factorial(x-1)
 else:
 return 1
>>> factorial(4)
24
```

- Se `return` non indica un valore o non viene usato, la funzione restituisce implicitamente `None`
- Passaggio dei parametri **per riferimento**
  - per oggetti immutabili questo è equivalente a passarli per valore
  - gli oggetti mutabili invece **possono essere modificati** dalla funzione

Data Intensive Applications - G. Moro, R. Pasolini - UNIBO, DISI (Cesena)

47



## Funzioni: Documentazione

- Inserendo **una stringa come prima istruzione** di una funzione, questa è usata per **descrivere testualmente cosa fa** (*docstring*)
  - simile ad un commento `/** ... */` in Java (javadoc)
  - per convenzione si usa una stringa con tripli apici, anche se di una riga
  - le docstring si applicano anche a moduli (vedi dopo) e classi
  - `pydoc nome_modulo #` per ottenere da shell la documentazione

```
def factorial(x):
 """Compute the factorial n! of a number n."""
 return x * factorial(x-1) if n > 1 else 1
```

- In modalità interattiva, usare la funzione `help` per leggere la documentazione di una funzione
  - tutte le funzioni standard di Python sono documentate

```
>>> help(print)
```



# Funzioni: Parametri con Valori di Default

- Ogni parametro di una funzione può avere un **valore di default**
- Se presente, è possibile chiamare la funzione senza specificarne il valore
- Se ci sono più parametri con default, per indicarne uno specifico nella chiamata si usa “*nome=valore*”
  - i parametri con valore di default si dichiarano sempre dopo quelli senza default

```
>>> def foo(x, m=1, q=0):
... return m*x + q
>>> foo(4, 3, 2)
14
>>> foo(4, 2) # q=0
8
>>> foo(4) # m=1, q=0
4
>>> foo(4, q=3) # m=1
7
```

Data Intensive Applications - G. Moro, R. Pasolini - UNIBO, DISI (Cesena)

49



# Funzioni: Parametri in Numero Variabile

- Dichiarando un parametro col prefisso “\*”, il suo valore sarà una **lista di tutti gli argomenti passati** oltre a quelli predefiniti

```
>>> def foo(a, b, *c):
... print(a, b, c)
>>> foo(1, 2, 3, 4, 5)
1 2 [3, 4, 5]
```

- Un parametro “\*\*” ha invece come valore un **dizionario** con gli argomenti “*nome=valore*” non assegnati ad altri parametri
  - una funzione può avere un parametro “\*” e uno “\*\*” insieme

```
>>> def foo(a, b=2, **c):
... print(a, b, c)
>>> foo(a=1, c=3, e=5)
1 2 {'c': 3, 'e': 5}
```

Data Intensive Applications - G. Moro, R. Pasolini - UNIBO, DISI (Cesena)

50



# Funzioni: Argomenti in Liste e Dizionari

- “\*” e “\*\*” possono anche essere usati in una **chiamata** a funzione per passare **molteplici argomenti in un unico oggetto**
  - un iterabile passato con “\*” è espanso ad argomenti senza nome
  - un dizionario passato con “\*\*” è espanso ad argomenti con nome
  - si possono usare insieme tra loro e ad argomenti passati normalmente

```
>>> def foo(a, b, c):
... print(a, b, c)
>>> x = [2, 3]
>>> foo(1, *x) # 2 e 3 sono assegnati a b e c
1 2 3
>>> y = {"b": 5, "c": 6}
>>> foo(4, **y) # idem ma restituisce la chiave
4 5 6
>>> foo(4, *y) restituisce invece il valore 4 b c
```

Data Intensive Applications - G. Moro, R. Pasolini - UNIBO, DISI (Cesena)

51



# Programmazione Funzionale: Riferimenti a Funzioni ed Espressioni Lambda

- Essendo oggetti, le funzioni possono essere **usate come argomenti o valori restituiti** da altre funzioni
  - queste ultime sono dette *funzioni di ordine superiore*
- Ad esempio, **map** applica una funzione agli elementi di un iterabile e ne restituisce un altro con i valori trasformati

```
>>> def double(x):
... return 2*x
>>> list(map(double, range(5))) # 2 * [0 ~ 4]
[0, 2, 4, 6, 8]
```

Questa non è una chiamata alla funzione double, ma un riferimento ad essa

- In alternativa ad usare una funzione esistente, si può usare un'**espressione lambda** per definire una funzione “al volo”
- ```
>>> list(map(lambda x: 2*x, range(5)))
[0, 2, 4, 6, 8]
```

Data Intensive Applications - G. Moro, R. Pasolini - UNIBO, DISI (Cesena)

52



Esempio di Programmazione Funzionale:

Definire l'Ordine degli Oggetti

- Quando si usa `sort/sorted/min/max`, l'ordine degli oggetti è implicito se comparabili con “`<`” e “`>`” (es. numeri e stringhe)
- È possibile specificare esplicitamente con una funzione `key` una caratteristica sulla base della quale ordinare gli oggetti
- Ad es., data una lista di tuple articolo+prezzo...

```
>>> items = [("Beer", 2.5), ("Juice", 2),
...     ("Soda", 2.25), ("Water", 1.5)]
```

- ...possiamo specificare il prezzo come chiave d'ordinamento

```
>>> items.sort(key = lambda x: x[1])
>>> items
[('Water', 1.5), ('Juice', 2), ('Soda', 2.25),
('Beer', 2.5)]
```

per ordinare le tuple,
considerare il secondo
elemento (il prezzo)



Generatori

- Un generatore è un oggetto iterabile definito in forma di funzione, con eventuali parametri
- Invece di `return`, la funzione deve eseguire un'istruzione `yield` per ciascun oggetto che il generatore fornirà
 - si può inoltre usare `yield from` per restituire tutti gli elementi da un altro iterabile

```
def fibonacci(n):
    a, b = 1, 1
    for i in range(n):
        yield a
        a, b = b, a+b
```

```
>>> for i in fibonacci(6):
...     print(i)
1
1
2
3
5
8
```



Aprire un File

- La funzione standard `open` da l'accesso ad un file su disco
- Il primo parametro di `open` indica il **nome o percorso del file**
 - il percorso può essere relativo alla directory corrente o assoluto
- Il secondo parametro indica la **modalità di apertura**, una stringa con una o più delle seguenti lettere:
 - `r` = lettura (default), `w` = scrittura (**sovrascrivendo** il file se esiste già), `a` = scrittura in append (scrivendo in coda al file se esiste già)
 - `t` = modalità testuale (default), `b` = modalità binaria
- `open` restituisce un oggetto che fa da **riferimento al file aperto**, che costituisce una risorsa impiegata dal programma
 - un “oggetto file” può essere aperto in altri modi e rappresentare altre risorse, es. lo standard output o un flusso di dati ricevuti via rete



Chiudere un File

- Appena finito di leggere o scrivere il file, è opportuno **chiuderlo** per confermare le modifiche e rilasciare il file
- Si può chiudere un file col metodo `close` dell'oggetto
`myfile = open("myfile.txt", "rt")`
`(operazioni su myfile...)`
`myfile.close()`
- Python fornisce un costrutto `with` per **chiudere in automatico i file aperti**, anche in caso di errori durante l'uso
 - equivalente al costrutto “try-with-resource” in Java
- Un pattern comune consiste nell'aprire un file con `with` e passare l'oggetto ad una funzione apposita che processi il file
`with open("myfile.txt", "rt") as myfile:`
 `data = parseDataInSomeFormat(myfile)`



Leggere da File di Testo

- I file di testo sono letti comunemente **riga per riga**
- Usando l'oggetto restituito da `open` come un **iterabile** (es. con `for`), **si scorrono le singole righe** del file in forma di stringhe
- Le stringhe restituite **includono i caratteri di fine riga** ("`\n`" o "`\r\n`"): si può usare il metodo `strip` per tagliarli
 - `strip` rimuove anche spazi e tabulazioni a inizio e fine riga!

```
# trova massimo numero di caratteri in un record
max_chars = 0
with open("myfile.txt", "rt") as myfile:
    for line in myfile:      # per ogni record
        max_chars = max(max_chars, len(line.strip()))
print(max_chars)
```



Scrivere su File di Testo

- Per scrivere su un file di testo, è possibile usare il metodo `write` passando una stringa di caratteri da scrivere
 - il testo non va a capo tra una chiamata a `write` e l'altra, a meno che non sia incluso un **fine riga** nelle stringhe passate

```
with open("myfile.txt", "w") as myfile:
    myfile.write("First line\r\n") # carriage return e new line
    myfile.write("Second line\r\n")
```

- In alternativa, si può usare la funzione `print` specificando il file di output col parametro opzionale `file`

– a differenza di `write`, `print` aggiunge un fine riga alla fine

```
with open("myfile.txt", "w") as myfile:
    print("First line", file=myfile)
    print("Second line", file=myfile)
```



Moduli

- Ogni file di codice Python costituisce un **modulo**
- Ogni modulo ha un proprio **namespace** (spazio di nomi) in cui sono definite variabili, funzioni ecc., **isolato dagli altri moduli**
 - diversi moduli possono usare gli stessi nomi senza interferenze
- All'avvio dell'interprete Python, viene creato un **modulo principale** in cui sono salvate le variabili dichiarate
 - in modalità interattiva, questo contiene le variabili definite nelle istruzioni eseguite man mano dall'utente
 - eseguendo un file, il modulo contiene gli oggetti dichiarati nel file stesso
- Tramite l'**importazione**, è possibile da un modulo richiamarne altri ed utilizzarne le funzionalità



Creare un Modulo

- Un file `.py`, oltre ad essere eseguito direttamente, può essere usato come modulo da un altro file
- Si prenda ad esempio un file `mymodule.py` in cui sono definite delle funzioni:

```
def factorial(n):
    return n*factorial(n-1) if n>1 else 1
def fibonacci(n):
    return (fibonacci(n-1) + fibonacci(n-2))
        if n>1 else 1
```

- Eseguendo direttamente questo file (“`python mymodule.py`”) non si ha alcun output
 - vengono dichiarate due funzioni, ma non sono usate in alcun modo



Importare un Modulo

- Avviando un interprete Python dalla directory in cui è salvato il file del modulo, è possibile *importarlo*

```
>>> import mymodule # senza estensione .py
```

- Una volta importato, è possibile accedere agli oggetti di un modulo con la sintassi “*modulo.oggetto*”

- Si può quindi usare la funzione `factorial` così:

```
>>> mymodule.factorial(4)
```

24

- Ogni oggetto modulo ha un attributo “`__name__`”, il cui valore è una stringa col nome del modulo

- il modulo principale ha sempre “`__main__`” come nome



Altre forme di import

- Con la clausola `as` si può *cambiare il nome* usato per riferirsi al modulo importato nel contesto corrente

```
>>> import mymodule as m
```

```
>>> m.factorial(5)
```

120

- Con la forma `from ... import` si possono *importare uno o più oggetti* da un modulo *direttamente nel namespace corrente*

```
>>> from mymodule import factorial, fibonacci
```

```
>>> factorial(6)
```

720

- si può scrivere “`from mymodule import *`” per importare *tutti* gli oggetti di un modulo, ma può causare conflitti tra oggetti di moduli diversi con nome uguale



Eseguire un Modulo

- Normalmente, il comando `python3` esegue uno script o modulo dato il nome (o percorso completo) **del suo file**
- Usando l'opzione `-m`, è possibile eseguire un modulo **dato invece il suo nome**
- L'opzione `-m` è usata per eseguire codice **definito in librerie esterne**, piuttosto che in un file scritto dall'utente
- Aggiungendo altri argomenti, questi sono passati come parametri al modulo
 - i parametri accettati dipendono dal modulo specifico
 - è spesso possibile ottenere un elenco dei parametri accettati passando “`-h`” o “`--help`” come argomento

`$ python3 -m mymodule arg1 arg2`



Package

- I **package** consentono di **organizzare i moduli in una gerarchia**
- Ogni package può contenere moduli e/o altri package
 - è comune che ogni libreria esterna sia contenuta in un package dedicato che contiene una propria gerarchia di package e moduli
- Un package è costituito da una directory contenente un file `__init__.py`, che contiene il codice per inizializzarlo
 - il file può essere vuoto, ma **dove** esistere perché la directory sia considerata un package
- La directory può contenere moduli (file `.py`) e altri package (directory con file `__init__.py`)
- Ogni package e modulo ha un nome completo costituito dalla sequenza dei nomi dei package, separati da “.”



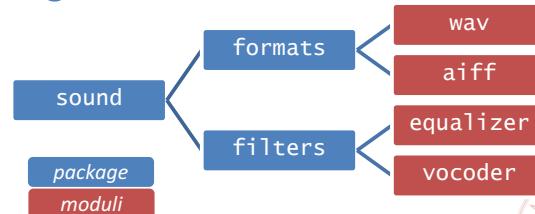
Esempio di Package Strutturato

- La seguente gerarchia di file e directory definisce un package “sound” contenente una gerarchia di package e moduli

```
sound/
    __init__.py
    formats/
        __init__.py
        wav.py
        aiff.py
        ...
    filters/
        __init__.py
        equalizer.py
        vocoder.py
        ...
```

Top-level package
Initialize the sound package
Subpackage for file formats

Subpackage for filters



Data Intensive Applications - G. Moro, R. Pasolini - UNIBO, DISI (Cesena)

65



Importare Moduli dai Package

- Si può importare un modulo da un package col suo nome completo e riferirsi ad esso sempre col nome completo

```
>>> import sound.formats.wav
>>> audio = sound.formats.wav.load("audio.wav")
```

- ...oppure si può importare il modulo nel namespace locale

```
>>> from sound.formats import wav
>>> audio = wav.load("audio.wav")
```

- Si può comunque importare con un nome a scelta

```
>>> import sound.formats.wav as w
>>> audio = w.load("audio.wav")
```



Libreria Standard di Python

- La *libreria standard di Python*, installata di default insieme all’interprete, fornisce moduli con funzionalità di uso comune
- Questi moduli possono essere importati con l’istruzione `import` e usati allo stesso modo di quelli creati dall’utente
- La documentazione di Python include informazioni approfondite su tutti i moduli della libreria standard

<https://docs.python.org/3/library/index.html>



Esempio d’Uso della Libreria Standard

- Il modulo `math` della libreria standard contiene diverse funzioni matematiche comuni
 - `sqrt` (radice quadrata), `log2` (logaritmo in base 2), ...
- Possiamo *importare il modulo intero* ed accedere a tutte le funzioni al suo interno...

```
>>> import math
>>> math.sqrt(25)
5.0
```

- ...così come possiamo *importare le singole funzioni* a cui siamo interessati e invocarle direttamente

```
>>> from math import log2
>>> log2(256)
8.0
```



Alcuni Moduli della Libreria Standard

- `collections`: strutture dati aggiuntive
- `re`: espressioni regolari (usate per trovare pattern in stringhe)
- `datetime`: rappresentazione di date e orari
- `random`: generazione di numeri casuali
- `csv`: lettura/scrittura file CSV (*Comma Separated Values*)
- `pickle`: (de)serializzazione di oggetti in forma binaria
 - usato per salvare/caricare oggetti Python arbitrari su/da file
- `argparse`: interpretazione di opzioni da linea di comando



Modulo `random`: Numeri Casuali

- Il modulo `random` fornisce funzioni per numeri e scelte casuali
- `seed(x)`: imposta lo stato del generatore di numeri
 - impostando lo stesso seed, esce la stessa sequenza di numeri “casuali”
- `random()`: numero casuale nell’intervallo [0, 1)
- `randint(a, b)`: intero casuale compreso tra a e b
- `gauss(mu, sigma)`: numero casuale con distribuzione normale di media mu e deviazione standard sigma
- `choice(x)`: elemento casuale dalla sequenza x
- `shuffle(x)`: mescola (modificandola) la lista x
- `sample(x, k)`: k elementi casuali distinti dalla lista x



Modulo `csv`: Leggere e Scrivere File CSV

- CSV (*Comma Separated Values*) è un semplice formato testuale per salvare dati in forma tabulare
 - Ogni riga del file costituisce una riga della tabella, le colonne in ogni riga sono separate da un carattere specifico
 - spesso virgola, punto e virgola o tabulazione
 - La prima riga del file di solito contiene i nomi delle colonne
- ```
ID;Product;Quantity;Price;Total
1;An item;3;2.50;7.50
2;Another item;1;3.50;3.50
...

```
- Il modulo `csv` di Python fornisce funzioni utili per leggere e scrivere file CSV



## Leggere un File CSV

- La funzione `reader`, dato un oggetto file aperto con `open`, restituisce un **iteratore con una tupla per ogni riga**
  - il parametro `delimiter` indica il separatore di colonna (default ";")
  - viene restituita anche la prima riga (spesso con i nomi delle colonne)

```
import csv
with open("myfile.csv", "rt") as myfile:
 for row in csv.reader(myfile):
 print(row[0]) # stampa solo la prima colonna
```

- In alternativa, la classe `DictReader` fornisce le **righe in forma di dizionari**, usando la prima riga come intestazione con i nomi

```
with open("myfile.csv", "rt") as myfile:
 for row in csv.DictReader(myfile):
 print(row["ID"]) # stampa solo colonna "ID"
```



# Scrivere un File CSV

- Dal modulo `csv` si possono creare oggetti con metodi appositi per scrivere oggetti in forma di righe di file CSV
  - `writerow` scrive una riga da un oggetto
  - `writerows` scrive più righe da una collezione di oggetti
- Con la funzione `writer` si trattano dati in sequenze (es. tuple)

```
with open("myfile.csv", "wt") as myfile:
 csvwriter = csv.writer(myfile)
 csvwriter.writerow([1, "An item", 5.5])
```

- Con la classe `DictWriter`, indicando la lista ordinata di nomi delle colonne, si trattano invece dati in forma di dizionari

```
with open("myfile.csv", "wt") as myfile:
 csvwriter = csv.DictWriter(myfile, ["ID", "Name"])
 csvwriter.writerow({"ID": 1, "Name": "An item"})
```



# Modulo `re`: Espressioni Regolari

- Un'*espressione regolare* (in breve *regex*) è uno schema (*pattern*) con cui una stringa può combaciare (*match*) o meno
- I pattern sono definiti essi stessi da stringhe, dove alcuni caratteri (detti *metacaratteri*) hanno significati speciali
- Le espressioni regolari sono presenti in molti linguaggi di programmazione con sintassi (quasi) identica
- Il modulo `re` fornisce diverse funzionalità basate sulle regex

| Esempi di regex   | Significato                                                                |
|-------------------|----------------------------------------------------------------------------|
| [A-Z] [a-z] +     | lettera maiuscola seguita da una o più minuscole                           |
| 0   [1-9] [0-9]*  | la cifra 0, oppure una cifra da 1 a 9 seguita da zero o più cifre da 0 a 9 |
| \d{4}-\d{2}-\d{2} | una data nel formato yyyy-mm-dd (\d = [0-9])                               |
| # [0-9A-Fa-f]{6}  | un colore in formato esadecimale (es. #FF0000)                             |



# Espressioni Regolari: Definizione di Pattern

## Caratteri

- `.` (punto): qualsiasi carattere
- `[abcd...]`: un carattere tra quelli indicati
  - `a-z`: un car. incluso tra a e z
- `[^abcd...]`: un carattere **non** tra quelli indicati
- `X|Y`:  $X$  oppure  $Y$
- `\d`: una cifra da 0 a 9
- `\s`: uno spazio, tab o simile
- `\X`: escape metacar.  $X$  (*ad es. pattern "\." → stringa "* $.$ *"*)

## Modificatori

- `X`: un carattere o un gruppo delimitato da parentesi `(...)`
  - `X?`: opzionale (0 o 1 volta)
  - `X*`: ripetuto 0, 1 o più volte
  - `X+`: ripetuto almeno una volta
  - `X{m}`:  $m$  volte esatte
  - `X{m,}`: almeno  $m$  volte
  - `X{m,n}`: dalle  $m$  alle  $n$  volte
- (questi sono i caratteri e modificatori più comuni, ne esistono altri)*



# Espressioni Regolari: Ricerca di un Pattern

- La funzione `compile` crea un oggetto pattern su cui possono essere chiamati i metodi illustrati sotto

```
>>> import re
>>> dates = re.compile(r"(\d{4})-(\d{2})-(\d{2})")
 – il pattern indica stringhe tipo "XXXX-XX-XX", dove ogni X è una cifra
```

- Il metodo `search` cerca il pattern in una stringa data

```
>>> datematch = dates.search("Today is 2018-02-21")
```

- `fullmatch` verifica che un'intera stringa combaci col pattern

```
>>> datematch = dates.fullmatch("2018-02-21")
```

- Entrambi i metodi restituiscono un oggetto "match" se la verifica del pattern riesce, `None` altrimenti

```
>>> if datematch: # se si è ottenuto un match
... print("found a date in the string!")
```



# Espressioni Regolari: Estrazione di Elementi

- Gli oggetti “match” danno informazioni sulla stringa trovata
- I metodi `start` e `end` danno la posizione iniziale e finale del match, `span` restituisce una tupla con entrambe

```
>>> dates = re.compile(r"(\d{4})-(\d{2})-(\d{2})")
>>> today = dates.search("Today is 2018-02-21")
>>> today.span() ↑0 ↑9 ↑19
(9, 19)
```

- Il metodo `group` senza argomenti restituisce la stringa trovata

```
>>> today.group()
'2018-02-21'
```

- Indicando un indice, viene restituito un **singolo componente della stringa** tra quelli **delimitati tra parentesi** nel pattern

```
>>> today.group(1)
'2018'
```



# Librerie Esterne

- Le funzionalità di Python possono essere estese oltre quelle della libreria standard tramite **librerie di terze parti**
- Ogni libreria fornisce **nuovi moduli e package** importabili e utilizzabili all'interno dei propri programmi
- Ogni libreria può a sua volta **dipendere da altre librerie**
- Il *Python Package Index* (PyPI) è un database online di oltre 100.000 librerie disponibili per Python

<https://pypi.org/>



# pip

- L'utility `pip` inclusa in Python può essere usata da linea di comando per **installare package da PyPI**
- Ad es., per installare la libreria NumPy usare dal terminale (**non** dall'interprete Python) il comando:  
`$ pip install numpy`
  - si può indicare una versione specifica (es. “`numpy==1.14`”)
  - è possibile installare più package insieme
- Se una libreria **dipende da altre**, queste sono **installate automaticamente** insieme ad essa
- Una libreria può essere installata
  - a livello di sistema (servono diritti di amministratore)
  - solo per l'utente corrente (con l'opzione “`--user`”)

Data Intensive Applications - G. Moro, R. Pasolini - UNIBO, DISI (Cesena)

79



# Ambienti Virtuali

- Diversi progetti Python possono richiedere librerie diverse o persino versioni diverse della stessa libreria
- Un *ambiente virtuale* rappresenta una collezione di librerie **installate indipendente da quelle installate nel sistema**
  - gli ambienti virtuali condividono solo la libreria standard di Python
- Creando un ambiente virtuale specifico per ciascun progetto, **si evitano conflitti tra versioni dei package** da utilizzare
- Un ambiente virtuale è **contenuto in una directory**, che contiene i programmi eseguibili e le librerie installate in esso



# Creare un Ambiente Virtuale

- In Python 3 è stato introdotto il modulo `venv` per la creazione di ambienti virtuali
  - in Python 2 si usa un comando `virtualenv` separato
- Eseguendo il modulo `venv`, si inizializza un nuovo ambiente in una directory specificata (che viene creata se necessario)
 

```
$ python3 -m venv myvenv
```
- Nell'ambiente virtuale sono inserite nuove copie dell'interprete Python e dei programmi correlati (es. `pip`)
  - nella sottodirectory `bin` in Linux e Mac OS X
  - nella sottodirectory `Scripts` in Windows
- Questa copia di Python ha accesso alla libreria standard, ma inizialmente a nessuna libreria esterna



# Utilizzare un Ambiente Virtuale

- Usando la copia di `pip` nella directory dell'ambiente, si possono installare librerie in esso invece che nel sistema
 

```
$ myvenv/bin/pip install numpy (Linux / Mac)
```

```
> myvenv\Scripts\pip install numpy (Windows)
```
- Lanciando l'interprete Python presente nella stessa directory, si potranno usare le librerie installate
- L'ambiente virtuale fornisce anche uno script `activate` per rendere default l'uso degli eseguibili in esso
 

```
$ source myvenv/bin/activate (Linux / Mac)
```

```
> myvenv\Scripts\activate.bat (Windows)
```
- È così possibile avviare `python` e `pip` omettendone il percorso
 

```
(myvenv) $ pip install numpy
```

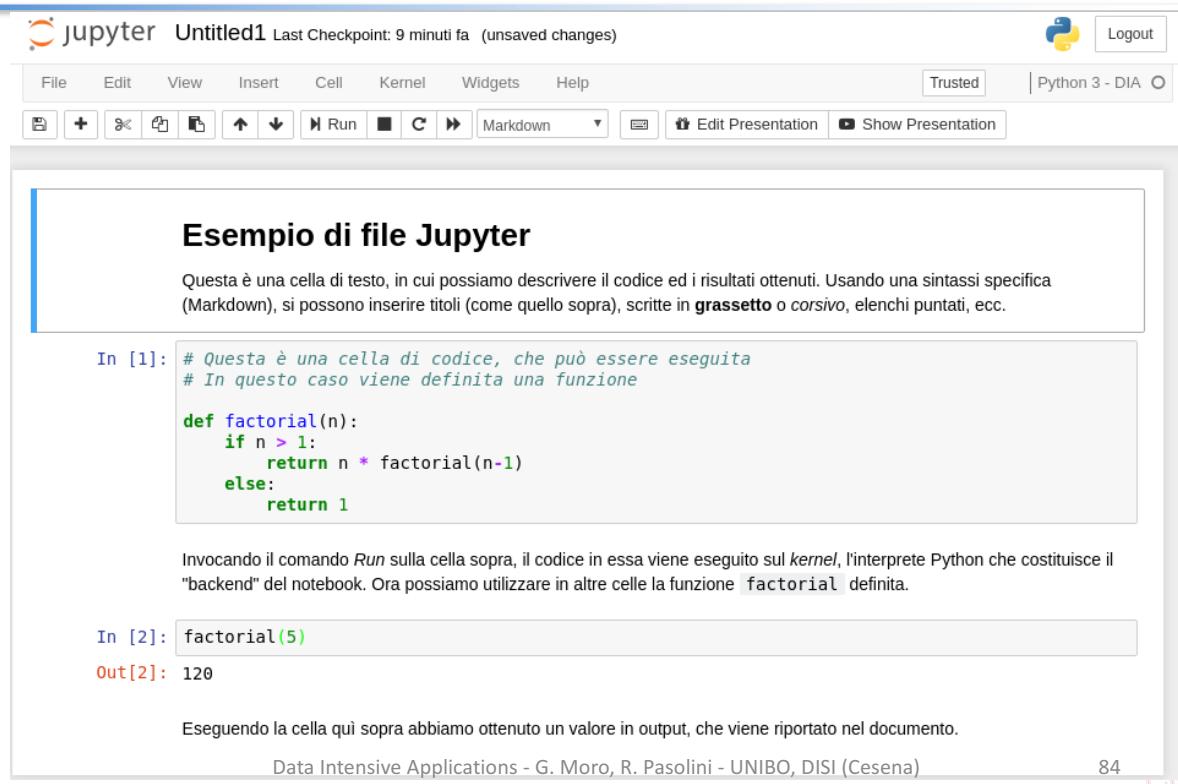


# Jupyter

- Il progetto *Jupyter* fornisce una modalità d'utilizzo di Python alternativa all'interprete e ai file script
- In Jupyter si possono scrivere documenti costituiti da una sequenza di *celle*, ciascuna contenente codice oppure testo
- Le celle di codice possono essere eseguite, riportando ogni eventuale output sotto la cella stessa nel documento
- Con Jupyter si può quindi eseguire codice interattivamente e documentarlo allo stesso tempo, ottenendo un documento unico con codice, risultati e testo descrittivo
  - i documenti sono esportabili in formati standard (HTML, PDF, ...)



## L'interfaccia di Jupyter

A screenshot of the Jupyter Notebook interface. The top navigation bar includes 'File', 'Edit', 'View', 'Insert', 'Cell', 'Kernel', 'Widgets', and 'Help'. On the right, there are buttons for 'Logout', 'Trusted', and 'Python 3 - DIA'. Below the toolbar, a menu bar has icons for file operations like 'New', 'Open', 'Save', etc., followed by 'Run', 'Kernel', 'Widgets', 'Markdown' dropdown, and presentation-related buttons. A large central area contains a code cell with the title 'Esempio di file Jupyter'. The cell content is a Markdown header and a Python code block defining a factorial function. Below the cell, a note explains that running the code creates a kernel and defines a function. Another code cell shows the execution of the factorial function with the argument 5, resulting in 120. A note at the bottom states that the output value is reported in the document.

```
In [1]: # Questa è una cella di codice, che può essere eseguita
In questo caso viene definita una funzione

def factorial(n):
 if n > 1:
 return n * factorial(n-1)
 else:
 return 1
```

Invocando il comando *Run* sulla cella sopra, il codice in essa viene eseguito sul *kernel*, l'interprete Python che costituisce il "backend" del notebook. Ora possiamo utilizzare in altre celle la funzione `factorial` definita.

```
In [2]: factorial(5)
Out[2]: 120
```

Esegundo la cella qui sopra abbiamo ottenuto un valore in output, che viene riportato nel documento.



# Jupyter: Kernel

- L'esecuzione del codice avviene sul *kernel*, un processo eseguito in background con cui la webapp interagisce
- Di default il kernel è un interprete Python (*IPython*) che ha accesso alle librerie installate nel sistema
  - per utilizzare le librerie installate in un ambiente virtuale, va aggiunto un kernel associato ad esso
- Sono disponibili kernel per molti altri linguaggi oltre a Python
  - R, Java, Matlab, JavaScript e tanti altri
- Ad ogni file notebook è associato un kernel specifico
  - per ogni file è eseguita un'istanza separata del kernel
- Se il kernel termina si perdono i dati in memoria, ma il contenuto del notebook (testo, codice e output) è salvato



# Programmazione ad Oggetti in Python

- Come in tutti i linguaggi ad oggetti, è possibile **definire nuovi tipi di oggetto** in Python dichiarando delle *classi*
  - da una classe si possono creare *istanze*, che forniscono attributi e metodi definiti dalla classe
- L'esistenza di un attributo di un oggetto è verificata **quando viene invocato, indipendentemente dal tipo** di oggetto
  - ad es. l'invocazione di un metodo chiamato *X* è sempre possibile nel codice e da errore a run-time solo se non esiste un metodo di nome *X*
  - questo approccio è noto come *duck typing*
  - per questo in Python (al contrario di Java) **non** esistono interfacce
  - questo consente di avere attributi dinamici: diversi oggetti, anche se della stessa classe, potrebbero avere attributi diversi



# Definire una Classe in Python

- Per **definire una classe** si usa il costrutto `class`
  - Possiamo definire una classe minimale “vuota”:
- ```
>>> class Foo:
...     pass
```
- “`pass`” si usa per indicare un blocco indentato vuoto
- Una volta definita, possiamo **creare istanze** di questa classe **invocandola come una funzione**
 - non si usano `new` o altre parole chiave

```
>>> x = Foo()
>>> x
<__main__.Foo object at 0x00000165B75E2048>
```



Definire Metodi in una Classe

- Un metodo è definito come **funzione all’interno della classe**
- Quando si dichiara un metodo, il suo **primo parametro** deve **sempre essere il riferimento all’oggetto su cui è chiamato**
 - convenzionalmente questo parametro ha nome “`self`”

```
class Foo:
    def say_hello(self):
        print("Hello!")
```

- Il metodo può quindi essere chiamato su un oggetto istanza della classe con la tipica sintassi `nome.metodo`

```
>>> x = Foo()
>>> x.say_hello()    # equivale a Foo.say_hello(x)
Hello!
```

- il valore del parametro `self` è implicitamente l’oggetto `x`



Modificare lo Stato dell’Oggetto

- Su ciascun singolo oggetto possono essere impostati degli attributi, che ne costituiscono lo stato
- Secondo la OOP lo stato si legge/modifica con i metodi definiti nella classe, tramite il riferimento “`self`” all’oggetto

```
class Counter:
    def set_count(self, n):
        self.count = n
    def get_count(self):
        return self.count
```

- È **sempre** possibile accedere ad attributi e metodi anche al di fuori della classe, non esistono attributi “privati” della classe
 - per convenzione, attributi o metodi il cui nome inizia per “`_`” vanno considerati privati, ma Python **non ne impedisce l’accesso** dall’esterno



Inizializzazione degli Oggetti

- In ogni classe si può definire un metodo speciale `__init__`, chiamato automaticamente su ogni oggetto appena creato
 - come gli altri metodi, deve avere come primo parametro il riferimento `self` all’oggetto appena creato
 - si usa ad esempio per assegnare valori iniziali agli attributi
- Il metodo `__init__` può accettare parametri (oltre a `self`), i cui valori sono passati dal costruttore dell’oggetto

```
class Counter:
    def __init__(self, n=0):
        self.count = n
>>> x = Counter(5)
>>> x.count
5
```



Metodi Speciali

- Oltre a `__init__`, Python definisce altri metodi speciali `__nome__` che possono essere implementati nelle classi
 - noti anche come metodi *magici* o *dunder* (“double underscore”)
- Questi metodi sono invocati automaticamente da Python quando sono usate determinate funzionalità del linguaggio
 - operatori (`+`, `-`, ...), funzioni standard, costrutti del linguaggio (`for`, ...)
- Ad esempio, il metodo `__len__` determina il risultato restituito dalla funzione `len` invocata su un’istanza

```
class Foo:
    def __len__(self):
        return 10
>>> x = Foo()
>>> len(x)      # equivale a x.__len__()
10
```



Esempi di Metodi Speciali

Espressione	Chiamata equivalente a metodo speciale
<code>str(obj)</code>	<code>obj.__str__()</code> <i>(analogo a <code>toString</code> in Java)</i>
<code>x == y</code>	<code>x.__eq__(y)</code> <i>(analogo a <code>equals</code> in Java)</i>
<code>x < y</code>	<code>x.__lt__(y)</code>
<code>obj(x, y)</code>	<code>obj.__call__(x, y)</code>
<code>obj[ind]</code>	<code>obj.__getitem__(ind)</code>
<code>obj[ind] = val</code>	<code>obj.__setitem__(ind, val)</code>
<code>del obj[ind]</code>	<code>obj.__delitem__(ind)</code>
<code>item in obj</code>	<code>obj.__contains__(item)</code>
<code>a + b</code>	<code>a.__add__(b)</code>

Ad es., per creare una classe di oggetti che si possano “sommare” con l’operatore `+`, basta implementare il metodo `__add__` e definire in esso la semantica della somma



Ereditarietà tra Classi

- Una classe *B* può estendere un'altra classe esistente *A*
- Un attributo o metodo di un oggetto di tipo *B*, se non trovato nella classe *B*, viene cercato nella classe *A*
- L'inizializzazione di un oggetto di tipo *B* esegue l'`__init__` definito in *A* solo se richiamato esplicitamente da quello in *B*

```
class Derived(Base):  
    def __init__(self, foo, bar):  
        super().__init__(foo)  
        self.bar = bar
```

- In Python una classe può anche ereditare da più classi

```
class Derived(Base1, Base2, Base3):  
    ...
```

- Python determina l'ordine in cui attributi/metodi vengono cercati nelle superclassi (*method resolution order*) con regole specifiche

