

# Moduli e Pacchetti

---



# Le Funzioni Finora

La funzioni forniscono una serie di **vantaggi significativi**

- Permettono di **astrarre un algoritmo**
  - ...E così di decomporre un problema complesso in problemi più semplici
- Rendono il codice più **ordinato**
- Rendono il codice più **facile da mantenere**
- Rendono il codice più **riutilizzabile**

Hanno però anche un grosso limite:

**Con le funzioni, non possiamo condividere codice tra "notebook" diversi**

- Per farlo, dobbiamo ancora copiare ed incollare il codice
- ...Con tutti gli svantaggi già discussi



# Moduli

## Una soluzione semplice: spostare le funzioni **in un altro file**

In questo modo:

- Diversi notebook possono fare riferimento allo **stesso gruppo di funzioni**
- Una modifica ad una funzione si ha effetto su **tutte le sue chiamate**
- Possiamo definire **diversi gruppi di funzioni**, rendendo il codice più ordinato

Python permette questo risultato attraverso i moduli

## Un **modulo** in Python

- Rappresenta un contenitore di funzioni e variabili
  - ...E di altri oggetti che incontreremo nel tempo
- È implementato con un **file di testo** con estensione `.py`



# Definire ed Utilizzare Moduli

Vediamo come definire in un modulo la funzione `potenza`:

```
In [2]: def potenza(a, b):  
        return a**b  
  
        print(potenza(2, 4))
```

16

**Nella cartella corrente **va creato un file di testo** con estensione `.py`**

- Potete farlo da "Esplora Risorse" su Windows o Finder su OS X
- ...Oppure dalla home page di Jupiter
  - Prima usate il pulsante "new" per create un file di testo (`.txt`)
  - Poi cambiatene il nome e dategli l'estensione `.py`
  - Il nome deve essere un identificatore valido in Python



# Definire ed Utilizzare Moduli

**Nel modulo, si può definire la funzione come al solito**

- Per questa lezione, è già stato creato un modulo "esempio modulo.py"
- Potete aprirlo usando il link, oppure dall'home page di Jupyter
- Potete verificare che contiene il codice della nostra funzione

In particolare il contenuto del file è:

```
def potenza(a, b):  
    return a**b
```



# Definire ed Utilizzare Moduli

**Il nostro obiettivo è chiamare la funzione da questo notebook**

- Per farlo, dobbiamo dire all'interprete Python di utilizzare il modulo
- Si usa l'istruzione `import`, con la sintassi:

```
import <nome modulo>
```

- Il nome del modulo coincide con il nome del file
- ...Senza l'estensione `py`

Nel nostro caso abbiamo:

```
In [3]: import esempio_modulo
```



# Definire ed Utilizzare Moduli

Ogni modulo definisce uno **spazio di nomi (namespace)**

...Ossia agisce come un contenitore di identificatori

■ La nostra funzione `potenza` è accessibile nel namespace `esempio_modulo`

**Per accedere un namespace in Python, si usa la **notazione puntata****

La sintassi è:

```
<namespace>.<identificatore>
```

Nel nostro esempio, avremo:

```
In [4]: esempio_modulo.potenza(2, 4)
```

```
Out[4]: 16
```

Con questo metodo possiamo usare `potenza` **da qualunque notebook**



# Varianti di `import`

L'istruzione `import` ha alcune varianti

Per esempio, permette di rinominare localmente il modulo

■ Si usa la sintassi:

```
import <modulo> as <nome alternarnativo>
```

Nel nostro caso possiamo usare per esempio

```
In [4]: import esempio_modulo as em  
  
em.potenza(2, 4)
```

```
Out[4]: 16
```

Si tratta di una funzionalità comoda per accorciare il nome di un modulo





# Ri-caricamento di Moduli

## Quando un modulo viene importato

...L'interprete verifica se l'operazione non sia già stata fatta

- Se il modulo non risulta già "caricato" si procede con l'importazione
- ...Altrimenti non avviene nulla

```
In [3]: import esempio_modulo as em  
  
em.potenza(2, 4)
```

```
Out[3]: 16
```

- Di solito si tratta di una funzionalità utile (fa risparmiare tempo)
- ...Ma a tempo di sviluppo può causare problem

Vediamo di capire quali e come risolverli...



# Ri-caricamento di Moduli

Provate ad eseguire la cella seguente:

```
In [5]: import esempio_modulo as em  
em.potenza(2, 4)
```

```
Out[5]: 16
```

Modificate ora il codice nel modulo come segue:

```
def potenza(a, b):  
    return 2*a**b
```

Ripetete ora l'esecuzione della cella:

```
In [6]: import esempio_modulo as em  
em.potenza(2, 4)
```

```
Out[6]: 16
```



**Il risultato non è cambiato!**

# Ri-caricamento di Moduli

**Il risultato non è cambiato perché il modulo non è stato ri-caricato**

- Possiamo risolvere il problema ri-caricando esplicitamente il modulo
  - Esiste una funzione apposta in un modulo predefinito
- ...Ma si tratta di una operazione scomoda

**Durante lo svilppo è più facile usare una funzionalità extra di Jupyter**

In particolare, useremo l'estensione **autoreload**

```
In [9]: %load_ext autoreload
        %autoreload 2
```

- `%load_ext autoreload` prepare l'estensione autoreload
- `%autoreload 2` la attiva
- ...E la configura per ricaricare tutti i pacchetti prima dell'esecuzione



# Ri-caricamento di Moduli

## Una volta che l'estensione è attiva

...Le modifiche ai moduli hanno effetto immediato in Jupyter

```
In [10]: import esempio_modulo as em  
em.potenza(2, 4)
```

```
Out[10]: 32
```

- Utilizzeremo `autoreload` in tutti gli esercizi d'ora in avanti
- ...Ma ricordate che per avere la massima performance, è bene disabilitarla



# Moduli e Script

---



# Codice Libero in un Modulo

## Eventuale **codice libero** (non in una funzione) in un modulo

...Viene eseguito (come al solito) durante la `import`

- Di solito lo si sfrutta per **definire variabili utili**
- E.g., potremmo voler rendere disponibile non solo la funzione `potenza`
- ...Ma anche la costante  $e$

## Il codice del modulo può essere modificato come segue:

```
def potenza(a, b):  
    return a**b  
  
e = 2.71828
```

Il codice modificato è disponibile in [esempio\\_modulo2.py](#)



# Codice Libero in un Modulo

## Quando importiamo il modulo

- L'istruzione `e = 2.71828` viene eseguita come al solito
- ...E la variabile `e` diventa disponibile \_all'interno del modulo:

```
In [5]: import esempio_modulo2

res = esempio_modulo2.potenza(esempio_modulo2.e, 2)
print('e^2 = ', res)

e^2 = 7.3890461584
```

Di solito si usa questa caratteristica per:

- Definire variabili importanti
- Eseguire codice di configurazione per il modulo



# Codice Libero in un Modulo

...Ma qualcuno l'ha usato anche per definire degli "easter egg"

Provate a riavviare il kernel ed eseguire:

```
In [6]: import this
```





# Script

**In generale, eseguire codice direttamente da un file può essere molto utile**

- Jupyter è utilissimo per sperimentare
- ...E per insegnare un nuovo linguaggio di programmazione

**...Ma una volta che del codice è consolidato**

- Dover avviare Jupyter ogni volta è scomodo
- ...E richiede inoltre di aver installato Jupyter

**Sarebbe molto meglio poter eseguire direttamente un file `.py`**

- Un file pensato per questo scopo si chiama `script`
- È identico (o quasi) ad un modulo con codice libero
- Semplicemente, è pensato per essere eseguito da terminale



# Script

Nel file esempio\_script.py trovate il codice:

```
def potenza(a, b):  
    return a**b  
  
e = 2.71828  
  
if __name__ == "__main__":  
    print(f'e^2 = {potenza(e, 2)}')
```

- `__name__` è una variabile che viene assegnata automaticamente dall'interprete
- Quando si importa il modulo contiene una stringa con il nome del modulo
- Quando si esegue un file `.py` da terminale, contiene la stringa `"__main__"`
- Usando `if __name__ == "__main__":`
- ...Il codice nell'`if` viene eseguito **solo** in caso di chiamata da terminale



# Script

## Proviamo ad eseguire lo script

- Aprite un terminale nella cartella con lo script
- ...Quindi eseguite il comando:

```
python esempio_script.py
```

Verrà stampato su terminale  $e^2 = 7.3890461584$

## Possiamo farlo anche dall'interno di Jupyter

Si usa il comando:

```
In [7]: !python esempio_script.py
```

```
e^2 = 7.3890461584
```

- Il "!" dice a Jupyter che il comando che segue va eseguito da terminale



# Script

## **Gli script sono il modo "normale" di eseguire codice Python**

Ogni programma Python compiuto viene di solito eseguito come script:

- Un server web
- Un sistema di diagnostica predittiva in contesto industriale
- Un sistema per il riconoscimento di immagini
- ...

Uno script può ricevere anche argomenti da linea di comando:

- L'indirizzo web (URL) da esporre
- L'identificativo del PLC da cui recuperare i valori dei sensori
- La telecamera da utilizzare
- ...

**In questo corso, però, gli script saranno usati molto poco**



# Pacchetti

---



# Moduli e Pacchetti

**Più moduli possono essere raggruppati in un pacchetto (package)**

- Un pacchetto è un contenitore di moduli
- È implementato come una **directory**

## **Per definire un pacchetto**

- Nella cartella corrente, va **creata una nuova cartella**
  - Il nome della cartella deve essere un **identificatore valido**
- Nella cartella va creato un file `__init__.py`
  - Questo file viene eseguito quando il pacchetto viene "importato"
  - Può essere anche vuoto
  - È comunque necessario per indicare che la directory è un pacchetto



# Pacchetti

Un pacchetto di esempio è disponibile nella directory esempio\_pacchetto

Contiene:

- Un file `__init__.py` vuoto
- Un modulo `esempio_modulo.py`

**I pacchetti si possono importare esattamente come i moduli**

```
In [2]: import esempio_pacchetto
```

- Quando si importa un pacchetto il suo file `__init__.py` viene eseguito
- ...Ma i moduli contenuti non vengono automaticamente importati



# Pacchetti

**Per importare un modulo da un pacchetto, occorre farlo esplicitamente**

```
In [4]: import esempio_pacchetto.esempio_modulo  
  
        esempio_pacchetto.esempio_modulo.potenza(2, 4)
```

Out[4]: 16

- Si usa la notazione puntata perché i pacchetti introducono un namespace

**Per maggior leggibilità, si usa spesso l'istruzione `from ... import`**

```
In [6]: from esempio_pacchetto import esempio_modulo  
  
        esempio_modulo.potenza(2, 4)
```

Out[6]: 16

- In questo modo importiamo il modulo

-  Senza passare per il namespace del pacchetto



# Sottopacchetti

## Un pacchetto può avere dei sottopacchetti

- Corrispondono semplicemente a delle sotto-directory
- Introducono nei namespace, come al solito
- Non richiedono di specificare un nuovo file `__init__.py`

## Un esempio è disponibile nella cartella esempio\_pacchetto2

- Il pacchetto principale è vuoto (c'è solo `__init__.py`)
- Il sottopacchetto `esempio_sottopacchetto` contiene `esempio_modulo.py`

```
In [7]: from esempio_pacchetto2.esempio_sottopacchetto import esempio_modulo as em  
  
em.potenza(2, 4)
```

```
Out[7]: 16
```

Per lavorare con i sottopacchetti si usa la notazione puntata, come al solito



# Compilazione Just in Time

## I pacchetti vengono gestiti in modo speciale dall'interprete Python

- Anziché essere semplicemente interpretati
- ...Vengono compilati al momento dell'importazione

Il risultato della compilazione:

- Viene inserito una cartella `__pycache__` all'interno della directory
- ...E consiste di uno o più file con estensione `.pyc`

## Si parte di **compilazione "just in time" (JIT)**

...Ossia fatta al momento del bisogno

- La compilazione JIT introduce un po' di overhead al momento del caricamento
- ...Ma permette di accelerare il tempo di esecuzione in seguito



# Pacchetti ed Ecosistema Python

## Python ha moltissimi pacchetti pre-installati

- `math` per fare calcoli
- `random` per generare numeri casuali
- `os` per lavorare con i file
- ...

## In aggiunta, altri pacchetti sono installabili da collezioni centralizzate

Di solito, vengono gestiti mediante un programma detto package manager

- `pip` per Python classico
- `conda` per Anaconda

Ci sono pacchetti per fare tantissime cose

## Insieme formano il cosiddetto ecosistema Python

