Paradossi della Probabilità

Un Tutorial di Programmazione

Impara Python Simulando l'Impossibile

Liceo Lugano 3 — Giornata Autogestita 2026

In questo tutorial imparerai a **programmare in Python** simulando cinque famosi paradossi della probabilità.

Non serve esperienza di programmazione — solo curiosità!

Alla fine avrai scritto le tue simulazioni,
generato grafici e *dimostrato* che la matematica può sorprendere.

Contents

Per Cominciare

Cos'è Python?

Python è uno dei linguaggi di programmazione più diffusi al mondo. Scienziati, ingegneri, analisti di dati e persino artisti lo usano ogni giorno. Perché?

- Si legge quasi come l'inglese
- È gratuito e funziona su qualsiasi computer
- Ha strumenti potenti per matematica, grafici e simulazioni

Configurazione

Il modo più semplice per iniziare è **Google Colab** — non serve installare nulla!

1. Vai su https://colab.research.google.com
2. Accedi con il tuo account Google
3. Clicca su **"Nuovo notebook"**
4. Sei pronto a programmare!

Importante

In Colab si scrive il codice in *celle*. Premi **Shift** + **Enter** per eseguire una cella. Provalo subito!

Il Tuo Primo Programma

Scrivi questo in una cella e premi **Shift** + **Enter**:

```
print("Ciao, mondo!")
print("Sto imparando Python!")
print(2 + 3)
```

Output

```
Ciao, mondo!
Sto imparando Python!
5
```

La funzione `print()` è il modo in cui Python mostra qualcosa sullo schermo. Tutto ciò che metti tra le parentesi viene stampato. Il testo va tra virgolette `"..."`, i numeri no.

Variabili: dare un nome ai valori

Una **variabile** è come un'etichetta che attacchi a un valore per poterlo riutilizzare:

```
nome = "Alice"       # una variabile di testo (stringa)
eta = 17             # una variabile numerica (intero)
altezza = 1.68       # un numero con la virgola (decimale)

print(nome)          # stampa: Alice
print(eta)           # stampa: 17
```

Come funzionano le variabili

Pensa a una variabile come a una **scatola con un'etichetta**. La riga `eta = 17` crea una scatola chiamata `eta` e ci mette dentro il numero 17. Puoi cambiare il contenuto in qualsiasi momento:

```
eta = 17
print(eta)    # 17
eta = 18      # il compleanno!
print(eta)    # 18
```

Il vecchio valore (17) viene semplicemente sostituito.

Matematica: Python come calcolatrice

Python sa fare tutti i calcoli che ti servono:

```
print(10 + 3)       # addizione: 13
print(10 - 3)       # sottrazione: 7
print(10 * 3)       # moltiplicazione: 30
print(10 / 3)       # divisione: 3.3333...
print(10 ** 2)      # potenza (10 al quadrato): 100
print(2 ** 10)      # 2 alla 10: 1024
```

Puoi anche salvare i risultati in variabili e usarli dopo:

```
base = 5
altezza = 3
area = base * altezza
print(area)          # 15
```

L'operatore +=

Spesso vogliamo *aggiungere* qualcosa a una variabile. Invece di scrivere `x = x + 1`, Python offre una scorciatoia:

```
punteggio = 0
punteggio += 1       # equivale a: punteggio = punteggio + 1
punteggio += 1       # ora punteggio vale 2
print(punteggio)     # 2
```

Useremo `+=` tantissimo per contare le vittorie nelle nostre simulazioni!

Stringhe formattate (f-string)

Per inserire variabili dentro un testo, si usano le **f-string**. Basta mettere una `f` prima delle virgolette e le variabili tra parentesi graffe `{...}`:

```
nome = "Marco"
eta = 16
print(f"Mi chiamo {nome} e ho {eta} anni")
```

Output

```
Mi chiamo Marco e ho 16 anni
```

Si possono anche formattare i numeri. Ad esempio, `:.1f` mostra un decimale, `:.1%` mostra una percentuale:

```
valore = 2/3
print(f"Il risultato e' {valore}")          # 0.6666666...
print(f"Il risultato e' {valore:.4f}")      # 0.6667 (4 decimali)
print(f"Il risultato e' {valore:.1%}")      # 66.7% (percentuale)
```

Condizioni: prendere decisioni con `if`

Il computer può prendere decisioni usando `if` ("se") e `else` ("altrimenti"):

```
temperatura = 35

if temperatura > 30:
    print("Fa caldo!")
else:
    print("Si sta bene")
```

Output

```
Fa caldo!
```

Python controlla la condizione dopo `if`. Se è **vera**, esegue il blocco indentato sotto `if`. Se è **falsa**, esegue il blocco sotto `else`.

L'indentazione conta!

In Python, gli spazi all'inizio di una riga *non sono decorazione* — definiscono la struttura. Tutto ciò che "appartiene" a un `if` o un ciclo `for` deve essere spostato a destra di 4 spazi. Colab lo fa automaticamente quando premi Invio dopo i due punti (`:`).

```
if 5 > 3:
    print("Questo e' DENTRO l'if")    # indentato = dentro
    print("Anche questo")              # indentato = dentro
print("Questo e' FUORI dall'if")      # non indentato = fuori
```

Si possono usare diversi **operatori di confronto**:

Operatore	Significato
`==`	uguale a (attenzione: due segni di uguale!)
`!=`	diverso da
`>`	maggiore di
`<`	minore di
`>=`	maggiore o uguale
`<=`	minore o uguale

Un errore classico: = vs ==

Un solo `=` *assegna* un valore: `x = 5` mette 5 nella scatola `x`.
Due `==` *confrontano* due valori: `x == 5` chiede "`x` è uguale a 5?".
Confonderli è l'errore più comune per chi inizia!

Valori booleani: `True` e `False`

Ogni confronto produce un valore **booleano**: può essere solo `True` (vero) o `False` (falso).

```
print(5 > 3)       # True
print(5 == 3)      # False
print(5 != 3)      # True

risultato = (10 > 7)
print(risultato)   # True
```

Si possono combinare condizioni con `and` ("e") e `or` ("o"):

```
eta = 16
ha_biglietto = True

if eta >= 12 and ha_biglietto:
    print("Puoi entrare!")    # entrambe le condizioni vere
```

Cicli: ripetere azioni con `for`

Un **ciclo** `for` ripete un blocco di codice un certo numero di volte:

```
for i in range(5):
    print(i)
```

Output

```
0
1
2
3
4
```

`range(5)` genera i numeri da 0 a 4 (5 numeri in tutto, partendo da 0). Ad ogni "giro" del ciclo, la variabile `i` assume il valore successivo.

```
# Somma dei numeri da 1 a 100
totale = 0
for numero in range(1, 101):   # da 1 a 100
    totale += numero
print(f"La somma e': {totale}")  # 5050
```

Perché si conta da 0?

In informatica, quasi tutto parte da 0 e non da 1. `range(5)` dà `0, 1, 2, 3, 4` — cinque numeri, ma partendo da zero. Ci si abitua in fretta!
Se vuoi partire da un numero diverso: `range(1, 6)` dà `1, 2, 3, 4, 5`.

Liste: collezioni ordinate

Una **lista** è una collezione di valori, racchiusa tra parentesi quadre:

```
colori = ["rosso", "blu", "verde"]
print(colori[0])     # rosso  (il primo elemento ha indice 0!)
print(colori[1])     # blu
print(len(colori))   # 3  (len = lunghezza della lista)
```

Si possono aggiungere elementi con `.append()`:

```
numeri = []           # lista vuota
numeri.append(10)     # aggiungi 10 alla fine
numeri.append(20)     # aggiungi 20 alla fine
numeri.append(30)     # aggiungi 30 alla fine
print(numeri)         # [10, 20, 30]
```

Cosa fa `.append()`?

`.append()` aggiunge un elemento *alla fine* della lista. Il punto (`.`) indica che stiamo "chiedendo alla lista" di fare qualcosa — in questo caso, di allungarsi. Useremo `.append()` per raccogliere i risultati delle nostre simulazioni e poi disegnare i grafici.

Si può controllare se un valore è nella lista con `in`:

```
frutti = ["mela", "banana", "arancia"]
print("mela" in frutti)     # True
print("kiwi" in frutti)     # False
```

Funzioni: blocchi di codice riutilizzabili

Una **funzione** è come una ricetta: la scrivi una volta e poi la puoi usare tutte le volte che vuoi. Si definisce con `def`:

```
def saluta(nome):
    print(f"Ciao, {nome}!")

saluta("Alice")    # Ciao, Alice!
saluta("Marco")    # Ciao, Marco!
saluta("Sofia")    # Ciao, Sofia!
```

Scomponiamo la struttura:

- `def` dice a Python: "sto definendo una funzione"
- `saluta` è il **nome** che diamo alla funzione
- `nome` tra parentesi è il **parametro**: un'informazione che la funzione riceve dall'esterno
- Il codice indentato sotto è il **corpo**: ciò che la funzione fa

Funzioni con `return`

Una funzione può anche **restituire un valore** con `return`. È come una macchina: entra qualcosa (parametri), esce un risultato:

```
def doppio(x):
    return x * 2      # restituisce il risultato

risultato = doppio(7)  # risultato ora vale 14
print(risultato)       # 14
print(doppio(100))     # 200
```

Quando Python incontra `return`, la funzione **si ferma** e manda indietro quel valore. Tutto ciò che sta dopo `return` non viene eseguito.

Ecco un esempio più utile — una funzione che calcola la media:

```python
def media(lista):
    somma = 0
    for valore in lista:
        somma += valore
    return somma / len(lista)

voti = [8, 7, 9, 6, 8]
print(f"La media e': {media(voti)}")    # La media e': 7.6
```

Importare librerie

Python ha migliaia di *librerie* — strumenti già pronti che puoi usare. Ne servono due:

```python
import random              # per generare numeri casuali
import matplotlib.pyplot as plt  # per creare grafici
```

In Colab queste sono già installate. Basta eseguire questa cella una volta all'inizio del notebook.

La libreria `random`

`random` permette di simulare il caso:

- `random.random()` — un numero casuale tra 0 e 1
- `random.randint(1, 6)` — un intero casuale (come lanciare un dado)
- `random.choice(["a", "b", "c"])` — sceglie un elemento a caso da una lista

Prova:

```python
import random

# Lancia un dado 10 volte
for i in range(10):
    lancio = random.randint(1, 6)
    print(f"Lancio {i+1}: {lancio}")
```

Eseguilo più volte — otterrai risultati diversi ogni volta! Questa è la casualità.

Paradosso I: Il Problema di Monty Hall

La Storia

Sei in un quiz televisivo. Ci sono **3 porte**. Dietro una c'è un'**auto**, dietro le altre due ci sono delle **capre**. Scegli una porta. Il conduttore (che *sa* cosa c'è dietro ogni porta) apre un'altra porta, rivelando una capra. Poi ti chiede:

"Vuoi cambiare porta?"

La maggior parte delle persone dice che non fa differenza — è 50/50, no?
Sbagliato. Cambiare vince **2/3** delle volte!

Sembra impossibile. *Dimostriamolo* con il codice.

Passo 1: Simulare una partita

Pensiamo a cosa succede in una partita:

1. L'auto viene piazzata dietro una porta a caso (0, 1 o 2)
2. Il giocatore sceglie una porta a caso
3. Il conduttore apre una porta con una capra (non quella del giocatore, non quella dell'auto)
4. Il giocatore decide: **restare** o **cambiare**

```
import random

# Piazza l'auto dietro una porta casuale (0, 1 o 2)
car = random.randint(0, 2)
print(f"L'auto e' dietro la porta {car}")

# Il giocatore sceglie una porta a caso
player_pick = random.randint(0, 2)
print(f"Il giocatore sceglie la porta {player_pick}")

# Il giocatore vince RESTANDO?
if player_pick == car:
    print("RESTARE vince! (ha scelto la porta giusta)")
else:
    print("CAMBIARE vince! (non ha scelto la porta giusta)")
```

Vediamo cosa fa ogni riga:

- `random.randint(0, 2)` genera un numero casuale tra 0 e 2 (cioè 0, 1 o 2) — serve per scegliere una porta
- `player_pick == car` confronta la scelta del giocatore con la posizione dell'auto. Se sono uguali, restare vince; altrimenti, cambiare vince

L'intuizione chiave

Nota una cosa: vinci **restando** solo se la tua prima scelta era l'auto. Vinci **cambiando** ogni volta che la tua prima scelta *non* era l'auto. Dato che ci sono 3 porte:

- Probabilità di indovinare l'auto al primo colpo: $\mathbf{1/3}$
- Probabilità di *non* indovinare l'auto: $\mathbf{2/3}$

Quindi cambiare vince $2/3$ delle volte!

Passo 2: Simulare 1000 partite

Una sola partita non dimostra molto. Giochiamo **1000 partite** e contiamo le vittorie:

```python
import random

stay_wins = 0        # contatore: vittorie restando
switch_wins = 0      # contatore: vittorie cambiando

for game in range(1000):              # ripeti 1000 volte
    car = random.randint(0, 2)          # piazza l'auto
    player_pick = random.randint(0, 2)  # il giocatore sceglie

    if player_pick == car:
        stay_wins += 1      # restare avrebbe vinto (+1 al contatore)
    else:
        switch_wins += 1    # cambiare avrebbe vinto (+1 al contatore)

# Alla fine, stampa i risultati
print(f"Partite giocate: 1000")
print(f"Vittorie restando:   {stay_wins}  ({stay_wins/10:.1f}%)")
print(f"Vittorie cambiando: {switch_wins} ({switch_wins/10:.1f}%)")
```

Output

```
Partite giocate:  1000
Vittorie restando:  332 (33.2%)
Vittorie cambiando:  668 (66.8%)
```

I numeri parlano da soli: cambiare vince circa il **67%** delle volte!
Vediamo cosa fa il codice passo per passo:

1. Creiamo due contatori a zero: `stay_wins` e `switch_wins`
2. Il ciclo `for game in range(1000)` ripete il blocco indentato 1000 volte
3. Ad ogni giro: piazziamo l'auto a caso e il giocatore sceglie a caso
4. Se la scelta è uguale all'auto (`==`), restare avrebbe vinto, quindi aggiungiamo 1 a `stay_wins`
5. Altrimenti (`else`), cambiare avrebbe vinto
6. Alla fine stampiamo i totali

Passo 3: Osservare la convergenza

Creiamo un **grafico** che mostra come le percentuali di vittoria si stabilizzano man mano che giochiamo più partite. Per fare questo, salviamo la percentuale dopo *ogni* partita in una lista:

```python
import random
import matplotlib.pyplot as plt

n_games = 5000          # numero totale di partite
stay_wins = 0           # contatore vittorie restando
switch_wins = 0         # contatore vittorie cambiando
stay_rates = []         # lista: percentuale di vittoria restando
switch_rates = []       # lista: percentuale di vittoria cambiando

for game in range(1, n_games + 1):    # da 1 a 5000
    car = random.randint(0, 2)
    pick = random.randint(0, 2)

    if pick == car:
        stay_wins += 1
    else:
        switch_wins += 1

    # Dopo ogni partita, salva la percentuale corrente
    stay_rates.append(stay_wins / game)
    switch_rates.append(switch_wins / game)

# Disegna il grafico
plt.figure(figsize=(10, 5))
plt.plot(stay_rates, color='#DC2626', linewidth=1,
         label='Restare', alpha=0.8)
plt.plot(switch_rates, color='#16A34A', linewidth=1,
         label='Cambiare', alpha=0.8)
plt.axhline(y=1/3, color='#DC2626', linestyle='--',
            alpha=0.5, label='Teoria: 1/3')
plt.axhline(y=2/3, color='#16A34A', linestyle='--',
            alpha=0.5, label='Teoria: 2/3')
plt.xlabel('Numero di partite', fontsize=12)
plt.ylabel('Percentuale di vittoria', fontsize=12)
plt.title('Monty Hall: Restare vs Cambiare',
          fontsize=14, fontweight='bold')
plt.legend(fontsize=11)
plt.ylim(0.15, 0.85)
plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()
```

Dovresti vedere la linea rossa (restare) stabilizzarsi intorno al 33% e la linea verde (cambiare) intorno al 67%.

Cosa hai appena fatto

Hai usato una **simulazione Monte Carlo**: ripetere un esperimento casuale migliaia di volte per stimare una probabilità. È una tecnica reale usata da scienziati, ingegneri e analisti finanziari ogni giorno!
Le due liste (`stay_rates` e `switch_rates`) raccolgono la percentuale dopo ogni partita con `.append()`. Poi `plt.plot()` le disegna come curve.

Sfida: 100 Porte!

E se ci fossero **100 porte** invece di 3? Ne scegli una, il conduttore apre 98 porte con capre, lasciando la tua porta e un'altra. Cambieresti?
Modifica la simulazione:

```
n_doors = 100
car = random.randint(0, n_doors - 1)
pick = random.randint(0, n_doors - 1)
# Restare vince solo se hai indovinato: 1/100
# Cambiare vince le altre 99/100 volte!
```

Prova con 10, 50 e 100 porte. Che schema noti?

Paradosso II: Ragazzo o Ragazza

La Storia

Un genitore ti dice: *"Ho due figli. Almeno uno di loro è un maschio."*

Qual è la probabilità che *entrambi* siano maschi?

La maggior parte delle persone risponde 1/2. Dopotutto, l'altro figlio è o maschio o femmina, no?
La risposta è 1/3.

Perché 1/3? Ragioniamoci

Una famiglia con due figli ha quattro possibilità equiprobabili:

Caso	Figlio 1	Figlio 2	Almeno un maschio?
1	Femmina	Femmina	No
2	Femmina	Maschio	Sì
3	Maschio	Femmina	Sì
4	Maschio	Maschio	Sì

Sappiamo che "almeno uno è maschio", quindi il Caso 1 è eliminato. Dei 3 casi rimanenti, solo 1 ha entrambi maschi. Dunque la probabilità è $\frac{1}{3}$!

Passo 1: Simulare famiglie casuali

Scriviamo una **funzione** che crea una famiglia con due figli casuali:

```
import random

def crea_famiglia():
    """Crea una famiglia con 2 figli casuali."""
    figlio1 = random.choice(["M", "F"])    # M = maschio, F = femmina
    figlio2 = random.choice(["M", "F"])
    return (figlio1, figlio2)      # restituisce i due figli
```

Cosa fa questa funzione:

- `random.choice(["M", "F"])` sceglie a caso tra "M" e "F" — come lanciare una moneta
- `return (figlio1, figlio2)` restituisce una **tupla** (coppia di valori)

Proviamola:

```
for i in range(10):
    famiglia = crea_famiglia()
    print(f"Famiglia {i+1}: {famiglia[0]}, {famiglia[1]}")
```

Output

```
Famiglia 1:  M, F
Famiglia 2:  F, F
Famiglia 3:  M, M
Famiglia 4:  F, M
Famiglia 5:  M, F
...
```

Passo 2: Filtrare e contare

Ora generiamo **10 000 famiglie**, teniamo solo quelle con almeno un maschio e contiamo quante hanno *entrambi* maschi:

```python
import random

n_famiglie = 10000
almeno_un_maschio = 0      # quante famiglie hanno almeno un maschio
entrambi_maschi = 0        # quante ne hanno entrambi maschi

for i in range(n_famiglie):
    figlio1 = random.choice(["M", "F"])
    figlio2 = random.choice(["M", "F"])

    # Questa famiglia ha almeno un maschio?
    if figlio1 == "M" or figlio2 == "M":
        almeno_un_maschio += 1

        # Sono ENTRAMBI maschi?
        if figlio1 == "M" and figlio2 == "M":
            entrambi_maschi += 1

# Calcola la probabilita' e stampa
probabilita = entrambi_maschi / almeno_un_maschio
print(f"Famiglie con almeno un maschio: {almeno_un_maschio}")
print(f"Di queste, entrambi maschi: {entrambi_maschi}")
print(f"Probabilita': {probabilita:.4f}")
print(f"Teoria: {1/3:.4f}")
```

Output

```
Famiglie con almeno un maschio:  7509
Di queste, entrambi maschi:  2498
Probabilita':  0.3327
Teoria:  0.3333
```

Nota come usiamo `or` ("o") per controllare se *almeno uno* è maschio, e `and` ("e") per controllare se *entrambi* lo sono.

Passo 3: Grafico di convergenza

Come per Monty Hall, salviamo la probabilità dopo ogni famiglia e disegniamo il grafico:

```python
import random
import matplotlib.pyplot as plt

n = 10000
almeno_un_maschio = 0
```

```
entrambi_maschi = 0
probabilita_nel_tempo = []     # lista per il grafico

for i in range(n):
    c1 = random.choice(["M", "F"])
    c2 = random.choice(["M", "F"])

    if c1 == "M" or c2 == "M":
        almeno_un_maschio += 1
        if c1 == "M" and c2 == "M":
            entrambi_maschi += 1

    # Salva la probabilita' corrente
    if almeno_un_maschio > 0:
        probabilita_nel_tempo.append(
            entrambi_maschi / almeno_un_maschio)
    else:
        probabilita_nel_tempo.append(0)

plt.figure(figsize=(10, 5))
plt.plot(probabilita_nel_tempo, color='#2563EB',
         linewidth=1, alpha=0.8)
plt.axhline(y=1/3, color='#DC2626', linestyle='--',
            linewidth=2, label='Teoria: 1/3')
plt.xlabel('Numero di famiglie', fontsize=12)
plt.ylabel('P(entrambi maschi | almeno uno maschio)', fontsize=12)
plt.title('Paradosso Ragazzo o Ragazza',
          fontsize=14, fontweight='bold')
plt.legend(fontsize=11)
plt.ylim(0.2, 0.5)
plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()
```

Sfida: una domanda diversa

E se il genitore dicesse: "*Il mio figlio* ***più grande*** *è un maschio*"? Ora qual è la probabilità che entrambi siano maschi?

Suggerimento: Stavolta fissiamo Figlio 1 = M. L'unica domanda riguarda il Figlio 2. Scrivi la simulazione e osserva — la risposta diventa $\mathbf{1/2}$!

Perché specificare *quale* figlio cambia la risposta?

Paradosso III: Il Problema del Compleanno

La Storia

Quante persone servono in una stanza perché ci sia una **probabilità superiore al 50%** che due di loro condividano il compleanno?
La maggior parte delle persone dice circa 180 (la metà di 365). La vera risposta?

Solo 23 persone!

Con sole 23 persone, c'è il 50,7% di probabilità che almeno due condividano il compleanno. Con 50 persone, si arriva al 97%!

Passo 1: Simulare una stanza

Scriviamo una funzione che genera compleanni casuali per un gruppo e controlla se ci sono coincidenze:

```
import random

def ha_coincidenza(n_persone):
    """Controlla se due persone del gruppo condividono
    il compleanno."""
    compleanni = []           # lista vuota: qui mettiamo i compleanni
    for i in range(n_persone):
        giorno = random.randint(1, 365)   # un giorno casuale
        if giorno in compleanni:
            return True       # coincidenza trovata! Esci subito
        compleanni.append(giorno)  # aggiungi alla lista
    return False              # nessuna coincidenza
```

Cosa fa questa funzione, passo per passo:

1. Crea una lista vuota `compleanni`
2. Per ogni persona, genera un giorno casuale (da 1 a 365)
3. Controlla se quel giorno è **già nella lista** con `in`
4. Se sì: `return True` — la funzione si ferma e restituisce "vero"
5. Se no: aggiunge il giorno alla lista e passa alla persona successiva
6. Se arriviamo alla fine senza coincidenze: `return False`

```
# Proviamo con 23 persone
risultato = ha_coincidenza(23)
print(f"Coincidenza trovata: {risultato}")
```

Esegui questa cella più volte. A volte otterrai `True`, a volte `False`. Questo perché 23 persone danno circa il 50% di probabilità.

Passo 2: Stimare la probabilità

Per avere una stima precisa, ripetiamo l'esperimento molte volte e contiamo quante volte troviamo una coincidenza:

```
def probabilita_compleanno(n_persone, n_prove=10000):
    """Stima P(coincidenza) per un gruppo di n_persone."""
    coincidenze = 0                    # contatore
    for prova in range(n_prove):       # ripeti molte volte
        if ha_coincidenza(n_persone):  # usa la funzione di prima
            coincidenze += 1
    return coincidenze / n_prove       # proporzione di successi

# Testiamo con diverse dimensioni del gruppo
for n in [10, 20, 23, 30, 40, 50, 60]:
    p = probabilita_compleanno(n)
    print(f"{n:3d} persone: P(coincidenza) = {p:.1%}")
```

Output

```
 10 persone:  P(coincidenza) = 11.8%
 20 persone:  P(coincidenza) = 41.0%
 23 persone:  P(coincidenza) = 50.4%
 30 persone:  P(coincidenza) = 70.5%
 40 persone:  P(coincidenza) = 89.2%
 50 persone:  P(coincidenza) = 97.1%
 60 persone:  P(coincidenza) = 99.4%
```

Nota come la funzione `probabilita_compleanno` *chiama* la funzione `ha_coincidenza` al suo interno. Le funzioni possono usare altre funzioni!

Passo 3: La formula esatta

Possiamo anche calcolare la probabilità *esatta* con la matematica:

$$P(\text{nessuna coincidenza con } n \text{ persone}) = \frac{365}{365} \times \frac{364}{365} \times \frac{363}{365} \times \cdots \times \frac{365 - n + 1}{365}$$

$$P(\text{almeno una coincidenza}) = 1 - P(\text{nessuna coincidenza})$$

In Python:

```
def compleanno_teoria(n):
    """Calcolo esatto di P(almeno una coincidenza)."""
    p_nessuna = 1.0                # partiamo da 1
    for k in range(1, n):          # per ogni persona dopo la prima
        p_nessuna *= (365 - k) / 365   # moltiplica
    return 1 - p_nessuna           # probabilita' complementare

print(f"P(coincidenza con 23 persone) = {compleanno_teoria(23):.4f}")
```

Output

```
P(coincidenza con 23 persone) = 0.5073
```

L'operatore `*=` funziona come `+=`: la riga `p_nessuna *= (365 - k) / 365` equivale a `p_nessuna = p_nessuna * (365 - k) / 365`.

Passo 4: Il grafico

Confrontiamo la simulazione con la teoria in un bel grafico:

```
import matplotlib.pyplot as plt

# Simulazione
dimensioni = list(range(2, 61))   # da 2 a 60 persone
simulato = [probabilita_compleanno(n, 5000) for n in dimensioni]

# Teoria
teorico = [compleanno_teoria(n) for n in dimensioni]

# Grafico
plt.figure(figsize=(10, 5))
plt.scatter(dimensioni, simulato, s=20, color='#2563EB',
            alpha=0.6, zorder=3, label='Simulazione (5000 prove)')
plt.plot(dimensioni, teorico, color='#DC2626',
         linewidth=2, zorder=2, label='Teoria esatta')
plt.axhline(y=0.5, color='gray', linestyle=':', linewidth=1)
plt.axvline(x=23, color='#16A34A', linestyle='--', linewidth=1.5,
            alpha=0.8, label='n = 23 (50.7%)')
plt.xlabel('Numero di persone nella stanza', fontsize=12)
plt.ylabel('P(almeno un compleanno condiviso)', fontsize=12)
plt.title('Il Problema del Compleanno',
          fontsize=14, fontweight='bold')
plt.legend(fontsize=10)
plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()
```

Perché è così sorprendente?

La nostra intuizione fallisce perché pensiamo alla possibilità che il *nostro* compleanno coincida con quello di qualcuno. Ma la domanda è se *due qualsiasi* persone coincidano. Con 23 persone ci sono $\binom{23}{2} = 253$ coppie possibili da confrontare — sono tantissime possibilità di coincidenza!

Sfida: la tua classe!

1. Quanti studenti ci sono nella tua classe? Usa la formula per calcolare la probabilità teorica.
2. Raccogli i compleanni reali dei tuoi compagni. C'è una coincidenza? Confronta con la teoria!

Paradosso IV: Il Paradosso di Simpson

La Storia

Nel 1973, l'Università della California a Berkeley fu accusata di **discriminazione di genere** nelle ammissioni. I numeri complessivi sembravano chiari:

	Uomini	**Donne**
Candidati	8.442	4.321
Ammessi	**44%**	**35%**

Caso chiuso? Non così in fretta! Analizzando *ogni dipartimento separatamente*, le donne avevano tassi di ammissione **più alti** nella maggior parte dei dipartimenti!
Com'è possibile fare meglio *ovunque* ma peggio *nel complesso*?

Il trucco: basi diverse

Il segreto: le donne facevano domanda soprattutto ai dipartimenti *più competitivi* (con tassi di ammissione bassi per tutti), mentre gli uomini facevano domanda ai dipartimenti *più facili*. Simuliamolo.

Passo 1: Creare i dati

Usiamo un **dizionario** — una struttura dati che associa nomi a valori (come un vocabolario):

```
import random

def simula_ammissioni(n_candidati=10000):
    """Simula ammissioni tipo Berkeley."""
    # Dizionario: chiave -> valore
    risultati = {
        "uomini": {"facile": [], "difficile": []},
        "donne":  {"facile": [], "difficile": []}
    }

    for i in range(n_candidati):
        # Meta' uomini, meta' donne
        if i < n_candidati // 2:
            genere = "uomini"
        else:
            genere = "donne"

        # Uomini: 80% al dip. facile, 20% al difficile
        # Donne: 20% al dip. facile, 80% al difficile
        if genere == "uomini":
            if random.random() < 0.80:
                dip = "facile"
            else:
                dip = "difficile"
        else:
            if random.random() < 0.20:
                dip = "facile"
            else:
```

```
                dip = "difficile"

        # Tassi di ammissione: facile ~60%, difficile ~20%
        # Le donne hanno un LEGGERO VANTAGGIO (+2%)
        if dip == "facile":
            tasso = 0.62 if genere == "donne" else 0.60
        else:
            tasso = 0.22 if genere == "donne" else 0.20

        # random.random() da' un numero tra 0 e 1
        # Se e' sotto il tasso -> ammesso (True)
        ammesso = random.random() < tasso
        risultati[genere][dip].append(ammesso)

    return risultati

dati = simula_ammissioni()
```

I dizionari in breve

Un **dizionario** usa `{chiave: valore}` per organizzare dati. Nell'esempio, `risultati["uomini"]["facile"]` è la lista dei risultati degli uomini nel dipartimento facile. Ogni elemento è `True` (ammesso) o `False` (respinto).

Passo 2: Analizzare i dati

Scriviamo una funzione che calcola il tasso di ammissione da una lista di `True`/`False`:

```
def tasso_ammissione(lista):
    """Calcola la percentuale di True in una lista."""
    if len(lista) == 0:        # lista vuota? evita errori
        return 0
    return sum(lista) / len(lista)
    # sum() conta i True (True=1, False=0)
    # len() conta il totale
```

`True` vale 1, `False` vale 0

In Python, `True` viene trattato come il numero 1 e `False` come 0. Quindi `sum([True, False, True, True])` dà 3 (tre valori "veri"). Dividendo per il totale otteniamo la proporzione!

```
# Tassi per dipartimento
print("=== PER DIPARTIMENTO ===")
for dip in ["facile", "difficile"]:
    u = tasso_ammissione(dati["uomini"][dip])
    d = tasso_ammissione(dati["donne"][dip])
    chi_vince = "Donne!" if d > u else "Uomini!"
    print(f"Dip. {dip:10s}: Uomini={u:.1%}, Donne={d:.1%} -> {chi_vince}")

# Tassi complessivi
print("\n=== COMPLESSIVO ===")
tutti_u = dati["uomini"]["facile"] + dati["uomini"]["difficile"]
tutte_d = dati["donne"]["facile"] + dati["donne"]["difficile"]
print(f"Complessivo: Uomini={tasso_ammissione(tutti_u):.1%}, "
      f"Donne={tasso_ammissione(tutte_d):.1%}")
```

Output

```
=== PER DIPARTIMENTO ===
Dip.  facile :  Uomini=60.2%, Donne=62.4% -> Donne!
Dip.  difficile :  Uomini=20.1%, Donne=22.3% -> Donne!

=== COMPLESSIVO ===
Complessivo:  Uomini=52.1%, Donne=30.2%
```

Le donne vincono in *ogni* dipartimento, ma perdono nel complesso! Questo è il Paradosso di Simpson.

Passo 3: Visualizzare il paradosso

```
import matplotlib.pyplot as plt

fig, axes = plt.subplots(1, 3, figsize=(12, 4.5), sharey=True)

categorie = ["Uomini", "Donne"]
colori = ['#2563EB', '#DC2626']

# Dip. facile
tassi = [tasso_ammissione(dati["uomini"]["facile"]),
         tasso_ammissione(dati["donne"]["facile"])]
axes[0].bar(categorie, tassi, color=colori, alpha=0.8)
axes[0].set_title("Dip. Facile", fontweight='bold')
axes[0].set_ylabel("Tasso di ammissione")
for i, t in enumerate(tassi):
    axes[0].text(i, t + 0.01, f'{t:.1%}',
                 ha='center', fontweight='bold')

# Dip. difficile
tassi = [tasso_ammissione(dati["uomini"]["difficile"]),
         tasso_ammissione(dati["donne"]["difficile"])]
axes[1].bar(categorie, tassi, color=colori, alpha=0.8)
axes[1].set_title("Dip. Difficile", fontweight='bold')
for i, t in enumerate(tassi):
    axes[1].text(i, t + 0.01, f'{t:.1%}',
                 ha='center', fontweight='bold')

# Complessivo
tassi = [tasso_ammissione(tutti_u), tasso_ammissione(tutte_d)]
axes[2].bar(categorie, tassi, color=colori, alpha=0.8)
axes[2].set_title("Complessivo", fontweight='bold')
for i, t in enumerate(tassi):
    axes[2].text(i, t + 0.01, f'{t:.1%}',
                 ha='center', fontweight='bold')

# Annotazioni
axes[0].text(0.5, 0.95, 'Donne meglio!',
             transform=axes[0].transAxes,
             ha='center', color='green', fontweight='bold')
axes[1].text(0.5, 0.95, 'Donne meglio!',
             transform=axes[1].transAxes,
             ha='center', color='green', fontweight='bold')
axes[2].text(0.5, 0.95, 'Uomini meglio!?',
             transform=axes[2].transAxes,
             ha='center', color='red', fontweight='bold')
```

```
45
46 for ax in axes:
47     ax.set_ylim(0, 0.85)
48     ax.grid(axis='y', alpha=0.3)
49
50 plt.suptitle("Il Paradosso di Simpson",
51              fontsize=14, fontweight='bold')
52 plt.tight_layout()
53 plt.show()
```

La lezione

I dati aggregati possono essere fuorvianti. I numeri complessivi nascondono il fatto che le donne sceglievano dipartimenti più difficili. La "variabile confondente" (la scelta del dipartimento) inverte la tendenza. Ecco perché gli scienziati cercano sempre variabili nascoste prima di trarre conclusioni!

Sfida: crea il tuo paradosso

Riesci a creare uno scenario con **tre** dipartimenti dove il paradosso si verifica? Oppure un esempio sportivo: un giocatore con una media migliore in ogni stagione ma peggiore nel complesso?

Paradosso V: Sei Gradi di Separazione

La Storia

L'idea è semplice e sorprendente: **due persone qualsiasi sulla Terra** possono essere collegate attraverso al massimo circa **6 conoscenze intermedie**.
Tu conosci qualcuno, che conosce qualcuno, che conosce qualcuno... e in soli 6 passaggi puoi raggiungere chiunque — il Presidente degli Stati Uniti, un contadino in Mongolia, un pescatore in Brasile.
Facebook lo ha confermato nel 2016: la distanza media tra due utenti qualsiasi era di **3,57 passaggi**.

Passo 1: La potenza della crescita esponenziale

Se ogni persona conosce circa 150 persone (il "numero di Dunbar"), quante persone puoi raggiungere?

```python
contatti = 150

print("Passi | Persone raggiungibili")
print("-" * 35)
for passo in range(7):
    raggiungibili = contatti ** passo  # 150 elevato a passo
    print(f"  {passo}    | {raggiungibili:>18,}")

print(f"\nPopolazione mondiale: 8,000,000,000")
```

Output

```
Passi | Persone raggiungibili
-----------------
0 | 1
1 | 150
2 | 22,500
3 | 3,375,000
4 | 506,250,000
5 | 75,937,500,000

Popolazione mondiale:  8,000,000,000
```

Al passo 5, puoi *teoricamente* raggiungere più persone di quante ne esistano sulla Terra!

Però...

Questa è una semplificazione! In realtà i tuoi amici si conoscono tra loro (sovrapposizione), quindi la portata reale è minore. Ecco perché dobbiamo simulare una vera *rete* per vedere cosa succede davvero.

Passo 2: Costruire una rete sociale

Costruiremo una rete "piccolo mondo" — un modello inventato da Watts e Strogatz nel 1998 che cattura come funzionano le reti sociali reali.

Una **rete** è un dizionario dove ogni persona (numero) è collegata a un insieme di altre persone:

```python
import random

def costruisci_rete(n_persone, n_contatti, p_ricollega=0.1):
    """Costruisce una rete piccolo-mondo."""
    # Crea un dizionario: persona -> insieme dei suoi contatti
    # set() e' un "insieme": come una lista, ma senza duplicati
    rete = {}
    for i in range(n_persone):
        rete[i] = set()

    # Collega ogni persona ai vicini piu' prossimi (anello)
    for i in range(n_persone):
        for j in range(1, n_contatti // 2 + 1):
            vicino = (i + j) % n_persone  # % = modulo (torna a 0)
            rete[i].add(vicino)      # aggiungi connessione
            rete[vicino].add(i)      # connessione reciproca

    # Ricollega alcune connessioni a caso (crea "scorciatoie")
    for i in range(n_persone):
        for j in range(1, n_contatti // 2 + 1):
            if random.random() < p_ricollega:
                vicino = (i + j) % n_persone
                if vicino in rete[i] and len(rete[i]) > 1:
                    rete[i].discard(vicino)
                    rete[vicino].discard(i)
                    # Scegli una nuova connessione casuale
                    nuovo = random.randint(0, n_persone - 1)
                    while nuovo == i or nuovo in rete[i]:
                        nuovo = random.randint(0, n_persone - 1)
                    rete[i].add(nuovo)
                    rete[nuovo].add(i)

    return rete
```

Cosa sono i `set()`?

Un `set` (insieme) è come una lista, ma con due differenze importanti:

- **Niente duplicati**: se aggiungi lo stesso elemento due volte, resta uno solo
- **Controllo veloce**: `x in mio_set` è molto più veloce che `x in mia_lista`

Perfetto per memorizzare i contatti di una persona (non vuoi la stessa persona due volte). `.add()` aggiunge un elemento, `.discard()` lo rimuove (se c'è).

Passo 3: Trovare il cammino più breve (BFS)

Per trovare il cammino più breve tra due persone, usiamo la **Breadth-First Search** (BFS, "ricerca in ampiezza") — uno degli algoritmi più importanti dell'informatica.

L'idea: partiamo dalla persona A ed esploriamo tutti i suoi contatti (distanza 1), poi tutti i contatti dei contatti (distanza 2), e così via, finché non troviamo la persona B:

```python
def cammino_piu_breve(rete, partenza, arrivo):
    """Trova il cammino piu' breve con BFS."""
    if partenza == arrivo:
        return [partenza]

```

```
    visitati = {partenza}  # persone gia' esplorate (set)
    # Coda: lista di (persona attuale, cammino percorso)
    coda = [(partenza, [partenza])]

    while len(coda) > 0:          # finche' c'e' qualcuno da esplorare
        attuale, cammino = coda.pop(0)  # prendi il primo dalla coda

        for vicino in rete[attuale]:     # per ogni contatto
            if vicino == arrivo:
                return cammino + [vicino]  # trovato!

            if vicino not in visitati:
                visitati.add(vicino)
                coda.append((vicino, cammino + [vicino]))

    return []      # nessun cammino trovato

# Costruiamo e testiamo
rete = costruisci_rete(200, 6, 0.1)
cammino = cammino_piu_breve(rete, 0, 100)
print(f"Cammino da persona 0 a persona 100: {cammino}")
print(f"Passi: {len(cammino) - 1}")
```

Passo 4: Misurare tutte le distanze

Misuriamo la distanza media tra tante coppie casuali:

```
import random

N = 500
rete = costruisci_rete(N, 10, 0.1)

distanze = []       # qui salviamo tutte le distanze trovate
for prova in range(2000):
    a = random.randint(0, N - 1)
    b = random.randint(0, N - 1)
    if a != b:
        cammino = cammino_piu_breve(rete, a, b)
        if cammino:
            distanze.append(len(cammino) - 1)

media = sum(distanze) / len(distanze)
print(f"Rete: {N} persone, 10 contatti ciascuna")
print(f"Distanza media: {media:.2f} passi")
print(f"Distanza massima: {max(distanze)} passi")
```

Passo 5: Istogramma delle distanze

```
import matplotlib.pyplot as plt

plt.figure(figsize=(8, 4.5))
plt.hist(distanze, bins=range(1, max(distanze) + 2),
         color='#16A34A', alpha=0.8, edgecolor='white',
         linewidth=1.2, density=True)
plt.axvline(x=media, color='#DC2626', linestyle='--',
            linewidth=2, label=f'Media = {media:.1f} passi')
plt.xlabel('Lunghezza del cammino (passi)', fontsize=12)
```

```
plt.ylabel('Frazione di coppie', fontsize=12)
plt.title(f'Sei Gradi: Rete di {N} Persone',
          fontsize=14, fontweight='bold')
plt.legend(fontsize=11)
plt.grid(axis='y', alpha=0.3)
plt.tight_layout()
plt.show()
```

L'effetto "piccolo mondo"

Anche se ogni persona conosce solo 10 altre persone, il cammino medio è molto breve! Le "scorciatoie" casuali riducono drasticamente le distanze. È esattamente ciò che succede nella vita reale: il tuo amico che si è trasferito all'estero ti collega a un circolo sociale completamente diverso.

Sfida: esplora i parametri

Prova a cambiare i parametri della rete e osserva:

1. Cosa succede se `p_ricollega = 0`? (Nessuna scorciatoia)
2. Cosa succede se `p_ricollega = 1`? (Rete completamente casuale)
3. Come influisce aumentare `n_contatti` sulla distanza media?
4. Riesci a trovare una combinazione dove la distanza media è inferiore a 3?

Cosa Hai Imparato

Abilità Python	Dove l'hai usata
Variabili e matematica	Ovunque!
Cicli `for`	Ripetere migliaia di simulazioni
Condizioni `if`/`else`	Controllare chi ha vinto
Funzioni `def` + `return`	`ha_coincidenza()`, `costruisci_rete()`
Liste e `.append()`	Raccogliere risultati per i grafici
Dizionari	Rete di contatti, dati raggruppati
Operatore `+=`	Contare vittorie e coincidenze
Libreria `random`	Simulare eventi casuali
Libreria `matplotlib`	Grafici di convergenza, barre, istogrammi
Algoritmo BFS	Trovare cammini minimi nelle reti
Simulazione Monte Carlo	Stimare probabilità empiricamente

Punti chiave

1. **Monty Hall:** Cambia sempre — vinci 2/3 delle volte, non 1/2.
2. **Ragazzo o Ragazza:** *Come* ottieni l'informazione cambia la probabilità. La risposta è 1/3, non 1/2.
3. **Problema del Compleanno:** Con sole 23 persone, P(coincidenza) $>$ 50%. La nostra intuizione sulle combinazioni è pessima.
4. **Paradosso di Simpson:** I dati aggregati possono invertire le tendenze. Cerca sempre le variabili nascoste.
5. **Sei Gradi:** Poche connessioni casuali creano scorciatoie che collegano intere reti. Il mondo è piccolo.

Per approfondire

- **Più Python:** https://www.codecademy.com/learn/learn-python-3
- **Probabilità:** 3Blue1Brown su YouTube — splendide spiegazioni visive
- **Data Science:** Kaggle.com — esercitati con dati reali
- **Reti:** "Linked" di Albert-László Barabási (libro divulgativo)

Buona programmazione!

Il modo migliore per imparare è *sperimentare*.
Cambia i numeri. Rompi le cose. Chiedi "e se?"
Questo è ciò che la programmazione — e la scienza — sono.