



**UNIVERSITÀ DEGLI STUDI  
DELLA BASILICATA**

*Corso di Sistemi Informativi  
A.A. 2018/19*

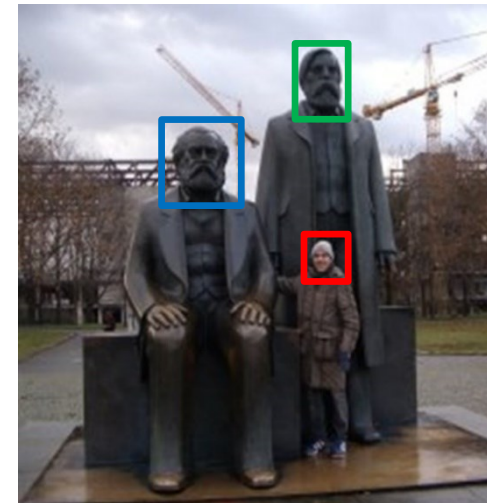
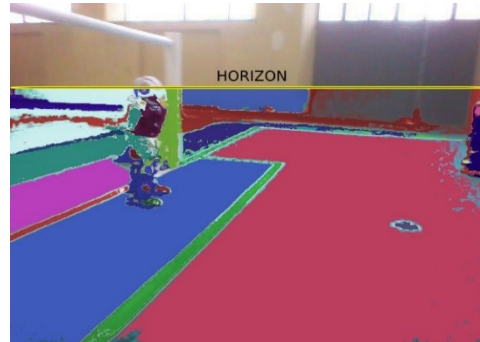
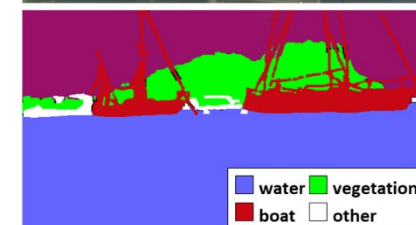
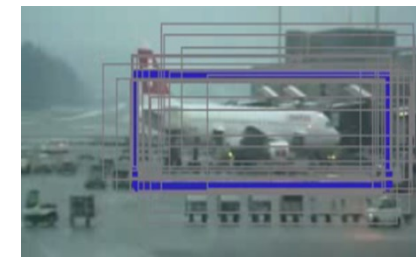
# Introduzione al Python

Docente

**Domenico Daniele Bloisi**



Parte del materiale deriva dai corsi dei proff. Paolo Caressa e Raffaele Nicolussi (Sapienza Università di Roma) e Giorgio Fumera (Università degli Studi di Cagliari)



Marzo 2019

# Il corso

---

- Home page del corso  
<https://dbloisi.github.io/corsi/sistemi-informativi.html>
- Docente: Domenico Daniele Bloisi
- Periodo: **Il semestre** marzo 2019 – giugno 2019  
Martedì 17:00-19:00 (Aula GUGLIELMINI)  
Mercoledì 8:30-10:30 (Aula GUGLIELMINI)

# Obiettivi del corso

---

Il corso intende fornire agli studenti conoscenze relative alla **programmazione in Python** per lo sviluppo di applicazioni basate sul sistema operativo ROS, sulla libreria per la percezione OpenCV e sulla libreria per il Deep Learning Keras



<https://www.youtube.com/watch?v=l9KYJlLnEbw>

# Il Python

---

È un linguaggio di programmazione:

- Interpretato
- Di alto livello
- Orientato agli oggetti
- Semplice da imparare e usare
- Potente e produttivo
- Ottimo anche come primo linguaggio (molto simile allo pseudocodice)
- Estensibile
- Con una tipizzazione forte e dinamica

Inoltre:

- È open source ([www.python.org](http://www.python.org))
- È multiplatforma
- È facilmente integrabile con C/C++ e Java

# Un po' di storia

---

Python è stato creato nel 1989 dall'informatico olandese Guido van Rossum (<https://gvanrossum.github.io/>)

Ideato originariamente come linguaggio di scripting, Python si è poi evoluto come linguaggio completo

Il nome fu scelto per via della passione di van Rossum per i Monty Python (<http://www.montypython.com>) e per la loro serie televisiva Monty Python's Flying Circus



# Versioni di Python

---

Sono al momento disponibili due versioni principali di Python:

- Python 2
- Python 3





# Python 2

---

Pubblicato alla fine del **2000**, Python 2 ha espresso un processo di sviluppo del linguaggio più trasparente e completo rispetto alle versioni precedenti di Python tramite l'implementazione della **PEP (Python Enhancement Proposal)**, una specifica tecnica che fornisce delle informazioni ai membri della comunità Python oppure descrive una nuova funzionalità del linguaggio.

Inoltre, Python 2 include un **garbage collector** in grado di rilevare cicli per automatizzare la gestione della memoria, un migliore supporto **Unicode** per uniformare i caratteri e la **list comprehension** per creare una lista basata su liste esistenti.

Nella versione 2.2 sono state aggiunte altre funzionalità, tra cui l'unificazione dei tipi e delle classi di Python in una gerarchia.

# Python 3

---

Python 3 è la versione del linguaggio attualmente in fase di sviluppo.

**Python 3** è stato rilasciato alla fine del **2008** per affrontare e modificare alcuni difetti di progettazione delle precedenti versioni del linguaggio. L'obiettivo dello sviluppo di Python 3 è stato quello di ripulire il codice base e rimuovere la ridondanza.

Le principali modifiche di Python 3 comprendono **il cambiamento dell'istruzione print in una funzione built-in**, il miglioramento del modo in cui gli interi sono divisi e un maggiore supporto a Unicode.

**Python 3 non è retrocompatibile con Python 2.**



# Python 2.7

---

Python 2.7 è stato pubblicato il **3 luglio 2010**, successivamente al rilascio di Python 3.0 (avvenuto nel **2008**)

Python 2.7 è previsto come **l'ultimo dei rilasci 2.x**

L'intento di Python 2.7 è stato quello di rendere più facile per gli utenti Python 2.x effettuare il porting di certe caratteristiche verso Python 3, fornendo **un certo grado di compatibilità tra i due**.

Questo supporto alla compatibilità include moduli avanzati per la versione 2.7 come **unittest** che supporta l'automazione dei test, **argparse** per fare il **parsing** delle opzioni della riga di comando e le più convenienti classi in **collections**.

Python 2.7 è da considerarsi un linguaggio legacy e il suo sviluppo, che oggi consiste principalmente in correzioni di bug, cesserà completamente nel 2020.

# Python 2.7

---

A causa della condizione unica di Python 2.7 quale release intermedia tra le versioni precedenti di Python 2 e Python 3.0, Python 2.7 continua ad essere una scelta molto popolare per i programmatori grazie alla sua compatibilità con tantissime librerie.

Quando si parla di Python 2, in genere si fa riferimento alla release 2.7 di Python essendo la versione 2.x più utilizzata.

**In questo corso utilizzeremo Python 3**

# Installare Python

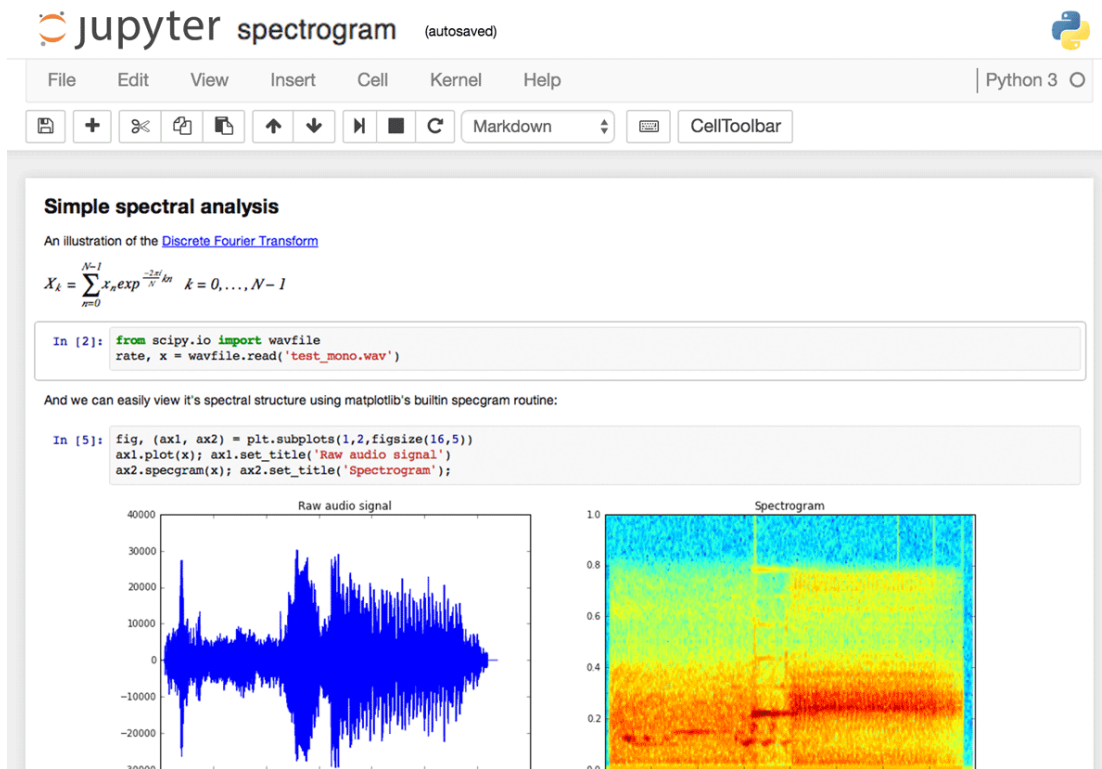
---

Python può essere installato su machine [Windows](#), [Linux](#) e [Mac](#) scaricando la versione desiderata dalla pagina ufficiale dei download <https://wiki.python.org/moin/BeginnersGuide/Download>

In alternativa, è possibile utilizzare distribuzioni di Python che includono già alcuni pacchetti per applicazioni specifiche (per esempio Anaconda <https://docs.anaconda.com/anaconda/> include pacchetti per applicazioni scientifiche)

# Jupyter Notebook

Jupyter Notebook è un ambiente di sviluppo interattivo, nel quale si possono combinare codice, testo, figure, formule matematiche e video.



Per installare Jupyter Notebook fare riferimento alla pagina <https://jupyter.org/install>



<https://www.dataquest.io/blog/jupyter-notebook-tips-tricks-shortcuts/>

# Google Colab

---

Colaboratory è un tool per la ricerca specifico per applicazioni di machine learning. É un ambiente Jupyter notebook (<https://jupyter.org/>) che non richiede l'installazione di alcuna libreria, poiché è già pronto per l'uso.



Colaboratory può essere usato con i più diffusi browser ed è testato sulle versioni desktop di [Chrome](#) e [Firefox](#).

Colaboratory è un progetto di ricerca che viene distribuito in accordo alla licenza [BSD 3-Clause "New" or "Revised" License](#).

Tutti i file Colaboratory sono memorizzati in [Google Drive](#).



# Google Colab

---

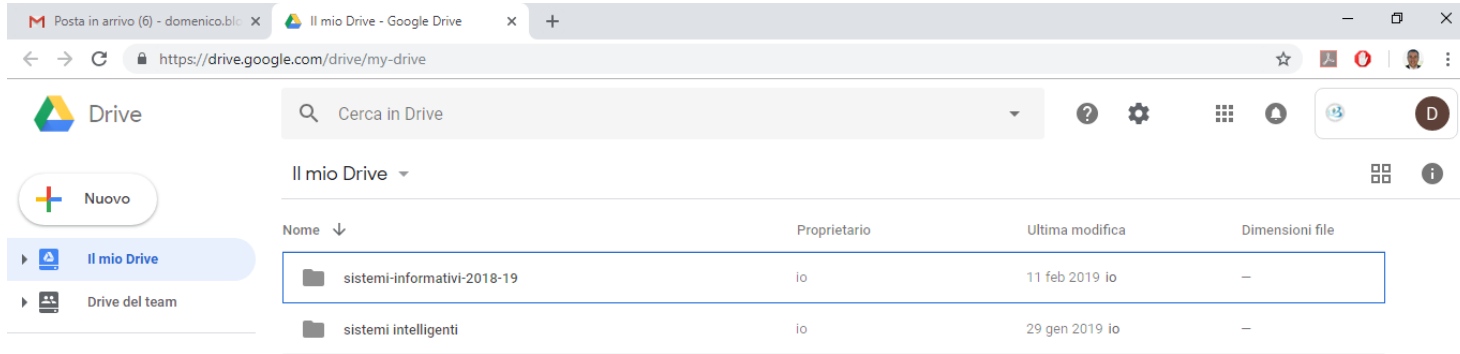
Al momento, Colaboratory supporta Python 2.7 e Python 3.6

Il codice viene eseguito su una macchina virtuale dedicata collegata all'account Google.

Google impone dei limiti temporali per l'uso continuativo della macchina virtuale (12 ore)

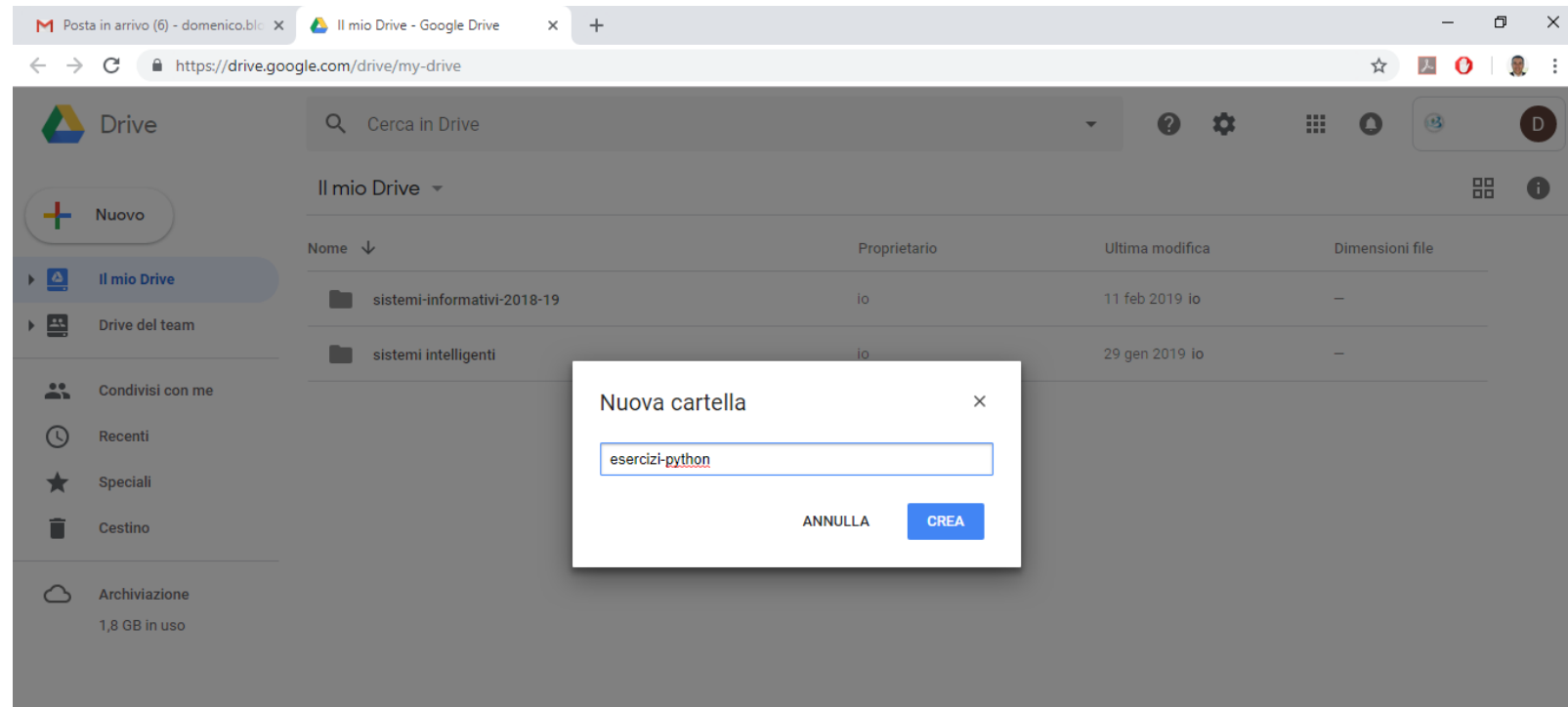


# Primo programma Python con Google Colab



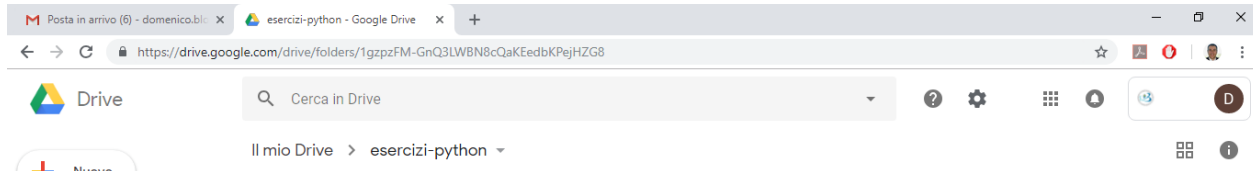
1. Apriamo il nostro Google Drive

2. Creiamo una nuova cartella denominata per esempio 'esercizi-python'

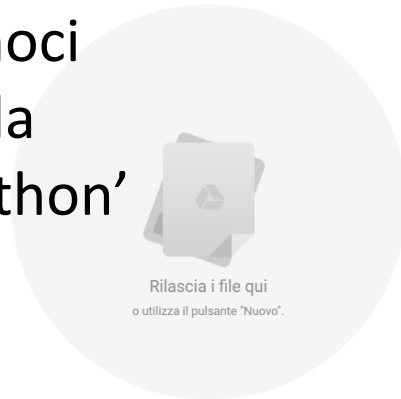




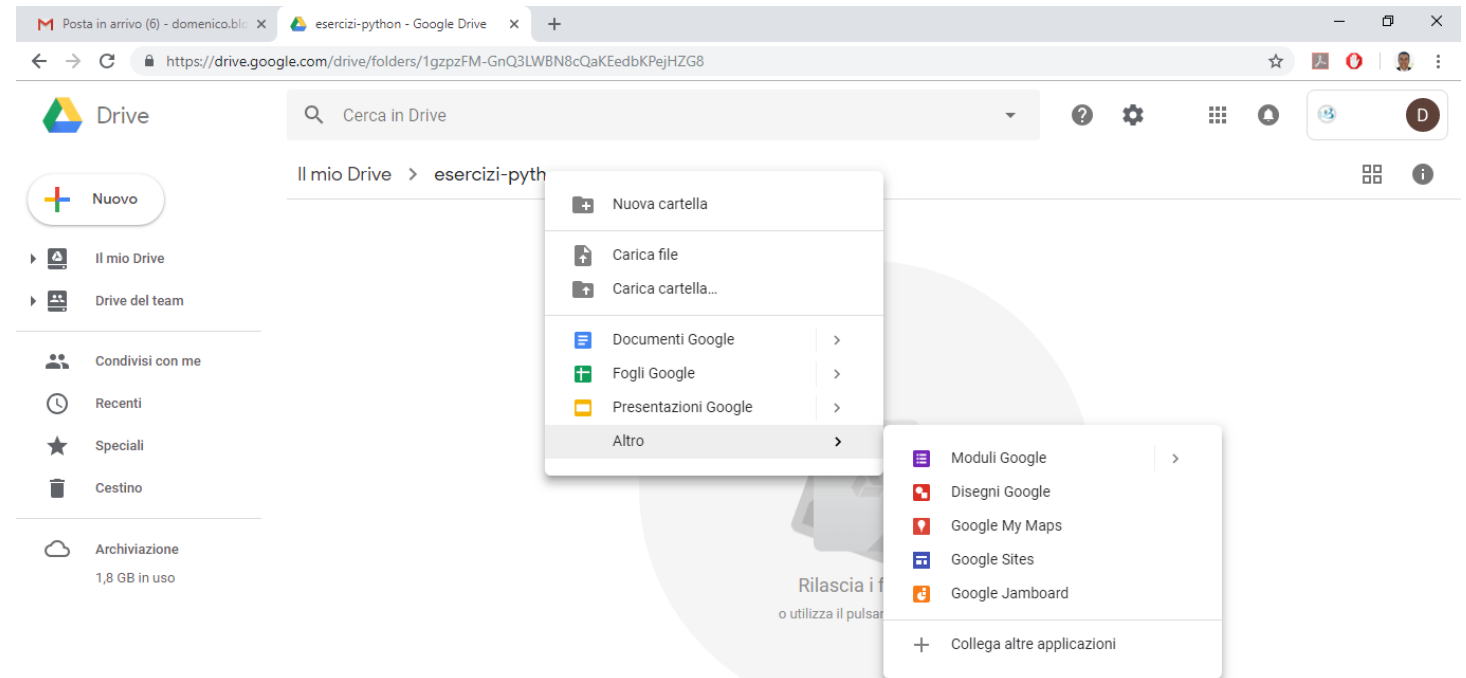
# Creazione del primo file sorgente



3. Spostiamoci  
nella cartella  
'esercizi-python'

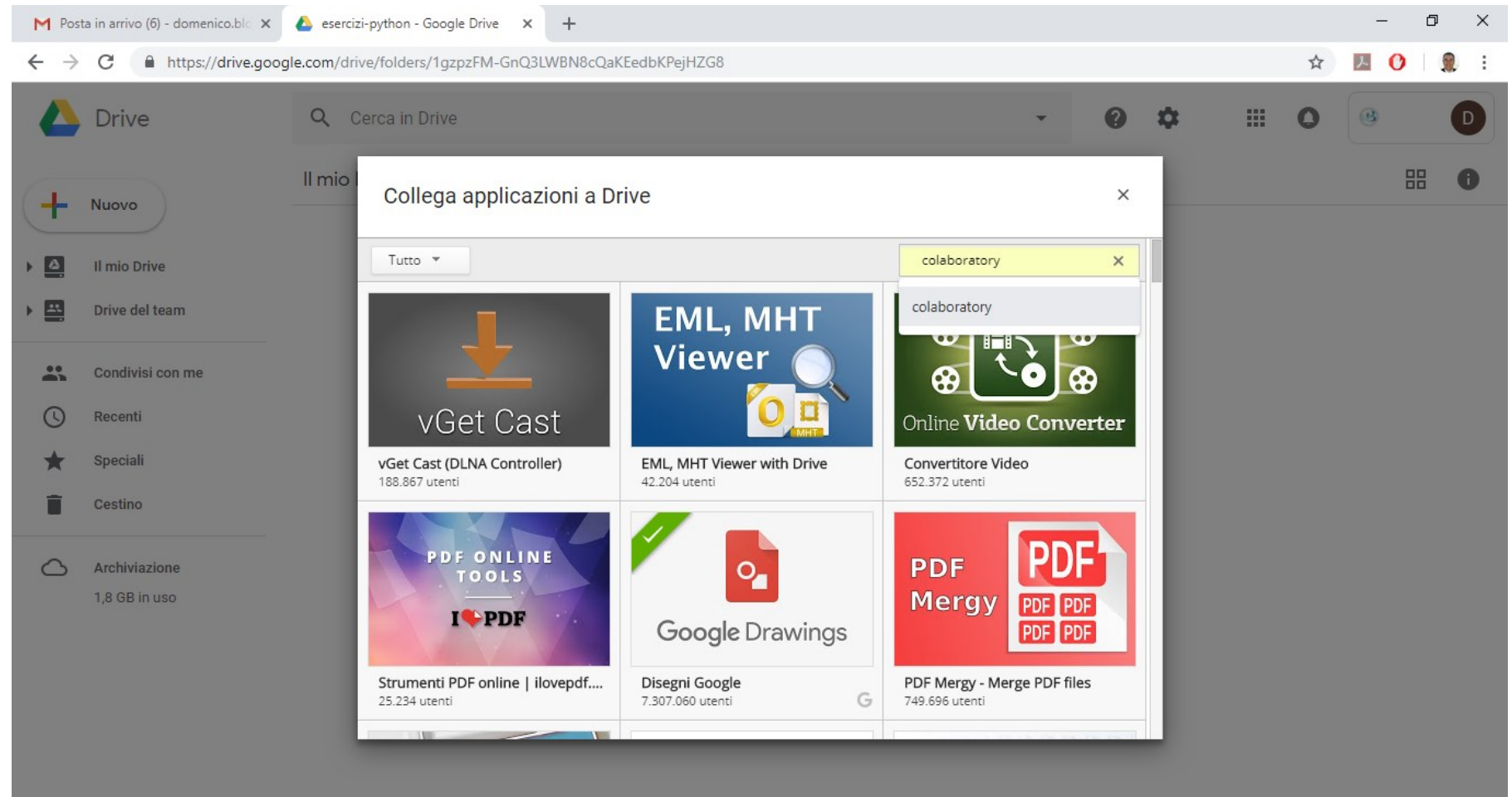


4. Usando il tasto destro del  
mouse, selezioniamo 'Altro' e  
poi 'Collega altre applicazioni'



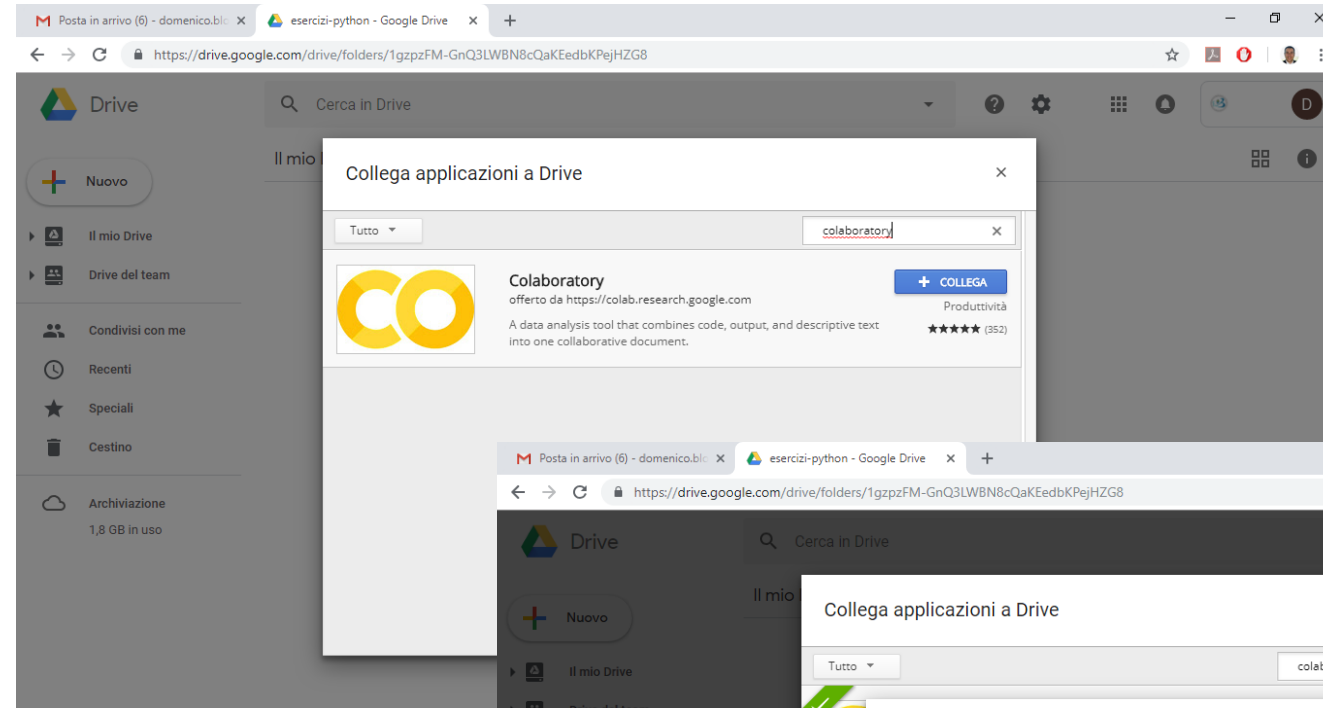
# Ricerca applicazione Google Colaboratory

5. Nella casella a destra inserire la parola 'colaboratory' per poter collegare l'applicazione Google Colaboratory al proprio Google Drive. Premere invio

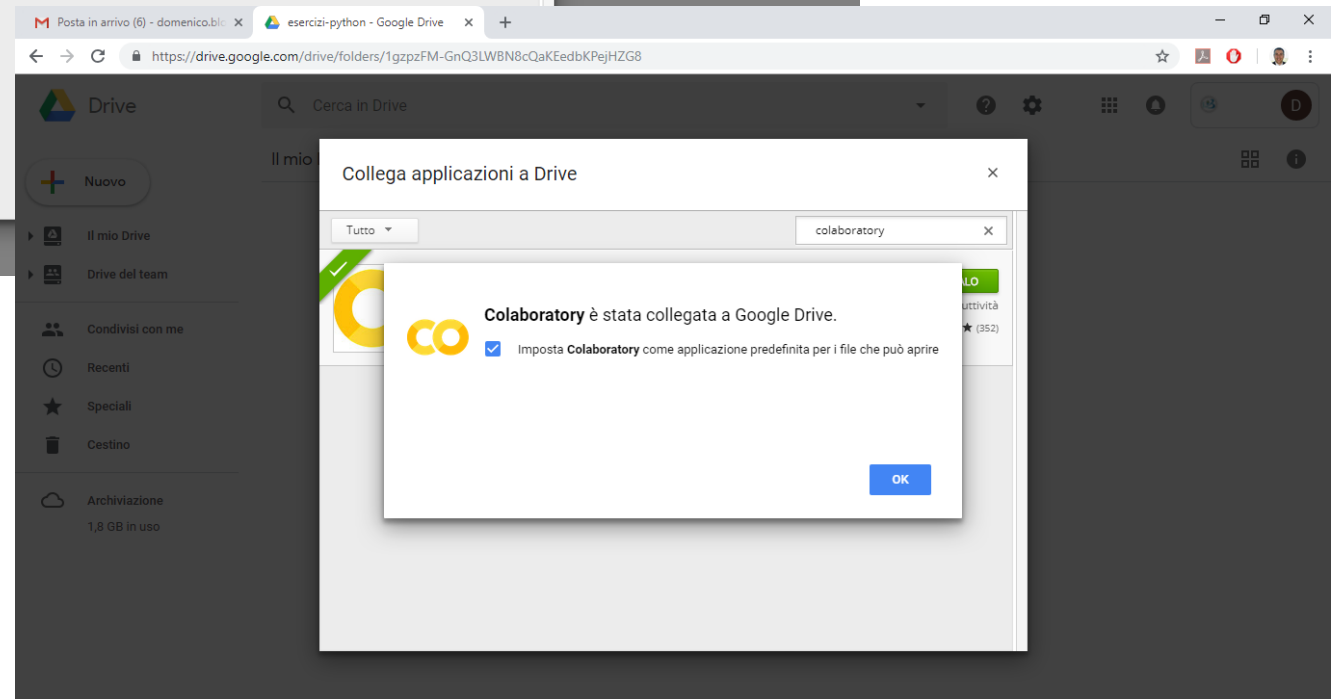


# Collegiamo l'applicazione al nostro Google Drive

## 6. Selezionare l'opzione 'collega'

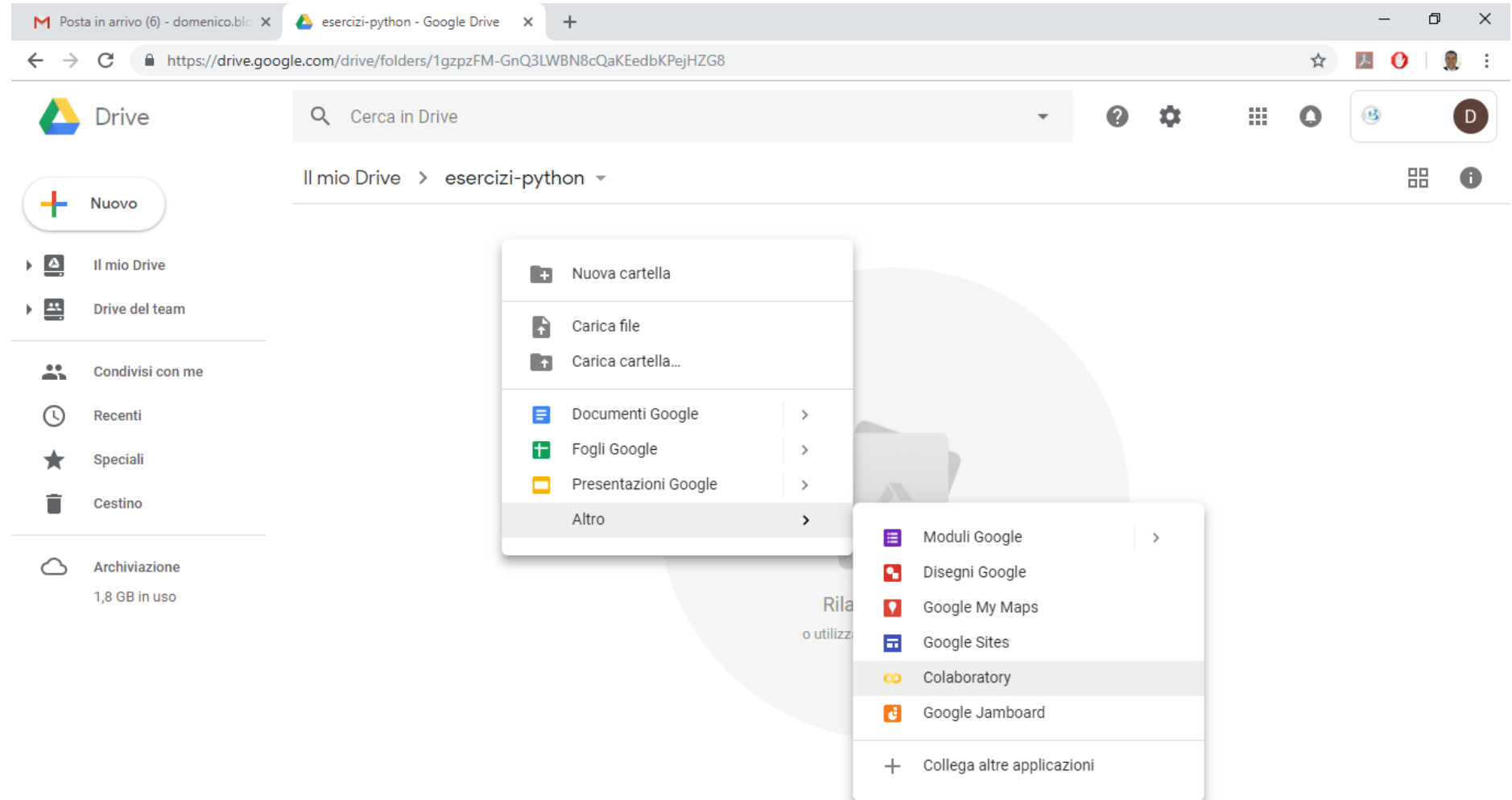


## 7. Selezionare 'OK'



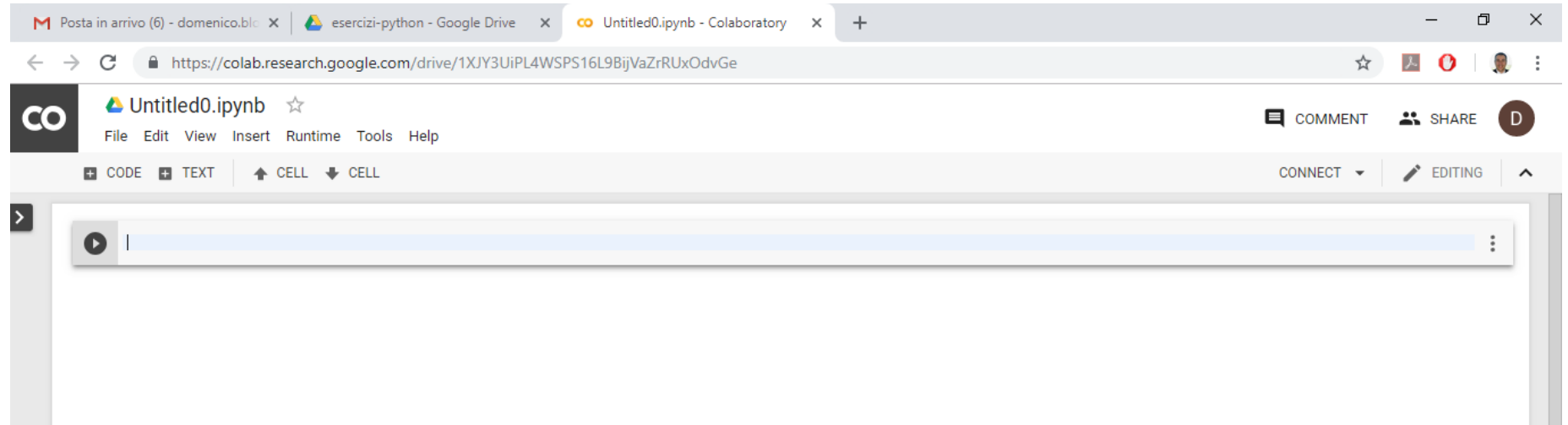
# Selezioniamo l'applicazione Colaboratory

8. Usiamo il tasto destro del mouse per selezionare 'Altro' e poi troveremo la nuova voce 'Colaboratory'

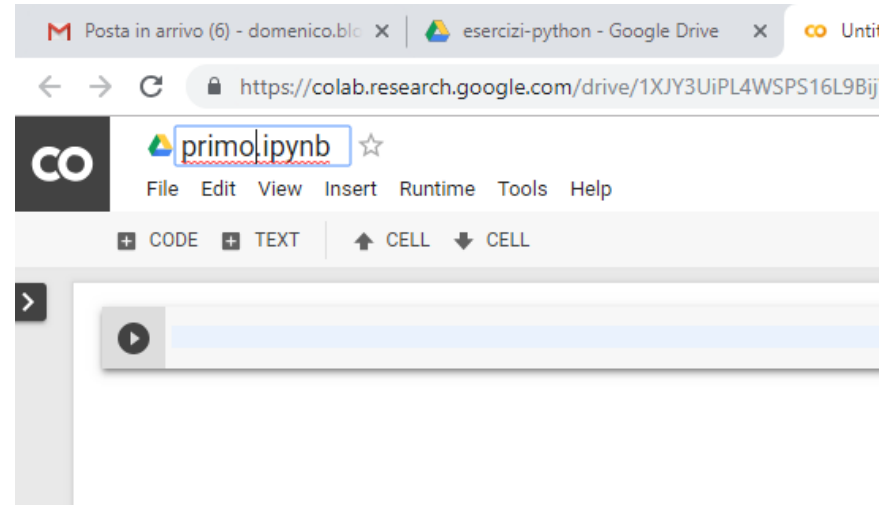


# Creiamo un file Colaboratory

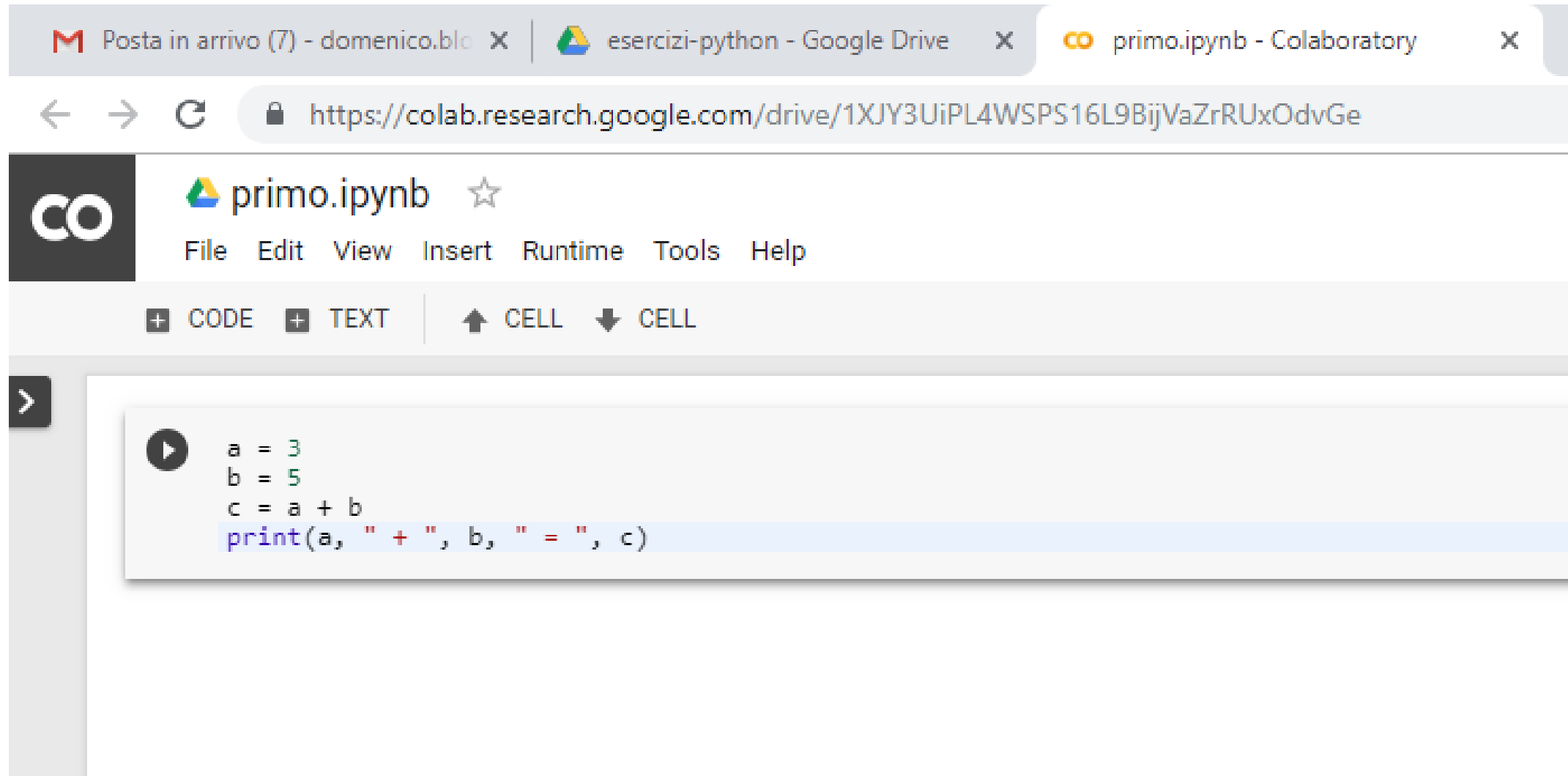
9. Verrà creato un file 'Untitled0.ipynb' che possiamo usare per creare il nostro primo programma in Python



10. Rinominiamo il file come 'primo.ipynb'



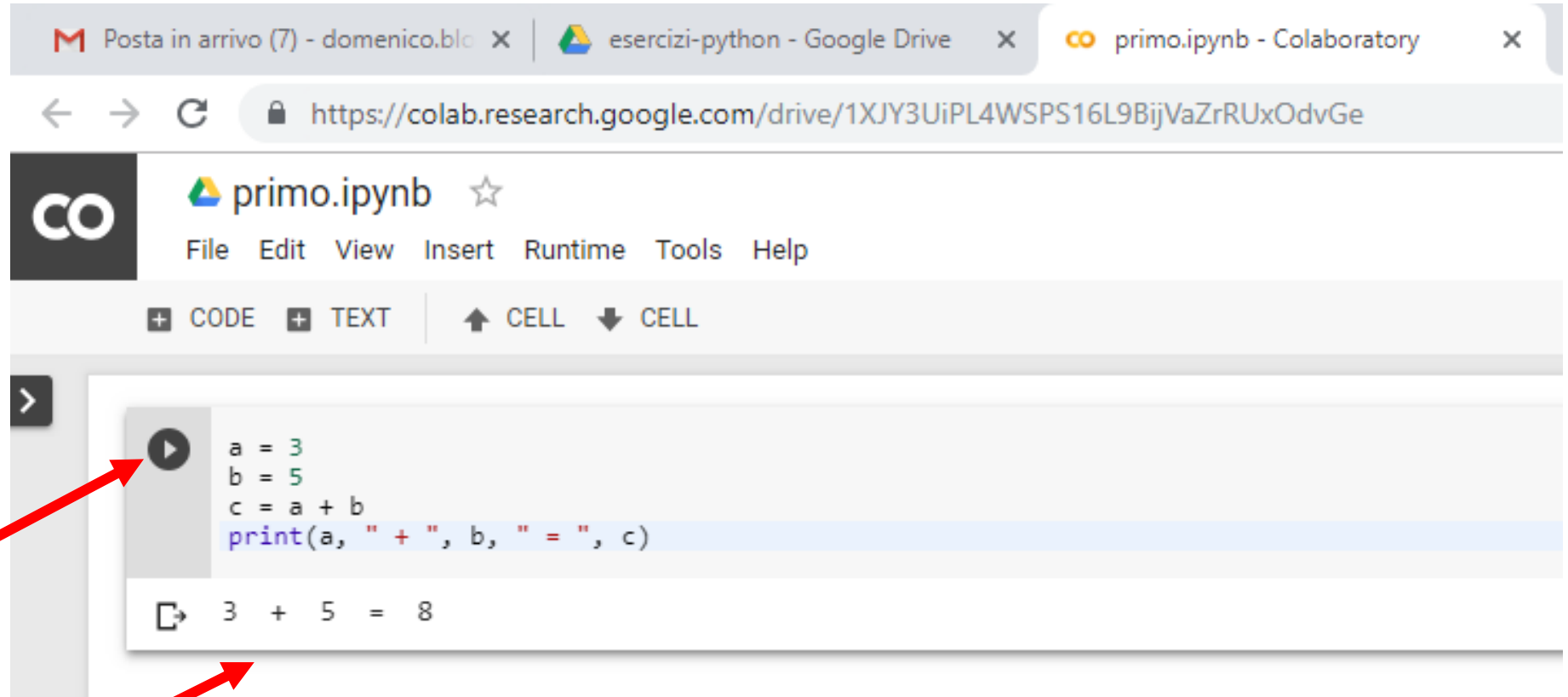
# Scriviamo il primo programma Colab



The screenshot shows a web browser with three tabs: "Posta in arrivo (7) - domenico.blo", "esercizi-python - Google Drive", and "primo.ipynb - Colaboratory". The address bar displays the URL <https://colab.research.google.com/drive/1XJY3UiPL4WSPS16L9BijVaZrRUxOdvGe>. The Colaboratory interface includes a "CO" logo, a star icon, and a menu bar with "File", "Edit", "View", "Insert", "Runtime", "Tools", and "Help". Below the menu bar are buttons for "+ CODE", "+ TEXT", "↑ CELL", and "↓ CELL". A code cell is active, containing the following Python code:

```
a = 3
b = 5
c = a + b
print(a, " + ", b, " = ", c)
```

# Esecuzione del primo programma Colab



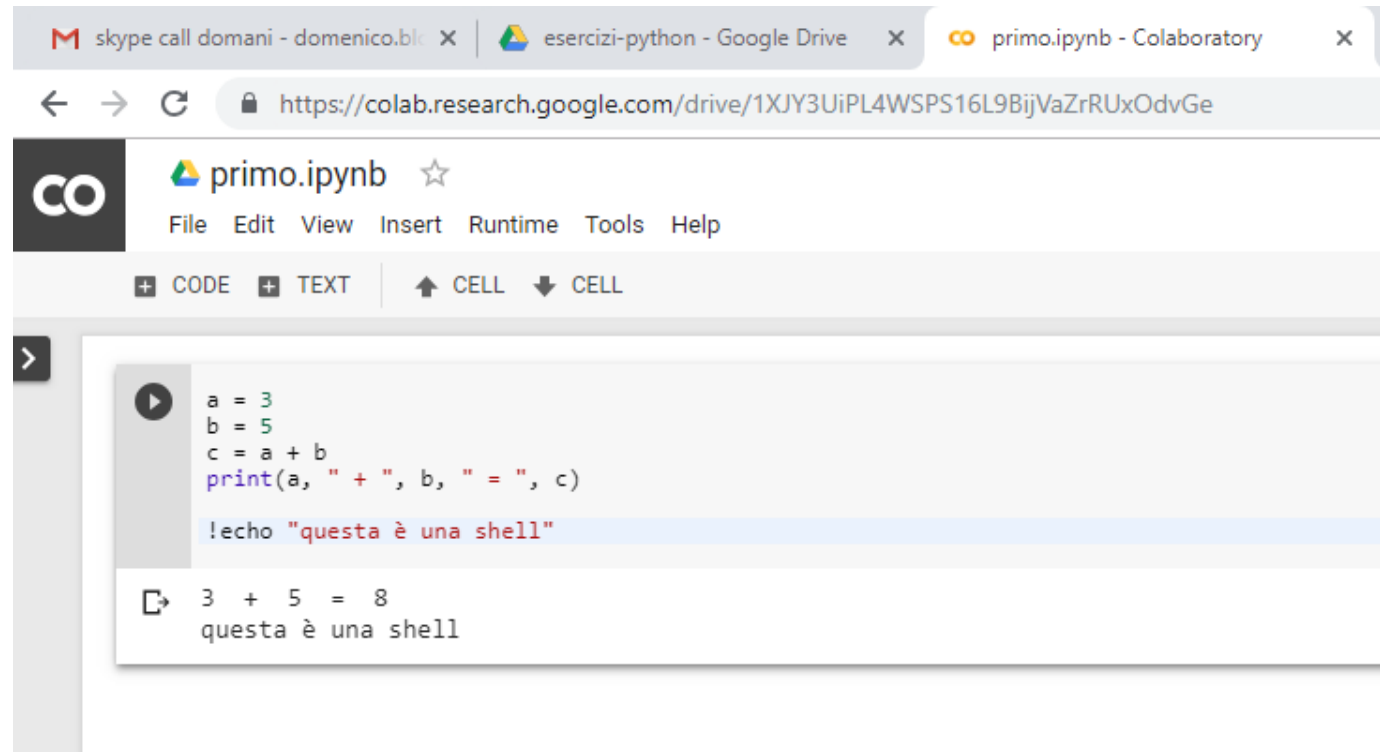
Selezionando 'Run cell' possiamo eseguire il nostro codice e ottenere l'output



# Comandi di shell in Colab

Possiamo utilizzare comandi di shell facendoli precedere dal simbolo '!'

Per esempio, per lanciare il comando `echo` possiamo inserire il simbolo '!' e poi procedere come se avessimo a disposizione una shell reale



The screenshot shows a web browser with three tabs: 'skype call domani - domenico.bl...', 'esercizi-python - Google Drive', and 'primo.ipynb - Colaboratory'. The address bar shows the URL 'https://colab.research.google.com/drive/1XJY3UiPL4WSPS16L9BijVaZrUxOdvGe'. The notebook interface for 'primo.ipynb' is visible, with a menu bar (File, Edit, View, Insert, Runtime, Tools, Help) and tabs for '+ CODE' and '+ TEXT'. A code cell is selected, containing the following Python code:

```
a = 3
b = 5
c = a + b
print(a, " + ", b, " = ", c)
!echo "questa è una shell"
```

Below the code cell, the output is displayed:

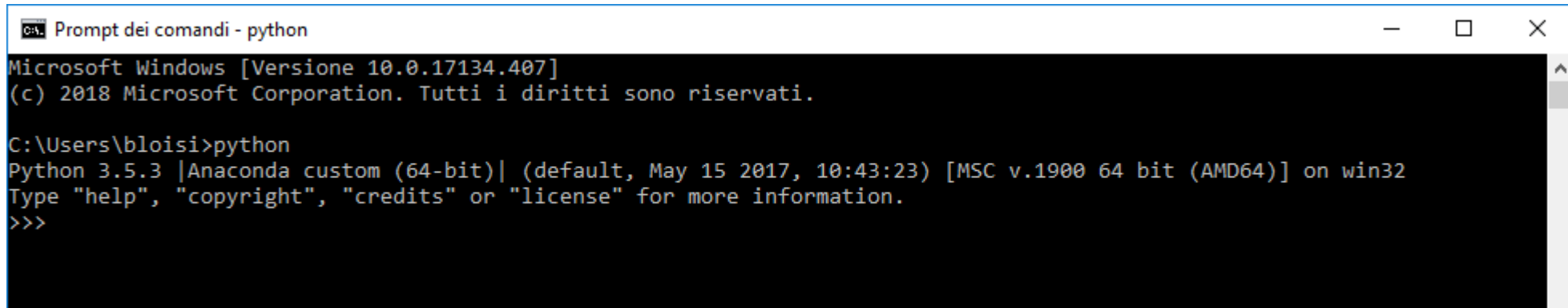
```
3 + 5 = 8
questa è una shell
```

# Interprete interattivo del Python

---

Python dispone di un interprete interattivo molto comodo e potente

Per attivarlo possiamo digitare la parola `python` al prompt di una shell su un pc in cui è installato Python



```
Prompt dei comandi - python
Microsoft Windows [Versione 10.0.17134.407]
(c) 2018 Microsoft Corporation. Tutti i diritti sono riservati.

C:\Users\bloisi>python
Python 3.5.3 |Anaconda custom (64-bit)| (default, May 15 2017, 10:43:23) [MSC v.1900 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Apparirà il simbolo `>>>` indicante che l'interprete è pronto a ricevere comandi

# Interprete interattivo del Python

---

L'interprete è un file denominato

- “python” su Unix
- “python.exe” su Windows → apre una console
- “pythonw.exe” su Windows → non apre una console

Se invocato senza argomenti presenta una interfaccia interattiva

I comandi si possono inserire direttamente dallo standard input

- Il prompt è caratterizzato da “>>>”
- Se un comando si estende sulla riga successiva è preceduto da “...”

I file sorgente Python sono file di testo, generalmente con estensione “.py”

# Interprete interattivo del Python

---

Introducendo “`#!/usr/bin/env python`” come prima riga su Unix il sorgente viene eseguito senza dover manualmente invocare l’interprete

Il simbolo “`#`” inizia un commento che si estende fino a fine riga

Le istruzioni sono separate dal fine riga e non da “`;`”

- Il “`;`” può comunque essere usato per separare istruzioni sulla stessa riga, ma è sconsigliato

Per far continuare un’istruzione anche sulla linea successiva è necessario inserire il simbolo “`\`” a fine riga

Se le parentesi non sono state chiuse, l’istruzione si estende anche sulla riga successiva

# Modalità interprete

---

Digitando dei comandi Python in modalità interprete interattivo si ottiene subito una risposta:

```
>>> 5 * 3
```

```
15
```

```
>>>
```

```
>>> a = 5
```

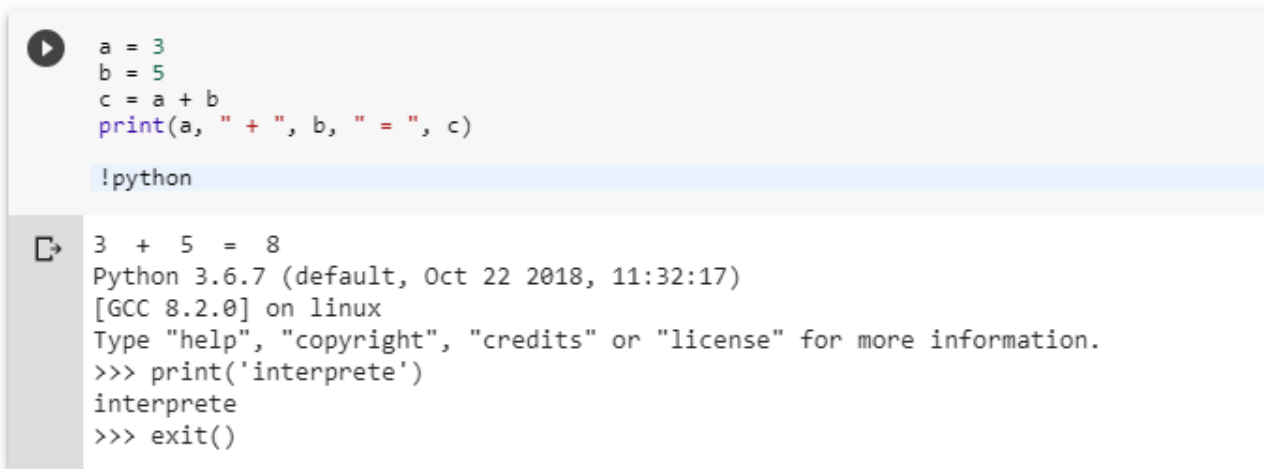
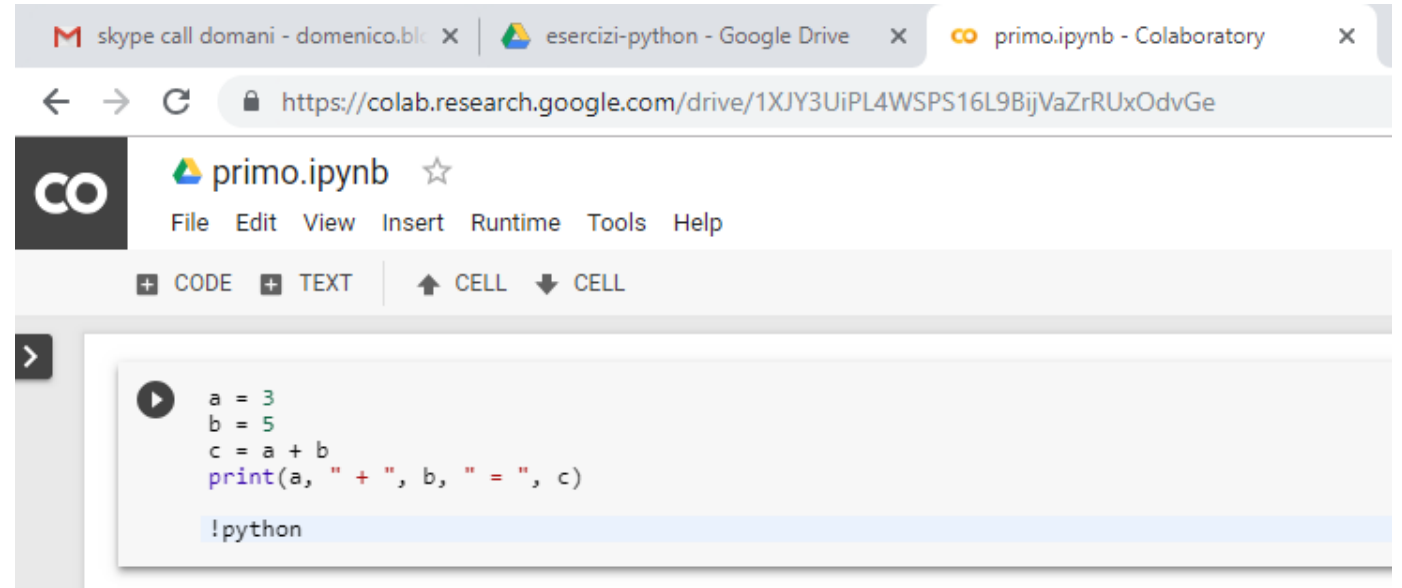
```
>>> b = 6
```

```
>>> 2 * (a+b) + 3*a
```

```
37
```

# Interprete in Colab

Per lanciare l'interprete Python su Colab possiamo scrivere  
`!python`



# Script in Python

---

L'interprete interattivo è in grado di leggere e valutare man mano le espressioni inserite dall'utente, ma è anche possibile eseguire script contenenti sequenze di istruzioni Python, digitando il comando:

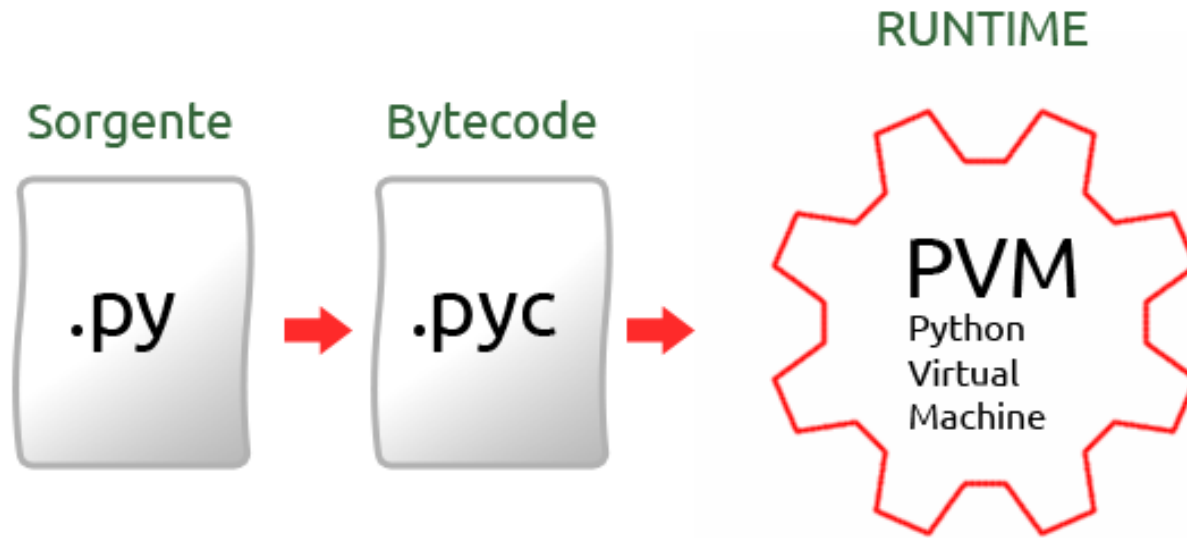
```
python script.py
```

Ogni volta che viene invocato il comando `python`, il codice scritto viene scansionato per token, ognuno dei quali viene analizzato dentro una struttura logica ad albero che rappresenta il programma.

Tale struttura viene, infine, trasformata in bytecode (file con estensione `.pyc` o `.pyo`). Per potere eseguire questi bytecode, si utilizza un apposito interprete noto come macchina virtuale Python (PVM).



# Bytecode



## Esempio

sorgente (script)

```
def hello()  
    print("Hello, World!")
```

bytecode

2	0 LOAD_GLOBAL	0 (print)
	2 LOAD_CONST	1 ('Hello, World!')
	4 CALL_FUNCTION	1

# Variabili e istruzione di assegnamento: sintassi

---

## Sintassi:

`variabile = espressione`

- non devono esserci spazi prima del nome della variabile
- `variabile` deve essere un nome simbolico scelto dal programmatore (con le limitazioni descritte più avanti)
- `espressione` indica il valore (per es., un numero) che si vuole associare alla variabile

## Esempi:

`x = -3.2`

`messaggio = "Buongiorno"`

`y = x + 1`

# Limitazioni sui nomi delle variabili

---

- possono essere composti da uno o più dei seguenti caratteri:
  - lettere minuscole e maiuscole (NB: «A» e «a» sono considerate lettere diverse)
  - cifre
  - il carattere `_` (underscore)

esempi: `x`, `somma`, `Massimo_Comun_Divisore`, `y_1`

- non devono iniziare con una cifra  
esempio: `12abc` non è un nome ammesso
- non devono contenere simboli matematici (`+`, `-`, `/`, `*`, parentheses)
- non devono coincidere con le keywords del linguaggio

<code>and</code>	<code>as</code>	<code>assert</code>	<code>break</code>	<code>class</code>
<code>continue</code>	<code>def</code>	<code>del</code>	<code>elif</code>	<code>else</code>
<code>except</code>	<code>exec</code>	<code>if</code>	<code>import</code>	<code>in</code>
<code>is</code>	<code>lambda</code>	<code>not</code>	<code>or</code>	<code>pass</code>
<code>print</code>	<code>raise</code>	<code>return</code>	<code>try</code>	<code>while</code>
<code>yield</code>	<code>True</code>	<code>False</code>	<code>None</code>	

Un nome di variabile può avere fino ad un massimo di 256 caratteri totali

# Tipi di dato ed espressioni

---

I **tipi di dato** *base* che possono essere rappresentati ed elaborati dai programmi Python, sono:

- numeri interi
- numeri frazionari (*floating-point*)
- stringhe di caratteri
- booleani

Le espressioni Python che producono valori appartenenti a tali tipi di dati, e che possono contenere opportuni **operatori**, sono le seguenti:

- espressioni aritmetiche
- espressioni stringa

# Espressioni aritmetiche

---

La più semplice espressione aritmetica è un singolo numero.

I numeri vengono rappresentati nelle istruzioni Python con diverse notazioni, simili a quelle matematiche, espressi in base dieci.

Python distingue tra due tipi di dati numerici:

- numeri interi, codificati nei calcolatori in complemento a due; esempi: 12, -9
- numeri frazionari ( floating point ), codificati in virgola mobile, e rappresentati nei programmi:
  - come parte intera e frazionaria, separate da un punto (notazione anglosassone)  
esempi: 3.14, -45.2, 1.0
  - in notazione esponenziale,  $m \times b^e$ , con base  $b$  pari a dieci ed esponente introdotto dal carattere E oppure e  
esempi: 1.99E33 ( $1,99 \times 10^{33}$ ), -42.3e-4 ( $-42,3 \times 10^{-4}$ ), 2E3 ( $2 \times 10^3$ )

**NOTA: i numeri espressi in notazione esponenziale sono sempre considerati numeri frazionari**

# Operatori aritmetici

---

Espressioni aritmetiche più complesse si ottengono combinando numeri attraverso operatori (addizione, divisione, ecc.), e usando le parentesi tonde per definire la precedenza tra gli operatori.

Gli operatori disponibili nel linguaggio Python sono i seguenti:

<b>simbolo</b>	<b>operatore</b>
+	somma
-	sottrazione
*	moltiplicazione
/	divisione
//	divisione (quoziente intero)
%	modulo (resto di una divisione)
**	elevamento a potenza

# Espressioni aritmetiche: esempi

---

Alcuni esempi di istruzioni di assegnazione contenenti espressioni aritmetiche.  
Il valore per ogni espressione è indicato in grassetto sulla destra.

<code>x = -5</code>	<b>-5</b>
<code>y = 1 + 1</code>	<b>2</b>
<code>z = (1 + 2) * 3</code>	<b>9</b>
<code>circonferenza = 2 * 3.14 * 3</code>	<b>18.84</b>
<code>q1 = 6 / 2</code>	<b>3</b>
<code>q2 = 7.5 / 3</code>	<b>2.5</b>
<code>q3 = 5 // 2</code>	<b>2</b>
<code>resto = 10 % 2</code>	<b>0</b>



# Rappresentazione del risultato

---

Se entrambi gli operandi di  $+$ ,  $-$  e  $*$  sono interi, il risultato è rappresentato come intero (senza parte frazionaria), altrimenti è rappresentato come numero frazionario.

Esempi:  $1 + 1 \rightarrow 2$ ,  $2 - 3.1 \rightarrow -1.1$ ,  $3.2 * 5 \rightarrow 16.0$


Anche se entrambi gli operandi di  $/$  sono interi, il risultato è invece sempre frazionario, come del resto accade se uno o entrambi gli operandi non sono interi.

Esempi:  $6 / 2 \rightarrow 3.0$ ,  $6.0 / 2 \rightarrow 3.0$ ,  $2 / 5 \rightarrow 0.4$ ,  
 $2 / 5.0 \rightarrow 0.4$ ,  $-2 / 3 \rightarrow -0.6666666666666666$

L'operatore  $//$  produce sempre il più grande intero non maggiore del quoziente, rappresentato come intero se entrambi gli operandi sono interi, altrimenti come numero frazionario.

Esempi:  $6 // 2 \rightarrow 3$ ,  $6.0 // 2 \rightarrow 3.0$ ,  $2 // 5 \rightarrow 0$ ,  
 $2 // 5.0 \rightarrow 0.0$ ,  $-2 // 3 \rightarrow -1$

# Prova su Colab

 **espressioni-aritmetiche.ipynb** ☆

File Edit View Insert Runtime Tools Help


COMMENT SHARE

+ CODE + TEXT ↑ CELL ↓ CELL

✓ RAM Disk EDITING


>

▼ Test del risultato per le espressioni aritmetiche



```
a = 5
b = 3
c = 4.5

print('a =', a)
print('b =', b)
print('a - b =', a - b)
print('a - c =', a - c)
print('a / b =', a / b)
print('a // b =', a // b)
print('a // c =', a // c)
```



```
a = 5
b = 3
a - b = 2
a - c = 0.5
a / b = 1.6666666666666667
a // b = 1
a // c = 1.0
```

# Espressioni: stringhe

---

I programmi Python possono elaborare testi rappresentati come sequenze di caratteri (lettere, numeri, segni di punteggiatura) racchiuse tra apici singoli o doppi, dette stringhe.

Esempi:

```
"Esempio di stringa"  
'Esempio di stringa'  
"A"  
'qwerty, 123456'  
"" (una stringa vuota)
```

È possibile assegnare una stringa a una variabile

Esempi:

```
testo = "Esempio di stringa"  
carattere = "a"  
messaggio = "Premere un tasto per continuare."  
t = ""
```

# Concatenazione di stringhe

---

Il linguaggio Python prevede alcuni operatori anche per il tipo di dato stringa. Uno di questi è l'operatore di concatenazione, che si rappresenta con il simbolo `+` (lo stesso dell'addizione tra numeri) e produce come risultato una nuova stringa ottenuta concatenando due altre stringhe.

Esempi:

```
parola = "mappa" + "mondo"
```

assegna alla variabile `parola` la stringa `"mappamondo"`

```
testo = "Due" + " " + "parole"
```

assegna alla variabile `testo` la stringa `"Due parole"` (si noti che la stringa `" "` contiene solo un carattere di spaziatura)

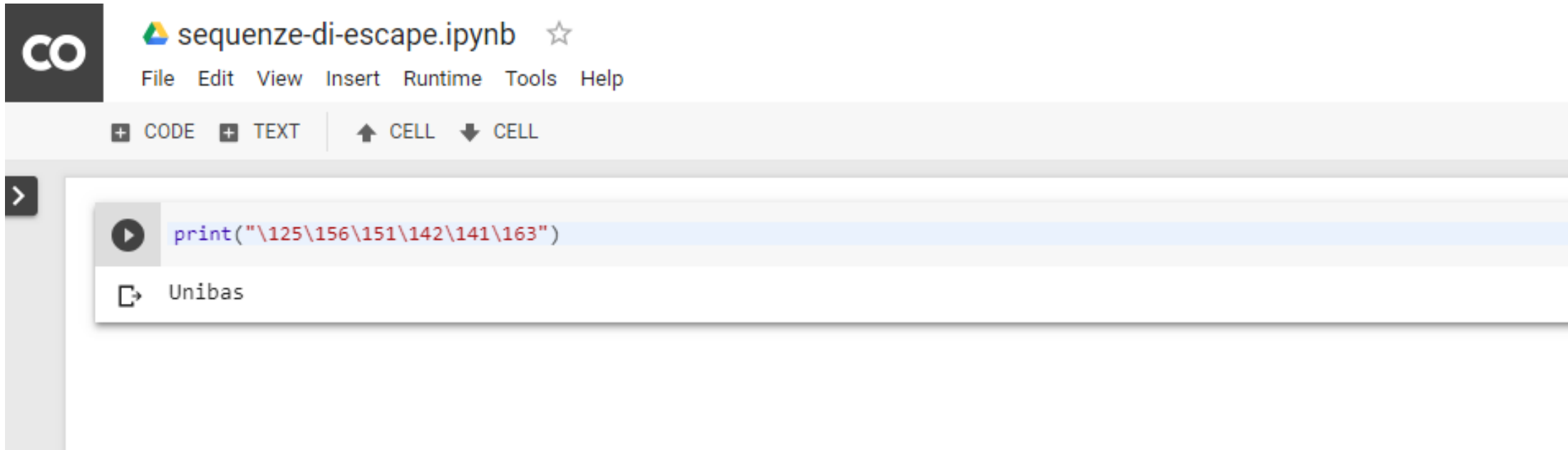
# Sequenze di escape

---

Sequenza di escape	Significato
\\	Backslash (\)
\'	Carattere di quotatura singola (')
\"	Carattere di quotatura doppia (")
\a	ASCII Bell (BEL)
\b	ASCII Backspace (BS)
\f	ASCII Formfeed (FF)
\n	ASCII Linefeed (LF)
\N{name}	Carattere chiamato name nel database Unicode (solamente in Unicode)
\r	ASCII Carriage Return (CR)
\t	ASCII Tab orizzontale (TAB)
\uxxxx	Carattere con valore esadecimale a 16 bit xxxx (valido solo per Unicode)
\Uxxxxxxxx	Carattere con valore esadecimale a 32 bit xxxxxxxx (valido solo per Unicode)
\v	ASCII Tab verticale (VT)
\ooo	Carattere ASCII con valore ottale ooo
\xhh	Carattere ASCII con valore esadecimale hh

# Sequenze di escape: esempio Colab

---



# Espressioni contenenti nomi di variabili

---

Una espressione contenente il nome di una variabile alla quale non sia stato ancora assegnato alcun valore è [sintatticamente errata](#).

Per esempio, assumendo che alla variabile `h` non sia ancora stato assegnato alcun valore, l'istruzione

```
x = h + 1
```

genererà il seguente messaggio di errore:

```
NameError: name 'h' is not defined
```

# La funzione `input()`

---

La funzione built-in `input` può essere utilizzata in due forme.

```
valore = input()
```

l'interprete resta in attesa che l'utente inserisca nella shell, attraverso la tastiera, una sequenza di caratteri che dovrà essere conclusa dal tasto <INVIO>. Questa sequenza di caratteri è poi inserita in una stringa che viene restituita come `valore`.

```
dato = input(messaggio)
```

l'interprete stampa `messaggio`, che deve essere una stringa, nella shell, poi procede come indicato sopra.

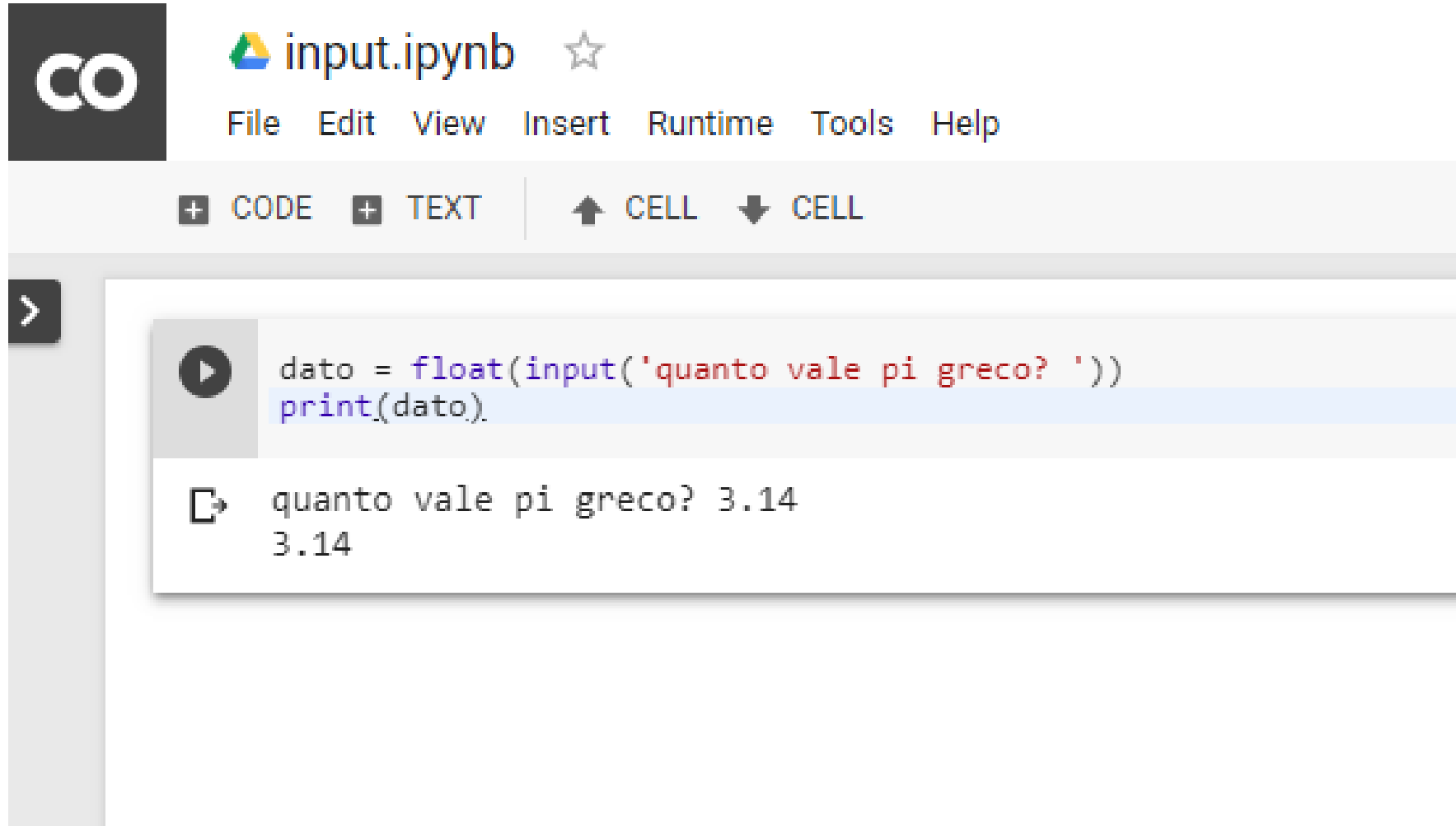
La stringa `messaggio` viene di norma usata per indicare all'utente che il programma è in attesa di ricevere un particolare dato in ingresso.

Nel caso di dati numerici, occorrerà convertire con `int` o `float` in intero o frazionario il valore `str` restituito da `input`

```
dato = float(input('quanto vale pi greco? '))
```



# La funzione `input()` : Esempio Colab



# La funzione `print`

---

Sintassi:

```
print(espressione)
```

- non devono esserci spazi prima di `print`
- `espressione` deve essere una espressione valida del linguaggio Python

Semantica:

viene mostrato sullo standard output il valore di `espressione`

È anche possibile mostrare con una sola istruzione `print` i valori di un numero qualsiasi di espressioni, con la seguente sintassi:

```
print(espressione1, espressione2, ...)
```

In questo caso, i valori delle espressioni vengono mostrati su una stessa riga, separati da un carattere di spaziatura.

# Commenti

---

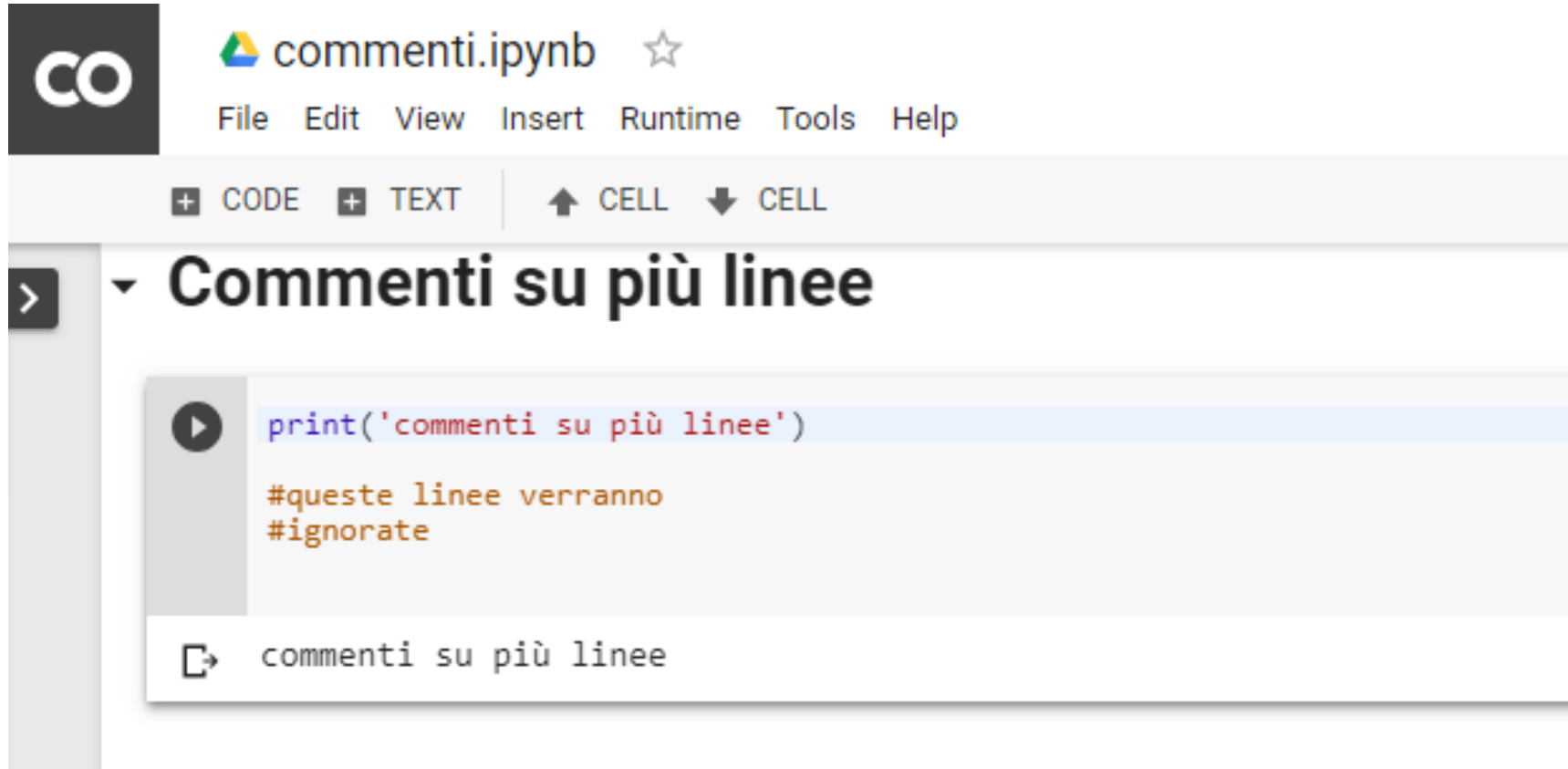
È sempre utile documentare i programmi inserendo commenti che indichino quale operazione viene svolta dal programma, quali sono i dati di ingresso e i risultati, qual è il significato delle variabili o di alcune sequenze di istruzioni.

Nei programmi Python i commenti possono essere inseriti in qualsiasi riga, preceduti dal carattere # (“cancellino”).

Tutti i caratteri che seguono il cancellino, fino al termine della riga, saranno considerati commenti e verranno trascurati dall’interprete.

# Commenti su più linee

---

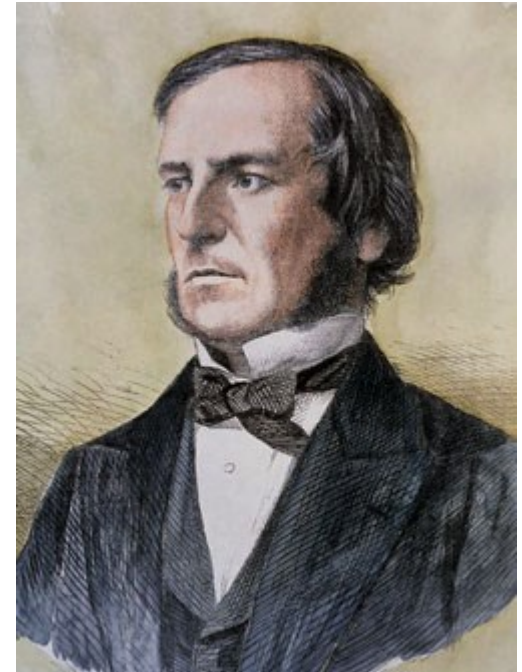


# Algebra di Boole

---

Il matematico britannico George Boole (1815-1864) trovò il modo di legare argomenti logici ad un linguaggio che potesse essere manipolato matematicamente.

Il sistema di Boole era basato su un approccio binario, in grado di differenziare gli argomenti in base alla presenza o all'assenza di una determinata proprietà.



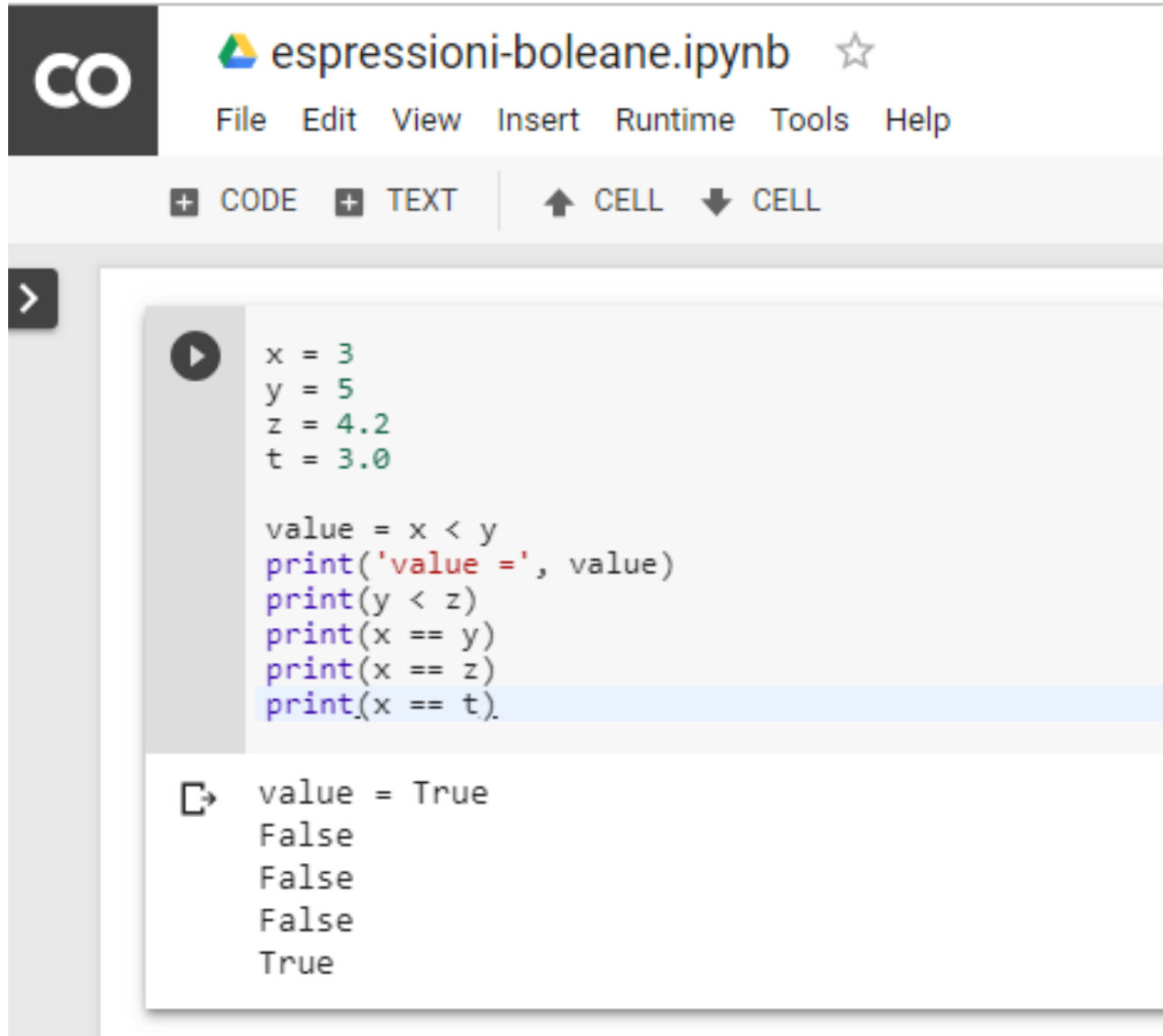
# Espressioni booleane

---

I valori booleani possono essere prodotti da:

- Le costanti `True` o `False`.
- Una variabile cui sia stato assegnato un valore booleano.
- Una relazione fra due espressioni per mezzo degli operatori relazionali come `==`, `!=`, `<`, `>`, `<=`, `>=`

# Espressioni booleane: esempio Colab



The screenshot shows a Google Colab notebook interface. At the top, the Colab logo is on the left, and the notebook title 'espressioni-boleane.ipynb' is in the center, followed by a star icon. Below the title is a menu bar with 'File', 'Edit', 'View', 'Insert', 'Runtime', 'Tools', and 'Help'. Underneath the menu bar is a toolbar with '+ CODE', '+ TEXT', '↑ CELL', and '↓ CELL'. The main area contains a code cell with a play button icon on the left. The code in the cell is as follows:

```
x = 3
y = 5
z = 4.2
t = 3.0

value = x < y
print('value =', value)
print(y < z)
print(x == y)
print(x == z)
print(x == t).
```

Below the code cell, the output is displayed, preceded by a copy icon:

```
value = True
False
False
False
True
```

# Operatore booleano and

---

Se  $x$  e  $y$  sono booleani, possiamo completamente determinare i valori possibili di  $x$  and  $y$  usando la seguente «tavola di verità»:

$x$	$y$	$x$ and $y$
False	False	False
False	True	False
True	False	False
True	True	True

$x$  and  $y$  è False a meno che le variabili  $x$  e  $y$  non siano entrambe True



# Operatore booleano `or`

---

Se `x` e `y` sono booleani, possiamo completamente determinare i valori possibili di `x or y` usando la seguente «tavola di verità»:

<code>x</code>	<code>y</code>	<code>x or y</code>
False	False	False
False	True	True
True	False	True
True	True	True

**`x or y` è `True` a meno che le variabili `x` e `y` non siano entrambe `False`**

# Operatore booleano `not`

---

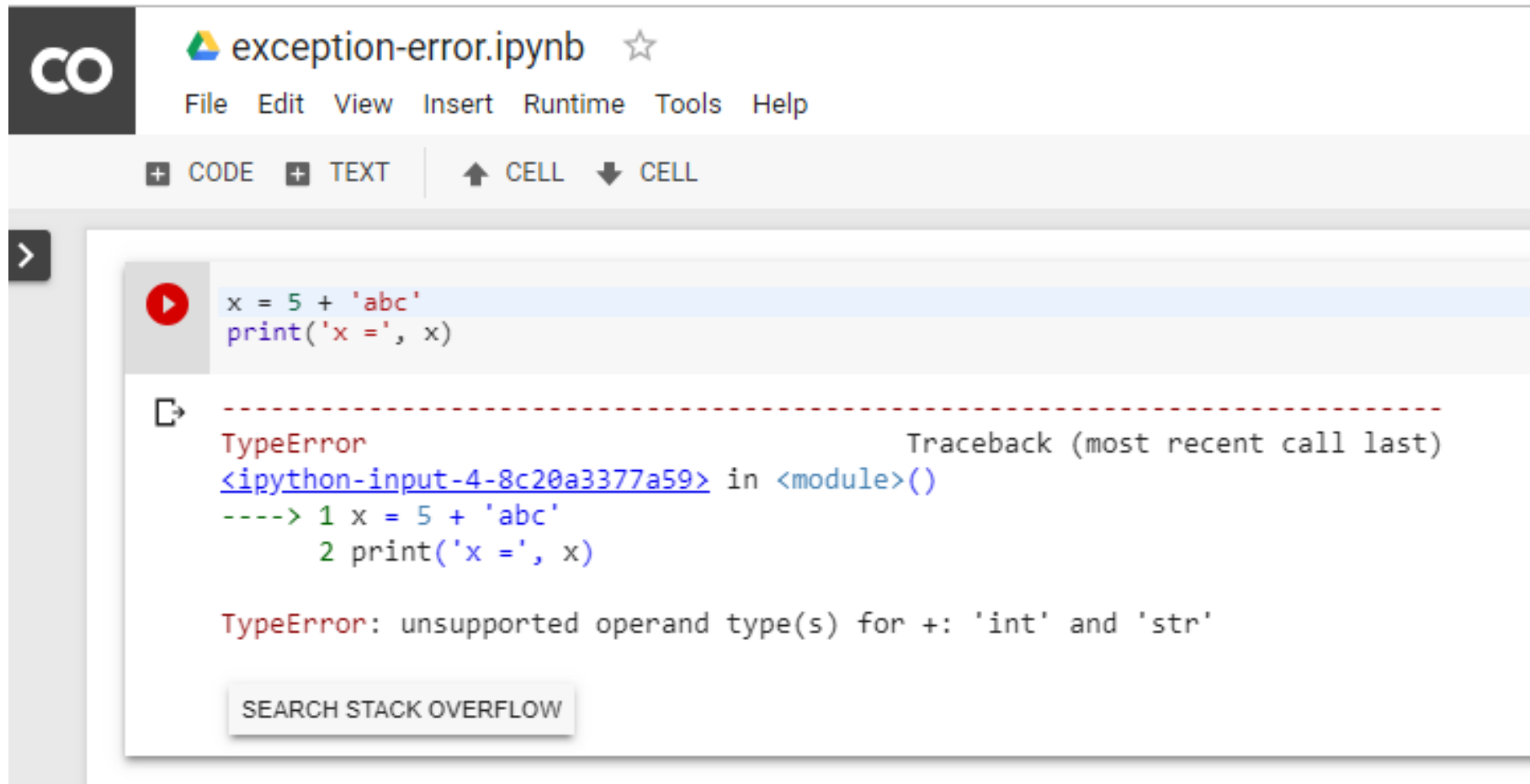
Se `x` è un booleano, possiamo completamente determinare i valori possibili di `not x` usando la seguente «tavola di verità»:

<code>x</code>	<code>not x</code>
False	True
True	False

**`not x` è True se `x` è False ed è False se `x` è True**

# Errori a runtime

Un *exception error* si verifica se si utilizza un operatore con tipi di dato non compatibili



The screenshot shows a Jupyter Notebook interface with a dark sidebar on the left containing a 'CO' logo and a right-pointing arrow. The main area has a header bar with the file name 'exception-error.ipynb' and a star icon, followed by a menu bar with 'File', 'Edit', 'View', 'Insert', 'Runtime', 'Tools', and 'Help'. Below the menu is a toolbar with buttons for '+ CODE', '+ TEXT', '↑ CELL', and '↓ CELL'. The notebook content area shows a code cell with a red play button icon. The code in the cell is:

```
x = 5 + 'abc'
print('x =', x)
```

Below the code cell, the output area displays a traceback for a `TypeError`. The traceback text is:

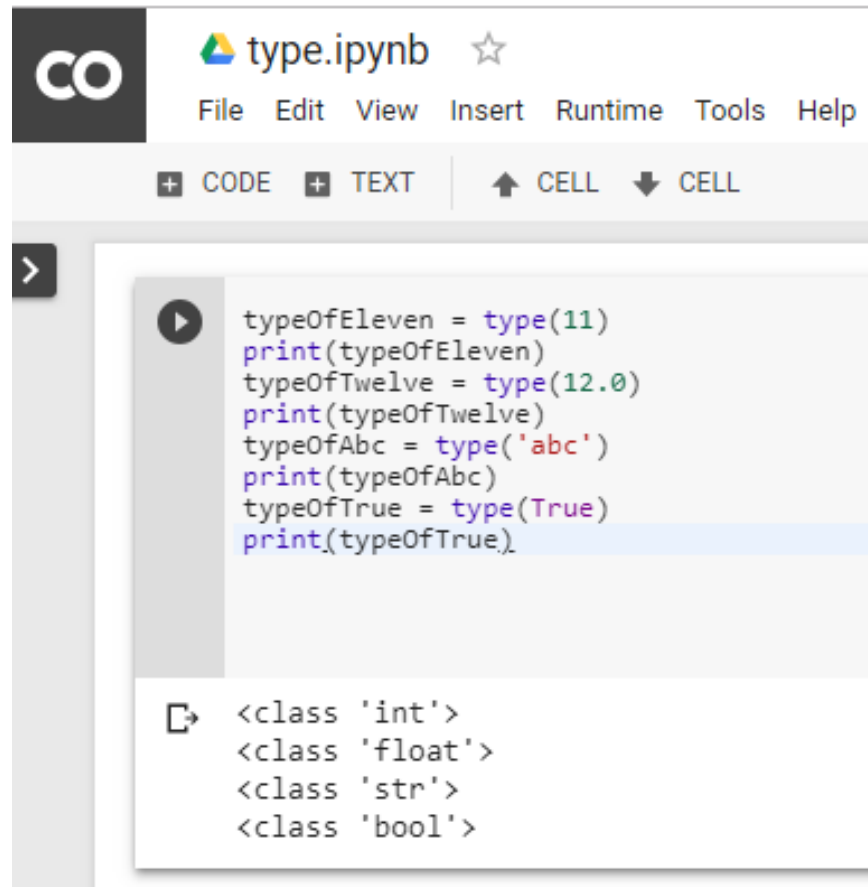
```
-----
TypeError                                Traceback (most recent call last)
<ipython-input-4-8c20a3377a59> in <module>()
----> 1 x = 5 + 'abc'
      2 print('x =', x)

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

At the bottom of the output area, there is a button labeled 'SEARCH STACK OVERFLOW'.

# La funzione `type`

Python ha una funzione built-in denominata `type` che può essere usata per ottenere il tipo di ogni variabile o valore costante.



The screenshot shows a Jupyter Notebook interface with a dark sidebar on the left and a main content area. The top bar includes the 'CO' logo, the file name 'type.ipynb', and a star icon. Below this is a menu bar with 'File', 'Edit', 'View', 'Insert', 'Runtime', 'Tools', and 'Help'. A secondary bar contains '+ CODE', '+ TEXT', and cell navigation icons. The main area displays a code cell with a play button icon on the left. The code in the cell is as follows:

```
typeOfEleven = type(11)
print(typeOfEleven)
typeOfTwelve = type(12.0)
print(typeOfTwelve)
typeOfAbc = type('abc')
print(typeOfAbc)
typeOfTrue = type(True)
print(typeOfTrue)
```

Below the code cell, the output is displayed in a separate box with a copy icon on the left. The output shows the results of the `print` statements:

```
<class 'int'>
<class 'float'>
<class 'str'>
<class 'bool'>
```

# Espressioni condizionali

---

Sintassi:

`espressione1 operatore espressione2`

- `espressione1` e `espressione2` sono due espressioni Python qualsiasi, definite come si è visto in precedenza (possono quindi contenere nomi di variabili, purché a tali variabili sia stato già assegnato un valore)
- `operatore` è un simbolo che indica il tipo di confronto tra le due espressioni

# Operatori di confronto

---

Nel linguaggio Python sono disponibili i seguenti operatori di confronto, che possono essere usati sia su espressioni aritmetiche che su espressioni composte da stringhe:

<b>simbolo</b>	<b>significato</b>
<code>==</code>	“uguale a”
<code>!=</code>	“diverso da”
<code>&lt;</code>	“minore di”
<code>&lt;=</code>	“minore o uguale a”
<code>&gt;</code>	“maggiore di”
<code>&gt;=</code>	“maggiore o uguale a”

# Espressioni condizionali composte

---

Sintassi:

- `espr_cond_1 and espr_cond_2`
- `espr_cond_1 or espr_cond_2`
- `not espr_cond`

`espr_cond_1`, `espr_cond_2` e `espr_cond` sono espressioni condizionali qualsiasi (quindi, possono essere a loro volta espressioni composte)

Gli operatori `and`, `or` e `not` sono detti operatori logici, e corrispondono rispettivamente ai connettivi logici del linguaggio naturale “e”, “oppure”, “non”

É possibile usare le parentesi tonde per definire l'ordine degli operatori logici

# L'istruzione condizionale

---

Sintassi:

```
if espr_cond:
    sequenza_di_istruzioni_1
else:
    sequenza_di_istruzioni_2
```

- le keyword `if` e `else` devono avere lo stesso rientro
- `espr_cond` è un'espressione condizionale
- `sequenza_di_istruzioni_1` e `sequenza_di_istruzioni_2` sono due sequenze di una o più istruzioni qualsiasi
- ciascuna istruzione deve essere scritta in una riga distinta, con un rientro di almeno un carattere; il rientro deve essere identico per tutte le istruzioni



# L'istruzione condizionale: varianti

---

Variante if:

```
if espr_cond:  
    sequenza_di_istruzioni
```

Variante elif:

```
if espr_cond_1:  
    sequenza_di_istruzioni_1  
elif espr_cond_2:  
    sequenza_di_istruzioni_2  
else:  
    sequenza_di_istruzioni_3
```

# Attenzione ai rientri!

---

I rientri sono l'unico elemento sintattico che indica quali istruzioni fanno parte di un'istruzione condizionale. Le istruzioni che seguono un'istruzione condizionale (senza farne parte) devono quindi essere scritte senza rientri.

Come esempio, si considerino le due sequenze di istruzioni:

```
if x > 0:
    print("A")
    print("B")
```

```
if x > 0:
    print("A")
print("B")
```

Nella sequenza a sinistra le due istruzioni `print` sono scritte con un rientro rispetto a `if`: questo significa che fanno entrambe parte dell'istruzione condizionale e quindi verranno eseguite solo se la condizione `x > 0` sarà vera.

Nella sequenza a destra solo la prima istruzione `print` dopo `if` è scritta con un rientro e quindi solo essa fa parte dell'istruzione condizionale; la seconda istruzione `print` verrà invece eseguita dopo l'istruzione condizionale, indipendentemente dal valore di verità della condizione `x > 0`

# Istruzioni condizionali annidate

---

Una istruzione condizionale può contenere al suo interno istruzioni qualsiasi, quindi anche altre istruzioni condizionali. Si parla, in questo caso, di **istruzioni annidate**.

L'uso di istruzioni condizionali annidate consente di esprimere la scelta tra più di due sequenze di istruzioni alternative.

Una istruzione condizionale annidata all'interno di un'altra andrà scritta tenendo conto che:

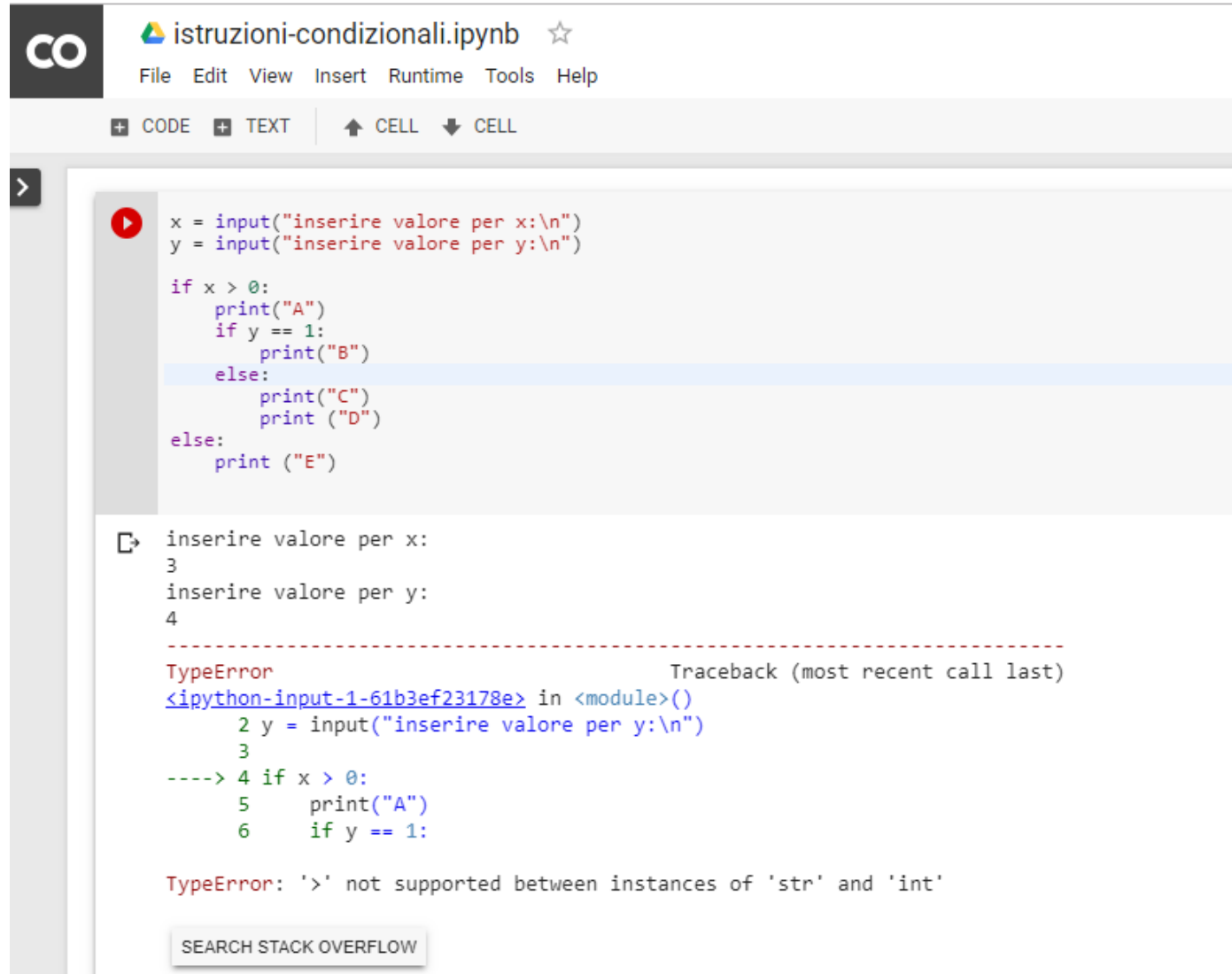
- le keyword `if`, `elif` e `else` (se presenti) devono essere scritte con un rientro rispetto a quelle dell'istruzione condizionale che le contiene
- le sequenze di istruzioni che seguono `if`, `elif` e `else` devono essere scritte con un ulteriore rientro

# Istruzioni condizionali annidate: esempio

---

```
if x > 0:
    print("A")
    if y == 1:
        print("B")
    else:
        print("C")
        print("D")
else:
    print("E")
```

# Esempio in Colab



The screenshot shows a Google Colab notebook interface. At the top, the file name is "istruzioni-condizionali.ipynb". Below the menu bar (File, Edit, View, Insert, Runtime, Tools, Help), there are tabs for CODE, TEXT, and CELL. The CODE tab is active. A code cell contains the following Python code:

```
x = input("inserire valore per x:\n")
y = input("inserire valore per y:\n")

if x > 0:
    print("A")
    if y == 1:
        print("B")
    else:
        print("C")
        print("D")
else:
    print("E")
```

Below the code cell, the input and output area shows the user's input and the resulting error:

```
inserire valore per x:
3
inserire valore per y:
4
```

-----

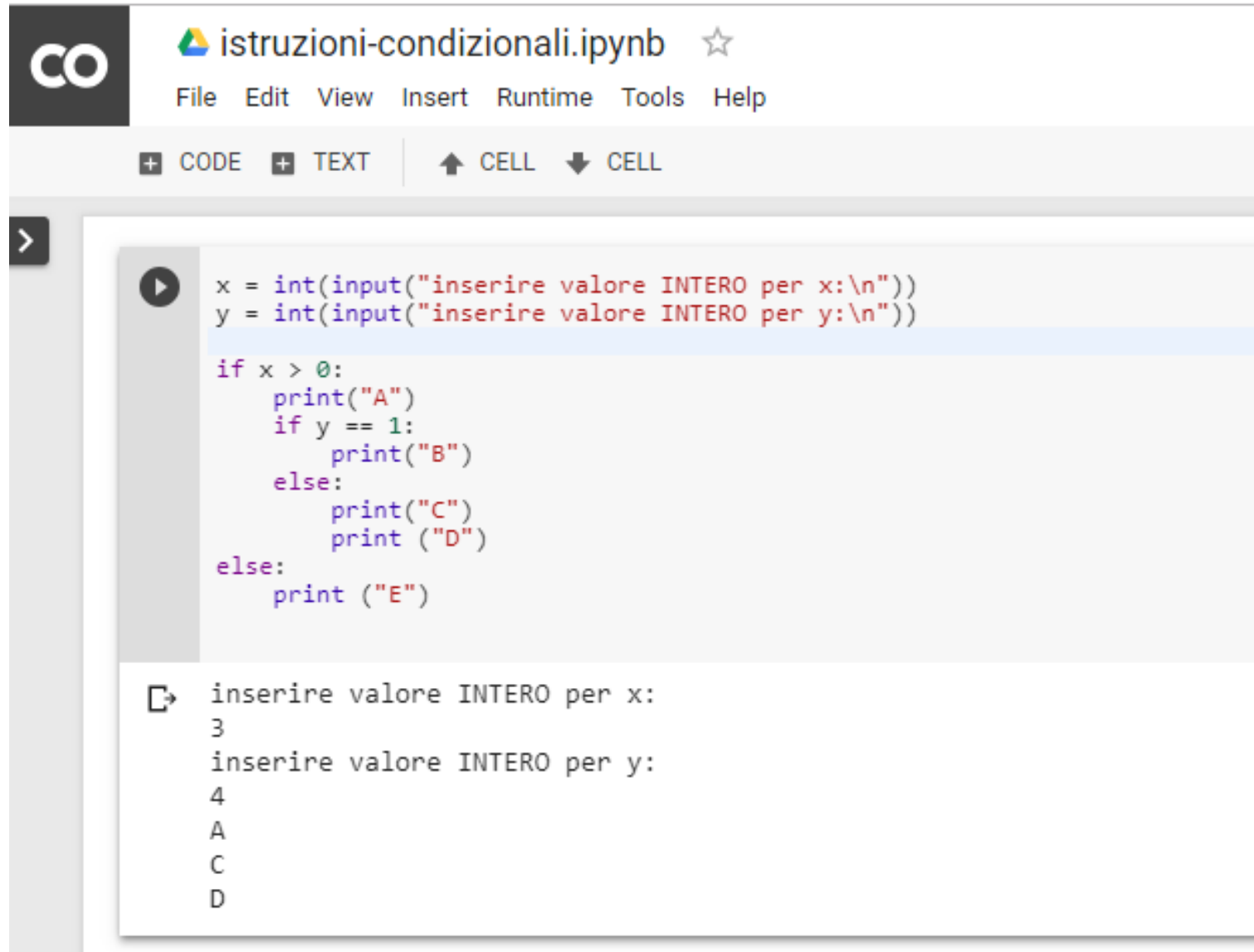
**TypeError** Traceback (most recent call last)

```
<ipython-input-1-61b3ef23178e> in <module>()
      2 y = input("inserire valore per y:\n")
      3
----> 4 if x > 0:
      5     print("A")
      6     if y == 1:
```

**TypeError: '>' not supported between instances of 'str' and 'int'**

At the bottom of the error message, there is a button labeled "SEARCH STACK OVERFLOW".

# Cast dei dati



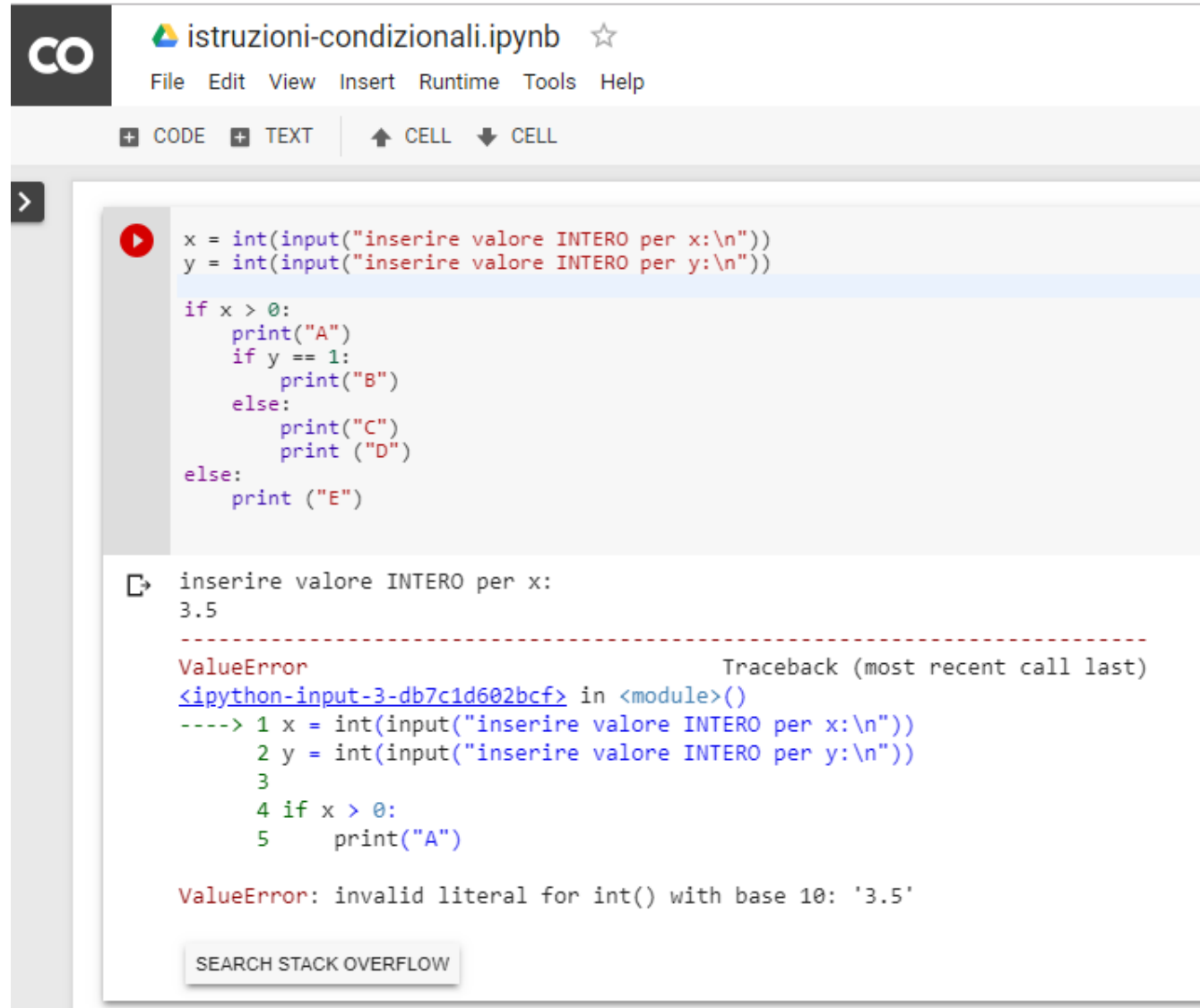
The screenshot shows a Jupyter Notebook interface. At the top, there's a header with a logo and the title "istruzioni-condizionali.ipynb". Below the header is a menu bar with options: File, Edit, View, Insert, Runtime, Tools, Help. Underneath the menu bar is a toolbar with buttons for adding code cells (+ CODE), text cells (+ TEXT), and navigating between cells (up and down arrows with CELL labels). The main area contains a single code cell with a play button icon on the left. The code in the cell is a Python script that takes two integer inputs, x and y, and prints a result based on conditional checks. The output of the code cell is displayed below it, showing the prompts and the user's input.

```
x = int(input("inserire valore INTERO per x:\n"))
y = int(input("inserire valore INTERO per y:\n"))

if x > 0:
    print("A")
    if y == 1:
        print("B")
    else:
        print("C")
        print("D")
else:
    print("E")
```

inserire valore INTERO per x:  
3  
inserire valore INTERO per y:  
4  
A  
C  
D

# Verifica del tipo di dato



The screenshot shows a Jupyter Notebook interface with a dark sidebar on the left containing a 'co' logo and a right-pointing arrow. The main area has a top bar with the file name 'istruzioni-condizionali.ipynb' and a star icon, followed by a menu (File, Edit, View, Insert, Runtime, Tools, Help) and a toolbar with buttons for adding code or text cells and moving between cells. Below this is a code cell with a red play button icon. The code in the cell is a Python script that takes two inputs, x and y, and prints different messages based on their values. The input for x is '3.5', which causes a 'ValueError' because it is not a valid integer literal. The output shows the input prompt, the value '3.5', a dashed line, the error message, and a traceback showing the execution steps up to the first print statement.

```
x = int(input("inserire valore INTERO per x:\n"))
y = int(input("inserire valore INTERO per y:\n"))

if x > 0:
    print("A")
    if y == 1:
        print("B")
    else:
        print("C")
        print("D")
else:
    print("E")
```

inserire valore INTERO per x:  
3.5

-----

ValueError Traceback (most recent call last)

<ipython-input-3-db7c1d602bcf> in <module>()

----> 1 x = int(input("inserire valore INTERO per x:\n"))

2 y = int(input("inserire valore INTERO per y:\n"))

3

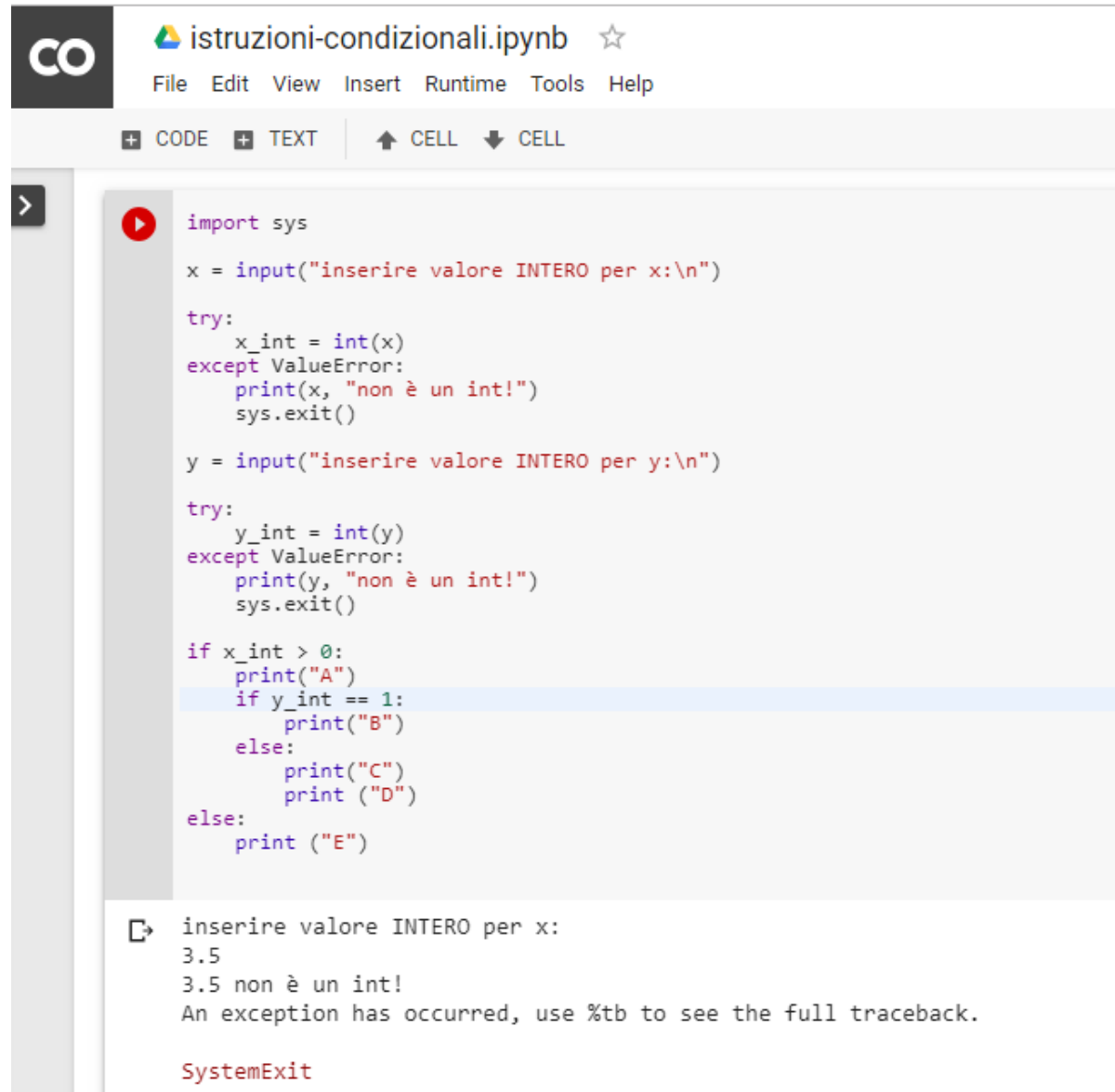
4 if x > 0:

5 print("A")

ValueError: invalid literal for int() with base 10: '3.5'

SEARCH STACK OVERFLOW

# Gestione delle eccezioni



The screenshot shows a Jupyter Notebook interface. At the top, there's a header with a logo and the text "istruzioni-condizionali.ipynb". Below this is a menu bar with "File", "Edit", "View", "Insert", "Runtime", "Tools", and "Help". A toolbar contains buttons for "CODE", "TEXT", "CELL", and "CELL". The main area displays a Python script with exception handling. The script prompts for two integers, x and y. If the input is not an integer, it prints an error message and exits. If both inputs are integers, it prints "A", "B", "C", "D", or "E" based on their values. The output shows the execution of the first part of the script, where the input "3.5" is not an integer, leading to a "SystemExit" exception.

```
import sys

x = input("inserire valore INTERO per x:\n")

try:
    x_int = int(x)
except ValueError:
    print(x, "non è un int!")
    sys.exit()

y = input("inserire valore INTERO per y:\n")

try:
    y_int = int(y)
except ValueError:
    print(y, "non è un int!")
    sys.exit()

if x_int > 0:
    print("A")
    if y_int == 1:
        print("B")
    else:
        print("C")
        print("D")
else:
    print("E")
```

inserire valore INTERO per x:  
3.5  
3.5 non è un int!  
An exception has occurred, use %tb to see the full traceback.  
SystemExit





**UNIVERSITÀ DEGLI STUDI  
DELLA BASILICATA**

*Corso di Sistemi Informativi  
A.A. 2018/19*

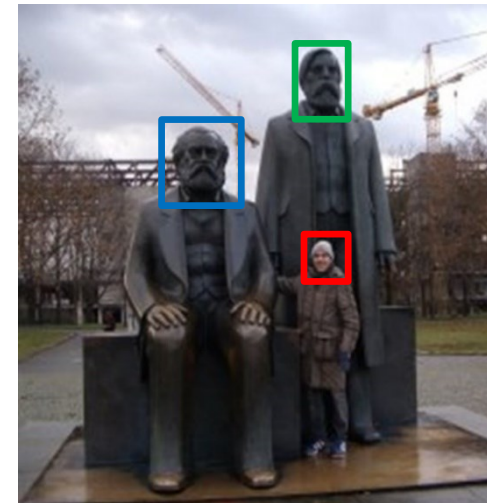
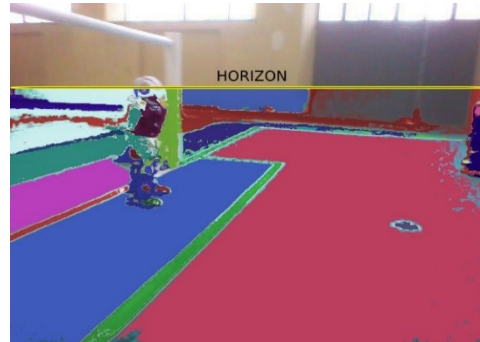
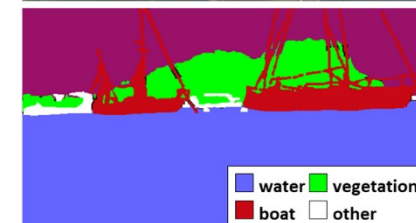
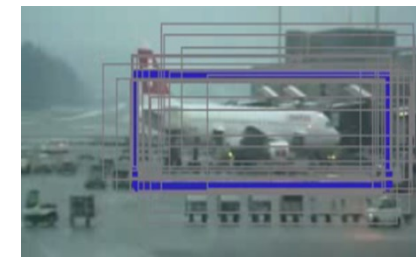
# Introduzione al Python

Docente

**Domenico Daniele Bloisi**



Parte del materiale deriva dai corsi dei proff. Paolo Caressa e Raffaele Nicolussi (Sapienza Università di Roma) e Giorgio Fumera (Università degli Studi di Cagliari)



Marzo 2019