



**UNIVERSITÀ DEGLI STUDI
DELLA BASILICATA**

*Corso di Visione e Percezione
A.A. 2019/2020*

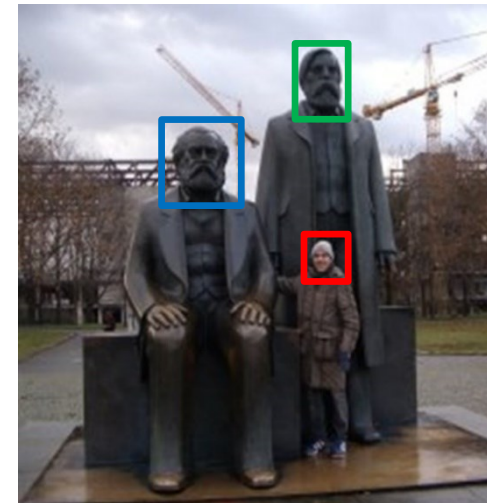
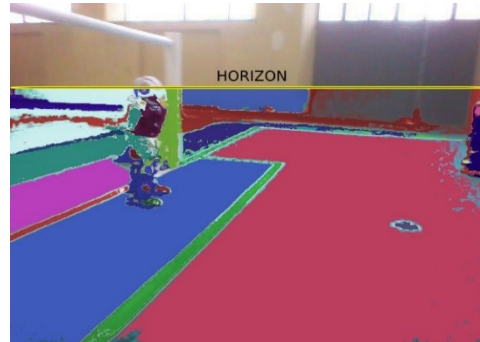
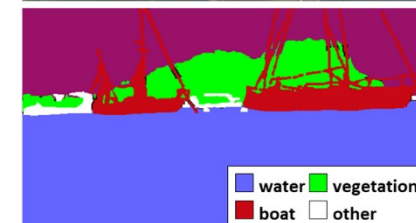
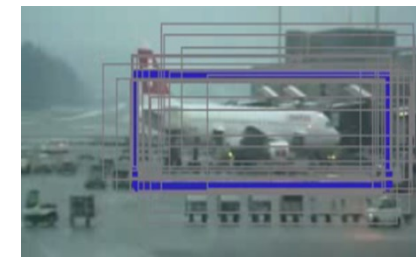
Funzioni in Python

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)



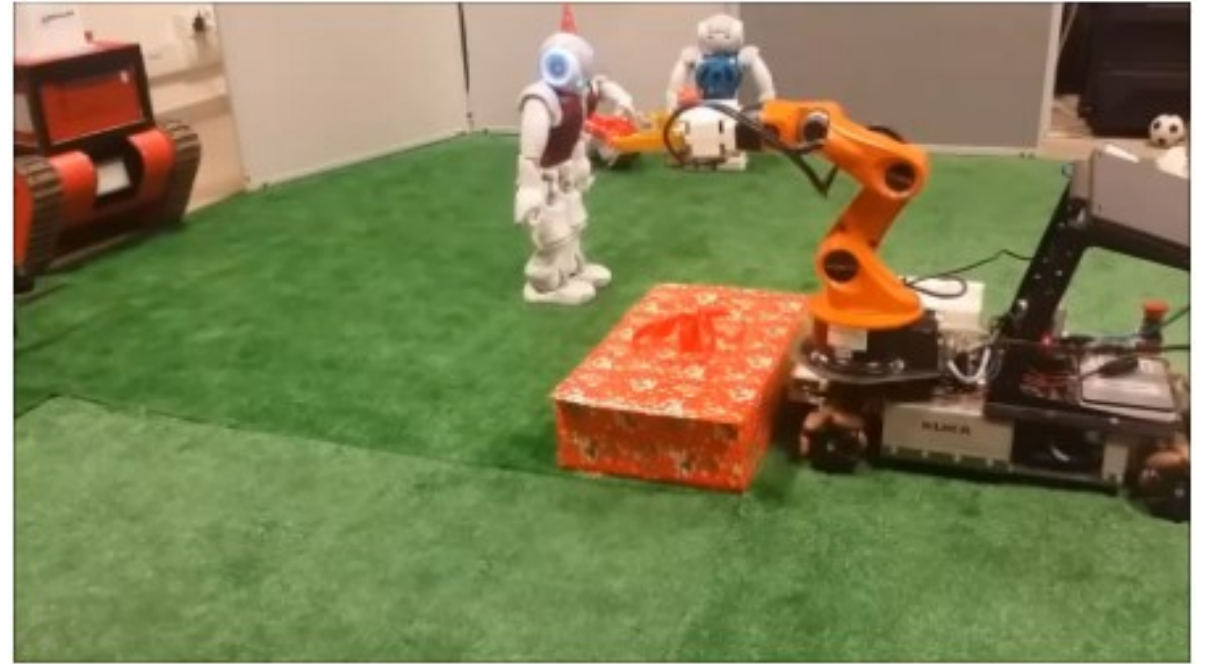
Marzo 2020

Il corso

- Home page del corso
<http://web.unibas.it/bloisi/corsi/visione-e-percezione.html>
- Docente: Domenico Daniele Bloisi
- Periodo: **Il semestre** marzo 2020 – giugno 2020
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=l9KYJILnEbw>

Funzioni built-in

Le funzioni disponibili in un linguaggio di programmazione sono dette [predefinite](#), o [built-in](#).

L'insieme di tali funzioni viene detto [libreria](#).

Esempi di librerie:

- funzioni matematiche (`math`)
- funzioni per la generazione di numeri pseudo-casuali (`random`)

La descrizione completa della libreria Python si trova nel documento [The Python Standard Library](#)

Chiamata di funzione

Sintassi:

```
nome_funzione(arg1, arg2, ..., argn)
```

- `arg1, arg2, ..., argn` sono espressioni Python i cui valori costituiranno gli argomenti della funzione
- il numero degli argomenti e il tipo di ciascuno di essi (per es., numeri interi, numeri frazionari, stringhe, valori logici) dipende dalla specifica funzione; se il tipo di un argomento non è tra quelli previsti, si otterrà un messaggio d'errore
- come tutte le espressioni, anche la chiamata di una funzione produce un valore: questo coincide con il valore restituito dalla funzione

Esempi di funzioni built-in

`len(stringa)`

restituisce il numero di caratteri di una stringa

`abs(numero)`

restituisce il valore assoluto di un numero

`str(espressione)`

restituisce una stringa composta dalla sequenza di caratteri

corrispondenti alla rappresentazione del valore di `espressione` (che può essere di un qualsiasi tipo: numero, stringa, valore logico, ecc.)

Esempi di funzioni built-in

`int(numero)`

restituisce la parte intera di un numero

`float(numero)`

restituisce il valore di numero come numero frazionario (floating point); può essere usata per evitare che la divisione tra interi produca la sola parte intera del quoziente, per es.: `float(2) / 3`

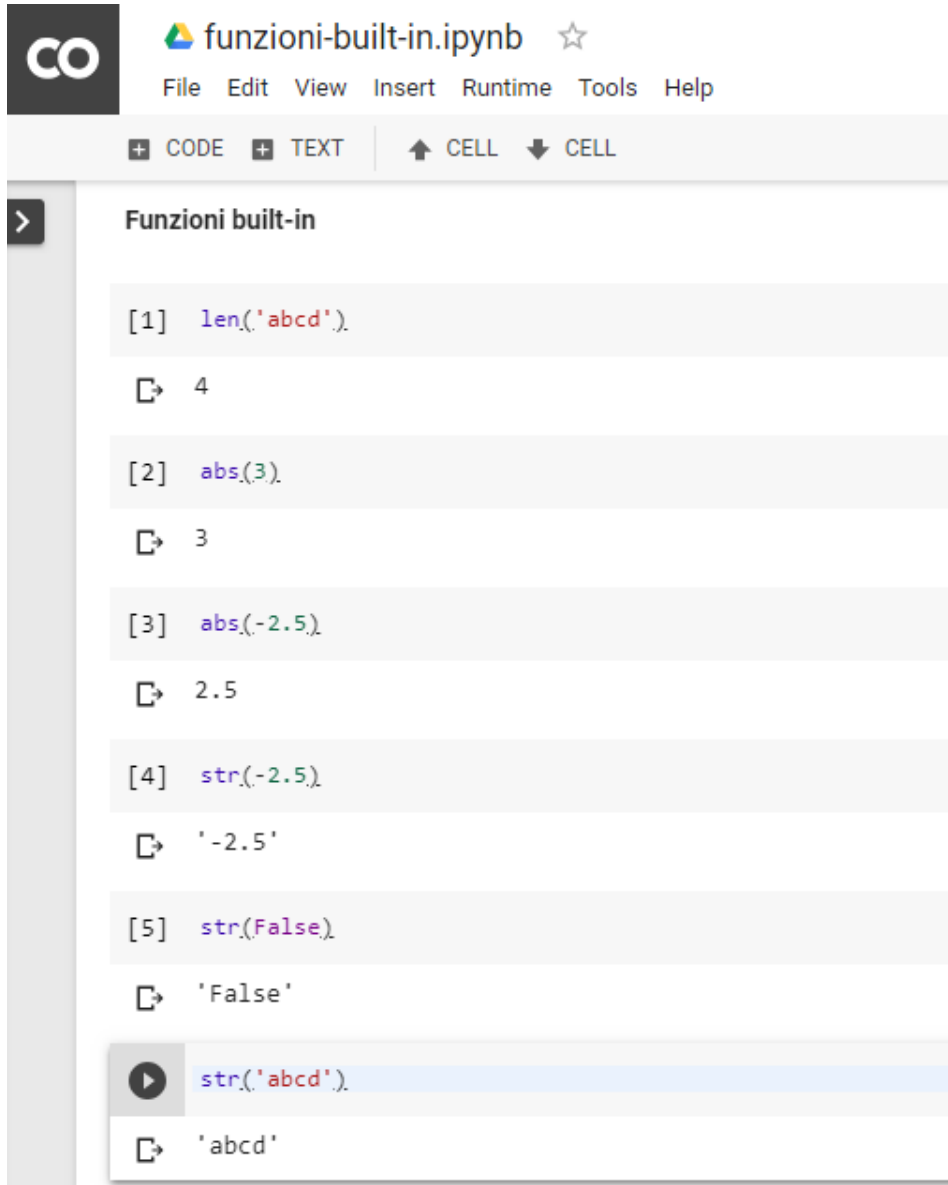
`int(stringa)`

Se `stringa` contiene la rappresentazione di un numero intero, restituisce il numero corrispondente a tale valore; in caso contrario produce un errore

`float(stringa)`

Se `stringa` contiene la rappresentazione di un numero qualsiasi (sia intero che frazionario), restituisce il suo valore espresso come numero frazionario; in caso contrario produce un errore

Esempi funzioni built-in



The screenshot shows a Jupyter Notebook titled "funzioni-built-in.ipynb". The interface includes a top bar with a "co" logo, a menu (File, Edit, View, Insert, Runtime, Tools, Help), and a toolbar with buttons for adding code or text cells, and navigating between cells. The notebook content is titled "Funzioni built-in" and displays five code cells, each with its input and output. The fifth cell is currently selected and highlighted in blue.

Funzioni built-in

[1] `len('abcd').`

↳ 4

[2] `abs(3).`

↳ 3

[3] `abs(-2.5).`

↳ 2.5

[4] `str(-2.5).`

↳ '-2.5'

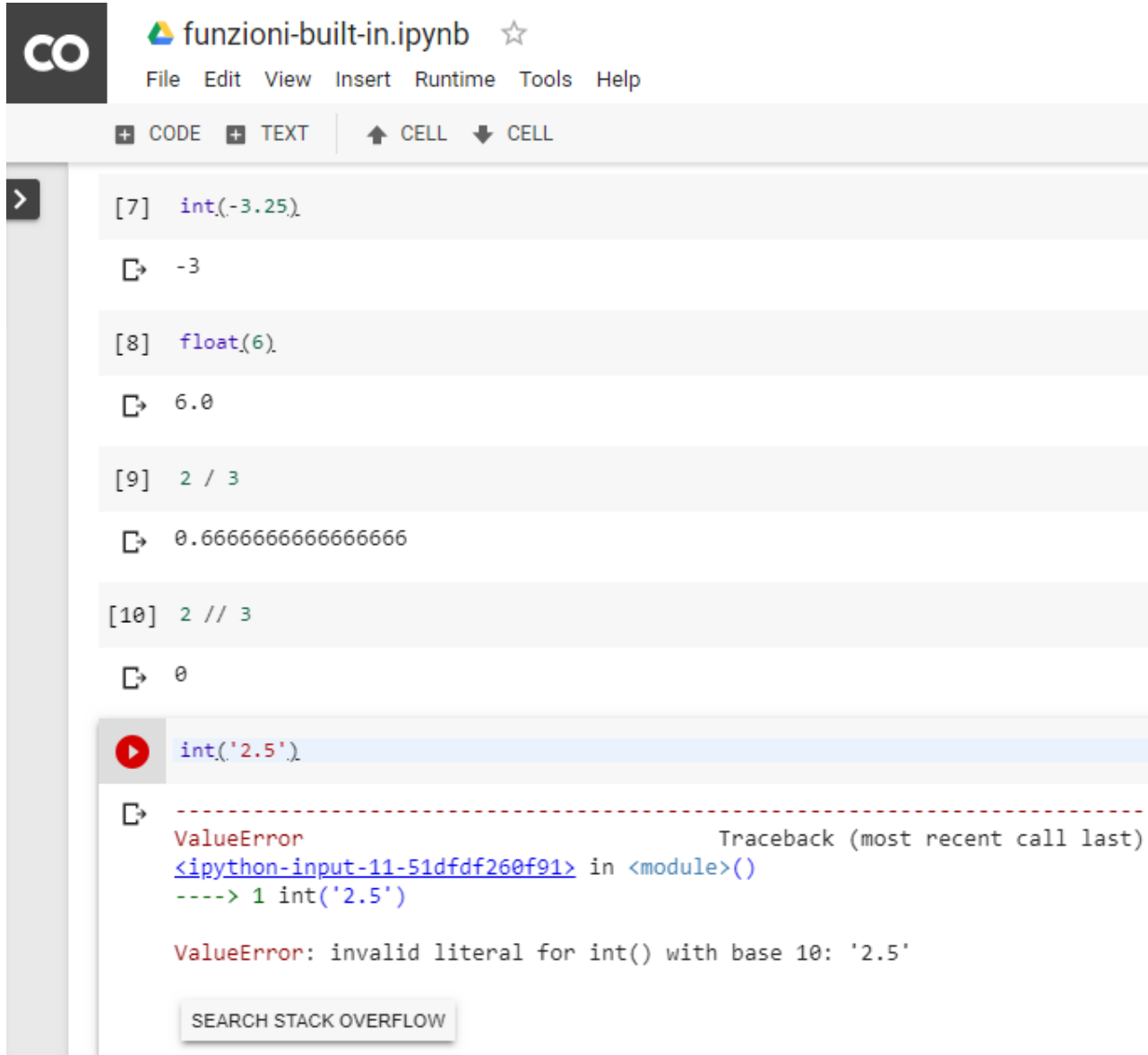
[5] `str(False).`

↳ 'False'

▶ `str('abcd').`

↳ 'abcd'

Esempi funzioni built-in



The screenshot shows a Jupyter Notebook titled "funzioni-built-in.ipynb". The interface includes a top bar with a "co" logo, a menu (File, Edit, View, Insert, Runtime, Tools, Help), and a toolbar with buttons for adding code or text cells, and moving between cells. The notebook contains five code cells. The first four cells show successful function calls: `int(-3.25)` returns `-3`, `float(6)` returns `6.0`, `2 / 3` returns `0.6666666666666666`, and `2 // 3` returns `0`. The fifth cell shows an error: `int('2.5')` raises a `ValueError`. The error message is displayed in a traceback format, indicating the error occurred in the current cell at line 1. A "SEARCH STACK OVERFLOW" button is located at the bottom of the error message.

co funzioni-built-in.ipynb ☆

File Edit View Insert Runtime Tools Help

+ CODE + TEXT ↑ CELL ↓ CELL

[7] `int(-3.25)`

↳ -3

[8] `float(6)`

↳ 6.0

[9] `2 / 3`

↳ 0.6666666666666666

[10] `2 // 3`

↳ 0

▶ `int('2.5')`

↳

```
-----  
ValueError                                Traceback (most recent call last)  
<ipython-input-11-51dfdf260f91> in <module>()  
----> 1 int('2.5')
```

ValueError: invalid literal for int() with base 10: '2.5'

SEARCH STACK OVERFLOW

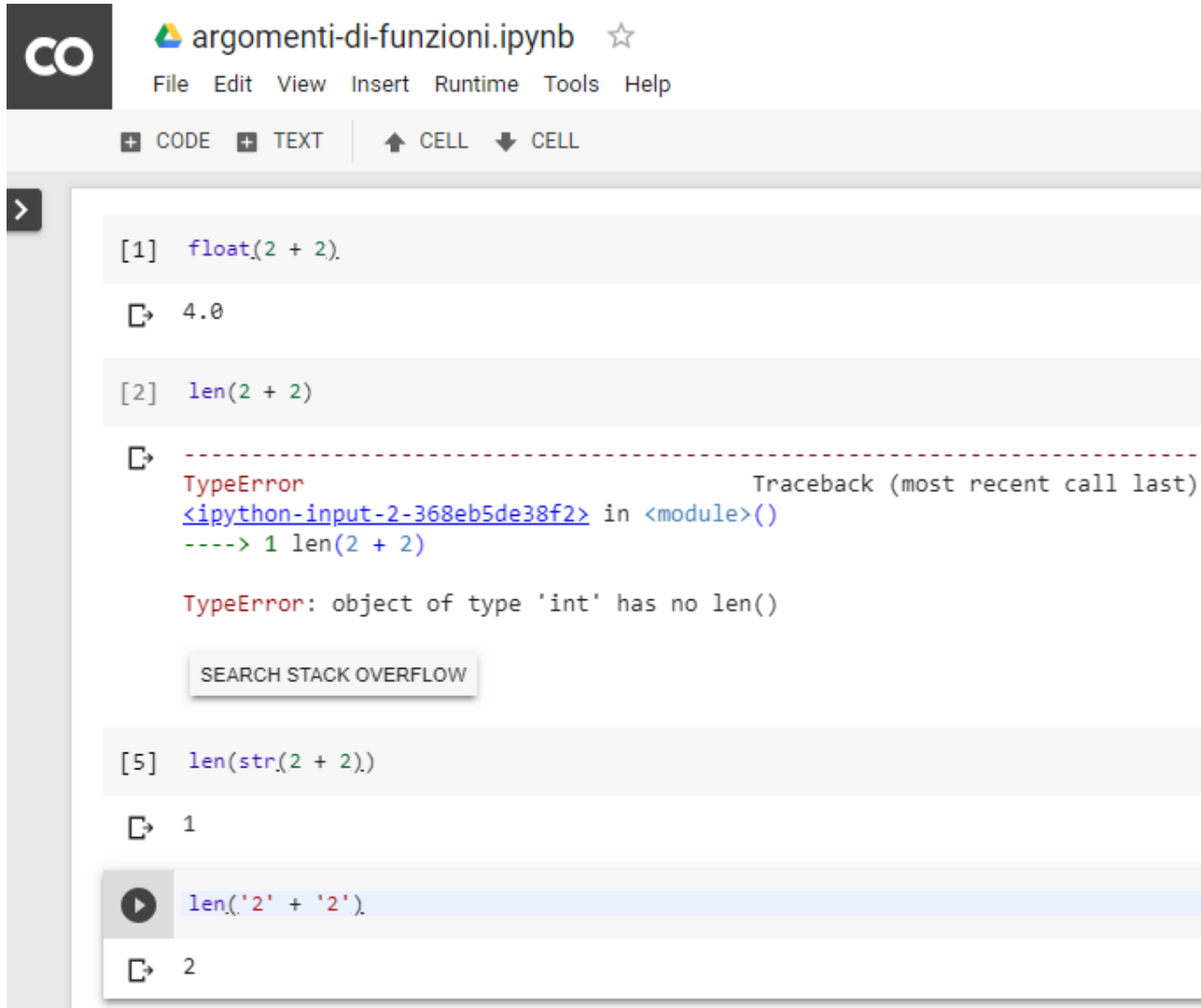
Argomenti di funzione

Gli argomenti di una chiamata di funzione sono costituiti da espressioni.

In ciascuno degli argomenti può quindi comparire un'espressione Python qualsiasi, purché produca un valore di un tipo previsto dalla funzione (in caso contrario si otterrà un messaggio d'errore).

Ne consegue, come caso particolare, che una chiamata di funzione può contenere tra le espressioni dei suoi argomenti altre chiamate di funzioni (chiamate annidate).

Esempi argomenti di funzione



The screenshot shows a Jupyter Notebook titled "argomenti-di-funzioni.ipynb". The interface includes a top bar with a "co" logo, a menu (File, Edit, View, Insert, Runtime, Tools, Help), and a toolbar with buttons for adding code or text cells and navigating between cells. The notebook contains four code cells:

- Cell [1]: `float(2 + 2).` Output: `4.0`
- Cell [2]: `len(2 + 2)` Output: A `TypeError` traceback. The error message is: `TypeError: object of type 'int' has no len()`. A "SEARCH STACK OVERFLOW" button is visible below the error.
- Cell [5]: `len(str(2 + 2))` Output: `1`
- Cell [6]: `len('2' + '2').` Output: `2`

Principali funzioni della libreria `math`

funzione	descrizione
<code>cos (x)</code>	coseno (x deve essere espresso in radianti)
<code>sin (x)</code>	seno (come sopra)
<code>tan (x)</code>	tangente (come sopra)
<code>acos (x)</code>	arco-coseno (x deve essere nell'intervallo $[-1, 1]$)
<code>asin (x)</code>	arco-seno (come sopra)
<code>atan (x)</code>	arco-tangente
<code>radians (x)</code>	converte in radianti un angolo espresso in gradi
<code>degrees (x)</code>	converte in gradi un angolo espresso in radianti
<code>exp (x)</code>	e^x
<code>log (x)</code>	$\ln x$
<code>log (x, b)</code>	$\log_b x$
<code>log10 (x)</code>	$\log_{10} x$
<code>pow (x, y)</code>	x^y
<code>sqrt (x)</code>	\sqrt{x}

Tutte le funzioni di questa libreria restituiscono un numero **frazionario**.

La libreria `random`

funzione	descrizione
<code>random()</code>	genera un numero reale nell'intervallo $[0, 1)$, da una distribuzione di probabilità uniforme (cioè, ogni valore di tale intervallo ha la stessa probabilità di essere “estratto”)
<code>uniform(a, b)</code>	come sopra, nell'intervallo $[a, b)$ (gli argomenti sono numeri qualsiasi)
<code>randint(a, b)</code>	genera un numero intero nell'insieme $\{a, \dots, b\}$, da una distribuzione di probabilità uniforme (gli argomenti devono essere numeri interi)

Ogni chiamata di tali funzioni produce un numero pseudo-casuale, indipendente (in teoria) dai valori prodotti dalle chiamate precedenti.

from import

Per poter chiamare una funzione di librerie come `math` e `random` è necessario utilizzare la combinazione `from import`

Sintassi:

```
from nome_libreria import nome_funzione
```

- `nome_libreria` è il nome simbolico di una libreria
- `nome_funzione` può essere:
 - il nome di una specifica funzione di tale libreria (questo consentirà di usare solo tale funzione)
 - il simbolo `*` indicante tutte le funzioni di tale libreria

Se la combinazione `from import` non viene usata correttamente, la chiamata di funzione produrrà un errore, come mostrato negli esempi seguenti.

Costanti matematiche

Oltre a varie funzioni, nella libreria `math` sono definite due variabili che contengono il valore (approssimato) delle costanti matematiche π (3,14. . .) ed e (la base dei logaritmi naturali: 2,71. . .):

- `pi`
- `e`

Per usare queste costanti è necessaria la combinazione `from import`, in una delle due versioni:

- `from math import *`
 - `from math import nome_variabile`
- dove `nome_variabile` dovrà essere `pi` oppure `e`

Esempi math

```
[1] cos(pi / 2)
```



```
-----  
NameError                                Traceback (most recent call last)  
<ipython-input-1-1c3767ed60fe> in <module>()  
----> 1 cos(pi / 2)  
  
NameError: name 'cos' is not defined
```

SEARCH STACK OVERFLOW

```
[2] from math import cos  
    from math import pi  
  
    cos(pi / 2)
```



```
6.123233995736766e-17
```

```
[6] from math import cos  
    from math import pi  
  
    if cos(pi / 2) == 0:  
        print('OK')
```



```
from math import cos  
from math import pi  
import sys  
  
if cos(pi / 2) < sys.float_info.epsilon:  
    print('OK')
```



```
OK
```


Esempi random

```
[2] from random import *  
     random()
```

0.6589012566357493

```
[3] random()
```

0.8476015372012984

```
[4] random()
```

0.09743656069098616

```
[5] uniform(-2, 2)
```


-1.8376847371489315

```
[6] uniform(-2, 2)
```

-1.9511529669296759

```
[7] randint(1, 10)
```

7

 randint(1, 10)

6

Definizione di nuove funzioni

Oltre ad usare le funzioni predefinite, è possibile creare nuove funzioni.

La definizione di una nuova funzione è composta dai seguenti elementi:

- il nome della funzione
- il numero dei suoi argomenti
- la sequenza di istruzioni, detta corpo della funzione, che dovranno essere eseguite quando la funzione sarà chiamata

La definizione di una nuova funzione avviene attraverso l'uso della keyword `def`

def

Sintassi:

```
def nome_funzione (par1, ..., parn):  
    corpo_della_funzione
```

- `nome_funzione` è un nome simbolico scelto dal programmatore, con gli stessi vincoli a cui sono soggetti i nomi delle variabili
- `par1, ..., parn` sono nomi (scelti dal programmatore) di variabili, dette parametri della funzione, alle quali l'interprete assegnerà i valori degli argomenti che verranno indicati nella chiamata della funzione
- `corpo_della_funzione` è una sequenza di una o più istruzioni qualsiasi, ciascuna scritta in una riga distinta, con un rientro di almeno un carattere, identico per tutte le istruzioni

La prima riga della definizione (contenente i nomi della funzione e dei parametri) è detta intestazione della funzione.

return

Per concludere l'esecuzione di una funzione e indicare il valore che la funzione dovrà restituire come risultato della sua chiamata si usa l'istruzione `return`.

Sintassi:

`return espressione`

dove `espressione` è un'espressione Python qualsiasi.

L'istruzione `return` può essere usata solo solo all'interno di una funzione.

Se una funzione non deve restituire alcun valore:

- l'istruzione `return` può essere usata, senza l'indicazione di alcuna espressione, per concludere l'esecuzione della funzione
- se non si usa l'istruzione `return`, l'esecuzione della funzione terminerà dopo l'esecuzione dell'ultima istruzione del corpo

Definizione e chiamata di una funzione

L'esecuzione dell'istruzione `def` non comporta l'esecuzione delle istruzioni della funzione: tali istruzioni verranno eseguite solo attraverso una chiamata della funzione.

L'istruzione `def` dovrà essere eseguita una sola volta, prima di qualsiasi chiamata della funzione. In caso contrario, il nome della funzione non sarà riconosciuto dall'interprete e la chiamata produrrà un messaggio di errore.

Esecuzione della chiamata di funzione

L'interprete esegue la chiamata di una funzione nel modo seguente:

1. copia il valore di ciascun argomento nel parametro corrispondente (quindi tali variabili possiedono già un valore nel momento in cui inizia l'esecuzione della funzione)
2. esegue le istruzioni del corpo della funzione, fino all'istruzione `return` oppure fino all'ultima istruzione del corpo
3. se l'eventuale istruzione `return` è seguita da un'espressione, restituisce il valore di tale espressione come risultato della chiamata

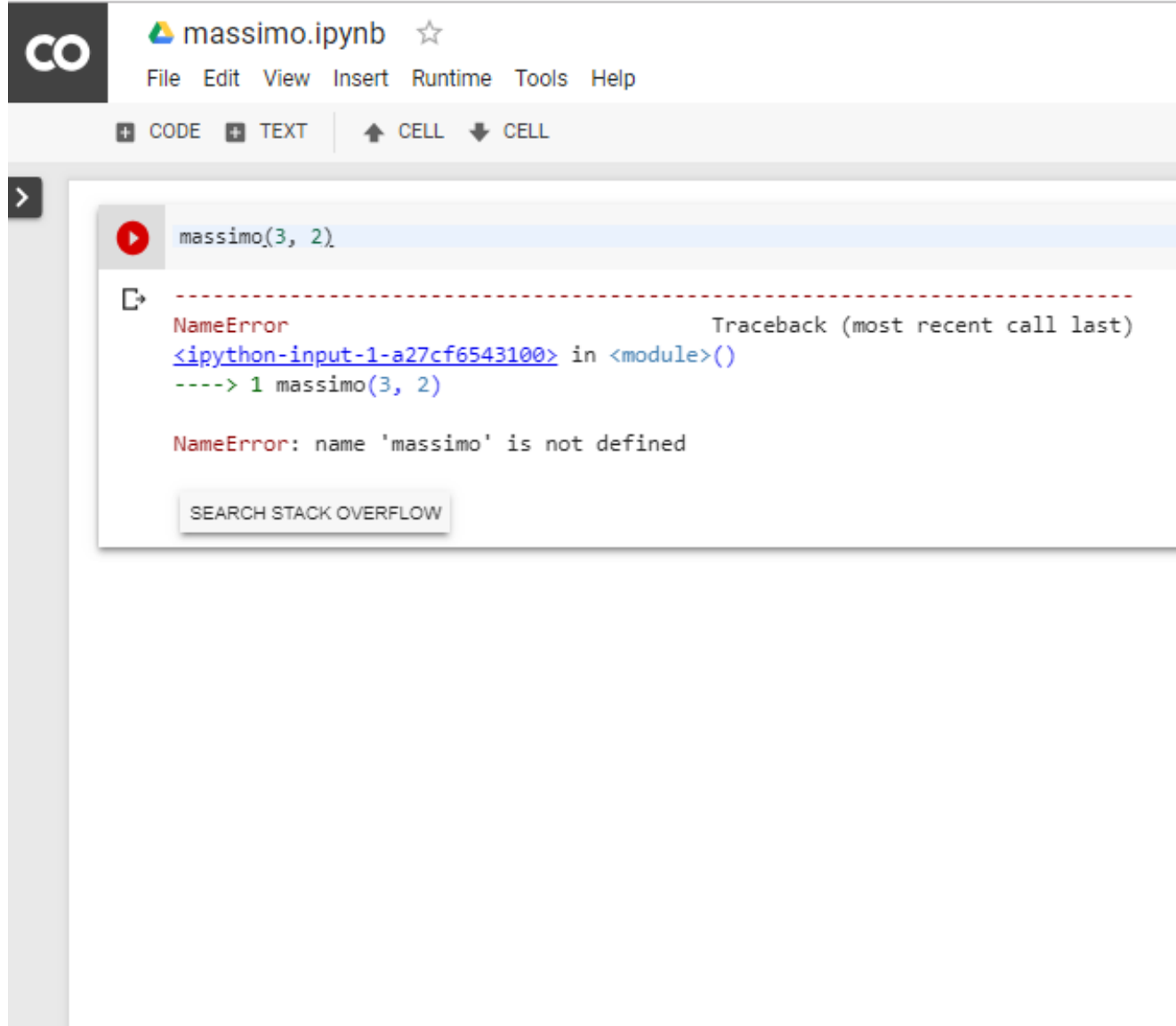
Definizione di funzioni: esempio

Si supponga di voler definire una funzione che restituisca il più grande tra due numeri ricevuti come argomenti.

Scegliendo `massimo` come nome della funzione, e `a` e `b` come nomi dei suoi parametri, la funzione può essere definita come segue:

```
def massimo(a, b):  
    if a > b:  
        return a  
    else:  
        return b
```

Esecuzione di funzione



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 filename 'massimo.ipynb', and a star icon. Below this is a menu bar with 'File', 'Edit', 'View', 'Insert', 'Runtime', 'Tools', and 'Help'. A secondary bar contains buttons for '+ CODE', '+ TEXT', and 'CELL' (with up and down arrows). The main area displays a code cell with a red play button icon and the code `massimo(3, 2)`. Below the code, a traceback is shown, indicating a `NameError` because the variable `massimo` is not defined. The traceback text includes 'Traceback (most recent call last)', the IPython input prompt, and the line of code that caused the error. At the bottom of the traceback, there is a button labeled 'SEARCH STACK OVERFLOW'.

```
massimo(3, 2)
```

NameError Traceback (most recent call last)
 <ipython-input-1-a27cf6543100> in <module>()
 ----> 1 massimo(3, 2)

NameError: name 'massimo' is not defined

SEARCH STACK OVERFLOW

Esecuzione di funzione



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 text 'massimo.ipynb' and a star icon, followed by a menu bar with 'File', 'Edit', 'View', 'Insert', 'Runtime', 'Tools', and 'Help'. Below the menu bar is a toolbar with buttons for '+ CODE', '+ TEXT', '↑ CELL', and '↓ CELL'. The notebook content is divided into two cells. The first cell contains a red play button icon followed by the code: `massimo(3,2)` and a function definition: `def massimo(a, b):`, `if a > b:`, `return a`, `else:`, `return b`. The second cell contains a copy icon followed by a traceback for a `NameError`. The traceback text is: `NameError` Traceback (most recent call last), `<ipython-input-2-22e4de05c284> in <module>()`, `----> 1 massimo(3,2)`, `2`, `3 def massimo(a, b):`, `4 if a > b:`, `5 return a`. Below the traceback is the message `NameError: name 'massimo' is not defined` and a button labeled 'SEARCH STACK OVERFLOW'.

```
massimo(3,2)

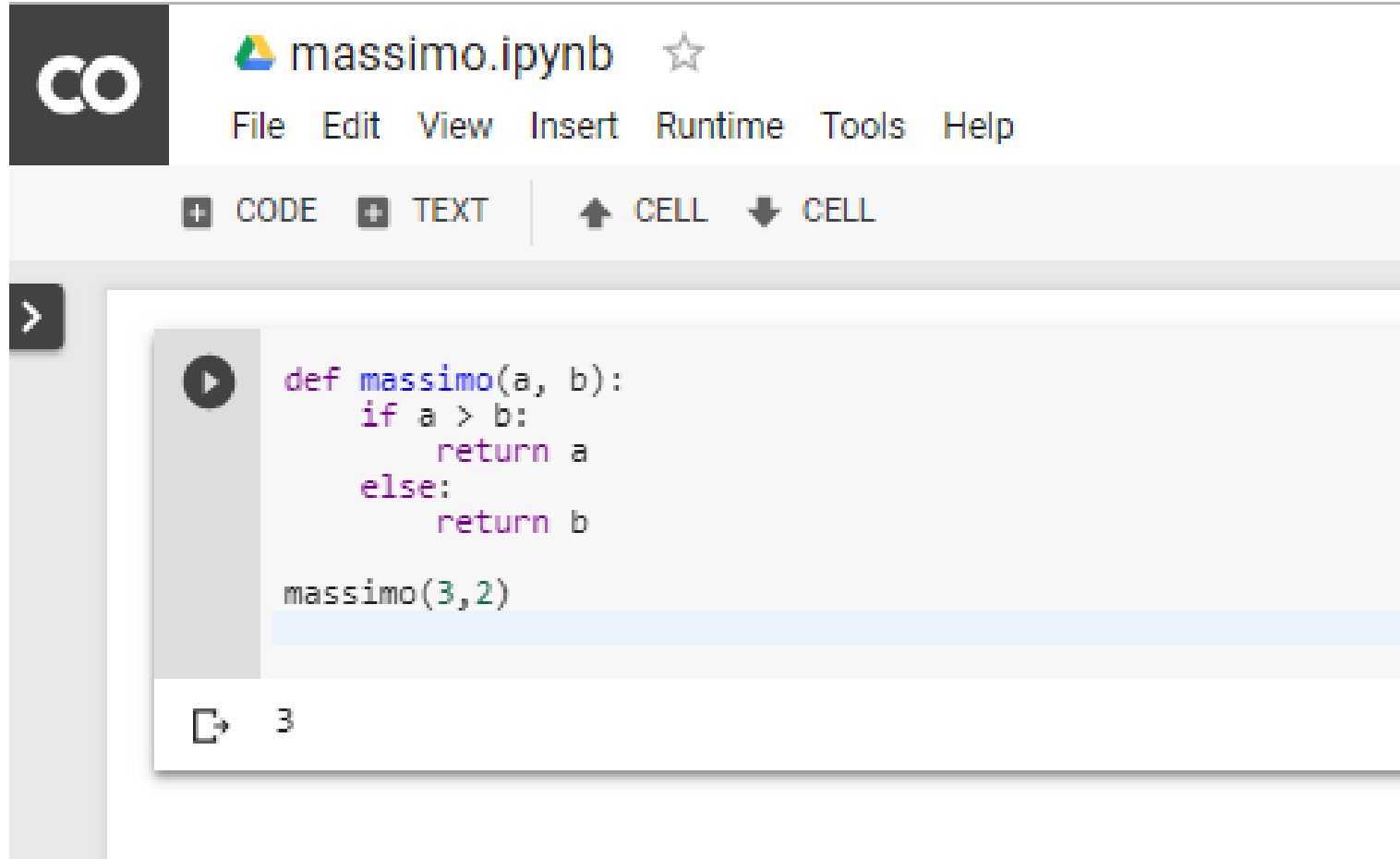
def massimo(a, b):
    if a > b:
        return a
    else:
        return b
```

```
-----
NameError                                Traceback (most recent call last)
<ipython-input-2-22e4de05c284> in <module>()
----> 1 massimo(3,2)
      2
      3 def massimo(a, b):
      4     if a > b:
      5         return a

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

SEARCH STACK OVERFLOW

Esecuzione di funzione



The screenshot shows a Jupyter Notebook interface. At the top left is the 'CO' logo. The notebook title is 'massimo.ipynb' with a star icon. Below the title is a menu bar with 'File', 'Edit', 'View', 'Insert', 'Runtime', 'Tools', and 'Help'. A toolbar contains buttons for '+ CODE', '+ TEXT', '↑ CELL', and '↓ CELL'. On the left side, there is a vertical sidebar with a '>' button. The main area displays a code cell with a play button icon. The code defines a function 'massimo(a, b)' that returns 'a' if 'a > b' and 'b' otherwise, followed by a call to 'massimo(3,2)'. Below the code cell, a small box shows the output '3'.

```
def massimo(a, b):  
    if a > b:  
        return a  
    else:  
        return b  
  
massimo(3,2)
```

3

while

Sintassi:

```
while espr_cond:  
    sequenza_di_istruzioni
```

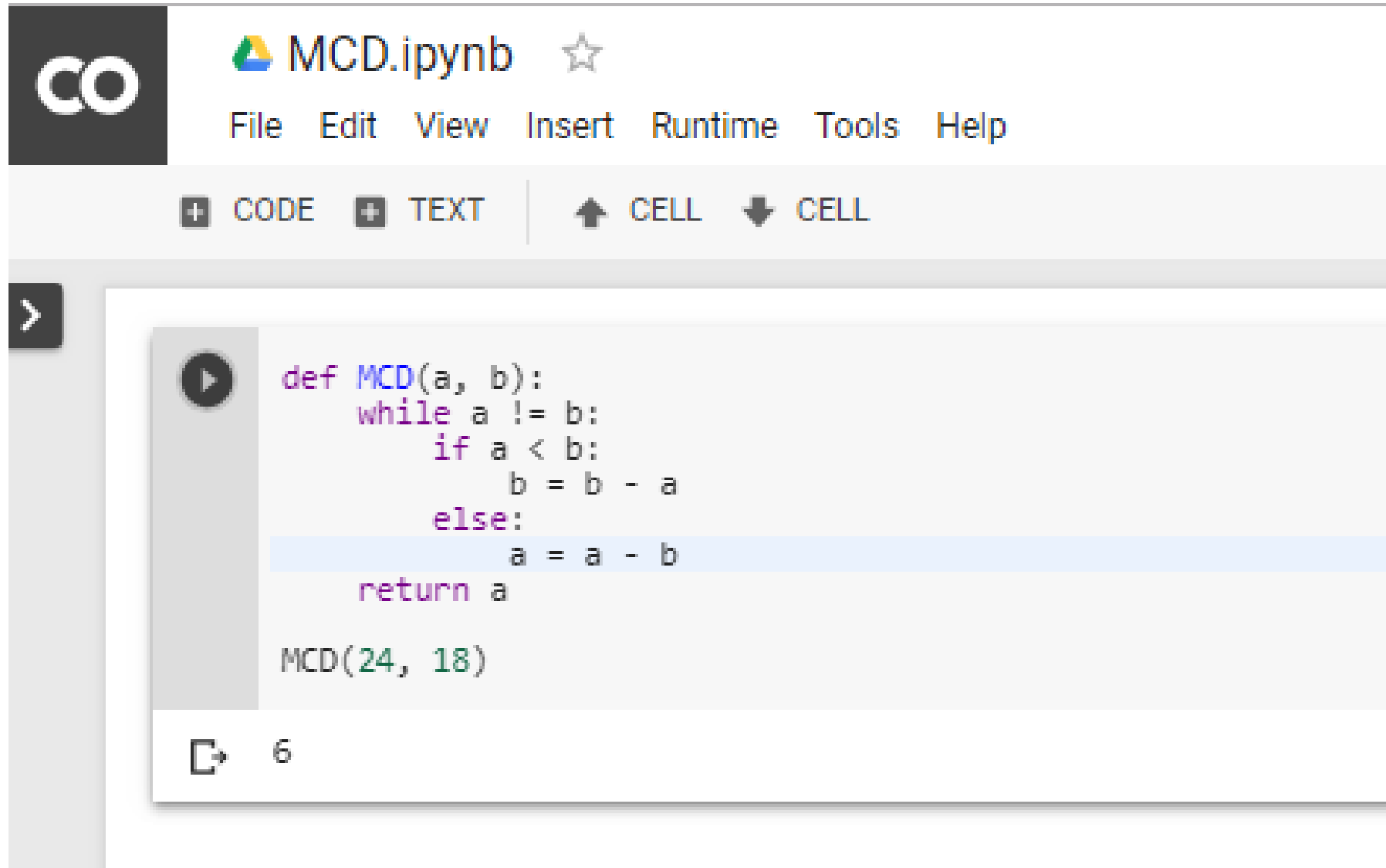
- la keyword `while` deve essere scritta senza rientri
- `espr_cond` è una espressione condizionale qualsiasi
- `sequenza_di_istruzioni` consiste in una o più istruzioni qualsiasi. Ciascuna di tali istruzioni deve essere scritta in una riga distinta, con un rientro di almeno un carattere. Il rientro deve essere identico per tutte le istruzioni della sequenza

Massimo comun divisore

Calcolo del massimo comun divisore con l'algoritmo di Euclide

```
def MCD(a, b):  
    while a != b:  
        if a < b:  
            b = b - a  
        else:  
            a = a - b  
    return a
```

Massimo comun divisore: esempio



The screenshot shows a Jupyter Notebook interface. At the top, there is a dark grey header with the 'CO' logo on the left and the file name 'MCD.ipynb' with a star icon on the right. Below the header is a menu bar with the following items: File, Edit, View, Insert, Runtime, Tools, and Help. Underneath the menu bar is a toolbar with buttons for adding code cells (+ CODE), adding text cells (+ TEXT), moving cells up (↑ CELL), and moving cells down (↓ CELL). The main area of the notebook contains a single code cell. On the left side of this cell is a play button icon. The code inside the cell is a Python function definition for the Greatest Common Divisor (MCD) and a function call. The function is defined as follows:

```
def MCD(a, b):  
    while a != b:  
        if a < b:  
            b = b - a  
        else:  
            a = a - b  
    return a
```

The line `a = a - b` is highlighted in light blue. Below the function definition, the function is called with the arguments 24 and 18:

```
MCD(24, 18)
```

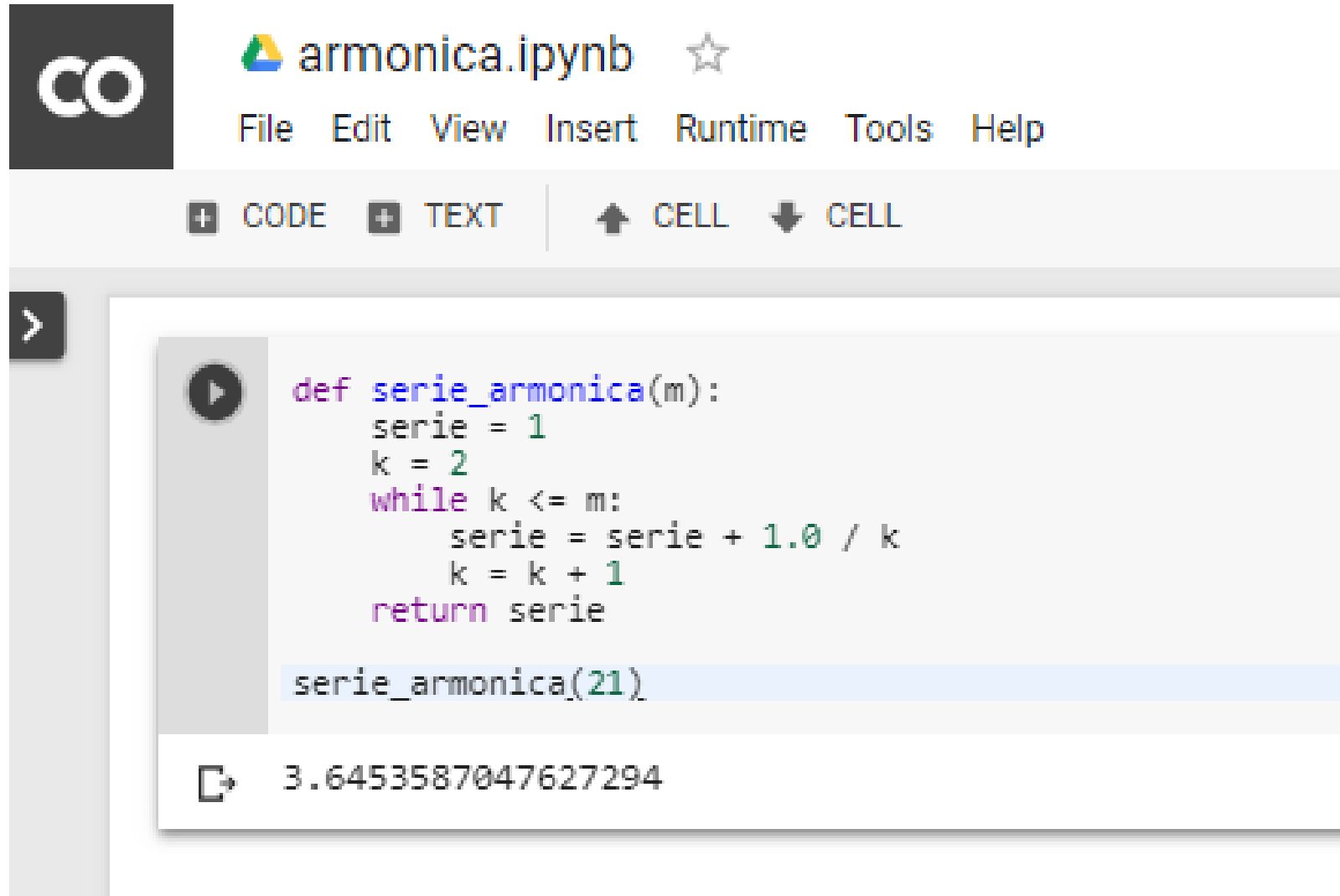
At the bottom of the code cell, there is a small icon of a document with an arrow pointing right, followed by the number 6, which represents the output of the function call.

Serie armonica

Calcolo della somma dei primi m termini della serie armonica:

```
def serie_armonica(m):  
    serie = 1  
    k = 2  
    while k <= m:  
        serie = serie + 1.0 / k  
        k = k + 1  
    return serie
```

Serie armonica: esempio



The screenshot shows a Jupyter Notebook window with the title "armonica.ipynb". The interface includes a menu bar with "File", "Edit", "View", "Insert", "Runtime", "Tools", and "Help". Below the menu bar is a toolbar with buttons for adding code or text cells, and moving between cells. The main area contains a single code cell with the following Python code:

```
def serie_armonica(m):  
    serie = 1  
    k = 2  
    while k <= m:  
        serie = serie + 1.0 / k  
        k = k + 1  
    return serie  
  
serie_armonica(21)
```

The output of the code cell is displayed below the code, showing the result of the function call: 3.6453587047627294.

Chiamate di funzioni all'interno di altre funzioni

Nelle istruzioni del corpo di una funzione possono comparire chiamate di altre funzioni, sia predefinite che definite dall'utente.

Se si vuole chiamare una funzione predefinita appartenente a una delle librerie Python (come `math` o `random`) sarà necessario inserire prima della chiamata la corrispondente combinazione `from import`

`from import` viene di norma inserita all'inizio del file che contiene il codice.

Esempio

```
from math import sqrt
```


```
def ipotenusa (a, b):
```

```
    return sqrt(a ** 2 + b ** 2)
```



elevamento a potenza

ipotenusa



ipotenusa.ipynb ☆

File Edit View Insert Runtime Tools Help

+ CODE + TEXT ↑ CELL ↓ CELL

>

```
from math import sqrt

def ipotenusa (a, b):
    return sqrt(a ** 2 + b ** 2)

ipotenusa(2, 3)
```

3.605551275463989

Esempio

La funzione seguente calcola la lunghezza della circonferenza di un cerchio, dato il suo raggio, usando la variabile `pi` definita nella libreria `math`

```
from math import pi
```

```
def circ(raggio):  
    circonferenza = 2 * pi * raggio  
    return circonferenza
```

Chiamare funzioni all'esterno

Per poter chiamare dall'interno di una funzione un'altra funzione definita dall'utente sono disponibili due alternative:

1. la definizione delle due funzioni deve trovarsi nello stesso file
2. le due funzioni possono essere definite in file diversi, ma tali file dovranno trovarsi in una stessa cartella e nel file che contiene la chiamata dell'altra funzione si dovrà inserire l'istruzione

```
from nomefile import nomefunzione
```

dove:

- `nomefile` è il nome del file che contiene la definizione dell'altra funzione (senza l'estensione `.py`)
- `nomefunzione` è il nome di tale funzione

is_numero_primo

```
def is_numero_primo(numero):  
    divisore = 2  
    while divisore <= numero / 2:  
        if numero % divisore == 0:  
            return False  
        else:  
            divisore = divisore + 1  
    return True
```

stampa_numeri_primi

```
def stampa_numeri_primi(n):  
    print("I numeri primi tra 1 e", n, "sono:")  
    k = 1  
    while k <= n:  
        if numero_primo(k) == True:  
            print(k)  
        k = k + 1
```

Definizione di funzioni nello stesso file

```
def is_numero_primo(numero):  
    divisore = 2  
    while divisore <= numero / 2:  
        if numero % divisore == 0:  
            return False  
        else:  
            divisore = divisore + 1  
    return True  
  
def stampa_numeri_primi(n):  
    print("I numeri primi tra 1 e", n, "sono:")  
    k = 1  
    while k <= n:  
        if is_numero_primo(k) == True:  
            print(k)  
        k = k + 1  
  
stampa_numeri_primi(13)
```

```
☞ I numeri primi tra 1 e 13 sono:  
1  
2  
3  
5  
7  
11  
13
```

Definizione di funzioni su file diversi

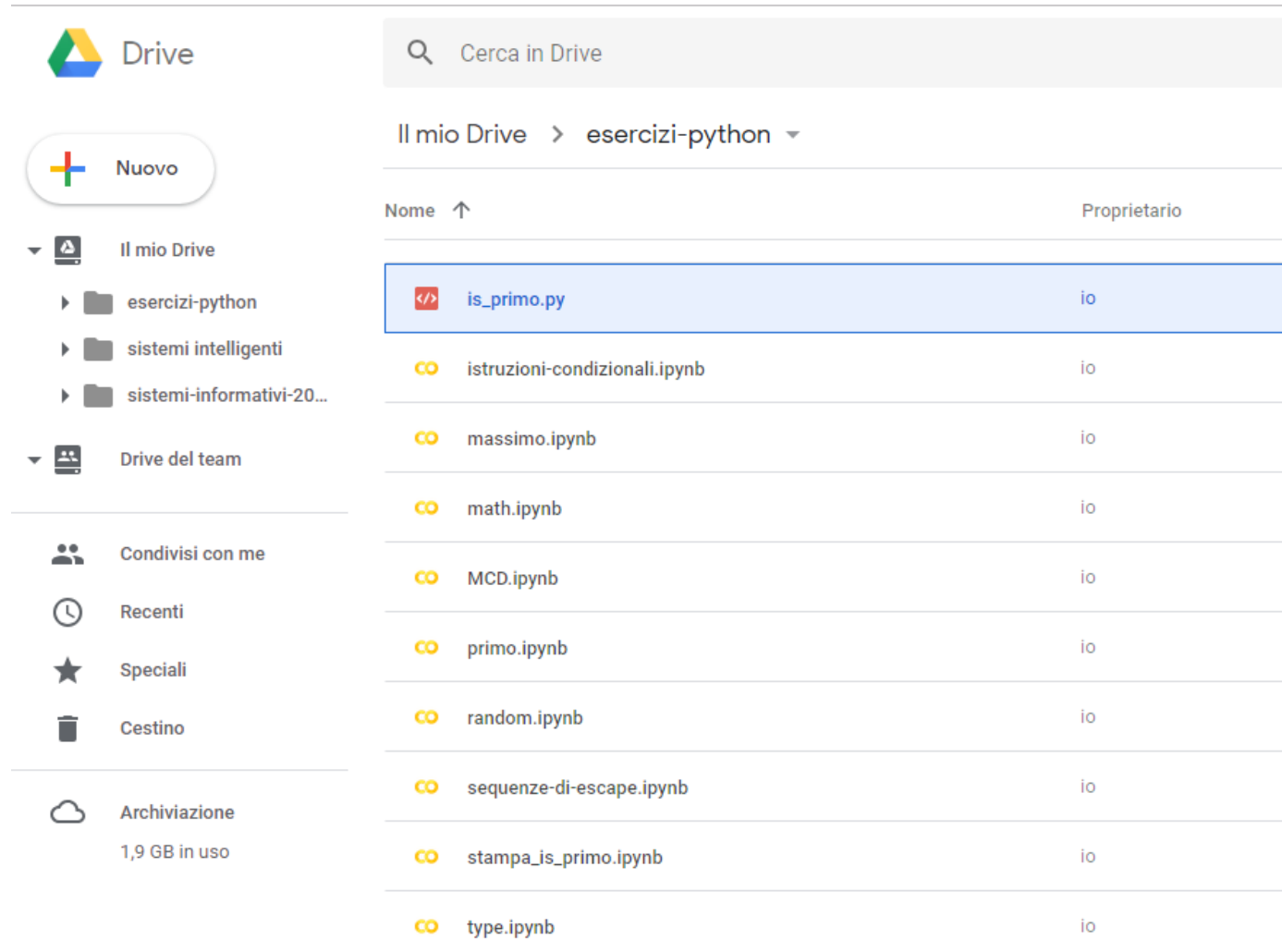
1. Creiamo sul nostro pc un file di testo con un editor che non inserisca informazioni di formattazione, per esempio Notepad++
<https://notepad-plus-plus.org>













```
File  Modifica  Cerca  Visualizza  Formato  Linguaggio  Configurazione  Strumenti  Macro  Esegui
[Icons]
is_primo.py x
1  def is_numero_primo(numero):
2      divisore = 2
3      while divisore <= numero / 2:
4          if numero % divisore == 0:
5              return False
6          else:
7              divisore = divisore + 1
8      return True
```


Definizione di funzioni su file diversi

2. Carichiamo il file is_primo.py su Google Drive



The screenshot shows the Google Drive web interface. On the left sidebar, the 'Il mio Drive' section is expanded, showing a folder named 'esercizi-python'. The main area displays the contents of this folder as a table. The file 'is_primo.py' is highlighted with a blue selection bar. Below it, several Jupyter Notebook files (ipynb) are listed, all owned by 'io'.

Nome	Proprietario
 is_primo.py	io
 istruzioni-condizionali.ipynb	io
 massimo.ipynb	io
 math.ipynb	io
 MCD.ipynb	io
 primo.ipynb	io
 random.ipynb	io
 sequenze-di-escape.ipynb	io
 stampa_is_primo.ipynb	io
 type.ipynb	io

Definizione di funzioni su file diversi

3. Creiamo un file Colab denominato stampa_is_primo.ipynb

```
from google.colab import drive
drive.mount('/content/gdrive', force_remount=True)

%cd '/content/gdrive/My Drive/esercizi-python/'
!ls

!cat '/content/gdrive/My Drive/esercizi-python/is_primo.py'

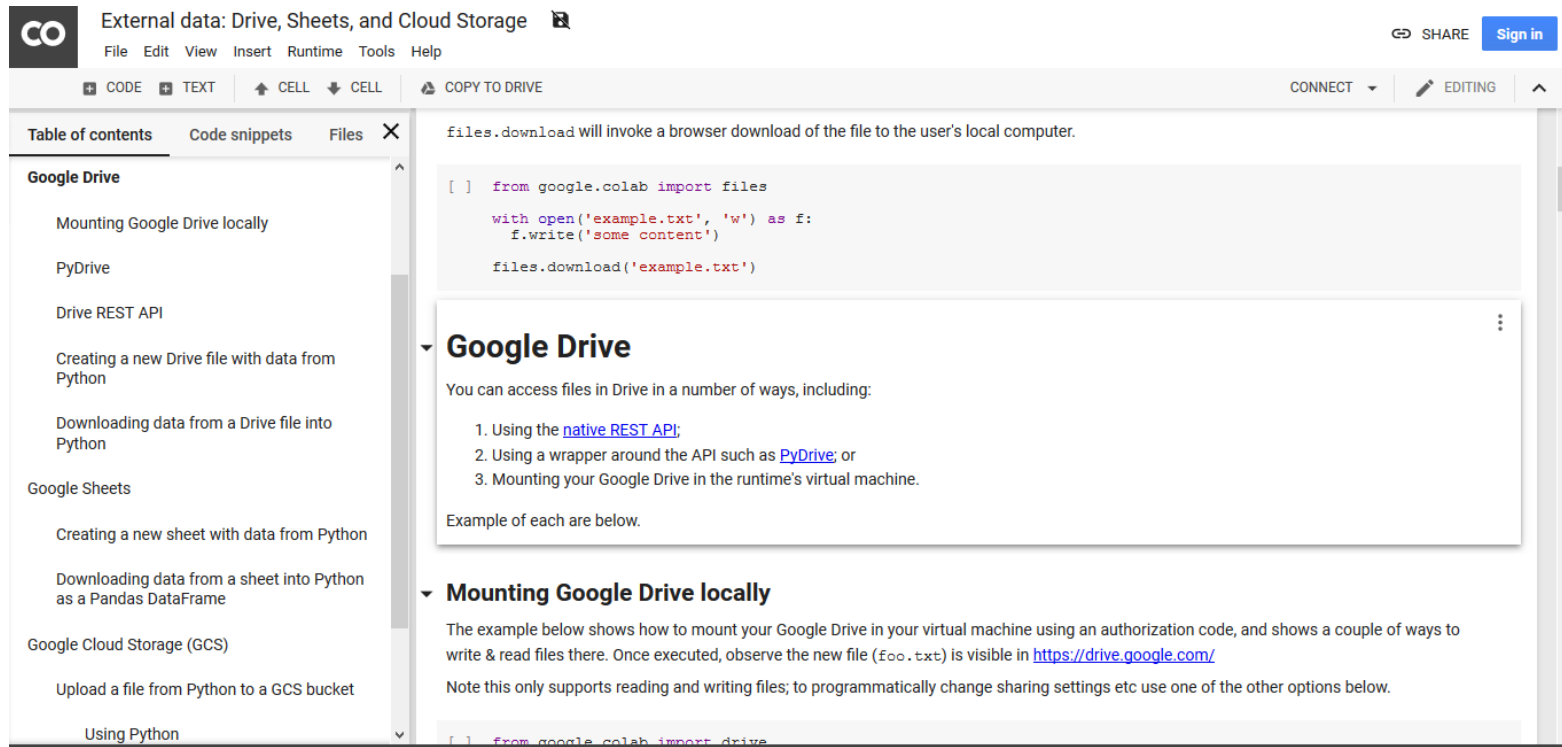
import is_primo

def stampa_numeri_primi(n):
    print("I numeri primi tra 1 e", n, "sono:")
    k = 1
    while k <= n:
        if is_primo.is_numero_primo(k) == True:
            print(k)
            k = k + 1

stampa_numeri_primi(13)
```

Google Drive REST API

Abbiamo utilizzato le Google Drive REST API per accedere al contenuto di Google Drive (è richiesta una fase di autenticazione)



The screenshot shows a Google Colab notebook titled "External data: Drive, Sheets, and Cloud Storage". The left sidebar contains a "Table of contents" with sections like "Google Drive", "Google Sheets", and "Google Cloud Storage (GCS)". The main area displays the "Google Drive" section, which includes a list of ways to access files in Drive and a code snippet for downloading a file.

files.download will invoke a browser download of the file to the user's local computer.

```
[ ] from google.colab import files  
with open('example.txt', 'w') as f:  
    f.write('some content')  
files.download('example.txt')
```

Google Drive

You can access files in Drive in a number of ways, including:

1. Using the [native REST API](#);
2. Using a wrapper around the API such as [PyDrive](#); or
3. Mounting your Google Drive in the runtime's virtual machine.

Example of each are below.

Mounting Google Drive locally

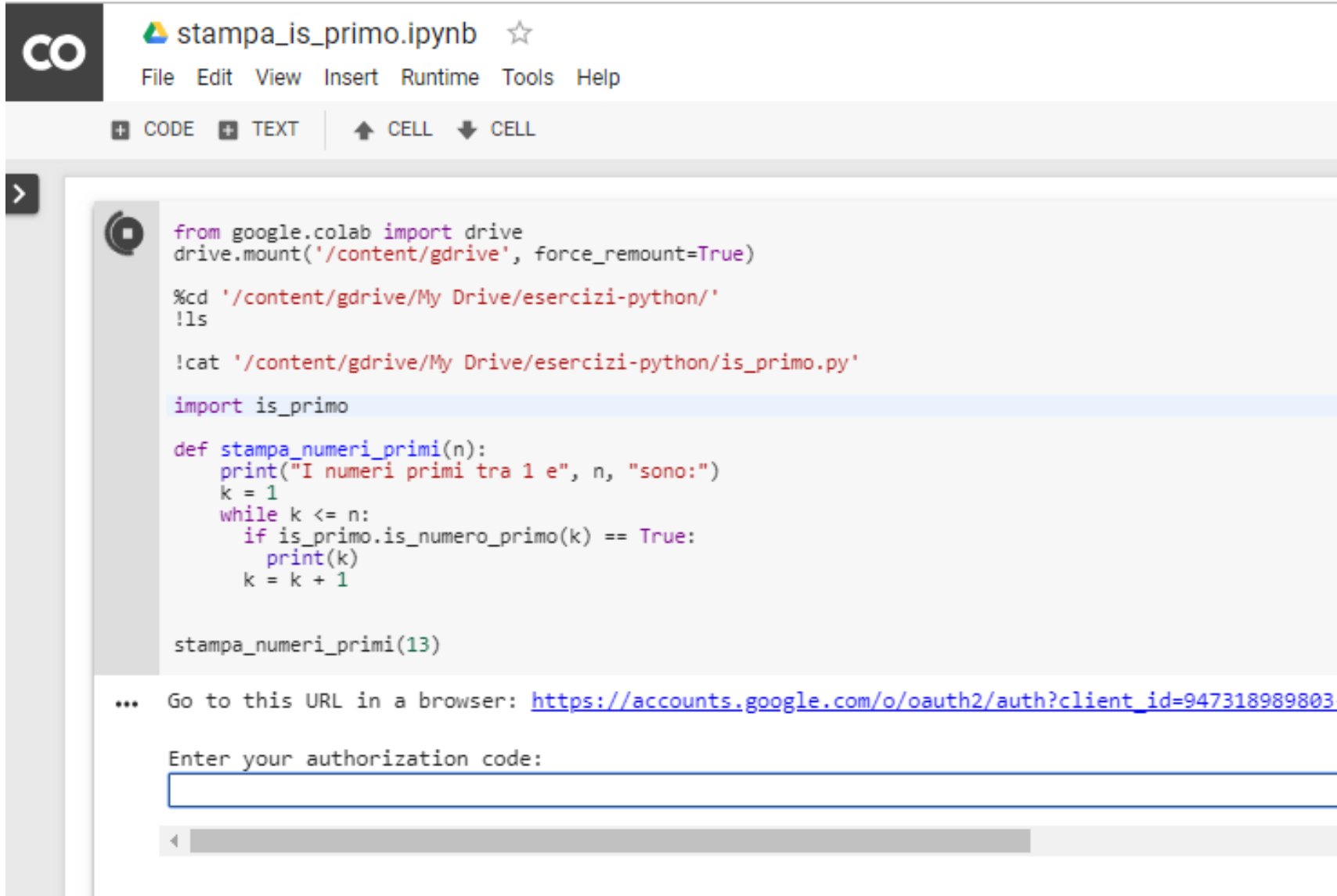
The example below shows how to mount your Google Drive in your virtual machine using an authorization code, and shows a couple of ways to write & read files there. Once executed, observe the new file (foo.txt) is visible in <https://drive.google.com/>

Note this only supports reading and writing files; to programmatically change sharing settings etc use one of the other options below.

```
[ ] from google.colab import drive
```

<https://colab.research.google.com/notebooks/io.ipynb#scrollTo=c2W5A2px3doP>

Montare il drive



The screenshot shows a Google Colab notebook interface. At the top, the notebook title is 'stampa_is_primo.ipynb' with a star icon. Below the title is a menu bar with 'File', 'Edit', 'View', 'Insert', 'Runtime', 'Tools', and 'Help'. A toolbar below the menu bar contains icons for adding code, text, and cells. The main area of the notebook displays a code cell with the following Python code:

```
from google.colab import drive
drive.mount('/content/gdrive', force_remount=True)

%cd '/content/gdrive/My Drive/esercizi-python/'
!ls

!cat '/content/gdrive/My Drive/esercizi-python/is_primo.py'

import is_primo

def stampa_numeri_primi(n):
    print("I numeri primi tra 1 e", n, "sono:")
    k = 1
    while k <= n:
        if is_primo.is_numero_primo(k) == True:
            print(k)
        k = k + 1


stampa_numeri_primi(13)
```

Below the code cell, there is a text prompt: "... Go to this URL in a browser: https://accounts.google.com/o/oauth2/auth?client_id=947318989803." followed by the instruction "Enter your authorization code:" and an empty text input field.

Output

```
Mounted at /content/gdrive
/content/gdrive/My Drive/esercizi-python
argomenti-di-funzioni.ipynb    istruzioni-condizionali.ipynb
armonica.ipynb                massimo.ipynb
commenti.ipynb                math.ipynb
espressioni-aritmetiche.ipynb  MCD.ipynb
espressioni-booleane.ipynb    primo.ipynb
exception-error.ipynb         __pycache__
funzioni-built-in.ipynb       random.ipynb
input.ipynb                   sequenze-di-escape.ipynb
ipotenusa.ipynb               stampa_is_primo.ipynb
is_primo.py                   type.ipynb
def is_numero_primo(numero):
    divisore = 2
    while divisore <= numero / 2:
        if numero % divisore == 0:
            return False
        else:
            divisore = divisore + 1
    return True
I numeri primi tra 1 e 13 sono:
1
2
3
5
7
11
13
```

Alternativa usando sys



The image shows a Google Colab interface. At the top left is the Colab logo (a black square with 'CO' in white). To its right is the file name 'stampa_is_primo.ipynb' with a star icon. Below this is a menu bar with 'File', 'Edit', 'View', 'Insert', 'Runtime', 'Tools', and 'Help'. Below the menu bar is a toolbar with '+ CODE', '+ TEXT', '↑ CELL', and '↓ CELL'. The main area contains a code cell with the following Python code:

```
from google.colab import drive
drive.mount('/content/gdrive', force_remount=True)

import sys
sys.path.append('/content/gdrive/My Drive/esercizi-python/')

import is_primo

def stampa_numeri_primi(n):
    print("I numeri primi tra 1 e", n, "sono:")
    k = 1
    while k <= n:
        if is_primo.is_numero_primo(k) == True:
            print(k)
            k = k + 1

stampa_numeri_primi(13)
```

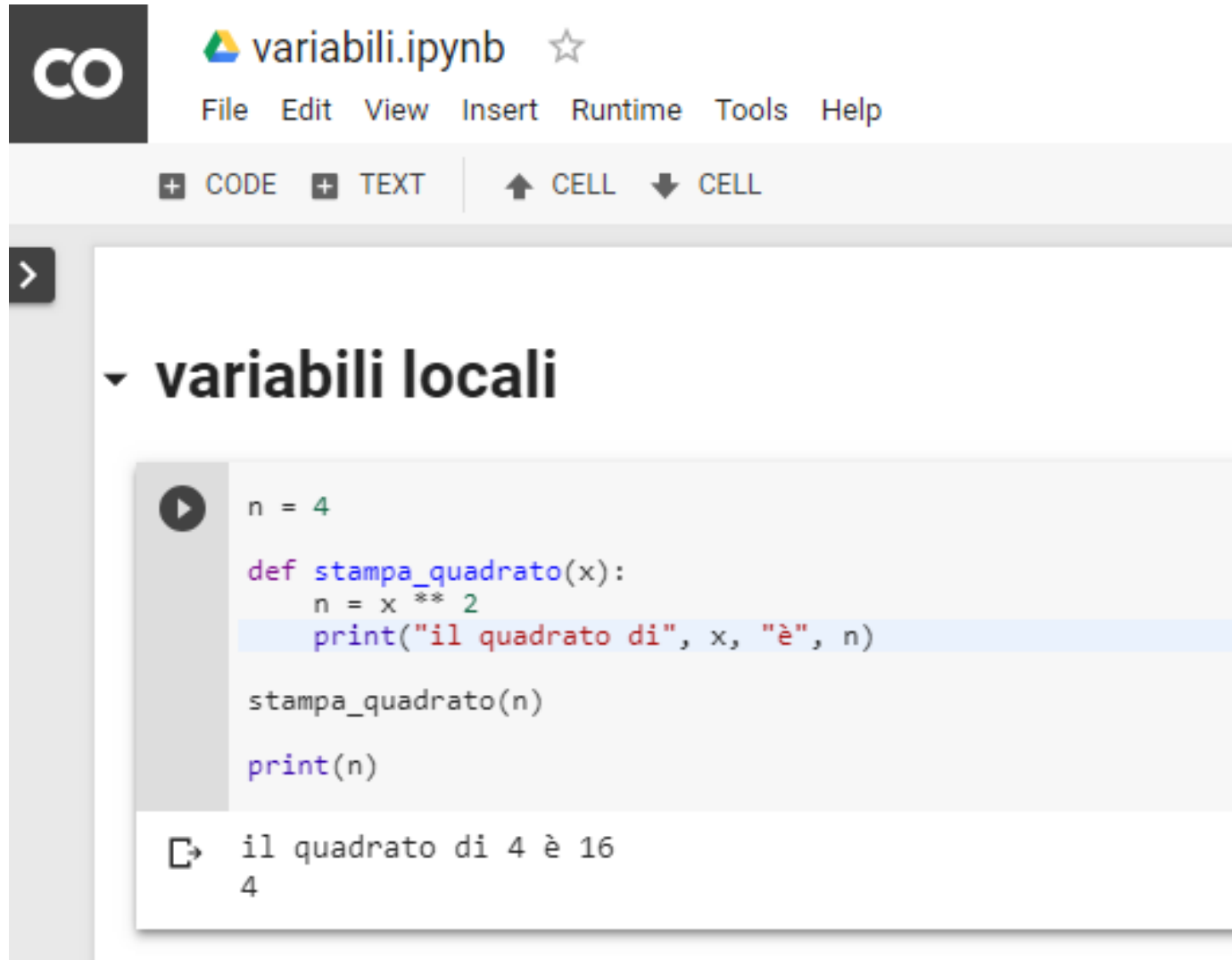
Below the code cell is a terminal output showing the execution results:

```
Mounted at /content/gdrive
I numeri primi tra 1 e 13 sono:
1
2
3
5
7
11
13
```

Variabili locali

I parametri di una funzione e le eventuali altre variabili alle quali viene assegnato un valore all'interno di essa sono dette locali, cioè vengono create dall'interprete nel momento in cui la funzione viene eseguita (con una chiamata) e vengono distrutte quando l'esecuzione della funzione termina.

Variabili locali: esempio



The screenshot shows a Jupyter Notebook interface. At the top, there is a logo with the letters 'co' and the filename 'variabili.ipynb' with a star icon. 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'. On the left side, there is a sidebar with a '>' button and a section titled 'variabili locali'. The main area displays a code cell with a play button icon on the left. The code in the cell is:

```
n = 4

def stampa_quadrato(x):
    n = x ** 2
    print("il quadrato di", x, "è", n)

stampa_quadrato(n)

print(n)
```

Below the code cell, the output is shown in a separate box with a copy icon on the left:

```
il quadrato di 4 è 16
4
```


Variabili globali

Se invece all'interno di una funzione il nome di una variabile (che non sia uno dei parametri) compare in una espressione senza che in precedenza nella funzione sia stato assegnato a essa alcun valore, tale variabile è considerata globale, cioè l'interprete assume che il suo valore sia stato definito nelle istruzioni precedenti la chiamata della funzione.

In questo modo, le istruzioni di una funzione possono accedere al valore di variabile definita nel programma chiamante (se tale variabile non esiste si ottiene un messaggio di errore).

In generale, è preferibile evitare l'uso di variabili globali nelle funzioni, poiché la loro presenza rende più difficile assicurare la correttezza di un programma.

Variabili globali: esempio

variabili globali

```
▶ n = 2

def stampa_globale():
    x = 2
    print(n + 1)

stampa_globale()

print(n)

def stampa_globale_2():
    print(x + 1)

stampa_globale_2()
```

```
✖ 3
  2

-----
NameError                                Traceback (most recent call last)
<ipython-input-9-1edbf4c07eab> in <module>()
     12     print(x + 1)
     13
--> 14 stampa_globale_2()

<ipython-input-9-1edbf4c07eab> in stampa_globale_2()
     10
     11 def stampa_globale_2():
--> 12     print(x + 1)
     13
     14 stampa_globale_2()

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

SEARCH STACK OVERFLOW



**UNIVERSITÀ DEGLI STUDI
DELLA BASILICATA**

*Corso di Visione e Percezione
A.A. 2019/2020*

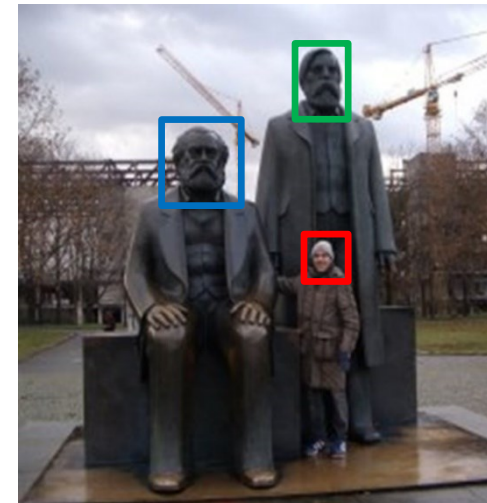
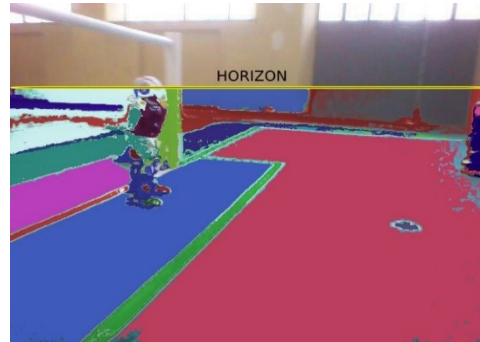
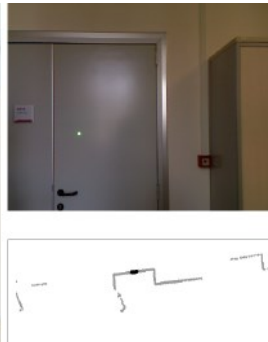
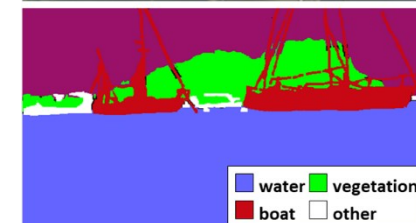
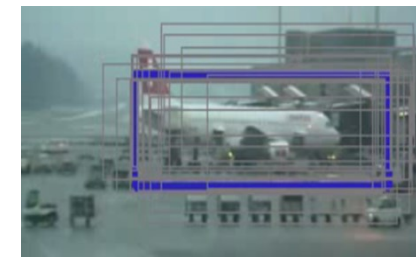
Funzioni in Python

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 2020